UiO : Department of Informatics
University of Oslo

Trade-offs of Adapting Binary Neural Network Ensembles for Multiclass Problems

Master's Thesis

Tor Jan Derek Berstad

Spring 2018

Abstract

The One Versus Rest (OVR) method of building classifiers has fallen out of favor with machine learning researchers in recent years. When used, it is mostly only with older binary linear classifiers such as Support Vector Machines. We suspected that one could also improve the classification performance of more complex neural network architectures by setting them up as individual binary classifiers for each class and combining the output.

In our research, we have tested the OVR style of building a classifier on several modern neural network architectures, including DenseNet, Inception v3, Inception ResNet v2, Xception, NASNet, and MobileNet. We have compared several aspects of their performance during training and testing, in both an OVR style and conventional multiclass single-network style. We have compared hardware resource use, classification speed, and several classification accuracy metrics. Also, we trained and tested a total of 186 networks; 50 of these were multiclass, and 136 were individual binary networks.

Using our final selection of 99 networks (11 multiclass and 88 binary), we compared the results of the 11 multiclass networks with the 11 OVR style networks built from the 88 individual binary classifiers using the Kvasir v2 dataset, which contains thousands of classified frames from colonoscopy videos. We chose The Kvasir dataset as it could provide a good indication of how applicable our method is to clinical research. We found that overall, there was a substantial increase in the average and median classification metrics when using our methods and applications.

Using the OVR style resulted in a 7% increase in average F_1 score, a 1% increase in average accuracy, a 6% increase in Matthews Correlation Coefficient (MCC), a 1% increase in precision and finally a 4% increase in average recall. Specificity remained mostly unchanged. Also, the median values for all of these metrics increased significantly in the OVR style, with the median F_1, MCC and recall scores increasing by over 15%. The most improved network in the OVR configuration saw an increase in F_1 of 45% and an MCC increase of 40%.

However, on average our OVR multi-network style was 7.6 times slower to classify than a single network multiclass implementation. These collective findings lead us to conclude that the OVR method can be applied to modern neural networks structures, and will often result in increased classification accuracy, but at the cost of classification speed.

Acknowledgments

First and foremost I would like to thank my supervisors; Michael Riegler, Pål Halvorsen, and Konstantin Pogorelov. They have provided me with invaluable assistance and guidance throughout the progression of my research.

I would also like to thank some of the other Master's students of the group, specifically Steve Hicks and Rune Borgli. Our discussions about the research material proved extremely helpful at times when the challenges seemed insurmountable.

I want to thank my parents, Elizabeth Glass and Olav Berstad, for giving the means and motivation to pursue this education, love and support my entire life, and an interest in technology.

I would like to thank one of my best friends, Ole Herman Elgesem for the support and guidance throughout our years of study. Although our paths eventually diverged, our friendship has been invaluable to me.

Finally, I would like to extend my gratitude to my partner Ingvild for being there for me when the light at the end of the tunnel was nearly imperceptible. Without you, I don't think this would have been possible.

Tor Jan Derek Berstad,
23.05.2018

Contents

List of Figures

List of Tables

Acronyms

ANN Artificial Neural Network . . . 11

AUC Area Under the Curve . . . 33

AUROC Area Under the Receiver Operating Characteristic curve, aka. AUC-ROC . . . 33

BMLBLN Based Model Last Block Layer Number . . . 38

CAD Computer-Aided Diagnosis . . . 4

CNN Convolutional Neural Network . . . 12

CPU Central Processing Unit . . . v

DNN Deep Neural Network . . . 12

FC Fully Connected . . . 52

FN False Negative . . . 30

FPF False Positive Fraction . . . ix

FPS Frames Per Second . . . 34

FP False Positive . . . 30

GIL Global Interpreter Lock . . . 40

GPU Graphics Processing Unit . . . v

GUI Graphical User Interface . . . 41

I/O Input/Output . . . 40

JSON JavaScript Object Notation 28

LR Learning Rate 38

MCC Matthews Correlation Coefficient i

MLP Multi-layer Perceptron 12

MMU Memory Management Unit 57

NVML Nvidia Management Library 41

OVO One Versus One 11

OVR One Versus Rest, aka. One Versus All 3

PR Precision-Recall 33

RAM Random Access Memory 19

RN Residual Network or ResNet 16

ROC Receiver Operating Characteristic 33

SM Streaming Multiprocessor 40

SNARC Stochastic Neural Analog Reinforcement Calculator 11

SVM Support Vector Machine 4

TDP Thermal Design Power 26

TNR True Negative Rate 31

TN True Negative 30

TPF True Positive Fraction ix

TPR True Positive Rate 30

TP True Positive 30

Part I

Introduction and Background

Chapter 1

Introduction

In recent years, the interest and research into machine learning have skyrocketed [1]. The potential applications of many types of machine learning are vast, with researchers envisioning new applications and solutions constantly. One of the most promising applications of many types of machine learning is in the area of image classification and processing. The hope is that, as research progresses, computers can assist and perhaps even replace humans in many types of image classification.

This would be beneficial in many circumstances as computers might have the potential to solve image classification problems more quickly than humans. Also, a computer might be able to provide a less costly alternative to having people do classification, so research in this area shows promise in cost-savings. However, developing newer and faster methods is a time-consuming and intensive process. Therefore, it might be beneficial to see if old techniques could be re-used with modern technologies.

In our research, we hope to re-examine and re-purpose an older technique that was used to get around the linearity of early binary classifiers and adapt them to multiclass applications, the One Versus Rest, aka. One Versus All (OVR) style. Here we have one classifier per class and combine the output of all of our classifiers to create a multiclass classifier. The OVR method was previously only used with simpler classifiers, but we intend to examine if this technique can also be applied to more modern complex neural networks and improve their results.

1.1 Motivation

It is not difficult to imagine several cases where computer-assisted image classification could be useful. For example, in a waste management sorting facility, where machines could recognize and extract special waste before recycling or other processes. Such a system might be much faster than manual sorting, and allow sorting to continue 24 hours a day. Another example would be a factory where computers vision could be used to detect manufacturing flaws as products progress down the line, saving money on defective product returns. However, one of the most potentially beneficial applications of computer vision is in the field of Computer-Aided Diagnosis (CAD).

Here we can use machine learning techniques to assist medical practitioners in disease detection and diagnosis. This has potentially massive benefits in patient outcomes, as many diseases have prognoses vastly improved by early detection and diagnosis. Much research has already been done in the field of medical image classification.

In general in image classification, one of the previously preferred methods of performing classification with multiple classes was to use several binary classifiers in an OVR configuration [2]. This means that we will have one binary classifier for each class, instead of a single classifier for all classes. In recent years this technique has fallen out of favor, and we were not able to find much research in which it was utilized. Also, all of the research we found using this technique was done using simpler classifiers such as Support Vector Machine (SVM)'s.

We were interested in investigating if we could apply the OVR technique to more complex classifiers by setting them up in a binary configuration and combining several classifiers into one large classifier. The theory being that the binary classifiers could become better at learning the specific features of the class in question, and the performance of the combined classifier could improve overall. We also believe that these single-class binary classifiers could be more robust against poor hyperparameter selection and un-optimized network structures. Also, this setup could increase the learning capacity of the network.

Also, we have noticed that the focus of research in machine learning is often on creating new and novel uses for machine learning or optimizing the accuracy of existing solutions. The amount of resources used and the drawbacks of increasing complexity with regards to classification speed are often not presented in much detail. This can be unfortunate because in most circumstances one would have several crucial requirements for a good classifier:

- The classifier must have **high classification accuracy**. A classifier is not very useful if it is highly inaccurate.

- It would be preferable if the classifier could function without using an unreasonable amount of **system resources**, both in the form of pure computational power and in the form of electrical power.

- In many cases, the classifier should be able to function in **real-time**. This means that it should be able to read some form of video input and perform classification on the frames as they arrive, and without a significant delay.

We aim to improve the classification accuracy of our existing solutions for the Kvasir v2 dataset. Also, our research will focus on the relationship between machine learning architecture, accuracy, and resource use. More specifically, we will focus on analyzing several modern deep learning architectures set up in both OVR and conventional multiclass configurations to reveal how this will affect the classification accuracy, classification speed, and hardware resource use.

1.2 Research Questions

Our principal hypothesis is that we can use modern neural network architectures in a binary OVR structure or style, and achieve better classification results. We hope to determine whether or not an OVR network style can, in fact, achieve better or at least similar classification results to a conventional multiclass network. Also, we aim to determine if the resource use increases and if so by how much. To put it plainly, we aim to examine the following:

- Can OVR produce desirable results with complex binary neural networks?
- How does using an OVR style affect how we fulfill the requirements of our classifier? Will our experimental setup still perform to the desired level when it comes to classification accuracy, resource use, and classification speed?

Resources will be monitored using several key metrics, for example, GPU usage, CPU usage, and system memory usage. In addition to the classification results, these metrics will inform us of the relationship between splitting the classification problem into several binary networks and the achieved performance. Our overall structures for comparison will be a single multiclass network, in contrast to an ensemble of binary networks which use the same architecture as the multiclass network. Our primary goal will be to examine the difference in classification accuracy achieved, and the tradeoff this has with resource use. We will also examine the different architectures to determine their resource efficiency on our dataset in comparison to their accuracy.

At the conclusion to our research, we should be able to determine whether or not OVR is a viable strategy for use with modern neural network architectures set up as binary classifiers.

1.3 Scope and Limitations

In our research, we will implement at testing and profiling suite that will allow us to train, test and log critical metrics about several modern neural network architectures, or applications. These applications will be set up in both a conventional multiclass configuration with a single network for all classes and a OVR binary configuration with one network per class. We will train and test the networks on the Kvasir v2 dataset, which is a medical dataset of anatomical landmarks and pathological findings of endoscopic procedures. We will log and calculate several vital metrics about these networks as they progress through all phases of our transfer learning based approach. These metrics will log both resource use on our system during all phases of use, and also important classification performance metrics. The metrics will hopefully allow us to answer our research questions about the performance of the OVR multi-network style.

There are, of course, some limitations to our methods. First of all, our methods are limited to a transfer learning approach, which means that we will not be training our applications from randomly initialized weights, but instead from weights pre-trained on the ImageNet dataset. There are other pre-trained weights available for transfer learning, but we will not examine those in our research. As a consequence of using transfer learning, the pre-trained weights and how we go about the transfer learning approach will affect the results. Secondly, we are only using a particular selection of modern neural network architectures. These include the most popular architectures from Google Brain, including select Inception variants. Additionally, we will examine the results for NASNet, MobileNet, DenseNet and the older VGG architectures. There are of course many architectures to choose from, and we cannot include them all. However, we feel that these architectures are a good representation of the current technological state of machine learning for image classification.

Furthermore, our experimentation is limited to the Kvasir v2 dataset, mostly in the interest of time. This will no doubt affect the results as some of our chosen architectures may be particularly

well, or poorly, suited for this dataset. The Kvasir v2 dataset is also limited in size, and while the transfer learning process can hopefully mitigate some of the downsides of this, it will not solve every issue. The results may change significantly with a larger dataset. The dataset also is limited in the type of images it contains, as it is a specific medical scenario and thus the images will be similar in many regards. The outcome may be different for a more diverse dataset.

We have only experimented using one system configuration, which as a CPU without hyperthreading and a single GPU. A more powerful GPU with hyperthreading and multiple GPUs might significantly affect the results. These setups would be an exciting experiment for future research in this area. On a related note, we are only using one main machine learning library. Keras, based on the TensorFlow API, is our choice and provides a solid foundation for us to build our framework. Consequently, other libraries which implement the base neural network building blocks differently might produce different results, at least when it comes to performance.

Finally, during our testing, we are reading test frames from the system memory, which we suspect is faster than basically any other method of feeding frames through our network. In a real-world application, the frames may be produced live by a camera or read from system storage into memory. Our test situation should provide a reasonable approximation of the latter case, but not necessarily the former. It would be of interest in future research to examine how live performance differs from our approach.

1.4 Main Contributions

In the following chapters, we will provide a comprehensive breakdown of the results of our experiments and the difference between an OVR multi-network and a conventional multiclass single-network approach when applied to our chosen dataset. These differences include both the difference in classification performance and the difference in resource use.

We have shown that, for many of our chosen applications, the OVR style does provide satisfactory classification performance at the expense of higher resource use and slower classification speed. We have shown what effect this change in style has on the performance of several different architectures. In summary, using the OVR style resulted in a 7% increase in average F_1 score, 1% increase in average accuracy, 6% increase in MCC, 1% increase in precision and finally a 4% increase in average recall. Specificity remained relatively unchanged. Also, the median values for all of these metrics increased significantly in the OVR style, with the median F_1, MCC and recall scores increasing by over 15%. The most improved network in the OVR configuration saw an increase in F_1 of 45% and an MCC increase of 40%. During testing, on average our OVR multi-network style was 7.6 times slower to classify than a single network multiclass implementation.

We created a framework, TFmetrics, which allows automated training and testing of neural networks in many different configurations, and on a user's chosen dataset. This framework may be useful in future research, and we plan to develop it further to support a broader spectrum of machine learning approaches. In total, during this research, we completed training, fine-tuning and testing sessions for 186 networks. 50 of these were multiclass, and 136 were individual binary networks. On our final test selection of 99 networks (11 multiclass and 88 binary), the metrics logged and calculated resulted in 16,029,806 total data points.

Also, we have created and tested configurations which improve the classification speed and accuracy significantly over previous research on this dataset. Our best networks achieved an F_1 score of 0.87, an accuracy of 0.97, MCC of 0.86, a precision of 0.88, recall of 0.87 and a specificity of 0.98 on the Kvasir dataset.

1.5 Thesis Structure

The structure of the remainder of this thesis will be as follows:

- **Chapter 2: Background**
  This chapter will give an introduction to the necessary theoretical background of our work. It will also attempt to put our work in the context of existing work and historical development in the field. We explain several of the main structures and methods used in this text.
- **Chapter 3: Tools and Processes**
  We detail and explain the different tools and processes used in this work, and summarize why we made these particular choices.
- **Chapter 4: Experiments**
  This section features a detailed explanation of the conducted experiments. Here we detail all metrics used and why they might or might not be useful. We also give a comprehensive breakdown of our experiments and the raw data the produced.
- **Chapter 5: Conclusion**
  We conclude with a summary of the study and what we have gleaned from the results. Also, we will suggest possible further expansions and continuations on these results, and what this could mean for future research.

Chapter 2

Background

To build comprehensive background knowledge of the field and the relevant structures, we analyzed several key texts about different neural network structures and their performance. Also, we looked at texts which specifically explore the use of neural nets for colonoscopy analysis. We chose Stephen Marsland's *Machine Learning: An Algorithmic Perspective* [3] (commonly referred to as Marsland) as an introductory background text on Machine learning. To gain an understanding of the chosen applications[1], as detailed in section 3.5.1, we analyzed their respective papers. This should allow us to establish similar expectations for each application.

We then analyzed several OVR structures relating to applying binary classifiers to multiclass problems. These texts gave us an insight into proper recombination strategies and final classification, in addition to inspiring us with proper metrics to determine what the results of our research were.

Finally, we look at work which discusses the Kvasir dataset. This work should give us an insight into the specific benefits and drawbacks, including potential limitations, of this dataset. It should also provide us with good background knowledge of what work has previously been done with the dataset. This dataset also provides us with a good insight into a specific situation where image classification is required and can be useful for medical practitioners.

In the following sections, we will outline the relevant parts of the texts mentioned above and how they relate to our work.

[1] Note: The use of the word applications here is to be consistent with Keras. In other research or texts, these may be referred to as models.

2.1 Machine Learning

Machine Learning is a popular subject in modern computer science, and a large quantity of research is focused on this field. One of the most promising and exciting uses for machine learning is in the area of image classification. Humans are naturally very good at image classification, but our classification speed is slow. This becomes an issue when there exists a significant amount of data to classify and little time to classify it.

Enter the modern computer, which can process vast quantities of data at high speeds, and remarkably efficiently, given an appropriate algorithm. Now, however, is when we encounter our first caveat. An appropriate algorithmic solution could be difficult or even impossible to design manually. That problem is especially true for image classification, which places several critical requirements on the algorithm chosen, including:

- The algorithm must often be able to differentiate between classes which are very similar. This limitation makes designing a manual image-classification algorithm based on traditional feature extraction and Bayesian probability calculation extremely difficult because a Bayesian classifier assumes feature independence [4], which we cannot guarantee. The changes in features and textures may be so minute between classes that it becomes impossible to differentiate them.

- Image classification must often be done with some expectation of speed and efficiency. It is no good to us if our classification algorithm never produces a result, or produces a result in a timeframe which is unrealistic for day to day use. Extracting features for each new frame might be so complex that it becomes very inefficient.

- Our solution should preferably be able to take new data into account and use this data to improve future results.

These challenges led to the desire in the scientific computing community to more efficiently harness the capacity of computers in the exploration of solutions in a stochastic manner. One of the first implementations of this theoretical capability was the Markov chain, which harnessed the power of stochastic processes using a sort of state machine configuration where the next state was independent on the previous states or configurations of the process. Each state has a certain probability of progressing to a specific next state or staying in the same state [3]. This was a useful step on the way to our current technologies.

2.1.1 Early History of Neural Networks and Deep Learning

People have often looked to nature for solving our technological problems. Even today there are things we can learn from the natural world, for example in aircraft design, where the shape of aircraft and wings might still be improved by taking inspiration from birds [5]. In 1943 Warren McCulloch and Walter Pitts published a critical work which postulated a mathematical model for human learning [6]. This, along with D.O. Hebb's work from 1949 [7] laid the groundwork for perhaps the most critical element of modern machine learning, the artificial neuron, seen in figure 2.1.

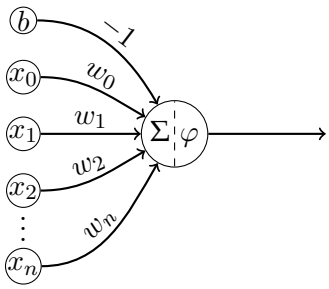


Figure 2.1: Artificial neuron and its components.[2]We see the neuron itself illustrated as the combination of a summing function Σ and an activation function φ. The inputs themselves are denoted as $x_0 \dots x_n$, which are multiplied by the weights $w_0 \dots w_n$. There is also a bias input b, typically -1.

The artificial neuron, rooted in biology, formed the basis for experimentation in building networks of such neurons, to emulate the structure of the brain. These advances could, theoretically, allow a machine implementing this structure to learn patterns on a given set of inputs. The first implementation of a neural network on physical hardware was Stochastic Neural Analog Reinforcement Calculator (SNARC), an Artificial Neural Network (ANN) machine consisting of approximately 40 randomly connected neurons [8].

In the years following these initial experiments, designs progressed and became more organized. In 1958 Frank Rosenblatt designed and implemented the single layer perceptron [9]. The network consisted of a single layer of neurons which together could function to linearly separate classes. The capacity of this network was limited, and it took several decades before more significant progress was made on neural networks. The problem was that to increase the learning capacity of a network; it was theorized that we would need to increase the number of weights, which required us to have additional layers. An example of a multiple layer neural network can be seen in figure 2.2. This means that to "teach" the network, we need to adjust the weights of several layers per iteration, and perhaps more critically, we need to know which weights to adjust.

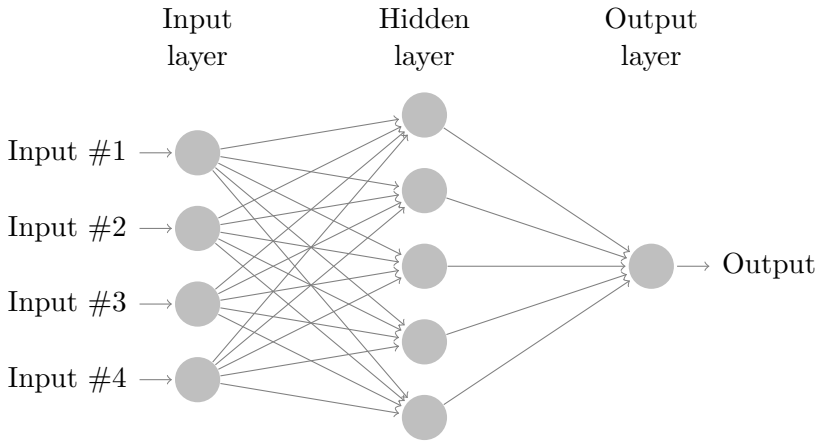


Figure 2.2: A simple neural network with one hidden layer.[3]

Another benefit of adding additional layers was that we could classify into multiple classes using a single network [3]. Previously, the best way to classify into multiple classes was to combine several binary classifiers, primarily using one of two different strategies. One Versus Rest, aka. One Versus All, which used a single classifier per class, and One Versus One (OVO), which used a classifier for

[2]Adapted from https://github.com/MartinThoma/LaTeX-examples/blob/master/tikz/artificial-neuron/
[3]Adapted from: http://www.texample.net/tikz/examples/neural-network/

every two classes. The first of these, OVR, is what we will examine later in the thesis and will be the primary focus of our research.

Classifying multiple classes was made possible in 1986 by David E. Rumelhard *et al.* when they invented the feed-forward Multi-layer Perceptron (MLP) and a way to backpropagate weight updates over multiple layers [3] [10]. The research also introduced a bias to the neurons as seen in figure 2.1. Backpropagation was useful as it functioned as a form of gradient descent to minimize the error of a network [11].

As time progressed and ANN's became more complicated with more layers, the term deep learning was introduced, along with the designation of Deep Neural Network (DNN)'s. These networks were termed as such because they could derive features from the input images, and thus learn higher-order correlations of the data, instead of just learning on the pure pixel values [3]. In the initial structures which implemented this, several autoencoders, derived from MLP's, could be stacked together in such a manner that the output of the hidden layer from one autoencoder was connected to the input of the hidden layer in the next. Many autoencoders can be stacked together, to form a DNN of unsupervised networks. At the top of the network, we can add an MLP, which produces the final classification.

After the resurgence of research into machine learning in the 1980's [11], massive leaps have been made in this field. Newer architectures achieve performance which the original perceptrons could not even compare to [11]. However, that increase in capability usually come at the cost of massively increased computational complexity, with many modern network architectures featuring millions [12] or even more than a hundred million trainable parameters or weights [13] and millions of multiplications and additions [13]. All of the excess computational costs meant more time spent at every step of the way, from training to final classification.

2.1.2 One vs. Rest

As mentioned in the previous section, OVR style classification has mostly fallen out of favor for multiclass use [2]. When OVR is implemented, it is usually by using more traditional binary classifiers that support regularization. An example of this is the SVM. In our search for previous work, we were not able to locate an example where someone had researched the application of an OVR strategy using the more advanced neural network styles which are common today.

We looked at several SVM implementations of OVR and these seemed to perform very well or at least comparable to some multiclass networks [2] [14] [15].

However, as we are not implementing SVM's, the technical implementation of these solutions is of less interest to us. What is interesting is that such a structure might be accurate at all, and it will be up to us to investigate if this property can be transferred to more complex networks, and what impact this will have on the classification accuracy and performance.

2.1.3 The Modern Era

Since the early days of machine learning a vast amount of progress has been made in the field, and the complexity of different strategies has increased immensely. We will look at a few of the key innovations of modern machine learning, and these will be particularly relevant as these architectures will also be used in our research. Most of these architectures are based on a form of Convolutional Neural Network (CNN), an example of a simple CNN can be seen in figure 2.3. Our networks usually consist of a considerable variety of convolutional layers (of different types), pooling layers, batch normalization layers and activation layers. Some will also feature fully-connected layers (FC-layers) and simple linear layers. Before beginning this section, it would,

therefore, be prudent to have a quick summary of the different layer types and what their purposes are.

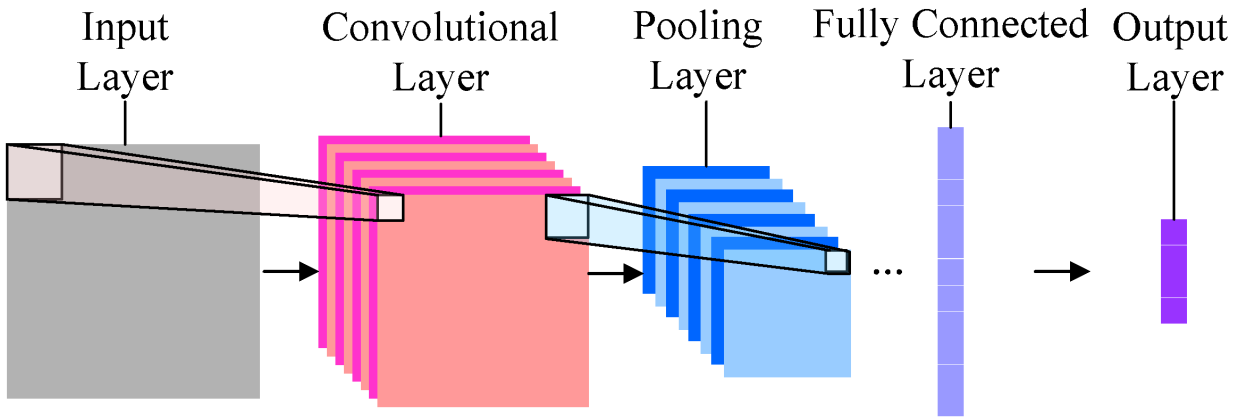


Figure 2.3: Illustration of a simplified CNN, with the different layers labeled. [4]

Linear Layer

Linear layers are the most straightforward layer style. They are also one of the most prominent building blocks in modern neural networks [16], despite it being almost directly based on the original single layer perceptron as discussed in section 2.1. The layer mainly consists of a single set of basic neurons, which have a set of weights W, and linearly output a scaled multiplication of the input [16]. Essentially, it is a matrix-vector operation [16]. We can interpret this to mean that a linear layer is the mathematical operation seen in equation 2.1. Here, y is the output vector, W is the weights of the layer, and x is the input vector.

$$y = xW \tag{2.1}$$

The linear layer performs a simple operation, but a computationally costly one, as it requires $O(N^2)$ operations to calculate the output [16].

Activation Layer

Activation layers are a bit more complicated to define, as there are dozens of activation choices available. We will not delve into the different activation functions here. However, the basic premise is to introduce non-linearity into the network, by having layers of neurons which are either activated or not activated, preferably by some non-linear function, depending on the input to the layer. Two common examples of this are ReLU (Rectified Linear Units) and Sigmoid, which are both efficient activation functions [17]. The most common activation function in the tested architectures is ReLU, which is an activation function defined as seen in equation 2.2.

$$f(x) = x^+ = max(0, x) \tag{2.2}$$

The output of the layer is the same as the input where the input is greater than 0. Otherwise, it is zero. This is an efficient operation as it is merely a comparison unless some form of approximation is used [17].

[4] From http://www.mdpi.com/2078-2489/7/4/61, CC BY 4.0

Convolutional Layer

Convolutional layers are the core building block of many modern neural networks, especially those used in image classification. Here, the primary purpose is to extract high-level features from the image using convolution. This operation iterates over the original input using a moving window or filter of a certain size. Typically the window could be a filter of size 5x5 or 3x3, which we can refer to as size $K \times K$. To understand this operation, we can view an image as a matrix of binary values of size $N \times N$. We then iterate our filter over our input image, which results in a convolved output of size $(N - (K - 1) \times N - (K - 1))$.

Depthwise separable conventional blocks are standard in some modern architectures. These consist of one depthwise convolutional layer and one pointwise convolutional layer [12]. First, the depthwise convolution applies a single filter per channel of the input image, as discussed above. Then the pointwise convolutional layer applies a simple 1x1 convolution to create a linear combination of the outputs from the depthwise layer [12].

Pooling Layer

A pooling layer is often inserted between successive convolutional layers to reduce the spatial size of the representation as we progress up the network. The purpose of this operation is to reduce the number of parameters needed and thus the complexity of the network. There are different types of pooling, which employ different strategies to reduce the spatial dimensions. We will not detail them here, but the most important thing to note is the purpose of these layers, which is to reduce computational complexity.

Batch Normalization Layer

These layers, simply put, increase the stability of the network. They do so by normalizing the output of the previous layer and then subtracting the batch mean, followed by a division by the batch standard deviation. Not all of our selected applications use batch normalization layers, but they are interesting to note regardless as the operation they perform can be computationally intensive.

Fully-connected Layer

Fully-connect layers are similar to the conventional layers we discussed in the section on neural network history. They have connections to all of the previous layer's outputs and all of the next layers inputs (if applicable). Otherwise, if used at the top of a network, it will have N output connections, where N is the desired number of classes. They are essentially the same as the output layer of the MLP discussed earlier in this regard.

2.1.4 Applications

Since the early days of machine learning, many new and exciting architectures and designs have been created. The designs have varying degrees of complexity, and the primary goal of the designers has been to create networks that are exceedingly capable of learning more complex problems. In our research, we will use a selection of these more modern applications, specifically ones that are available in Keras. The following is a summary of these applications. To more easily visualize the architectures described here, figures for each application are included in the appendix. Note that we use the term applications here, as that is the term used by Keras [18]. Many would refer

to these as models or architectures. However, the term application is used as these models are available from Keras with pre-trained weights.

VGG

Beginning with the oldest architecture tested, we reviewed the accompanying article for the VGG16 and VGG19 architectures, titled *Very Deep Convolutional Networks for Large-Scale Image Recognition* [13]. In this article, Karen Simonyan and Andrew Zisserman aim to improve the classification accuracy of modern Convolutional networks (ConvNets) and investigate the effects of network depth on classification accuracy.

The VGG architecture was established by pushing the network depth of a convolutional network to 16-19 layers, leading to a substantial increase in the number of trainable weight layers. The details regarding the network construction and what impact it has on classification accuracy and resource usage are of particular interest to us. The article contains detailed information about the number of parameters and the classification accuracy compared to previous networks architecture.

The article does not go into detail about processing performance. No details are presented regarding frames processed per second or hardware resource usage. The technical details of the Keras implementation are of interest to us in this section. We observe in table 3.4 that the VGG 16 implementation has over 138 million parameters with a depth of only 21. While the number of parameters is not the only aspect that can affect the performance, it is a rather high number for such a simple architecture. As we can see in the table, the two VGG architectures have the highest parameter count of all the tested styles, despite having the lowest number of layers.

Inception v3

Although VGG is a capable architecture, there are some limitations associated with it, and it does have a rather large number of trainable parameters. This lead to researchers desiring to make networks which could more efficiently use system resources, so that the resulting architectures would be more suitable for mobile vision and big data scenarios. The Inception v3 architecture is detailed in *Rethinking the Inception Architecture for Computer Vision* [19]. The application was designed to classify the ImageNet dataset [20].

This innovation involved a transition from conventional feed-forward convolutional networks, to a different structure, which used so-called Inception modules. These modules can be seen as small blocks of convolutional layers connected sequentially, and several of these sequential layer stacks are connected in parallel using concatenation layers [19]. This comprises a single block, and these blocks are connected in sequence, along with some conventional convolutional layers, and pooling layers to form the complete architecture.

The primary claimed benefit here, compared to VGG, is lower computational cost [19]. It will be interesting to see if this is reflected in our results. Inception v3 was also able to achieve a lower error rate than VGG on the ILSVRC 2012 classification benchmark [19].

DenseNet

In *Densely Connected Convolutional Networks* [21], Huang *et al.* propose an alternative solution which uses dense blocks. In these dense blocks, each layer is connected to all subsequent layers; this exploits the potential of a network through feature reuse [21]. The dense blocks are then connected sequentially using convolutional and pooling layers. It is similar to the approach used by residual networks. However, this approach should be more efficient [21].

The DenseNet article does not offer a good comparison to what sort of classification accuracy we can expect compared to Inception V3 or VGG applications, but it does contain claims that DenseNets are less prone to over-fitting than ResNets [21].

Xception

After the original Inception block structures were described and their performance analyzed, several researchers endeavored to improve their results. On such implementation was *Xception: Deep Learning with Depthwise Separable Convolutions* [20]. It uses a very similar structure to Inception v3, but it also incorporates depthwise separable convolutional operations. Although exactly what effect this has on the network complexity is outside the scope of our research, the network style is claimed to offer higher accuracy and lower loss on several key datasets. Interestingly, the researchers also speculated that Inception v3 might tend to over-fit on the ImageNet dataset, whereas Xception would not [20].

Inception ResNet v2

In another attempt to improve the performance of Inception blocks by including feature reuse, Szegedy *et al.* have designed a residual version of the Inception block in *Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning* [22]. The blocks are used to build a complete network, which is designed to be similar in computational cost to the Inception v4 network [22]. However, Inception v4 will not be tested in our research.

In this Residual Network or ResNet (RN) version of Inception, the researchers reported lower error rates on the ILSVRC 2012 dataset than Inception v3 and Inception v4. [22].

Mobilenet

The previous applications we have discussed have increased the complexity of the networks gradually increasing, having more and more mathematical operations from the bottom of the network to the top. MobileNet, detailed in *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications* [12], was designed to be efficient on low power hardware, presumably for mobile applications, and as such is much less complex. The layers are arranged in a purely sequential manner but, in contrast with for example VGG, it features depthwise separable convolutional layers. This allows approximately 1/10th the amount of Multiply-accumulate operations[5] of Inception v3 and 1/30th the amount VGG 16 has [12]. Optimizing in that manner should allow for much better performance on low power hardware.

NASNet

Our final applications are both based on the NASNet (Neural Architecture Search Net) structures. In *Learning Transferable Architectures for Scalable Image Recognition* [23] Zoph *et al.* propose an innovative way to optimize neural network architecture. In the previous examples, the search for a suitable architecture is performed on the dataset which one wants to classify. For example, Inception v3 was built to be the best possible classifier for ImageNet. However, this can be resource intensive and extremely time-consuming. Thus, it would be practical to be able to optimize our architecture on a smaller, less computationally intensive, dataset and then transfer this architecture to our main classification problem and dataset. The NASNet researchers' experiments are based

[5]By multiply-accumulate operation, we mean the common operation that calculates the product of two numbers and adds the result to an accumulator.

on using the CIFAR-10 dataset to optimize the network structure, and then transfer this structure to the ImageNet dataset [23].

NASNet-A (6 @ 4032), or NASNet Large as it is referred to by Keras and in our research, was able to achieve better results than Inception v3, Inception ResNet v2 and Xception on the ImageNet dataset. NASNet-A (4 @ 1056), or NASNet Mobile in our research, achieved results which were only slightly weaker, but with far fewer multiply-accumulate operations than those mentioned [23].

2.1.5 TensorFlow and Keras

Manually implementing many of described network structures and applications manually is complex and time-consuming. Luckily, several libraries and packages can create layers of neurons and connect them based on our input. One such library is TensorFlow, which was developed by the Google Brain team in 2015 to allow many machine learning algorithms to be represented as stateful dataflow graphs [24]. Data in the graph is represented as so-called tensors, which in the case of image classification are data structures of the format NHWC or NCHW. Here N is the number of images in a batch, H is the height of an image, W is the width of an image and C is the number of channels in the image, for example, three channels in the case of an RGB image [24]. The difference between NHWC and NCHW is merely the ordering of the dimensions.

Tensors are then fed through the network, where they are dimensionally altered depending on what type of operation the current layer does, but the format remains the same, i.e., although H, W, and C might change, the ordering will remain either NHWC or NCHW respectively. This tensor implementation makes it simpler to feed data through the network and keep track of the alterations in spatial size with using, for example, convolutional layers.

When it comes to calculations, TensorFlow can perform most of the calculation on the GPU, and as GPU's today usually have a tremendous number of cores, it allows for efficient performance of highly parallelizable tasks [24].

TensorFlow also has some pre-implemented layers that can be called and created with the desired dimensions, meaning that it is easy to create a stack of layers, and in turn create complex neural networks without having to program all of the layer functions and data structures ourselves. However, TensorFlow has some operations that are difficult to understand intuitively. Thankfully, Keras has been developed which allows us to import completed applications easily and quickly in a way that is easier to understand than TensorFlow [25] [26]. This is especially useful in our case as the primary focus of our research is to examine the difference between OVR and multiclass styles, not on the low-level implementation of each application.

2.1.6 Transfer Learning

Training a complex neural network structure such as those we will be examining can be extremely time-consuming at first, as we would usually begin with all of our weights randomly initialized. As there can be millions of weights to adjust and incredibly complex features in many image classification problems, it comes as no surprise that adjusting all of these weights from scratch takes time and system resources, especially as the number of multiply-accumulate operations increase into the billions [12].

It would, therefore, be practical if we did not have to train the network from scratch but instead could use the weights of an already trained network for a new problem by adjusting them. That way base image features which the network has already learned could be applied to the new problem. This would be especially useful if we do not have much training and testing data for our new problem [27].

In our research, we applied a transfer learning function which used networks previously trained on the ImageNet dataset. The ImageNet dataset or database is a large set of over 14 million images classified into over 20 thousand classes [28]. This means that during the different phases of our research we will be adjusting these previously trained networks to suit our chosen dataset. To complete our research, we must design and implement a piece of software which will create neural networks and take them through several key phases.

Building the network

Due to the particular nature of our transfer learning implementation, our phases of use are different from how things might conventionally be done. It would, therefore, be helpful to describe exactly how these phases will progress in our case. Initially, the networks will be initialized with fully trained weights, with the except final output layers of the network. These layers are also known as the top layers, and will not be included from the Keras implementation. This means that the last included layer from the Keras implementation will vary between the applications. For example, in the case of NASNet Mobile, the last included layer of the Keras implementation will be an activation layer with $7 \times 7 \times 1056$ outputs. We will then attach our top layers which conform to our desired number of output classes. Essentially, in the OVR layout, each class's network will first have one global average 2D pooling layer on the output of the pre-trained network. This operation will reduce the 4D tensors into a 2D output, which no longer has separate spatial data. Then a fully-connected layer will be added on top, which will further reduce the output into our desired number of classes. This will give us the final output prediction values.

Training

During the initial training, all layers except our top two added layers will be frozen, meaning that the imported ImageNet weights will be untrainable during this phase and we will only be training our final two layers. Freezing the layers limits how accurate the network can become, but will also allow us to achieve a basic amount of accuracy while substantially reducing the amount of hardware resources needed to train, as the weight calculation functions will not have to be run for the pre-trained network. During training itself, first training data will be propagated through the network. Then, during the backpropagation phase, the outputs of the network and the true classes of the training data will be run through a loss function. This loss or error is then used to update the weights, starting from the output, down the network until the input is reached or until the layers are no longer trainable. The process is done using an optimization algorithm, which seeks to optimize the loss (i.e., reduce it), and updates the weights of each neuron based on how much it contributed to the outcome. Please note that this is an elementary explanation, but the fine technical details are not the main focus of our research.

Fine-tuning

In the second phase of training, what we describe as fine-tuning, a large number of the weights of the imported network are unfrozen, typically a few blocks or a certain number of layers at the top of the network. This allows several more of the weights to be adjusted by the optimizer during training. Furthermore, it means that the network can learn more specific features of the training data but at the cost of increased computational intensity and resource use. We might also adjust the learning rate of the optimizer during this phase to reflect the change in the number of trainable parameters.

Testing

The final phase of use for our networks, testing, involved test images being quickly fed through the network and predictions stored for each image. As there is no loss or optimization function involved here, this phase is usually able to process images much more quickly than the training phases.

2.2 Resource Use

During all phases of network use, TensorFlow makes extensive use of the CPU, GPU and main system memory (Random Access Memory (RAM)) [24]. Although most of the actual computation happens on the GPU [24], how much of the CPU and memory resources are used is essential to know in the present study. As we are comparing two styles of classification which might have wildly different complexities, we should expect our resource use to increase, at least during the final classification when several networks are being used simultaneously.

The monitoring and evaluation of the resource use is essential because we are seldom blessed with an infinite amount of computational resources during actual use of a classification system. Besides, high power hardware can be prohibitively expensive, difficult to set up and use, and not always be available. Resource should also have a significant effect on the amount of time spent in all phases throughout use. Amid the training phases, high resource use might mean that we are unable to experiment as much as we would like with application and hyperparameter selection. Throughout testing, this might mean that the network is unable to classify images in a reasonable amount of time.

2.2.1 CPU

Several important operations during training and testing are performed mainly on the CPU, so the CPU usage is these phases is an important part of the performance of the network. For example, all the base Python code, along with the python interpreter itself run on the CPU. Also, TensorFlow uses the CPU to transfer data into and out of the GPU, and thus into and out of the main system memory. This is important as during training weights and gradients will be transferred back an forth by the CPU. One interesting thing to note here is that Python has a Global Interpreter Lock, meaning that the Python interpreter will run only one piece of code at a time. This will be important for us to consider when designing and building our testbench. Most of the more difficult calculations, however, are performed in the GPU.

2.2.2 GPU

During training and testing, the GPU is heavily used by TensorFlow [24]. This means that during all phases of our testing, to gain an understanding of the performance impact the OVR structure has, we should examine the GPU usage at the same time. It may be that some applications lend themselves better to the OVR structure than others, especially in the case of testing. During testing, networks will have to be run in parallel, and perhaps data from the networks be transferred to and from the GPU.

As GPUs are complicated devices, there are several areas of interest when it comes to their use. To begin with, we would want to monitor the volatile GPU core usage; this will likely be an important metric as it gives us an idea of how computationally complex our applications and styles are. We might also want to monitor the raw power draw of the GPU, which should also give us a similar idea of the computational complexity of our problem and how much of the GPU's resources are being

used by changing hardware states within the GPU. Although we will not perform an incredibly complex analysis of how TensorFlow uses the GPU, it will still be interesting to see the differences between the OVR and multiclass styles in this regard.

Contexts

We would, of course, prefer to be able to run the networks in parallel on the GPU in the testing phase, as this is potentially the most critical time for performance. We want the networks to produce results promptly. Unfortunately, it seems that the contexts used by CUDA prevent more than one instance from sequestering GPU resources simultaneously [29]. This limitation might be problematic for parallelization. Thus it would be interesting for us to see if it indeed is the case during testing.

2.3 Medical Uses and Kvasir

Medical science is one of the more promising uses for machine learning; ranging from using machine learning to identify statistical patterns in patient outcomes, to our use case, which is using machine learning methods for medical image classification as in the present study. Specifically, we are interested in the form of CAD where the computer can assist the medical practitioner in detecting, classifying, and diagnosing medical issues which may be present in images.

There are many types of CAD, and for several forms, the stakes can be high, as early detection can be a crucial factor for patient outcomes in many diseases. Such a system can be useful and alert the doctor if it detects an anomaly or recognizes a medical issue [30]. In particular, we are interested in forms of CAD which are primarily an exercise in image classification and feature detection. Due to the extensive focus on image classification in modern machine learning, we can perhaps create systems that can assist the practitioner.

One of the potential applications for a medical image classifier is assisting doctors with identifying anomalies in the digestive system. Anomalies can be critical to identify, as many serious diseases and issues can manifest themselves in the digestive system, and can have consequences ranging from a severe decrease in life quality to death. Colorectal cancer (CRC), esophageal and stomach cancers, for example, account for over 2.8 million new cases each year. The cancers can have quite poor outcomes, as they result in over 1.8 million deaths per year [31]. For cancers of the digestive system, early detection and treatment can be vital when it comes to having a good patient outcome.

With this in mind, we wanted to select a dataset which presented a useful real-world scenario for our approach, and one which could perhaps be used to improve the outcomes in the previously discussed scenarios. In *KVASIR: A Multi-Class Image Dataset for Computer Aided Gastrointestinal Disease Detection* [32] , Pogorelov *et al.* present a dataset called Kvasir which contains thousands of classified frames from colonoscopy videos. This dataset is an improvement over the previously available data in the field [32].

Also, the dataset has related research which provides us with exciting findings we can compare our final results with, to evaluate how our classifiers might function in a clinical setting [33] [34] [35].

With these ideas in mind, selected the Kvasir dataset for our experiments.

2.4 Summary

In this chapter, we have provided an introduction to many of the basic concepts and tools used in our research. We looked at the historical path machine learning has taken, and why re-examining OVR

style networks in a modern context might be an interesting research opportunity. In particular, we can note that no previous research that we could find has attempted to use the OVR technique with modern neural network architectures. We have provided a breakdown of the various modern architectures we will examine in our comparisons, and why they represent a good selection of the current cutting edge of machine learning.

We discussed the different hardware resources involved and why they are essential for us to monitor in our experiments. These will provide us with an excellent basis to determine how efficient or inefficient our solutions are. These metrics are especially crucial for the performance aspect of our research questions. Finally, we outlined the main classification problem we will be examining and why this classification problem is of particular importance.

However, to answer our central research questions, we will require more information and a comprehensive selection of experiments. The experiments will provide us with the data we need to answer our questions. In the next chapter, we will detail a testing framework which will give us the basis to answer our research questions.

Part II

Implementation and Discussion

Chapter 3

Tools and Implementation

In this chapter, we will outline the details of our implementation and research methods. Of particular note here is the design and creation of TFmetrics, our research framework. We have decided to examine our hypothesis using 11 neural network applications, which means that to train and test all of these, we will need to train 99 networks in total, test them and log metrics about them. Given that it can take up to 29 hours to train a single network as we have configured them, it would not have been possible without our framework as it would have been too time-consuming otherwise. Also, we will detail all the metrics we have logged during the experiments, which external frameworks we have used and critical specifications about our chose applications. The following sections will outline exactly how we planned to answer our research questions.

3.1 Test Systems

To run the necessary experiments, we required a few main resources. First, we required a physical system to run the experiments on, and secondly, we required a software framework which would create the neural networks and run them. In the following section, we will provide details about the hardware and software used.

3.1.1 System Specifications

We used a system configured with a single GPU as detailed in table 3.1. At the time of writing, this system was the most appropriate system available for our use. The system should be capable of adequate performance for our use, in addition to being representative of modern high-performance hardware. Importantly our GPU had a cuDNN compute compatibility of 6.1 [36], which was the highest available at the time of writing for a consumer GPU. Note that the system is only equipped with a single GPU, while TensorFlow is built to optimize use on several GPUs in parallel [24].

Component	Model	Specifications
CPU	Intel i5-4590[6]	Logical Cores: 4 Clock speed: 3.3GHz (3.7 GHz max) Thermal Design Power (TDP): 84W
GPU	NVIDIA GTX 1080 Ti[7]	CUDA®Cores: 3584 Boost clock speed: 1.582GHz TDP: 250W Memory: 11 GB GDDR5X Memory speed: 11 Gbps
System memory	N/A	16GB DDR3

Table 3.1: A list of the major system components

[6] https://ark.intel.com/products/80815/Intel-Core-i5-4590-Processor-6M-Cache-up-to-3_70-GHz
[7] https://www.nvidia.com/en-us/geforce/products/10series/geforce-gtx-1080-ti/

3.1.2 Frameworks and Packages

Many frameworks and packages enabled us to complete our research. An overview of the most relevant ones can be seen in table 3.2. We make extra note of the version numbers as the resource usage characteristics might change during development, so all metrics are provided as the frameworks stood at the time of writing.

Package name	Version	Source
Ubuntu	16.04.2 LTS Server	Canonical Ltd.[8]
TFmetrics	0.0.1	Github[9]
Keras	2.1.5	Github (Fork)[10]
CUDA Toolkit	9.0	Nvidia Corp.[11]
Nvidia Driver	390.30	Nvidia Corp.[12]
Python	3.5.2	apt-get
nvidia-ml-py3	7.352.0	pip
tensorflow-gpu	1.5	pip
psutil	5.4.3	pip
scikit-learn	0.19.1	pip
scikit-plot	0.3.4	pip
numpy	1.14.2	pip
scipy	1.0.1	pip
opencv-python	3.4.0.12	pip
matplotlib	2.1.2	pip
graphviz	0.8.2	apt-get
pydot	1.2.4	pip
seaborn	0.8.1	pip

Table 3.2: A list of the frameworks and packages used in this thesis, along with their respective version numbers.

TFmetrics

TFmetrics is a framework we developed to encapsulate Keras instances and compile metrics about their performance. Keras itself is explained in more detail in section 3.5. Our framework was created with the intention of being able to efficiently run many experiments sequentially, including all phases of network use. Also, the framework can log the metrics which are specified more closely in this section.

The motivation for creating our framework was primarily to make it easier to run the number of

[8] https://www.ubuntu.com/download/desktop
[9] https://github.com/Berstad/TFmetrics
[10] https://github.com/Berstad/keras
[11] https://developer.nvidia.com/cuda-downloads
[12] https://www.nvidia.com/Download/index.aspx

experiments required, and to create a piece of software that could set up and use several networks in an OVR configuration, as no such software currently exists to our knowledge. As training and testing these networks can be extremely time-consuming, we wanted to develop a system which could read a list of desired jobs as inputs and process these automatically in succession without additional input from the user.

This system was important because we would be training, fine-tuning and testing a total of 99 neural networks, in addition to running 11 tests of the OVR networks running simultaneously in the networks parallel. We also wanted to be able to run tests using several different combinations of hyperparameters without having to start and configure the networks each time manually. Therefore we desired a software suite which could perform the following actions without intervention from the user:

- Read and interpret jobs from some form of an input file.
- Set up the specified applications in Keras using the specified hyperparameters.
- Perform the action specified in the job file, which could, for example, be to run all phases of the network or to load previous weights and only perform testing.
- While performing these actions, log hardware metrics as specified in the job file. The metrics are detailed in section 3.6.
- Log any applicable network metrics, for example, accuracy during training.
- During testing, calculate accuracy metrics as defined in section 3.4.1.
- Store all metrics collected during all phases in appropriate formats, and automatically generate metric plots and figures if it is specified in the job file.

With TFmetrics, we have achieved most of the design goals. Based solely on JavaScript Object Notation (JSON) standard parameter files, the framework creates, trains, fine-tunes, tests and generates metrics for the specified network. The framework supports all available architectures in Keras, in addition to all available optimizers. The metrics are logged in separate threads using the threading module, with each metric type having its own thread. The project ended up having 2812 lines of code and required a fork of a few external repositories to provide the desired functionality.

In the future, we plan to develop the framework further so that it can run as a server and save all metrics and data to a database. This change would allow a client to connect to the server, select different network parameters to test and then do so. Also, the user should be able to monitor whichever metrics they like in approximately real-time. We would also like to create a graphical client for the project, similar to Tensorboard, but more specific to our use-case. Eventually, we hope to develop the framework to use other learning methods than transfer learning and to support custom network structures.

Plotting and Visualization

Mostly, all plots are generated using matplotlib with a custom library to translate stored metric history into coherent plots. The only exception is the model visualization, which is generated by Keras' model visualizer, which has been forked and modified by us to include layer numbers for all layers. It made it easier to visualize the networks and select layers to freeze during fine-tuning.

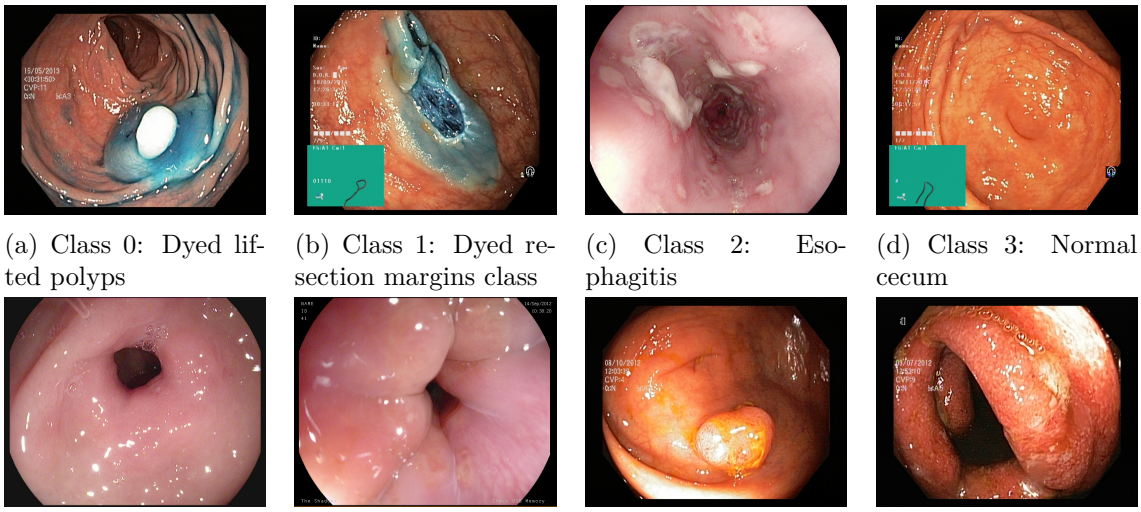
Some plots are generated using Seaborn, including most statistical plots such as bar plots, violin plots, and box plots. The plots enable us to visualize the effects of the OVR style networks have had on performance.

3.2 Dataset

3.2.1 Selecting a Dataset

As detailed in section 2.3 we have selected the Kvasir v2 dataset to perform our tests. We chose this dataset as the supervisors and group have extensive experience using this dataset, and have previously assembled a selection of classification results [32] which we can compare with our results. We selected this dataset because of its medical potential, and because relevant research on this dataset includes classification results that we can compare with our own.

The Kvasir v2 dataset consists of 8000 images, which belong to 8 classes showing anatomical landmarks and pathological findings of endoscopic procedures in the GI tract [32]. The classes are balanced, i.e., there are 1000 images per class. The encoding settings of the images vary across the dataset, which reflects the *a priori* unknown endoscopic equipment settings [32]. Sample images of each of the classes can be seen in figure 3.1.



(a) Class 0: Dyed lifted polyps

(b) Class 1: Dyed resection margins class

(c) Class 2: Esophagitis

(d) Class 3: Normal cecum

(e) Class 4: Normal pylorus

(f) Class 5: Normal Z-line

(g) Class 6: Polyps

(h) Class 7: Ulcerative colitis

Figure 3.1: Example images from the Kvasir v2 dataset.

3.2.2 Dividing the Data

Marsland outlines the need for separate testing and validation sets, and for example, using some form of K-fold Cross-Validation to avoid overfitting [3]. We have decided, based on this information, to use separate validation and test sets. We have 8000 total images, as discussed in the previous section. We chose to divide the images into three sets using a seeded random process, which creates symlinks to existing data in a pseudorandom manner. The first set, the training set, is used for training directly. The second set, the validation set, is used for the validation phase of training, where the validation accuracy is calculated and is primarily used for our early-stopping function. Finally, we have the test set, which is not used for training at all and is not seen by the network until the final testing phase. This process uses *numpy*'s random selection. We selected a 50-25-25 split, as suggested by Marsland [3]. Thus, we have 4000 training images, 2000 validation images, and 2000 test images. The images are selected proportionally per class, i.e., each set will have the same number of images for each class.

The same images are used to train, validate, and test both the multiclass and binary networks. Although Kvasir is a rather small dataset, we felt it would still be prudent to have a test set.

3.3 Overall Structure

In section 1.2, we outlined our fundamental research question: Will the OVR multi-network style offer comparable results to the single-network multiclass style? We must now define basic network structures that can be used to evaluate and potentially answer this question. Because we have chosen to use pre-defined network topologies, which we detail in section 3.5.1, the multiclass network structure is simplified. We can use the automatically generated structures from Keras for both the multiclass and binary structures. In principle, we are using the same architectures for both our binary and multiclass classification, but for the binary case, we are merely using two classes, positive and negative. Our segmentation of the data means that we are training 8 OVR classifiers on all of the training data.

3.4 Classification Performance Metrics

We used several metrics, as outlined by Marsland [3], as inspiration to define our final metric list, seen in section 3.4.1. Also, Marsland gives examples of how the divide data and evaluate a classifier on this data in a way that can give us an idea of how generalized the performance of the network is [3].

A systematic analysis of performance measures for classification tasks [37] was used for inspiration. Ultimately several metrics from both sources were used, although with a slight modification to the metrics from the latter paper, as we have chosen to use macro-averages for all metrics.

3.4.1 Post-selection Accuracy

In the pursuit of properly assessing the final classification performance of both the binary and multiclass networks, we chose a selection of metrics which together create a decent summary. Some of these metrics are better for assessing the performance of multiclass networks, and some are better for binary problems. If we look at the multiclass problem as a series of binary problems both styles can be helpful. The metrics are primarily based on Marsland's book [3].

- **True Positive (TP):**
  TP is the number of correctly identified samples or the number of frames with a specific endoscopic finding which are correctly identified as a frame with that endoscopic finding [32] [3].
- **True Negative (TN):**
  TN is the number of correctly identified negative samples or the number of frames without a specific endoscopic finding which are correctly identified as a frame without that endoscopic finding [32] [3].
- **False Positive (FP):**
  Here we have the number of positively identified samples for a specific class which were, in fact, negative, commonly called a "false alarm" [32] [3].
- **False Negative (FN):**
  FN is the number of negatively identified samples for a specific class which were, in fact, positive [32] [3]. This metric can be critical in a medical setting, where a false negative can lead to undiscovered issues.
- **Recall (REC):**
  Recall is frequently called sensitivity, the probability of detection and True Positive Rate

(TPR). It is the ratio of samples that are correctly identified as positive to the total number of positive samples [32] [3]:

$$REC = \frac{TP}{TP + FN} \tag{3.1}$$

- **Precision (PREC):**
  Precision is frequently called the positive predictive value. It is the ratio of correctly identified positive samples to the total amount of positively identified samples [32] [3]:

$$PREC = \frac{TP}{TP + FP} \tag{3.2}$$

- **Specificity (SPEC):**
  Specificity is frequently called the True Negative Rate (TNR), and shows the ratio of correctly identified negative samples to the total amount of negative frames [32] [3]:

$$SPEC = \frac{TN}{TN + FP} \tag{3.3}$$

- **Accuracy (ACC):**
  Accuracy is the the percentage/ratio of correctly identified true and false samples [32] [3]:

$$ACC = \frac{TP + FP}{TP + FP + TN + FN} \tag{3.4}$$

- **MCC:**
  MCC takes into account true and false positives and negatives. It is especially useful for us in the binary case because of the unbalanced dataset [32] [3]:

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{3.5}$$

- **F1 score (F1):**
  The F_1 score is the harmonic mean of the precision and recall. It is a useful metric for combining the precision and recall scores [32] [3]:

$$F_1 = \frac{2TP}{2TP + FP + FN} \tag{3.6}$$

- We also calculated classification loss on validation and training sets (loss) as defined by the selected loss function. The figures will show us the progression of the loss and accuracy during training, at the end of each epoch. It gives us a good idea of the efficiency of our hyperparameter selection.
- Also, we generate confusion matrices on test set [3]. The figures show us how often each class was confused for another, but plotting the network's predictions on the test set in a matrix where the rows represent the true class labels, and the columns represent the output predictions for the network. An example of this can be seen in figure 3.2 The confusion matrix, compared with all the other metrics we have chosen, will give us a good idea of which classes the classifiers struggle with the most.
- **Frames Per Second:**
  How many frames of test data the network is able to predict per second. The metric is measured using the Keras predict method.

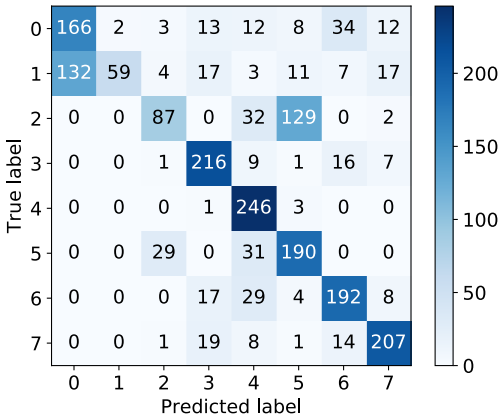


Figure 3.2: Example confusion matrix for Inception-ResNet-v2, multiclass, generated by Scikit-plots.

3.4.2 Averaging

In the following sections, two different methods of averaging will be discussed. The two methods are important in different ways: macro-averaging is useful when the classes are balanced as it treats all classes in the same way, while micro-averaging, on the other hand, is useful when the classes are unbalanced.

- **Micro-average:**
  In order to calculate an average score, curve or metric, we add up the base scores:

  $$\sum_{i=1}^{n} TP_i = TP_{total}$$
  $$\sum_{i=1}^{n} TN_i = TN_{total}$$
  $$\sum_{i=1}^{n} FP_i = FP_{total}$$
  $$\sum_{i=1}^{n} FN_i = FN_{total}$$

  Where n is the number of classes, and the base numbers are as defined in 3.4.1. We then place these sums in the formula instead of the original sums, so the micro-averaged F_1 score would become:

  $$F1_{micro-avg} = \frac{2TP_{total}}{2TP_{total} + FP_{total} + FN_{total}} \tag{3.7}$$

- **Macro-average:**
  Macro-averaging is a more conventional averaging strategy, basically we sum the individual scores and divide the number of classes, r.g. for F_1 this would be.

  $$F1_{macro-avg} = \frac{F_{11} + F_{12} \ldots F_{1n}}{n} \tag{3.8}$$

The metrics mentioned in section 3.4.1 are then macro-averaged for each class, for both the OVR networks and the multiclass network. Macro-averaging was deemed acceptable because our classes are balanced for the test data.

3.4.3 Pre-selection Accuracy

To evaluate the performance of our networks regardless of which strategy we use to select the output class, we can use metrics which directly utilize the prediction probabilities. For example,

if we were to select a threshold strategy for our final classification, we would like to be able to compare the network probabilities directly, without choosing a threshold.

- **Receiver Operating Characteristic (ROC) curves:**
  A receiver operating characteristic curve, an example of which can be seen in figure 3.4, is a two-dimensional plot that plots the TPF against the FPF [38]. The formula for these ratios can be seen in equation 3.3. This plot allows us to get a good idea of what the accuracy would

$$TPF = TPR = REC = \frac{TP}{TP + FN} \tag{3.9}$$

$$FPF = 1 - SPEC \tag{3.10}$$

Figure 3.3: Equations for the TPF and FPF. For an explanation of TP, RN, REC, and SPEC, see section 3.4.1

  be with a specific threshold (or operating point) selection [38]. Each ROC figure contains the following curves:

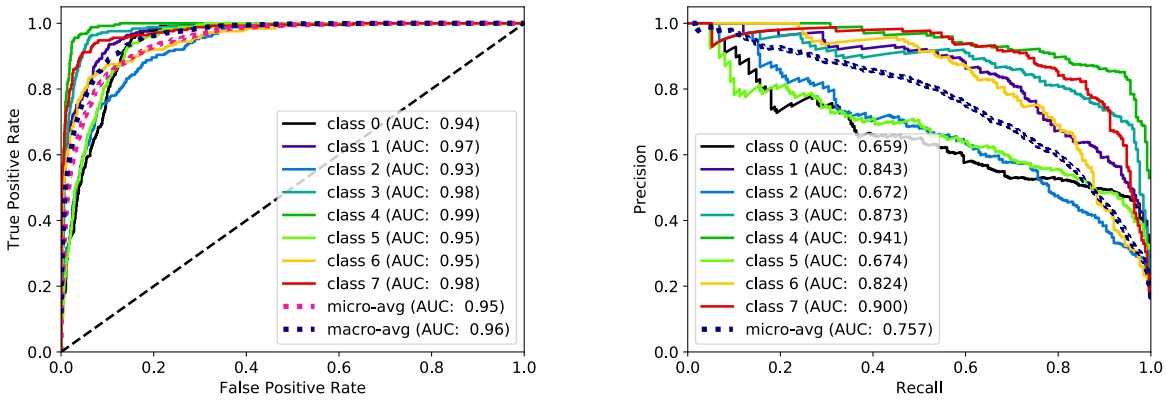
  - **Per class ROC:**
    One curve is included per class, along with the accompanying Area Under the Receiver Operating Characteristic curve, aka. AUC-ROC (AUROC).
  - **Micro-average ROC:**
    The average ROC curve for all classes using the micro-averaging method.
  - **Macro-average ROC:**
    The average ROC curve for all classes using the macro-averaging method.

  The AUROC is included for each class, and for the micro and macro averages. In general, we wish to maximize the AUROC.

- **Precision-Recall (PR) curves:**
  As each curve for each class of the ROC curves can be interpreted as the curve for a separate binary classifier, the curves will often not tell the whole story. This is especially true when the dataset is skewed [39], which it is in our case as we have, for each class, eight times more negative samples than positive ones. It would be useful for us to have another plot which could show the relationship between the precision and recall for any given threshold, i.e., the Precision-Recall. Each PR figure contains the following curves:

  - **Per class PR:**
    One curve is included per class, along with the accompanying Area Under the Curve (AUC).
  - **Micro-average PR:**
    The average PR curve for all classes using the micro-averaging method.

  There can be large differences between the apparent performance on the PR and ROC curves [40]. An example of this can be seen in figure 3.4, where the ROC curves appear to be acceptable, with AUROC scores over 0.9 for all classes. However, the PR curves tell a different story. Here the goal is for each curve to be as close to the upper right-hand corner as possible, and we can see that the curves for class 0,2 and 5 are much worse than the others, lowering the average curve and the micro-average AUC. This is not easy to see on the ROC figure, where the AUROC for these classes seems to be only marginally less than for all other classes.



(a) Multiclass ROC curves

(b) Multiclass PR curves

Figure 3.4: Example Receiver Operating Characteristic and Precision-Recall curves for Inception-ResNet-v2, multiclass, generated by Scikit-plots.

3.4.4 Metric Goals

As our trained networks have the potential for use in a clinical setting [32], it will be helpful for us to define a threshold for all, or at least several, of our metrics that show us if the resulting classification is suitable for diagnostic work. The threshold for when a network becomes a good classifier is difficult to define exactly, as these thresholds vary based on the data in question and the application. As our dataset is used in a medical setting, some metrics might be more important than others, particularly metrics that place a high value on avoiding false negatives. Metrics where false negatives heavily reduce the score would be more suitable to optimize.

One such metric is recall or sensitivity, as discussed in section 3.4.1. Research suggests that a reasonable threshold, for a network to be used for diagnostic purposes, is for recall to be greater than 0.85 [41]. The same research also suggests that the specificity should be higher than 0.85. Based on this, when evaluating our results, we will look for values higher than these thresholds for recall and specificity. We should also aim to have F_1 and MCC scores which are as high as possible; these metrics are commonly used in machine learning to determine the quality of classification [37].

Besides, we have set a goal that our networks should be able to process data in real-time. This means that they should be able to process frames at least as quickly as the equipment can produce them. After searching for several different makes and models of endoscopic equipment, we found that the number of frames per second varied considerably. Some wireless pill-type cameras produced as little as 3 Frames Per Second (FPS) [42], whereas high-end wired equipment produced up to 60 FPS [43]. It appeared to us that the higher frame rate was a selling point for many of the high-end cameras and that this might be a trend towards the future. Some of the cameras also featured a 30 FPS output [44], so we have set this as the minimum goal.

For the other metrics, we decided to compare our results to those found in the original paper for the Kvasir dataset, and aim to achieve better, or at least as good, results [32]. With these specifications in mind, we have set ourselves the goals seen in table 3.3 for metrics and processing performance.

We have not set any specific goals for the AUC and AUROC, as there was little research available for comparison. It would be interesting for future researchers to examine the relationship between the ROC and PR curves and the diagnostic suitability of a method.

Metric	Minimum diagnostic	Desired
Recall	0.85 [41]	>0.85
Precision	N/A	>0.748 [32]
Specificity	0.85 [41]	>0.964 [32]
Accuracy	N/A	>0.959 [32]
MCC	N/A	>0.711 [32]
F_1	N/A	>0.747 [32]
FPS	30 [44]	60 [43]

Table 3.3: A list of our major metrics and our desired values.

3.4.5 Selection Method

There were many potential methods available to us when we were looking for an appropriate way to select our output class from the ensemble of OVR classifiers. We reviewed several of these methods, however with the inclusion of the ROC and PR curves in our report, we determined it would be prudent to select the most straightforward method. This was done both to save time and to avoid introducing errors/bias into the statistical analysis.

The challenge here is that while the multiclass networks output a probability vector of length n, where n is the number of classes, the binary classifiers each output a probability vector of length 2, one probability for negative and one probability for positive.

Considered methods

- We considered thresholding of the outputs for both the binary and multiclass classifiers. In this case, thresholding would mean that any positive output more than our set threshold would be labeled as true, and any below it would be labeled as false. The method seems interesting, but the tuning of the threshold would be challenging and time-consuming and was thus decided to be beyond the scope of this thesis.
- We also considered training a "top" network on the outputs of the binary networks, which itself would provide the same type of output as the multiclass network. Again, this would be an interesting experiment, but we decided not to pursue this.

Selected Method

In the current study, we decided to select an output, after discussion with the Media Processing Group at Simula Research Laboratory, by selecting the binary network that had the highest positive probability output. To do this, we combine all positive probabilities of the binary networks into a vector and select the maximum value along that vector as our output class. This strategy seems to be common in OVR classification [2] [45].

3.5 Main Machine Learning Framework

To answer our research questions in an efficient and surmountable manner, we required a suitable framework to build our networks and run them. We decided that it would be best to select a framework which allowed ease of use and with which we were already familiar. As it would be impractical to implement our framework during the allotted time for a short thesis, Keras seemed to be a promising alternative. Keras offers several benefits when it comes to straightforwardly

implementing machine learning algorithms. Keras is a model-level library that can use several backends [18].

For several reasons, it seemed appropriate to use Tensorflow GPU as a backend; the backend was already familiar to us and should offer acceptable performance on our hardware, using a general-purpose GPU [24]. In addition to the ease of using different backends, Keras has several built-in network structures and applications. Our thinking was that using several of these applications would offer more opportunities for comparison of results, and it would not be very time consuming to implement. Thus, a selection of the available applications was chosen, as detailed below.

Unfortunately, Tensorflow is not the fastest machine learning library available, so in future research, it might be prudent to use different libraries and re-examine the results [46].

3.5.1 Applications

In table 3.4 we have compiled a summary of the chosen applications, detailing the amount of parameters, the number of layers, the depth and the size on disk of each application. There is a large variety of complexities and network sizes. Interestingly, this shows us one of the first issues with a binary implementation, which requires one fully trained model per class. The number of classes and network size means that, in the case of NASNet Large, we will end up with approximately 8 GB of applications. These models must be stored on the disk and, perhaps even more importantly, must be loaded into video memory during normal operation/classification. The large size of these models will pose a challenge on systems where video memory is limited.

Also, model visualizations for several of the models used have been provided in the appendix in section A.1.[13] These visualizations, generated as outlined in table 3.2, give us an insight into several of the network structures and the balancing which has gone on between network depth and network width. It also shows us how the networks are structured including residual connections, parallel blocks/layers and so on. Also, these visualizations afford us the ability to envision the size and complexity of each structure more intuitively.

Network	Parameters (millions)	Depth	Number of Layers	Size on disk
VGG16	138.35 [18]	21	21	115.6MB
VGG19	143.6 [18]	24	24	155.7MB
Inception v3	23.85 [18]	159	313	177.1MB
DenseNet 121	8 [18]	121	428	81.3MB
DenseNet 169	14.3 [18]	169	596	69MB
DenseNet 201	20.242 [18]	201	708	94.4MB
Xception	22.9 [18]	126	134	121.9MB
Inception RN v2	55.87 [18]	572	782	261.3MB
MobileNet	4.25 [18]	88	98	32.4MB
NASNet Large	88.9 [23]	768	1021	1002MB
NASNet Mobile	5.3 [23]	384	751	54.8MB

Table 3.4: A list of the chosen applications with specifications, depth and layers adjusted to match our actual implementation. Size is based on the size of trained models.

3.5.2 Transfer Learning

Keras offers us the ability to import networks with pre-trained weights, i.e., an easy way to implement transfer learning. We decided to use this ability, and as our problem is an excellent

[13] Some of these visualizations have been truncated/edited so that they will be easier to include in this document.

example of image classification based on features we decided to use networks which were already trained on the ImageNet dataset, as these were available. This should allow us to train our networks much more quickly and efficiently.

In our implementation, we do not include the top of the pre-defined networks, and instead, build our own top layers: a pooling layer and a fully connected layer with two or eight outputs depending on the configuration. Then, during the initial training, we freeze all weights other than the ones in our top layers. This allows us to quickly gain some accuracy while retaining much of the pre-trained information. Afterward, we unfreeze a certain number of blocks at the top of the imported network, and "fine-tune" with a lower learning rate. Unfreezing more of the weights allows the network to learn more about the data.

This method is especially important in our case as we have a relatively small dataset, and as mentioned in section 2.1.6, transfer learning is useful in such cases.

3.5.3 Hyperparameters

If we wish to have any chance of decent network performance, certain hyperparameters must be selected. Because automatic hyperparameter optimization is not a part of this thesis, much of it was down to trial and error. After communication with the media processing group, some basic hyperparameters that worked well in previous work were chosen. The rest of the hyperparameters were based on those used initially, after adjustments to make the networks perform better. In the end, we were able to achieve an accuracy score, as defined in 3.4.1, of 0.85 or more on the validation set for all multiclass network styles. The binary hyperparameters were then based on those used for the multiclass networks.

There is more work to be done here, as one could optimize the multiclass networks further, and optimize the binary networks individually for each class. The process would take very long time, or require an automated approach, so further optimization was determined to be outside the scope of this thesis. Furthermore, we will not go into a significant amount of detail on how each of the hyperparameters impacts the results, how they work, or their benefits compared to other choices, as that could be several theses in and of itself.

- **Batch size (BS):**
  For the batch size, we selected the largest possible batch size which would still function with the chosen network style and image dimensions. Some applications require so much video memory during training that they would cause a "Resource exhausted: OOM" (Out-of-memory) error in Keras. We selected the largest functional size so that the gradient estimates for the optimizer would be the most accurate.
- **Input image dimensions (Dims):**
  Here, we selected the default input image dimensions from Keras for each application [18]. These, in turn, are based on the image dimensions used in each style's respective research.
- **Loss function:**
  We selected a loss function which seemed to perform well in previous experimentation with Keras. Specifically, we selected categorical cross-entropy as each of our samples belong to a single class.
- **Activation function:**
  We selected a Softmax activation function. Softmax is a common activation function in machine learning and seems to work well in our case.
- **Optimizer (Opt.):**
  We selected Nadam here. Nadam is a variant of Adam (Adaptive Moment Estimation). It incorporates Nesterov Momentum and seems to work well for our applications.

- **Learning Rate (LR):**
  We adapted and tuned the learning rates for each application and phase. Often the base training will have a higher learning rate than the fine tuning; this seemed to work the best in our experience. Some applications use the default learning rate for Nadam, 0.002, but many do not. The other variables for the Nadam optimizer β_1, β_2 and ϵ, were set as the default in the Keras implementation [18].

- **Based Model Last Block Layer Number (BMLBLN):**
  This hyperparameter selected the number of layers unfrozen during the fine-tuning phase. We, for the most part, selected values which would unfreeze the last few blocks of the given applications structure. Some applications had other values which we found to work well after brief experimentation.

- **Number of training epochs:**
  Although our training regimen implements early stopping, we set a limit to the amount of training and fine-tuning epochs for each style. During the initial training, we use one fifth the amount of epochs as during fine-tuning. As many of these networks take an extremely long amount of time to train, as we will discuss further in the results section , the number of epochs for the binary networks was set to be 1/8th of that of the multiclass networks.

- **Patience:**
  Patience is the number of epochs the network will tolerate not improving the validation loss. After this number is reached, the early-stopping function terminates training. This value was selected primarily to save time.

In table 3.5 we can see the finally selected hyperparameters for each of our applications. As we can see the hyperparameters for the binary networks that will make up our OVR structure are the same as for our multiclass applications.

Network	BS	Dims	Opt.	LR (Train)	LR (Tune)	BMLBLN	Epochs (Max)
VGG16 Multi	64	224x224	Nadam	0.002	1e-06	15	200
VGG19 Multi	64	224x224	Nadam	0.002	1e-06	17	200
Inception v3 Multi	64	299x299	Nadam	0.002	1e-05	249	200
DenseNet 121 Multi	64	224x224	Nadam	1e-04	1e-05	394	200
DenseNet 169 Multi	16	224x224	Nadam	1e-04	1e-05	530	200
DenseNet 201 Multi	64	224x224	Nadam	1e-04	1e-06	642	200
Xception Multi	16	299x299	Nadam	0.002	1e-05	126	200
Inception RN v2 Multi	64	299x299	Nadam	0.002	1e-05	64	200
MobileNet Multi	64	224x224	Nadam	1e-03	1e-04	762	200
NASNet Large Multi	8	331x331	Nadam	1e-05	1e-06	100	100
NASNet Mobile Multi	64	224x224	Nadam	1e-05	1e-06	0	200
VGG16 Binary	64	224x224	Nadam	0.002	1e-06	15	25
VGG19 Binary	64	224x224	Nadam	0.002	1e-06	17	25
Inception v3 Binary	64	299x299	Nadam	0.002	1e-05	249	25
DenseNet 121 Binary	64	224x224	Nadam	1e-04	1e-05	394	25
DenseNet 169 Binary	16	224x224	Nadam	1e-04	1e-05	530	25
DenseNet 201 Binary	64	224x224	Nadam	1e-04	1e-06	642	25
Xception Binary	16	299x299	Nadam	0.002	1e-05	126	25
Inception RN v2 Binary	64	299x299	Nadam	0.002	1e-05	64	25
MobileNet Binary	64	224x224	Nadam	1e-03	1e-04	762	25
NASNet Large Binary	8	331x331	Nadam	1e-05	1e-06	100	25
NASNet Mobile Binary	64	224x224	Nadam	1e-05	1e-06	0	25

Table 3.5: A list of the chosen hyperparameters for each network style

3.6 Hardware Metrics

We have chosen several hardware metrics to measure which resources Keras is using during all phases of the process. We chose these metrics to give an impression of the type of system resource impact one might expect when using these specific architectures, and how using an OVR strategy affects this use. For all metrics, we chose to sample every 0.5 seconds. This sampling rate was deemed appropriate given the amount of time it takes to train one of these networks (up to 30 hours) and also given the number of metrics logged, 15. Also, we determined that this would have a negligible impact on system resources on its own, i.e., a low polling rate would not influence the results.

3.6.1 General System Metrics

Using psutil, we can collect many different system metrics [47]. We selected the most interesting ones to log:

- CPU Use per Core
  CPU use per core is the volatile use percentage of the CPU measured individually per core.
- CPU Use Average:
  The average CPU use is the volatile use percentage of the CPU averaged over all cores.
- CPU Temperature:
  The CPU temperature is the temperature of the die on package id 0, be aware this might be platform dependent [47].

- System Memory used:
  Additionally, we measure the amount of system memory used at the time of measurement in bytes.
- Disk Input/Output (I/O):
  Finally, we measure the number of bytes read/written to the disk.

3.6.2 GPU Metrics

On the GPU, a vast quantity of metrics is available via NVML [48]. We selected a subset of these to implement in our testing software. Not all of these metrics will be useful in our final analysis, but we will still collect data about them for the sake of analysis.

- GPU Volatile Percentage:
  The GPU volatile percentage is the percent of the time over the last sample period during which one or more kernels was executing on the GPU [48]. The sample period is set to be 0.5 seconds.
- GPU Memory Percentage:
  The GPU memory usage is the percent of the time over the last sample period during which global (device) memory was being read or written [48]. The sample period is the same as above.
- GPU Memory actual in B:
  We can retrieve the amount of used memory on the device, in bytes.
- GPU temperature:
  Here, we get the device temperature on the GPU die in °C.
- GPU Fan speed:
  We can retrieve the intended operating speed of the device's fan in percent. This percentage may differ from the actual operating speed [48].
- GPU Power usage in mW:
  We retrieve power usage for this GPU in milliwatts and its associated circuitry (e.g., memory). This should be within +/- 5% of current power draw [48].
- GPU Device Clocks:
  - Graphics clock speed in Mhz.
  - Streaming Multiprocessor (SM) clock speed in Mhz.
  - Memory clock speed in Mhz.
  - Video Clock speed in Mhz.

3.6.3 Parallelization

In our results we are interested in the output speed and performance for each network structure, it is interesting for us to examine how parallelizable these network instances are when it comes to processing frames. As discussed in section 2.2.1 the python interpreter is equipped with a Global Interpreter Lock (GIL) which might hamper parallelization.

Also, the CUDA library does not allow more than one context, as discussed in section 2.2.2, to sequester the GPU resources at the same time. This will likely also significantly affect our ability to parallelize. It will be of interest to us to confirm this using our available metrics.

3.6.4 Calibration

In the interest of determining whether or not our metric measurements affected their own results, we ran multiple calibration measurements to determine what the baseline resource use was for the system with a single network loaded into the memory. This test was run for 1 minute before every training, fine-tuning and testing session. This was done to ensure that in the case of abnormal results, we could refer back to the calibration phase and make sure that other processes were not taxing the system. As the system in question was not running a Graphical User Interface (GUI) or any resource heavy background processes, these results should read minimal resource usage across the board except for memory usage, as a single network (DenseNet 121) was loaded.

GPU Usage

In figure 3.5, we can see that our volatile GPU during the calibration period is 0% for the entirety of the period. This means that unless training or testing is creating a context on the GPU, it is not being used by Keras (or any other process.)

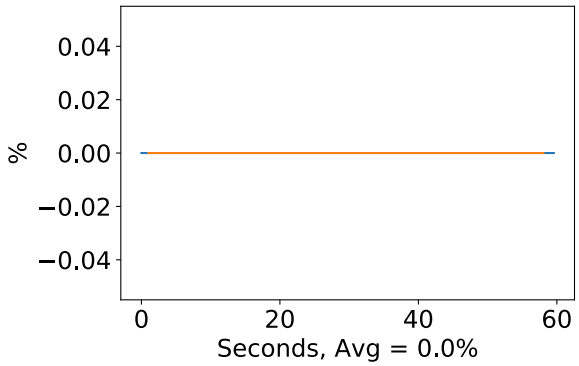


Figure 3.5: GPU volatile usage during calibration, with a moving window average over 40 measurements overlaid

Strangely, the same can not be said for the volatile memory usage, as seen in figure 3.6. Here the usage is 0% at first, then increases to 2% for the remainder of the period. This result was consistent for all tests we checked, so it may be that this increase is due to the interactions between Nvidia Management Library (NVML) and the GPU memory to read the results. Hence, this is something we should keep in mind when interpreting the results later on.

Looking at the GPU power measurement in figure 3.6, it starts at over 50W, then declines after around 10 seconds. It then remains stable at a little less than 20W. 20W would appear to be the idle power usage for the GPU. When we looked at the clock metrics, we observed that all the GPU clock speeds were reduced at this time. This reduction would appear to confirm what we speculated earlier: the clock speeds are reduced to conserve energy.

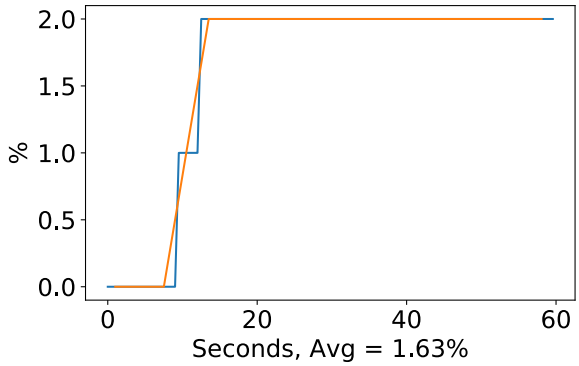


Figure 3.6: GPU memory usage during calibration, with a moving window average over 40 measurements overlaid

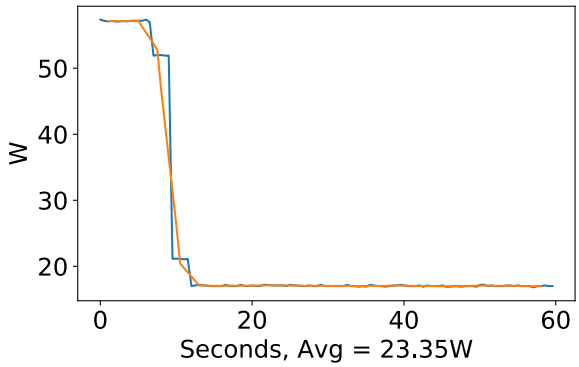


Figure 3.7: GPU power usage during calibration in W, with a moving window average over 40 measurements overlaid

CPU and memory usage

Moving on to the CPU usage in figure 3.8, we can see that the average usage is around 1% during the test time. This seems reasonable considering that we have three threads running during the calibration, each logging their own system metrics. It does have a spike of 15% at the first index, but this does not seem problematic to us as this is seems to be an aritifact of the way psutil gives data for the first measurement [47]. We should be able to assume a baseline use of around 1% when considering our results.

The memory usage during calibration, seen in figure 3.9 is interesting. Because of the relatively large size of the DenseNet model, we have a memory usage with the model loaded of around 1.65GB. We should expect this to change depending on what model is chosen. After looking at the other results, we can confirm that it does, which brings us back to what was mentioned in section 3.5.1. Loading 8 NASNet models into the system memory at one time means that on a system such as ours, over half of the memory is already used before the test data is loaded. If we wish to process large amounts of data, this could be problematic as we would need to either load the data into batches or only use ImageDataGenerators.

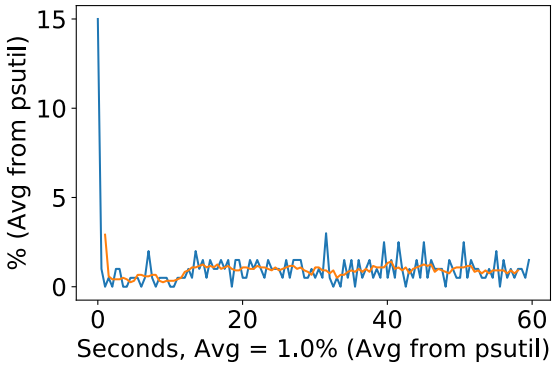


Figure 3.8: CPU usage (averaged over 4 cores) during calibration, with a moving window average over 40 measurements overlaid

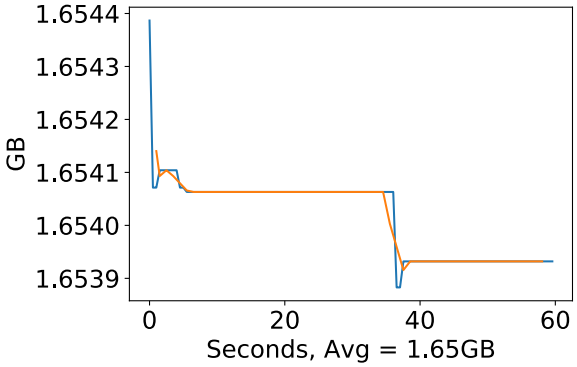


Figure 3.9: Memory usage during calibration, with a moving window average over 40 measurements overlaid

3.7 Summary

We created a comprehensive system to test our hypothesis and collect metrics about our results. The system, TFmetrics, can automatically run our experiments and compile results in a human-readable format. It will collect several metrics about all aspects of our network's use and will be useful for us in our final analysis and conclusion. By using the classification metrics outlined here, we should be able to get a good idea of how useful our OVR style implementation could be in a clinical setting. On the other hand, our hardware metrics provide us with an insight into what is influencing the performance of our networks, and what kind of hardware would be required to use our solution in a real-world scenario. Together these metrics will provide us with a basis to answer our central research questions. Now that we have designed a testing framework, it is time for us to begin our experiments. We will run a substantial number of experiments, and collect a significant amount of data. In the next chapter, we will attempt to break down all of these experiments and provide a useful context for them. In addition, we will attempt to show how these experiments help us answer our research questions.

Chapter 4

Experiments

During the progression of our research, we performed a rather substantial amount of experiments and collected a significant amount of data. In this chapter, we hope to present it concisely and explain what this data means for our research questions. We aim to examine the differences between an OVR multi-network approach and a multiclass single-network approach. These differences will include resource use during all phases, time to train, and finally classification performance, including both speed and accuracy. We aim to provide an idea of the tradeoffs involved in selecting our OVR style and the conventional multiclass style. It will aid us in answering our primary research question; is OVR a viable strategy using modern neural networks?

The chapter is structured such that we will first examine all aspects of both training phases, then will we will examine the resource use of the testing phase. Finally, we will examine the classification performance. Hopefully, the chapter will provide a good breakdown of all phases of use for our network styles.

4.1 Common Results

4.1.1 Included Results

In total, we collected data on 186 networks' training, fine-tuning and testing sessions. 50 of these were multiclass, and 136 were individual binary networks. On our final test selection of 99 networks (11 multiclass and 88 binary), the metrics logged and calculated resulted in 16,029,806 total data points collected and 19,974 plots being generated by our testing framework. Although most of the hardware metrics are interesting in some regard, some of them were especially interesting, and thus we will concentrate on presenting those in this chapter. More specifically we will focus on the following metrics.

- CPU Volatile use percentage averaged over all cores:
  We find that this metric provides a reasonable estimation of the CPU usage during all phases. As this is one of the primary system components, we conclude that this resource use is valuable to investigate, especially considering the potential additional CPU overhead involved in the OVR style during testing.
- System Memory used:
  The metric provides an impression of the amount of system memory required during all phases of use, which will be helpful, as we suspect the required memory will increase drastically in the OVR configuration.
- GPU Volatile Percentage:
  This is perhaps the most critical hardware metric, as it directly expresses how much of the GPU operational resources are being used at any given time. When we see a difference here, we usually expect to see a corresponding difference in power usage and temperature fluctuations.
- GPU Power usage in mW:
  This metric is important as it gives us a good idea of what some of the real world consequences might be of selecting a certain network style. Electricity is a valuable resource, and we should endeavor to select a style that provides us with satisfactory classification results in addition to being environmentally conscious.

As can be expected in such an experiment, not all of the metrics we collected were particularly useful. Many of our collected metrics will be omitted from the discussion, as they are not of particular interest in this research. More specifically, the following system metrics, as discussed in section 3.6 will be omitted from this section in the interest of brevity and succinctness:

- CPU Percentage used per core:
  These plots are almost unreadable within any significant sample period, and will be omitted. We determined that the average CPU usage metric would be specific enough for our use.
- CPU Temperature:
  Similar to the GPU temperature metric, this seemed highly related to the volatile use percentage.
- Disk I/O:
  Although this metric might be useful for analyzing the way Keras' ImageDataGenerator object streams training, validation and test data from the disk, we determined it to be outside the scope of our research.
- GPU Memory actual in bytes:
  After reviewing the stored data, we determined that this metric was not as useful as the volatile memory usage. In particular, because the memory seemed to be constant at the maximum storage of the device. Furthermore, it seems likely this phenomenon caused

by TensorFlow mapping all device memory immediately after starting. It appears that TensorFlow can be programmed to not do this, but unfortunately that was not implemented during our testing.

- GPU Memory Percentage:
  Although the memory usage percentage on the GPU was more interesting than the actual memory used in bytes, we will not be examining it in this section as we found that the other metrics provide a good summary. These plots are included in the appendix for reference.
- GPU temperature:
  This metric seems, perhaps unsurprisingly, to be a function of the GPU volatile usage over time, and as such does not directly relate to our research.
- GPU Fan speed in percentage:
  Configured fan speed is directly related to the device temperature so that it will be omitted as well.
- GPU device clocks:
  - Graphics clock in Mhz.
  - SM clock in Mhz.
  - Memory clock in Mhz.
  - Video Clock in Mhz.

  These are somewhat interesting, as they do vary slightly over time, likely as a function of the GPU trying to optimize power usage. However we decided it would be outside the scope of this thesis to analyze these optimizations, and as such these will be omitted.

4.1.2 The Appendices

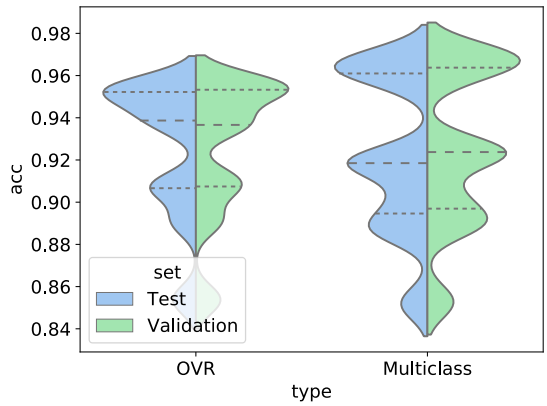
In the interest of brevity, we will only be examining some of the results in great detail. We will examine those networks which displayed the most exciting results or representational results. There are several plots that are not included directly in this chapter but are instead included in the appendix. The plots, tables, and figures are of the same type as those we have included here. We have included them to support our conclusions, and provide the reader with reference material should that be required.

4.1.3 Performance on the Testing and Validation Sets

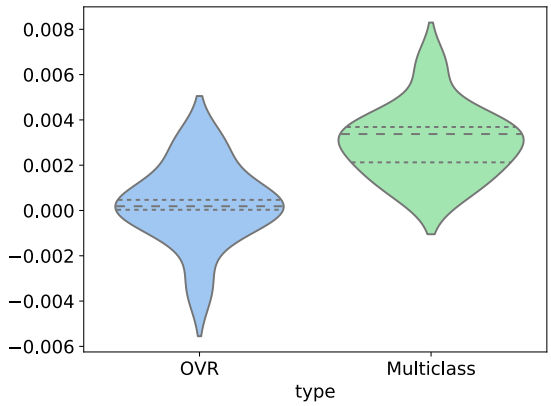
In section 3.2.2, we discussed our choice of having three sets of data with a separate untouched test set. In theory, it would provide us with a better idea of how generalized our performance is on unseen data. The validation data is used for early stopping and adjusting hyperparameters. It will as such be of interest to see if there is a difference in performance between the validation and test sets. As we have a limited number of networks in our final selection, it is by no means a comprehensive test. We still thought it was pertinent to include, but it should perhaps be investigated more closely in future research.

Because the number of results here is limited we will not draw too many conclusions here, however, based on figure 4.1 there might be some difference between the sets, at least for the multiclass networks. For the OVR networks, it appears the difference is insignificant.

A note on accuracy:
In the following sections, we will be discussing two different metrics of accuracy. In our plots that display accuracy as a function of the number of epochs, the accuracy metric is calculated by Keras



(a) Violin plot of the accuracy on the validation and test sets for both OVR and multiclass network styles, with quartiles imposed.



(b) Violin plot of the difference in accuracy on the validation and test sets for both OVR and multiclass network styles, with quartiles imposed. Positive values indicate that the network performed better on the validation set than the test set.

Figure 4.1: Plots of the difference between the validation and test sets

and is a batch estimate. Keras' accuracy is equivalent to our Recall metric when macro-averaged. This difference might seem confusing, but we felt it would be even more confusing to refer to it as recall since Keras does not use this term in their metric calculation [18].

4.2 Training

4.2.1 Accuracy

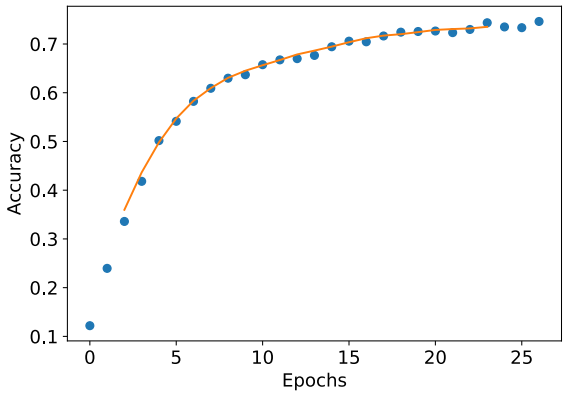
During training, we are interested in many aspects of how the Keras is using the system resources and the differences between network styles in this regard. We also want to analyze how much time is used to do the initial training of the networks. Of interest here is how long each epoch is on average for each network style, and how much time in total it took to achieve the attained performance.

As the training progresses, we expect to see the accuracy increase and loss decrease on both the training and validation sets. For the test set, an example of this can be seen in figure 4.2. Here we see the Keras accuracy rise steadily for each epoch, and the loss falling as we expect. Interestingly, we can also see that our hyperparameter selection is well optimized for the multiclass classifier, but poorly optimized for the OVR classifier in questions. Despite this, both classifiers end up at approximately the same Keras accuracy and loss, despite the Binary classifier having trained for less than 1/5th the number of epochs of the multiclass network.

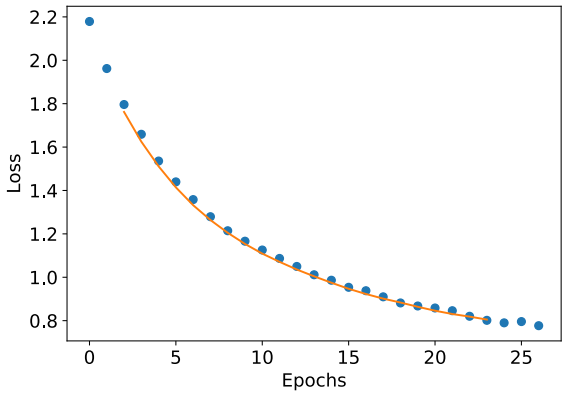
The previous plots illustrate an issue with the style of accuracy Keras chooses: it can be misleading, as it was in this case. In our experience, and after reviewing the data, to expect a high total output accuracy of the OVR ensemble, we want to see higher numbers for Keras accuracy from the individual binary networks.

At the same time as the training accuracy is increasing, we would expect the validation accuracy to grow, although perhaps not as much, as the network is not learning directly from the validation data. The accompanying results to the previous figure can be seen in figure 4.3.

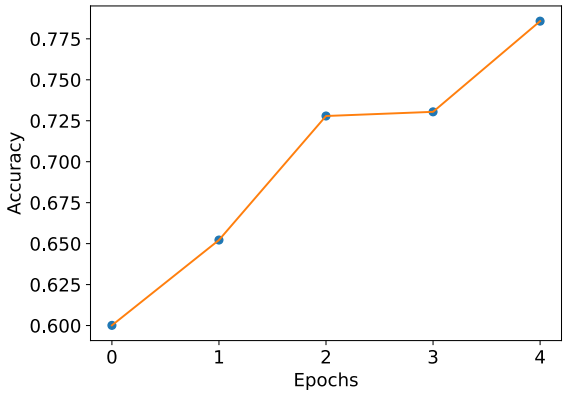
The findings were somewhat unexpected. On one hand, for the multiclass network, we can see what we expected, the accuracy does increase, but not nearly as much as for the training set. The



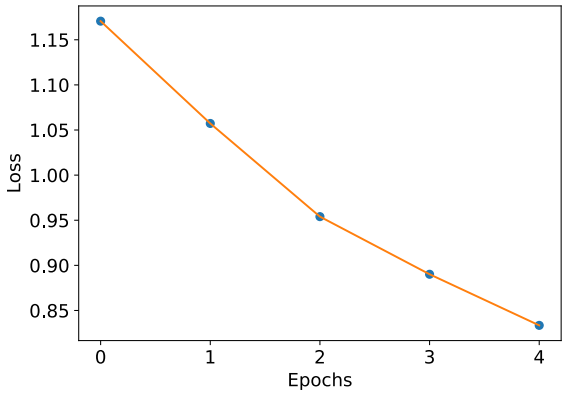
(a) Multiclass network accuracy



(b) Multiclass network loss



(c) Binary network "polyps" accuracy



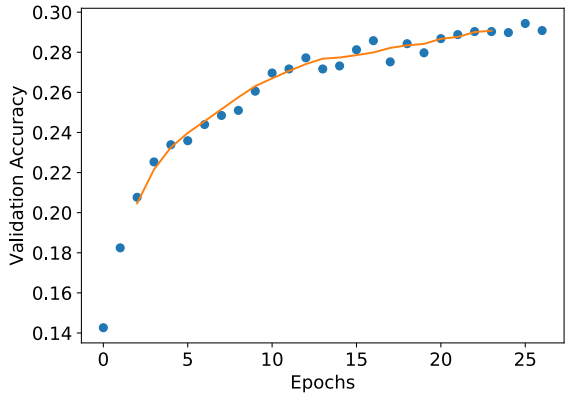
(d) Binary network "polyps" loss

Figure 4.2: NASNet Mobile Training classification Keras accuracy (Recall) and loss history for the training data

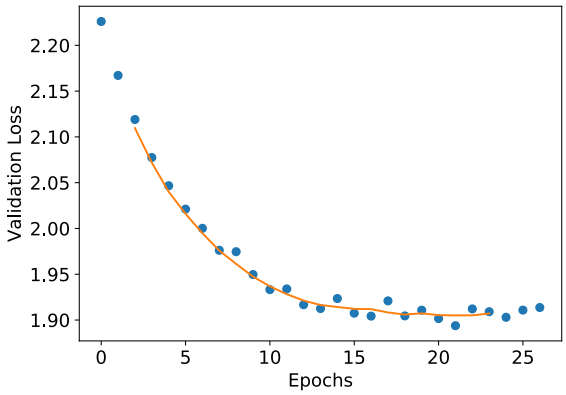
growth in accuracy is also more volatile, which makes sense as the optimizer corrects by updating the weights. This means that accuracy will jitter around the learning curve. As we can see, after training, our accuracy on the training set for the multiclass network is over 0.7, whereas it is less than 0.3 for the validation set.

The binary network's performance is more surprising though, the validation accuracy (≈ 0.85) and loss (≈ 0.38) are quite a lot better than for the training data (≈ 0.775 and 0.85, respectively). We would typically expect that the training accuracy would be higher than the validation accuracy, as the training data is used directly to update the weights. Interestingly, this difference is relatively standard across our experiments, the validation data accuracy for the binary networks is often at least as high, or higher than for the training data. The opposite is true for the multiclass networks.

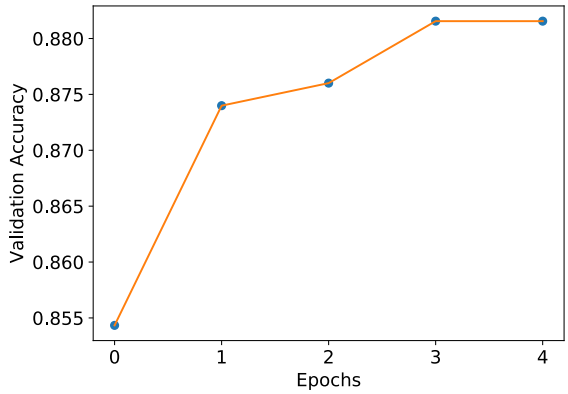
We have included examples of all runs in the appendices, in section A. We would speculate that this phenomenon is due to the use of dropout and other regularization mechanisms during training, but not during the validation calculations [18]. The effect seems to be greater for the binary networks than for the multiclass ones.



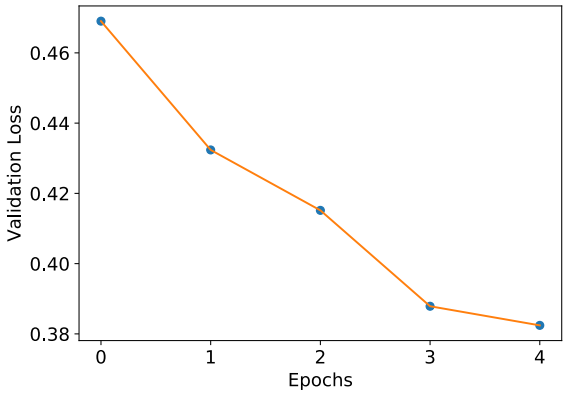
(a) Multiclass network accuracy



(b) Multiclass network loss



(c) Binary network "polyps" accuracy



(d) Binary network "polyps" loss

Figure 4.3: NASNet Mobile Training classification Keras accuracy (Recall) and loss history for the validation data

4.2.2 Resource Use

As mentioned previously, we are also interested in how system resources are used during the different phases of network use. We will look at the use percentage of the GPU, CPU and system memory. The use percentage should give us a good idea of how the binary and OVR networks compare, in addition to how the different applications compare.

GPU Usage

In figure 4.4 we can see a breakdown of the average GPU volatile use for each network style and application. In this figure, the OVR usage is the average of the usage for each class's network during training. Perhaps unsurprisingly, the volatile usage is similar for both the OVR and multiclass styles for most applications, within a few percentage points. In some cases, the usage is slightly higher for OVR, and others it is somewhat higher for multiclass. We can see that, on average, the multiclass use is somewhat higher. Interestingly, there is a rather substantial difference in the case of NASNet Large, where the resource use is much higher for the multiclass network.

Another interesting revelation from this chart is just how little GPU resources are used on average here, and especially in the case of MobileNet and NASNet mobile. These applications are designed to use fewer resources, and as we can see, this is also the case during training. The other network styles seem to be within a few percentage points of the overall average. It appears that DenseNet

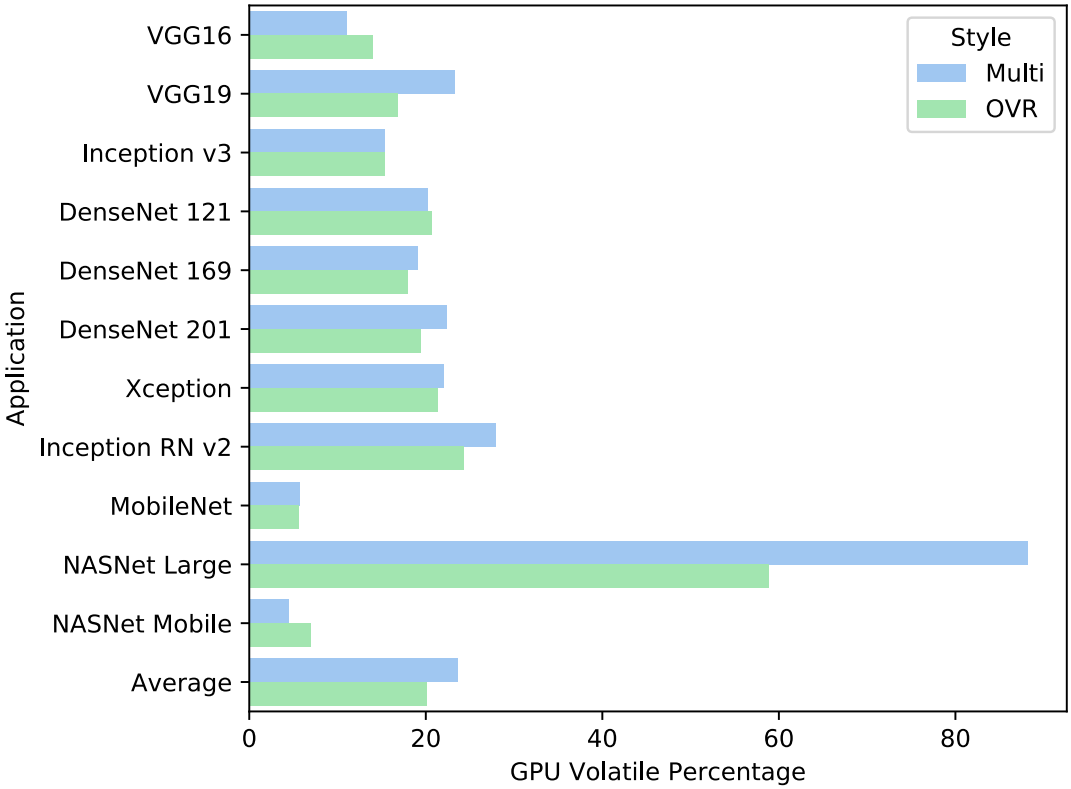
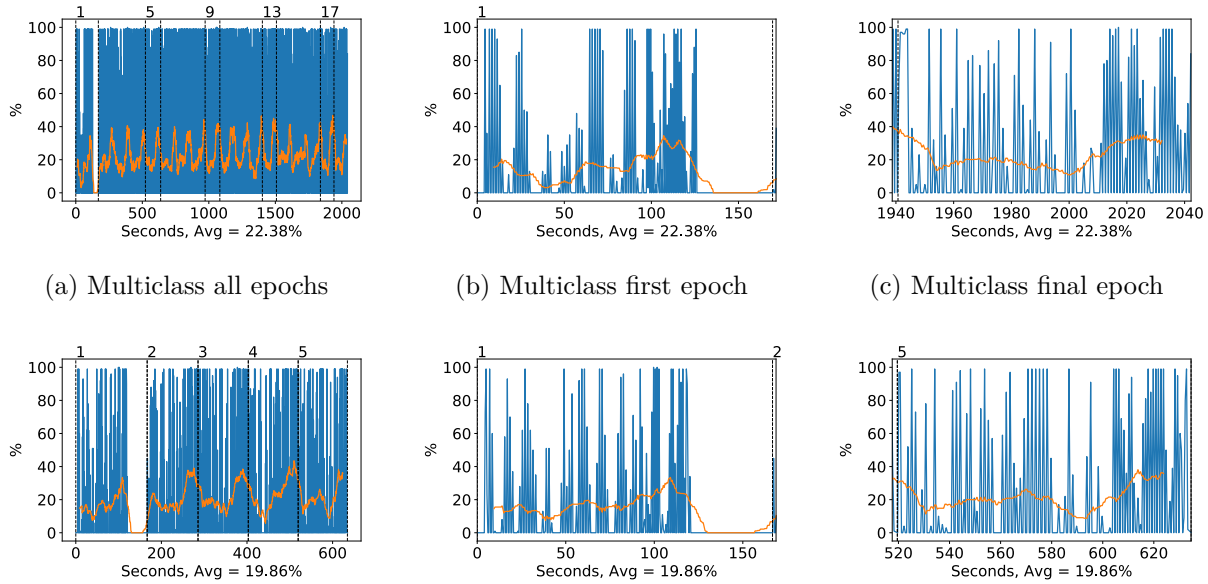


Figure 4.4: Training: GPU Volatile Percentage, including all applications and an average value. For both binary and multiclass styles.

201 is an excellent example of the average case. Of course, there might not be a "typical" case here, as the way resources are allocated could change drastically between applications. We should, therefore, look at a few of the most exciting examples.



(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

Figure 4.5: DenseNet 201 GPU volatile usage during training, with a moving window average over 40 measurements overlaid

Note:
In each of the following figures, which plot resource usage during each phase, the binary network chosen as a comparison is trained on the polyps class of the Kvasir dataset. For all of the results we examined, the difference between classes with regards to resource use was negligible.

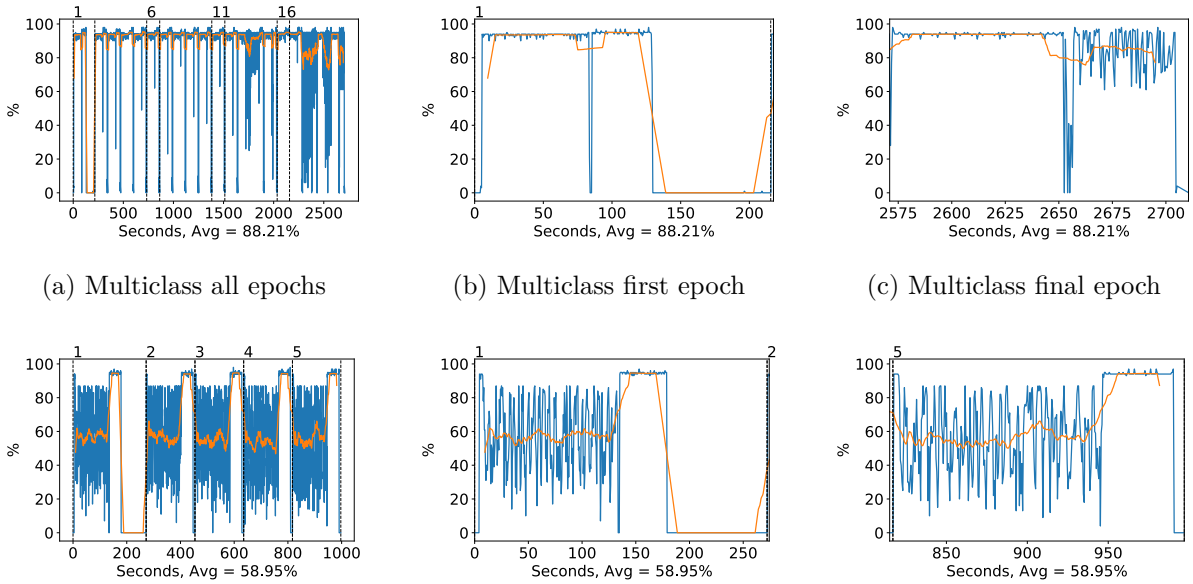
As we mentioned, DenseNet 201 could be representative of a typical case in our experiments, so we have plotted the GPU volatile usage during training in figure 4.5. The plots reflect the results from the summary figure (4.4) nicely and offer some interesting insights into the GPU usage during training.

We can, for example, see the different phases of each epoch play out as training progresses. There is a slight peak at the end of most epochs, where the validation loss calculations take place. There is also a significant dip at the end of the first epoch, and as we can see the first epoch takes longer than the remaining epochs. This dip to 0% usage in the first epoch was also observed with a few of the other applications. After examining the other resource plots, we were not able to find any reasonable explanation for this.

In figure 4.6 we have plotted the GPU volatile usage for the NASNet Large structure. Again, we see the dip at the beginning of the first epoch. Interestingly, there is also a dip at the end of each subsequent epoch, which is also difficult to explain. It may be the case that the drop in the first epoch takes place for the same reason as for subsequent epochs, but lasts longer.

For us, the most interesting relationship in figure 4.6 is the difference in GPU use during the training of the binary and multiclass network. We can see that the average usage during training is much more stable for the multiclass network than the binary network. On average, the use is higher than for the binary network. This might be due to the unbalanced classes of the binary network, although it is difficult to say. Having checked the network structures, we can confirm that the network structure for the multiclass and binary network is the same apart from the top Fully Connected (FC) layer, which has two outputs for the binary network, and 8 for the multiclass. The images are the same as well, with both networks reading symlinks to the same training set.

Contrasting with the NASNet Large example above, the NASNet Mobile volatile GPU usage shown



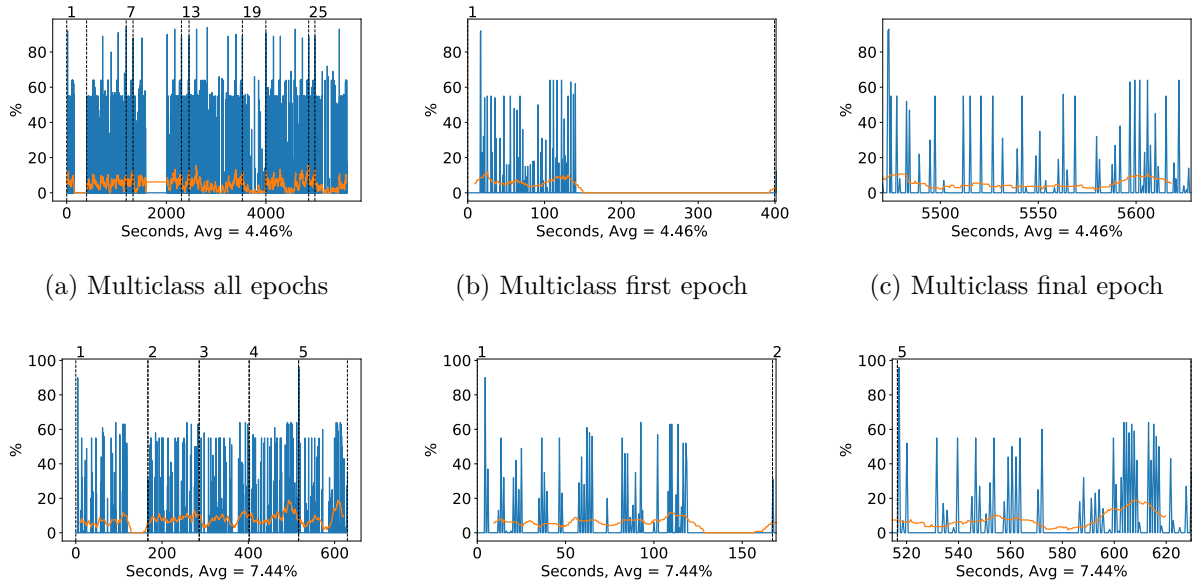
(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

Figure 4.6: NASNet Large GPU volatile usage during training, with a moving window average over 40 measurements overlaid

in figure 4.7 shows much lower GPU use across the board. The NASNet mobile application is less complicated than the large, so this might partially explain it. The overall GPU usage is very low, with much time at 0% usage and spikes to around 60% usage. The dip discussed earlier during the first epoch seems to be a lot longer for the multiclass networks, which affects the overall training time, especially if we were to have fewer epochs. Overall though, we can see that the NASNet mobile application uses minimal GPU resources during training, possibly to the detriment of overall performance and training speed.

However, this also means that, as was the claimed goal for the application, it is suitable for low performance/low power hardware. In future research, running these tests on less capable hardware would be of interest.



(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

Figure 4.7: NASNet Mobile GPU volatile usage during training, with a moving window average over 40 measurements overlaid. Note that the multiclass measurements contain a gap in the data before 2000 seconds. We suspect this might have something to do with the GIL but it is difficult to be sure.

Another interesting metric to look at on the GPU is the power P used in W, and the averages for this can be seen in figure 4.8. We noticed that the relationship between OVR and multiclass network styles remained the same, in comparison to the volatile usage.

We can also see in figure 4.8 that it appears that the difference between applications is more substantial here than for the volatile usage percentage in figure 4.4. In the case of DenseNet 121, DenseNet 169 and DenseNet 201 this difference is particularly large. These three had very similar volatile usage percentages, but different power use percentages, with DenseNet 169 having far less than the others. To get a better idea of the relationship, we can look at the actual recorded data for two of the network styles we examined in the case of GPU volatile percentage.

In figures 4.9 and 4.10 we can see the power draw during training for NASNet Large and NASNet Mobile. Interestingly, the difference is smaller here than when we examined the volatile usage.

We also see the same behavior as before on the NASNet Large application, where the power use is higher on the multi-class structure. It seems likely this behavior is due to the more stable volatile usage during the initial phase of each epoch, where the power draw is relatively constant for the multiclass structure and very volatile for the binary structure.

The same behavior is displayed for NASNet Mobile, the power used is highly reflective of the volatile GPU percentage. The GPU power makes up the bulk of the power usage in this system during all phases, which means that our NASNet Mobile and MobileNet applications likely consume far less power during training. Thus, provided that they can attain a reasonable accuracy, they will be much more efficient.

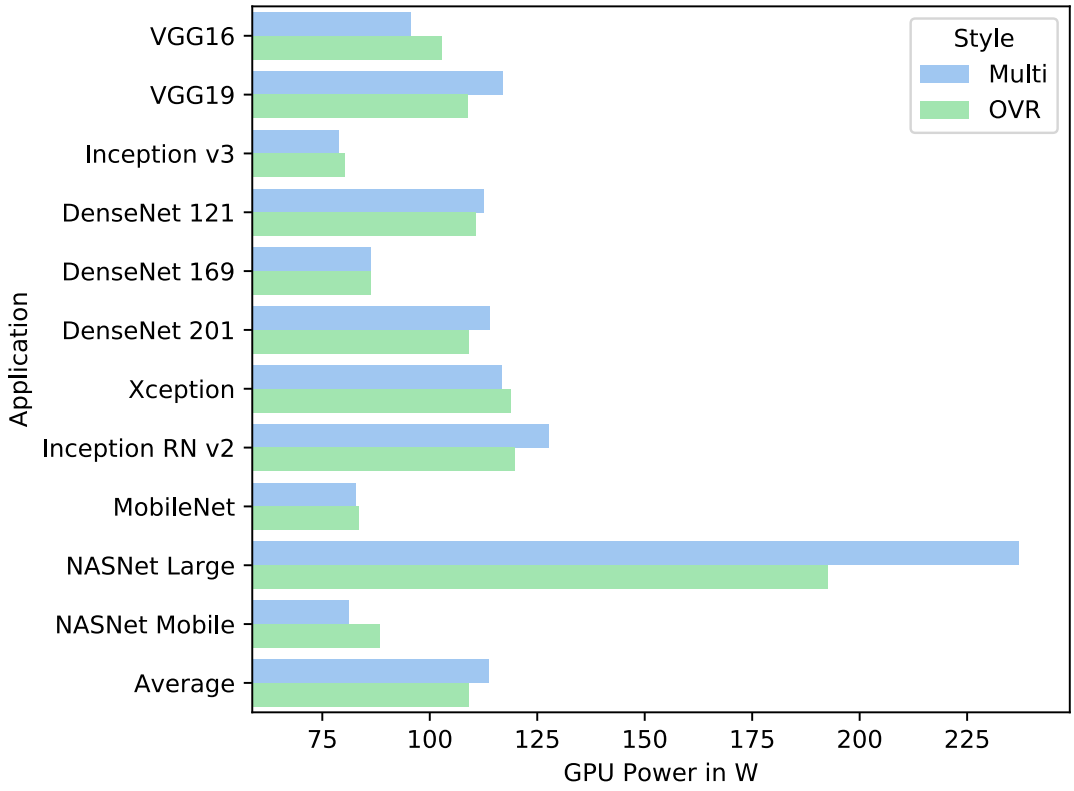
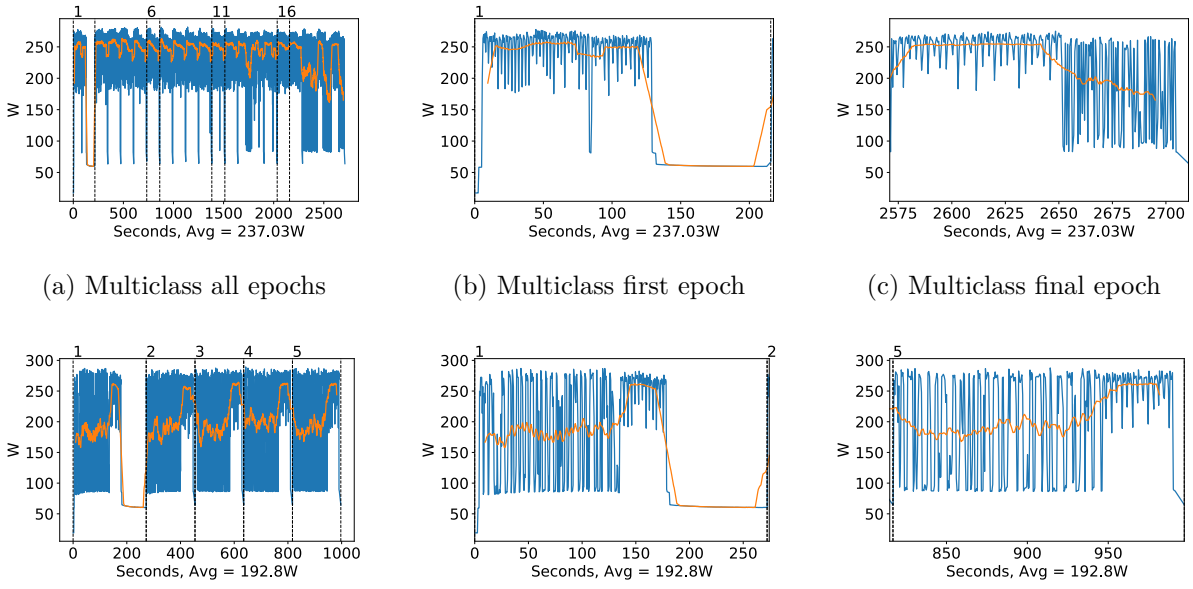


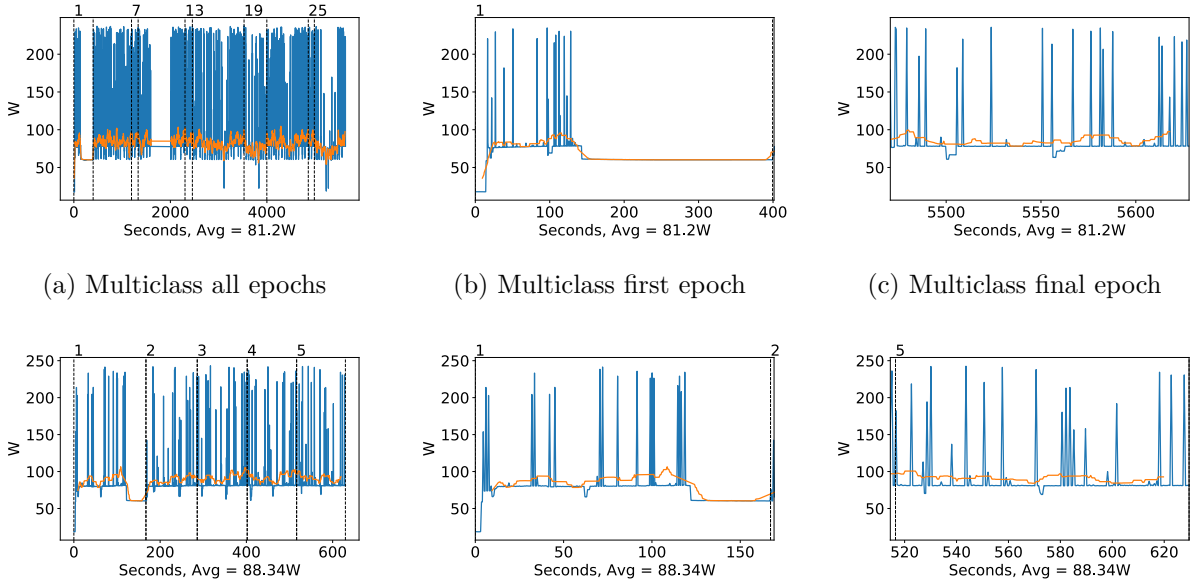
Figure 4.8: Training: GPU Power in W, including all applications and an average value. For both binary and multiclass styles.



(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

Figure 4.9: NASNet Large GPU power usage during training in W, with a moving window average over 40 measurements overlaid



(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

Figure 4.10: NASNet Mobile GPU power usage during training in W, with a moving window average over 40 measurements overlaid

Other Hardware

It is also interesting and informative to look at the other system resources during the training phase and see if any of them might be bottlenecking the performance of our networks, and also to compare across styles. Aside from GPU resources, the most valuable resource during training is the CPU. As the Memory Management Unit (MMU) is implemented here, all memory operations also play a role in CPU use, hence that the loading and unloading of testing data into the network might be problematic. In figure 4.11 we can see the average CPU usage during training for each application and both OVR and binary styles.

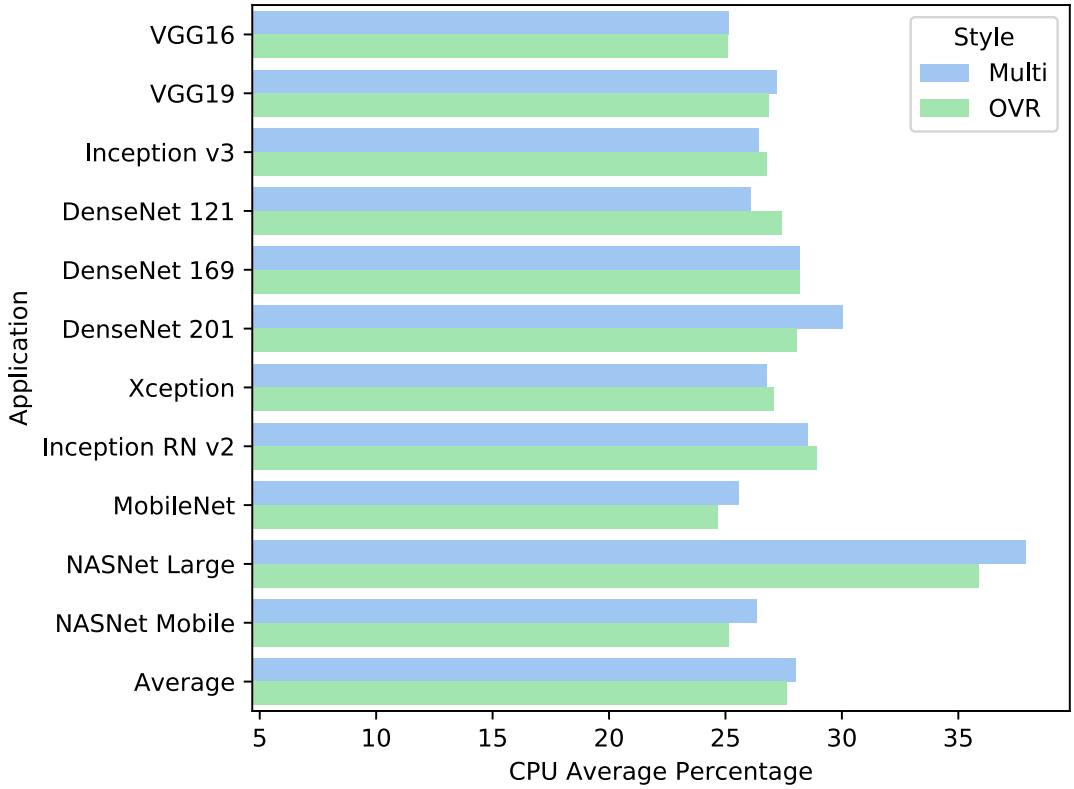


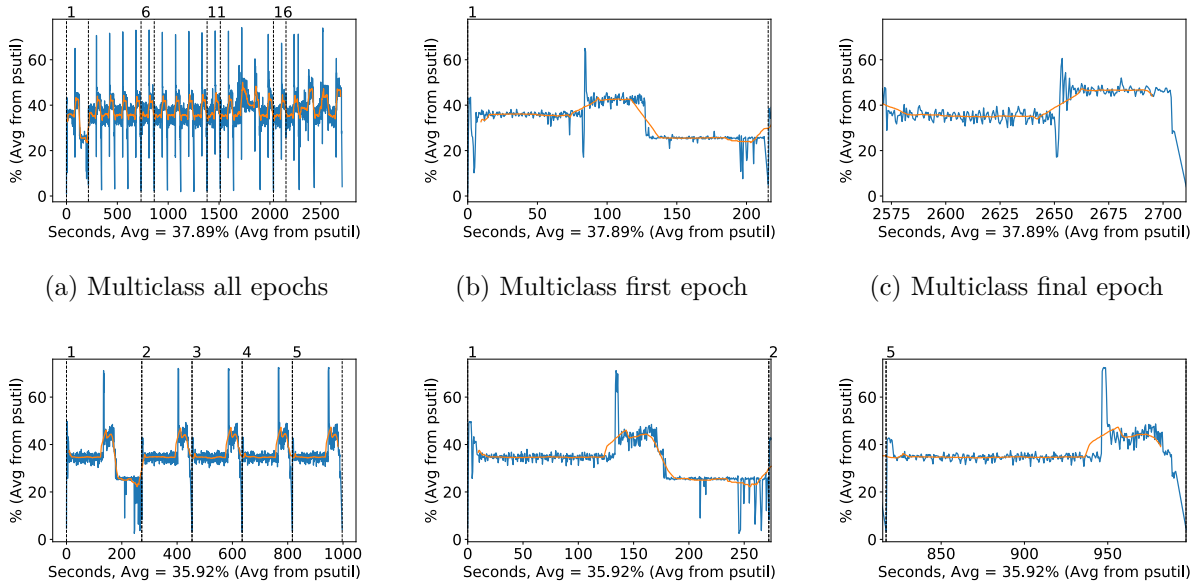
Figure 4.11: Training: CPU Average Percentage, including all applications and an average value. For both binary and multiclass styles.

What we can observe is that, apart from NASNet Large, most of our networks have incredibly similar usage, with the differences between network styles and binary/multiclass seemingly negligible. To confirm this difference, we might be interested in re-examining two examples that we have previously examined.

In figures 4.12 and 4.13, we can see the difference between one of the more normal CPU usage profiles, NASNet Mobile, and the uncommonly high one, NASNet Large. We can also see that the CPU usage is not particularly volatile, at least not to the same degree as the GPU usage. It remains relatively constant, with certain spikes and dips which seem to be related to each phase of the epoch. Comparing these figures to the figures of the GPU usage, we can see that when the GPU usage dips at the end of each epoch, there is a corresponding dip in the CPU usage. It is unexpected, as it is not entirely clear what is happening at this point in the training.

We can also see that in the case of NASNet Mobile, the CPU usage is very stable except for the small bursts about 3/4ths through the epoch. It might be that when the validation test is being run, the validation data is loaded from the disk or memory and into the GPU. In future research, it would be interesting to monitor specific system busses, in addition to logging the system calls

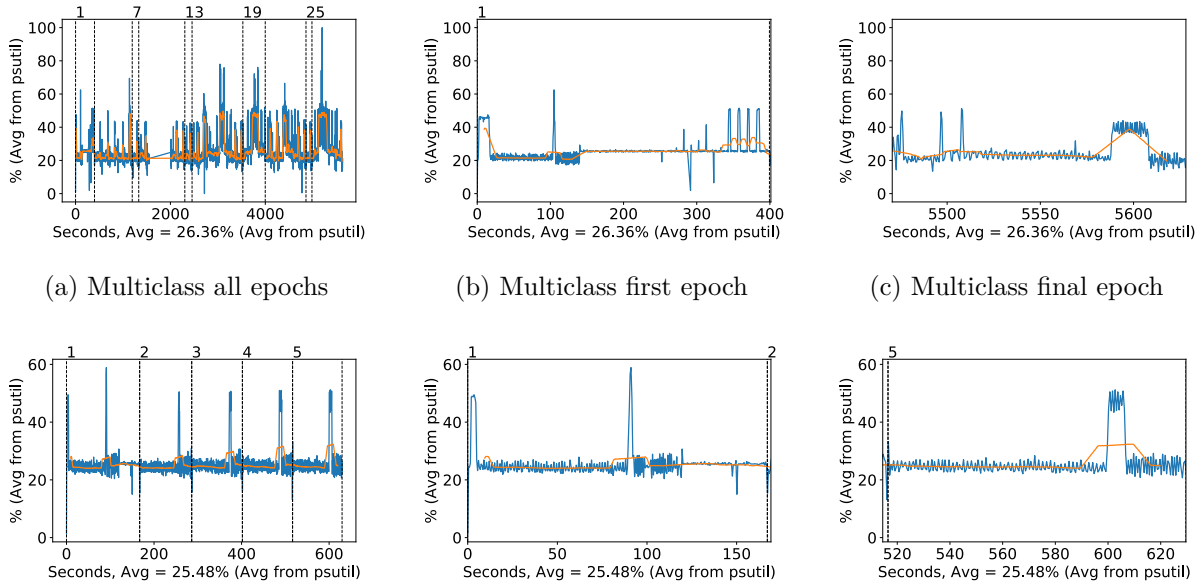
made by TensorFlow.



(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

Figure 4.12: NASNet Large CPU usage (averaged over 4 cores) during training, with a moving window average over 40 measurements overlaid



(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

Figure 4.13: NASNet Mobile CPU usage (averaged over 4 cores) during training, with a moving window average over 40 measurements overlaid

Finally, we can observe the system memory used for each network style and application in figure 4.14. Here we can see that the memory usage varies quite substantially between the applications, and in fact, it also varies whether or not the binary or multiclass network uses the most system memory. After reviewing the plotted memory usage, we were unable to come to any conclusions on why this might be the case. There would seem to be no discernable reason why the

system memory usage would vary so much; it might have something to do with the way TensorFlow stores variables in the system memory, in addition to the way Keras' ImageDataGenerator function stores and loads training and validation data.

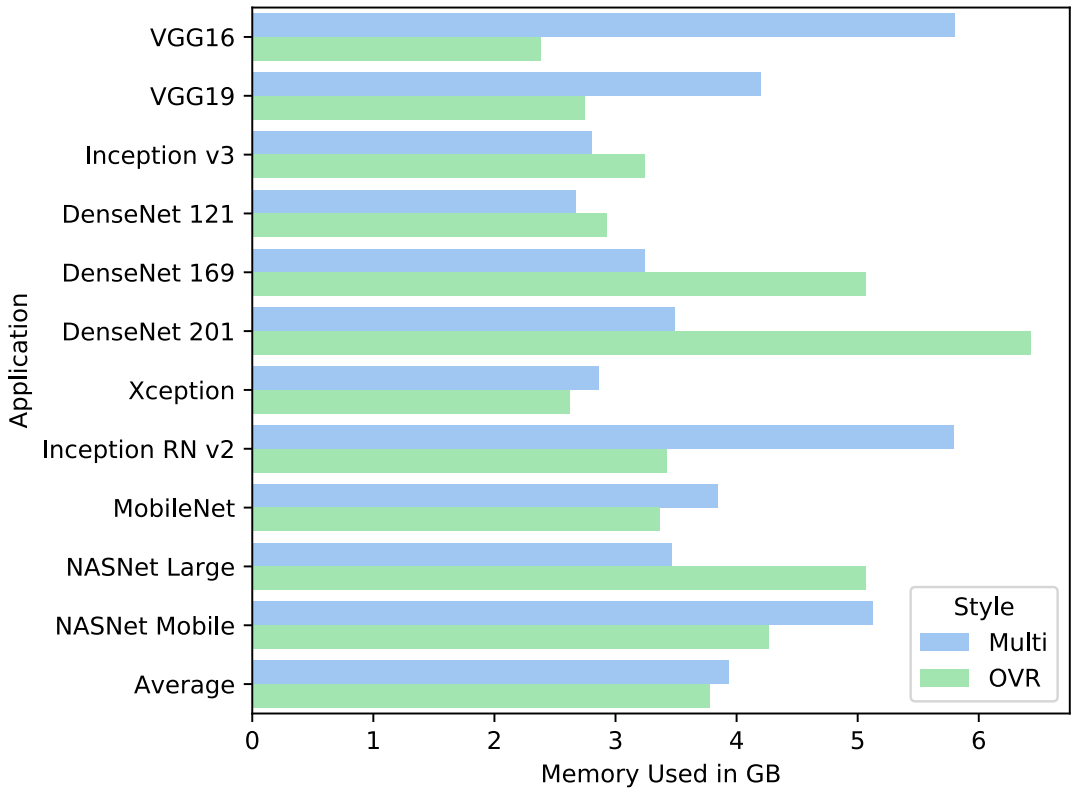
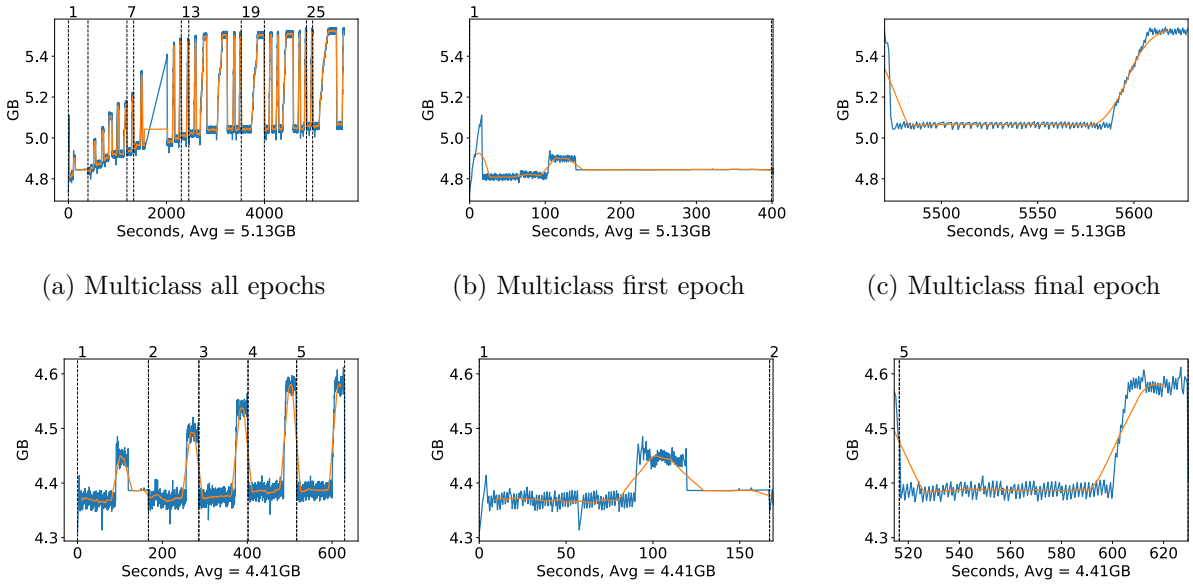


Figure 4.14: Training: Memory Used in GB, including all applications and an average value. For both binary and multiclass styles.

To examine this more closely, we would need to examine how much memory is being allocated by Keras/TensorFlow specifically and profile the memory allocations somehow. In the interest of curiosity, we have included a plot of the memory use during training for NASNet Mobile in figure 4.15. What we can see is, that much like the GPU and CPU usage, the memory usage rises and falls in phases during each epoch. We can also observe, however, that the memory use steadily rises as training progresses, by for example 200MB in the multiclass case. This might be due to some form of history stored by TensorFlow.

We initially suspected this steady rise in memory across epochs might be due to the stored logging of the metrics, but this data is only a few MB for the entire training history, so this seems unlikely.



(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

Figure 4.15: NASNet Mobile Memory usage during training, with a moving window average over 40 measurements overlaid

4.2.3 Time

Another crucial aspect of our research is time. The amount of time it takes to train a network can be important, for example, if one desires to update the network frequently once new classified data becomes available. In general, it is also important because it determines how much energy we are using in total, dependant on the average energy use of the network. There is also reason to believe that the circumstances of training a network put more wear on the components of the system, as temperature fluctuations force the materials of the system to expand and contract. This could mean a premature physical component failure, for example of a capacitor on the GPU board. An example of these temperature fluctuations can be seen in figure 4.16.

In addition to the above factors, having a shorter training time allows us to try more strategies, hyperparameters, dataset splits and other factors which may impact the final classification. For example, in research funding and similar factors might limit the amount of available time. Thus, if one application takes several days to train, while another takes a few minutes, it would be extremely advantageous to select the least time-intensive network.

In figure 4.17 we have the total amount of time taken to train each network style and application. For the OVR method, the total time to train all the binary networks is shown. It is apparent when viewing the plot, that the total time spent training the OVR networks is much higher than for the multiclass network, despite the fact that we limited the number of epochs in the binary case to be 1/8th that of the multiclass case. The likely reason for the increased time use is that the patience for the early stopping function was set to the be the same for the binary and multiclass cases, which means that for the initial training the binary networks were seldom stopped early, while the multiclass networks were. It is an unsurprising outcome, given that the multiclass networks had a limit of 40 epochs for the initial training and the binary networks had a limit of 5.

To confirm this thought, we should be able to see that the amount of time spent on each epoch is roughly similar to the total amount of training time for the binary case, but not similar in the multiclass case. To examine this, we can look at figure 4.18 which features the average epoch time for each application and style. From this figure, we can confirm visually that these do roughly

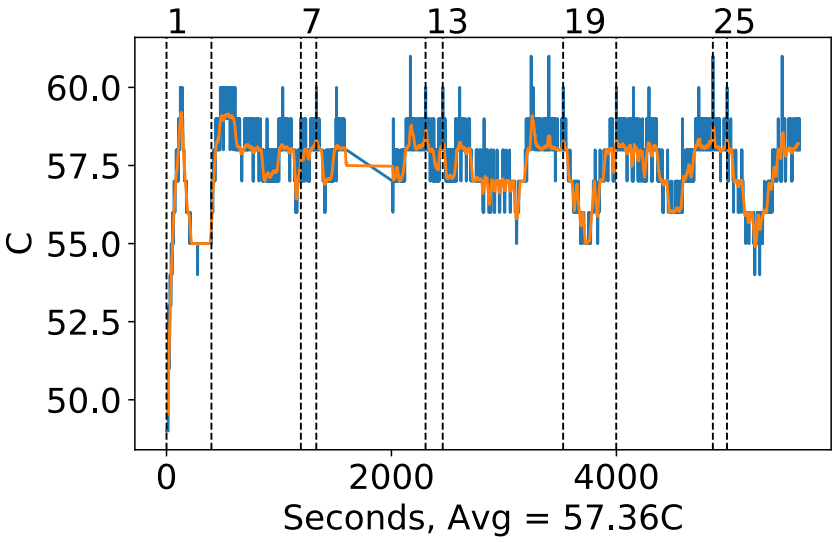


Figure 4.16: NASNet Mobile GPU temperature during training, with a moving window average over 40 measurements overlaid

align, except for VGG19. After consulting with the test data, we confirmed that this inconsistency was due to a mistake in the parameter files, where the number of epochs allowed for VGG19 was set to 100 instead of 25.

We can also see that in the case of the multiclass networks, the total amount of time does not seem to line up with the epoch duration. It is likely that most of these networks were stopped early. Another interesting detail is that there actually is a substantial difference in epoch duration for some of the network styles, and it is not always the same difference. For example, the VGG19 application has longer binary epochs than multiclass epochs, as does the NASNet Large style. On the other hand, the NASNet Mobile has the opposite situation.

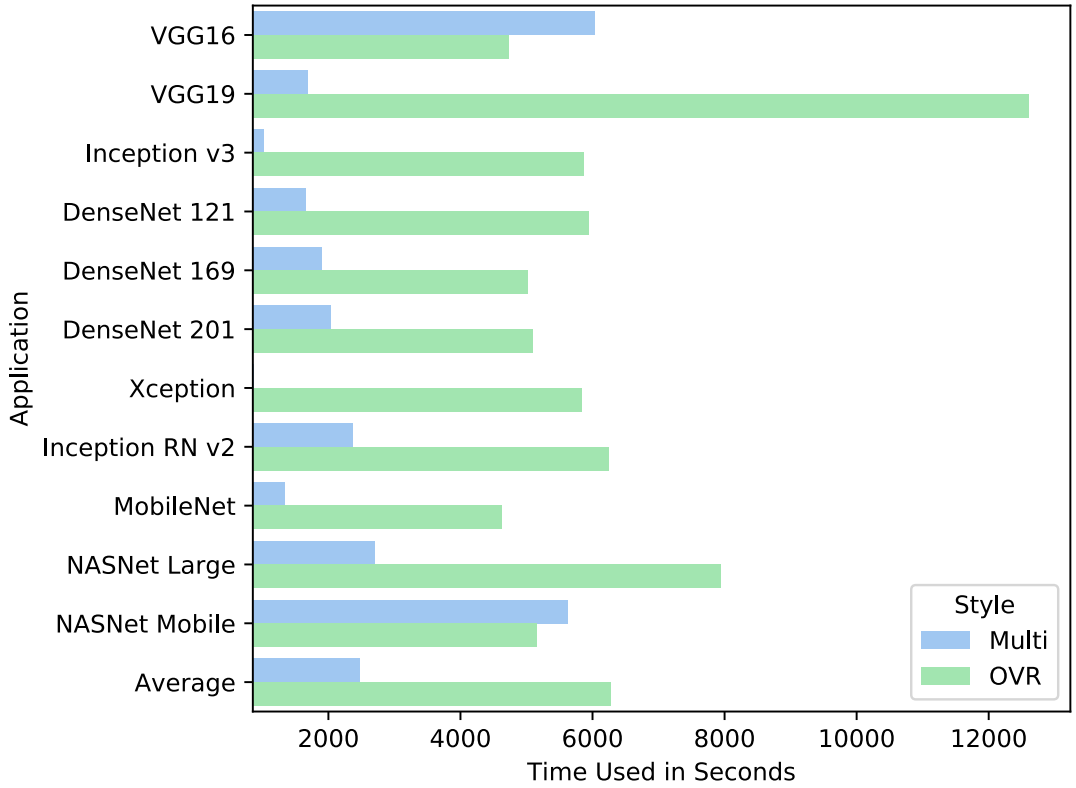


Figure 4.17: Training: Time Used in Seconds, including all applications and an average value. For both binary and multiclass styles.

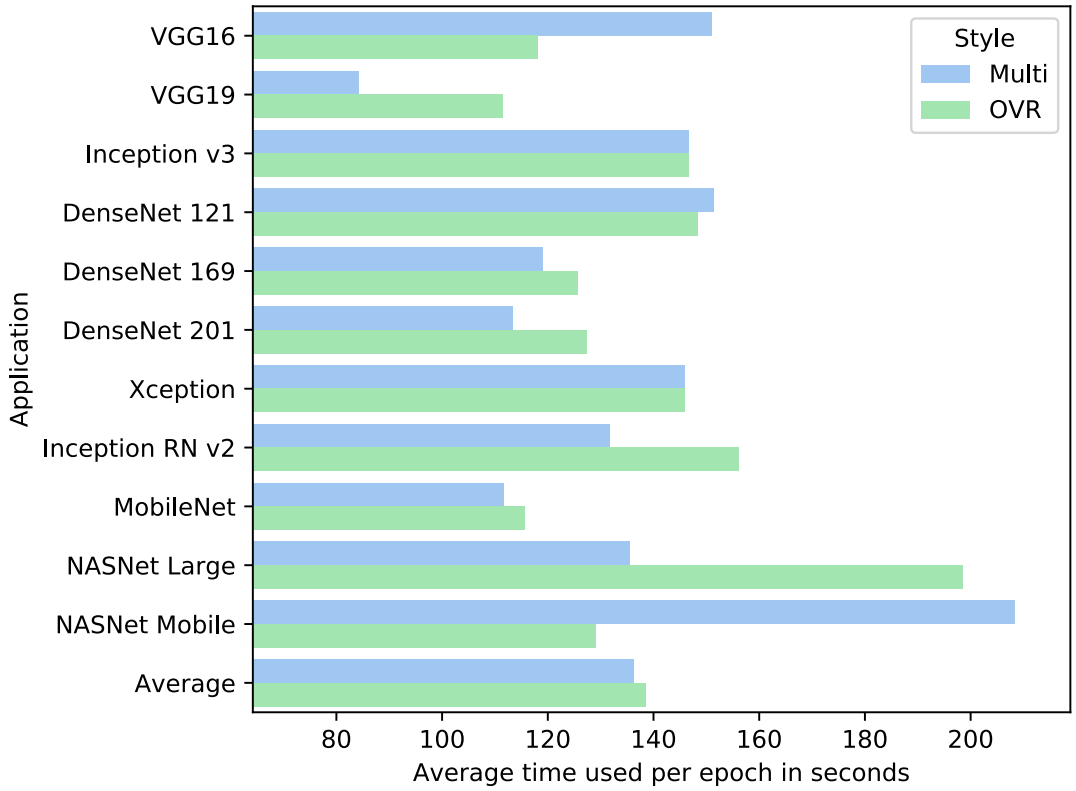
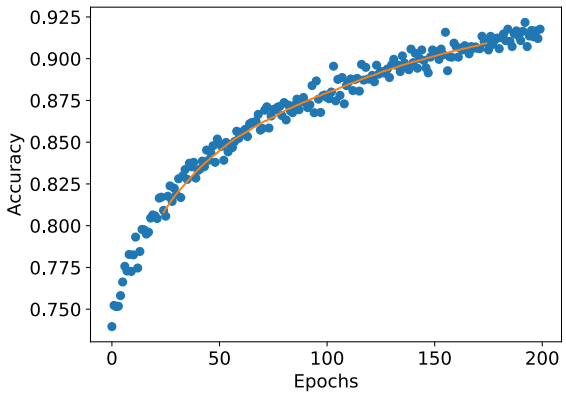


Figure 4.18: Training: Average time used per epoch in seconds, including all applications and an average value. For both binary and multiclass styles.

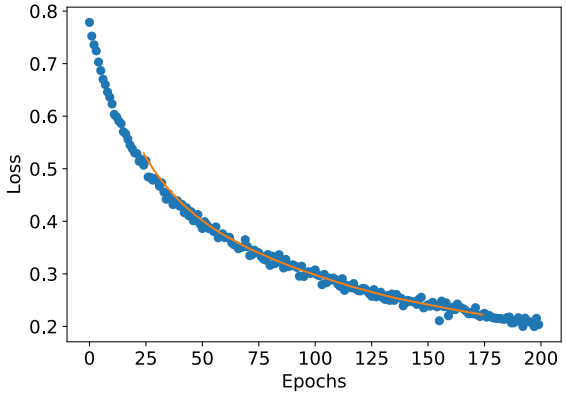
4.3 Fine-tuning

During the fine-tuning phase of each network, many layers are unfrozen and made trainable, and thus many more layers of weights are updated during backpropagation. We might expect all phases of training to take longer and use more resources, and as such, it is the most resource-intensive phase for our networks. It also usually lasts far longer than the initial training phase.

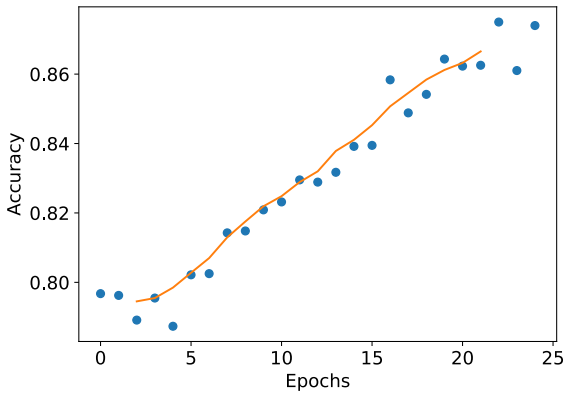
We should first look at what is precisely going on during the fine-tuning phase. To determine this, it might be helpful to look at the accuracy and loss plots of the same network we looked at in the previous section, NASNet Mobile. A plot of the training set accuracy and loss can be seen in figure 4.19.



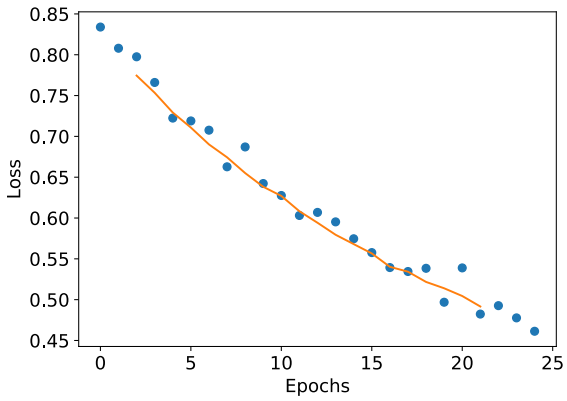
(a) Multiclass network accuracy



(b) Multiclass network loss



(c) Binary network "polyps" accuracy

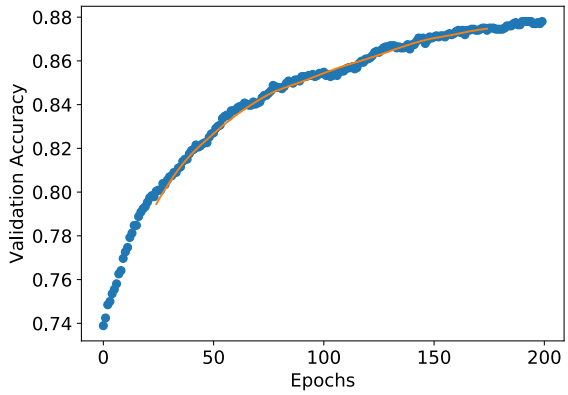


(d) Binary network "polyps" loss

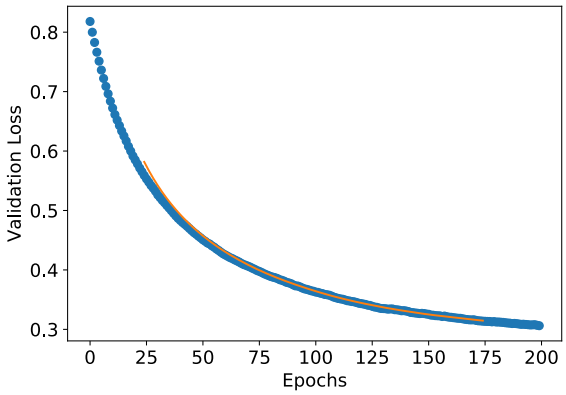
Figure 4.19: NASNet Mobile Fine-tuning classification accuracy and loss history for training data

In this figure, we can see that the training set accuracy and loss resume basically where they left off after the initial training. They then steadily rise and fall, respectively, until our limit of 200 epochs is reached for the multiclass network, and 25 epochs for the binary network. Interestingly this is one of the few applications that was not stopped early in either the multiclass our binary cases. The characteristic of the curve for the multiclass network indicates a good selection of hyperparameters, but not so much for the binary case. It should be mentioned that the polyps class is consistently one of the most difficult for the network to learn, as we will see in the testing section. Thus, the performance might not be as bad as it initially appears, but it also illustrates how the performance perhaps could be improved by individually tuning the hyperparameters for each binary network.

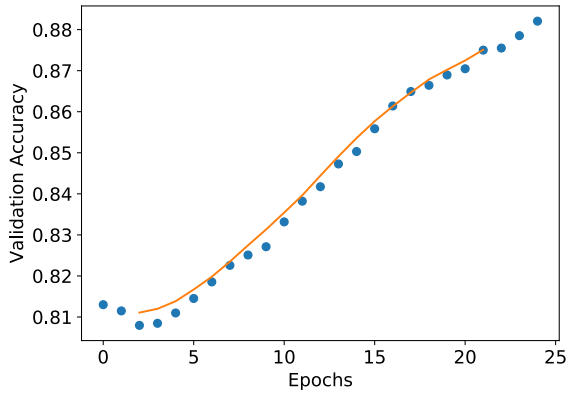
In figure 4.20 we see similar characteristics for the validation data. Notice that the validation accuracy appears to rise much more steadily than the training accuracy did in figure 4.19. Although



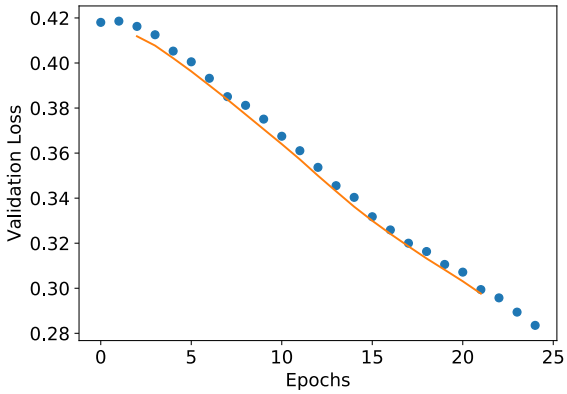
(a) Multiclass network accuracy



(b) Multiclass network loss



(c) Binary network "polyps" accuracy



(d) Binary network "polyps" loss

Figure 4.20: NASNet Mobile Fine tuning classification accuracy and loss history for validation data

it does seem to move around the average overlaid curve slowly, the large jitter from the previous figure is missing. It may be because the optimizer is making more substantial corrections based on the data from the training set, which leads to slight overfitting during each epoch and then corrections in the next epoch. Meanwhile, the validation accuracy rises steadily.

4.3.1 Resource Use

As mentioned previously, the resource use during the fine-tuning phase is important because it is so time-consuming, so we should examine the most significant metrics during this phase as well. In figure 4.21 we can see the average volatile GPU usage during fine-tuning. In general, we can see that on average, the OVR networks now have a slightly higher average GPU usage. However, the difference is small so this result is probably not significant. We can also see that the difference between the binary and multiclass NASNet Large has almost disappeared, whereas for NASNet Mobile the difference is much more drastic. Interestingly, it would seem that the large difference in the resource use due to the number of classes is particular to the NASNet applications. It also might be related to the rather substantial amount of unfrozen layers for these networks. We cannot find any particular reason why this would affect the binary network so much more than the multiclass one.

We examined the raw GPU usage data for NASNet Large and Mobile, and as expected the usage is higher for the binary NASNet Large and Mobile networks during fine-tuning. In addition, we

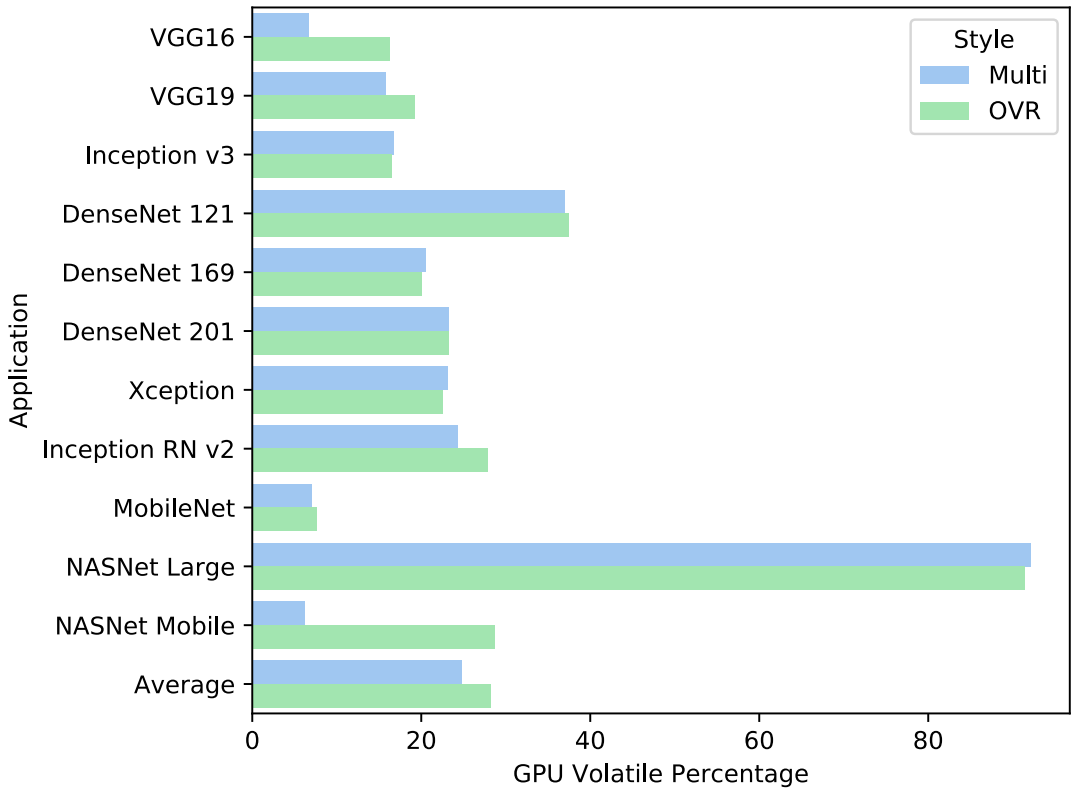


Figure 4.21: Fine-tuning: GPU Volatile Percentage, including all applications and an average value. For both binary and multiclass styles.

examined the GPU power usage for all networks during fine tuning and concluded that as before, it is highly tied to the GPU volatile percentage. We have included the figure A.99 in the appendix which shows these numbers.

Other Hardware

As before, the CPU usage during this phase is important to examine. We present the numbers in figure 4.22. For the most part, the numbers are very similar to those in the initial training, we can see that on average the CPU use has risen by about 5%. This is not entirely unexpected, as the CPU usage would presumably increase when the gradients and variables are updated during each epoch, due to the substantial increase in the number of trainable parameters.

We also examined the system memory usage, but these numbers seemed to be very similar to the initial training, and as such not particularly attractive to examine in detail. The numbers here are included in the appendices A.100. In summary, we can conclude that the resource use during fine tuning has increased, as we initially suspected, although the increase was mostly seen in the case of the CPU use.

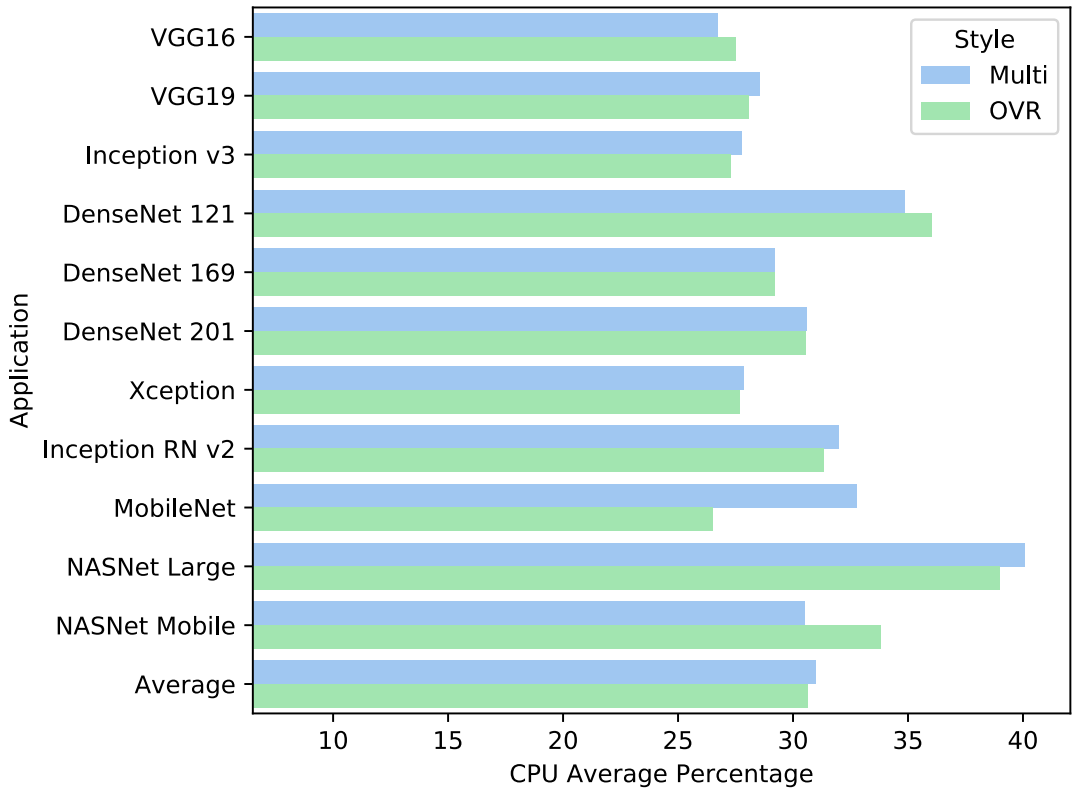


Figure 4.22: Fine-tuning: CPU Average Percentage, including all applications and an average value. For both binary and multiclass styles.

4.3.2 Time

Finally, we should examine the time spent fine-tuning each network style. The total time taken to fine-tune each network style and application can be seen in figure 4.23. It is clear that the time taken is quite substantial for most of the OVR style networks, and a few of the multiclass ones. The average time spent is over 5 hours, which is a substantial amount of time. The NASNet Large OVR networks, for example, took over 20 hours to fine tune, which is a considerable amount of time since the average GPU power draw during this time was 220W.

Although the total amount of time spent training is interesting, it does not tell the whole story and is likely very dependant on the hyperparameter selection, which we have not optimized as much as one would perhaps usually like. In figure 4.24 we can instead observe the amount of time spent per epoch.

These numbers are more interesting for a general use-case, as it shows the difference between fine-tuning our binary networks and the multiclass networks, in a manner which isn't so dependant on hyperparameter selection. Here we can see that, by and large, the difference is not very substantial between the network styles. However, there are two exceptions, namely VGG 16 and NASNet Mobile, where the multiclass networks are immensely more time consuming to fine-tune per epoch than the binary equivalent. This also partially explains the sizeable total time increase for these networks.

The plots for the GPU usage and power draw are included in the appendices A.3. After examining these the usage plots, we determined that the amount of time needed for each epoch slowly increased during fine tuning. We could not find any discernable reason for this, as the operating temperature of the GPU was well within acceptable parameters and thus throttling was eliminated as a cause. This effect could be interesting to examine in future research.

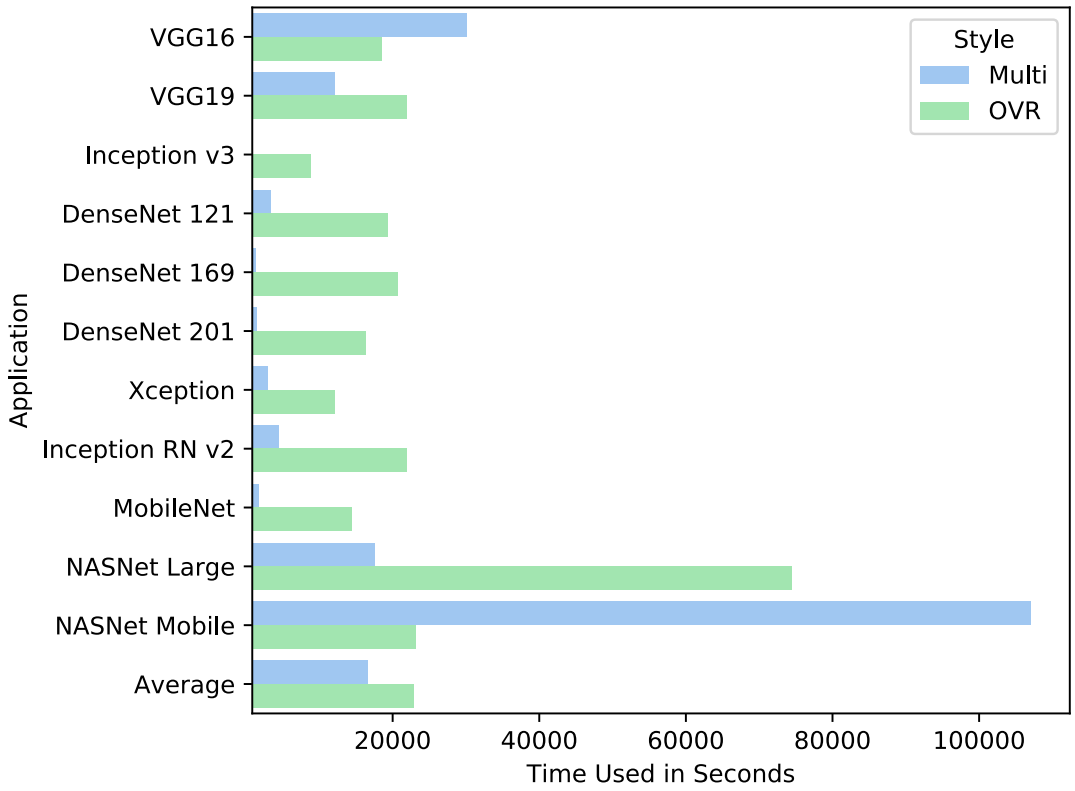


Figure 4.23: Fine-tuning: Time Used in Seconds, including all applications and an average value. For both binary and multiclass styles.

4.3.3 Tradeoffs

During the initial training and fine-tuning, we can see that the most important and immediate trade-off is time spent training. We have seen that the time spent training is often a great deal longer in total for the OVR styles, probably because the OVR styles are less likely to be stopped early during their smaller number of epochs. If the number of epochs for the OVR networks was increased to match the multiclass networks, the difference would probably be much larger, making training multiple binary networks extremely time-consuming if we have many classes.

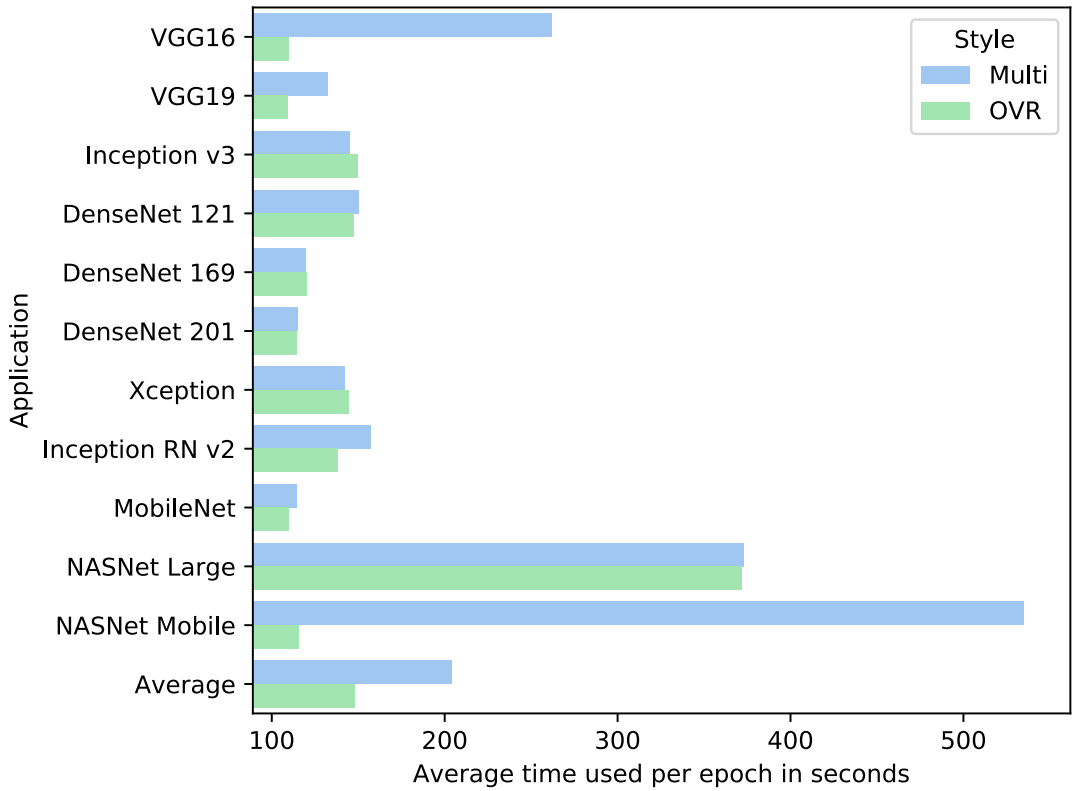


Figure 4.24: Fine-tuning: Average time used per epoch in seconds, including all applications and an average value. For both binary and multiclass styles.

4.4 Testing

When we tested of our neural nets on the test set, we focused on examining several crucial factors which could determine how well each network style and application is to any given final use. Classification accuracy is of paramount importance and is often the focus of research in this field. However the hardware resources used, and classification speed, are also essential in many cases. In the following section, we will look at the final performance of the fully trained networks in all respects: resource use, to speed, and accuracy. Hopefully, it will provide us with a satisfactory overview of what we can expect from these networks in real-world situations, naturally with some limitations. We ran a series of 10 tests for each network style and application.

4.4.1 Resource use

The final phase of use for a neural network is perhaps the most important when it comes to resource use. During actual classification, the network could be expected to run continuously and, in some circumstances, in real time. This means that network with excessive resource use will most likely be unable to classify images quickly enough on low power or older hardware. Specifically, in our case, we are interested in determining how well Tensorflow can delegate resources so that the OVR networks can classify images in parallel. The networks are not connected together in one large network. Hence several model instances must be run on the GPU at the same time, with each model generating its probabilities during testing.

This might be problematic if the interpreter or scheduler is unable to process these requests in parallel. However, it might also mean that when one network is not actively using resources, they may be used by another. In the following sections, we should look for signs that the binary and

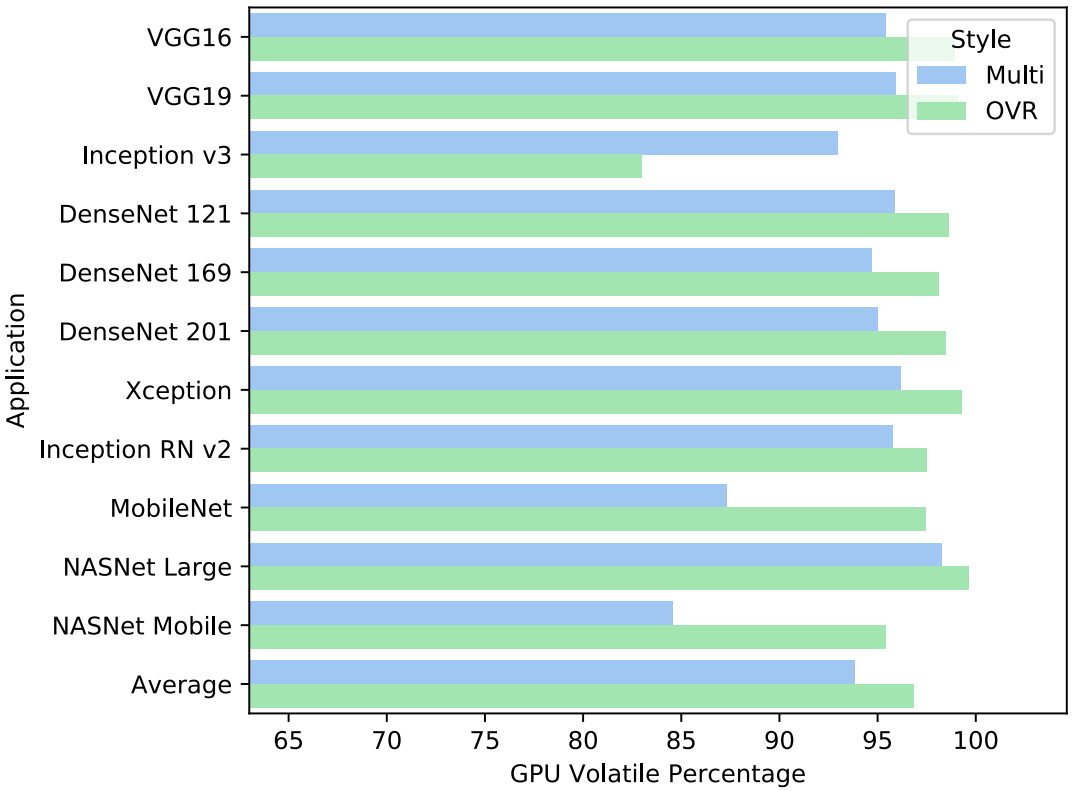


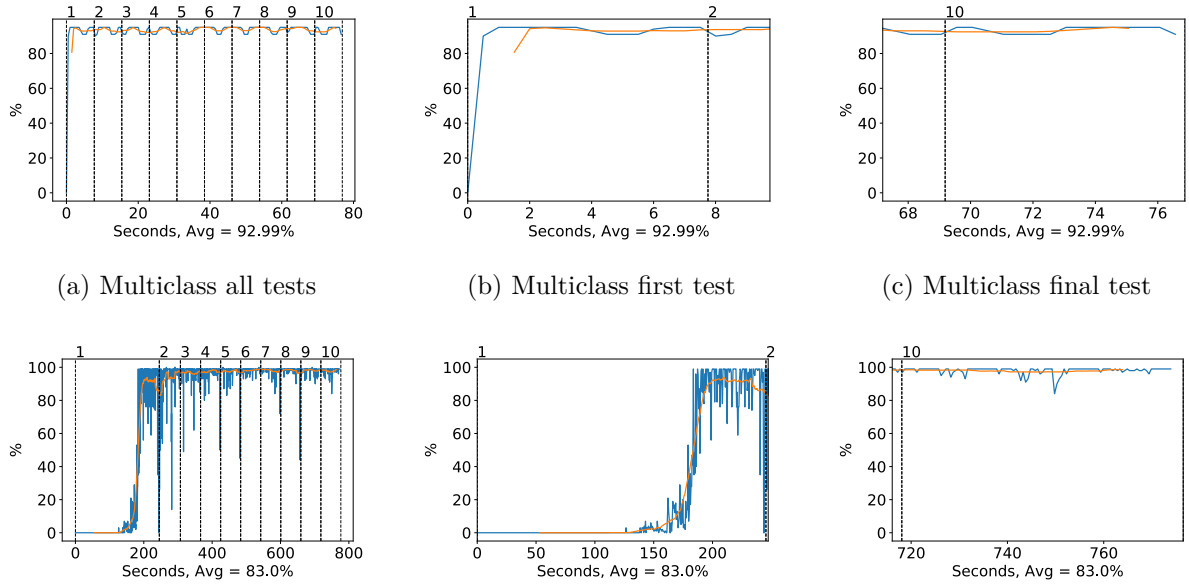
Figure 4.25: Testing: GPU Volatile Percentage, including all applications and an average value. For both binary and multiclass styles.

multiclass networks are efficiently using the hardware during testing. We expect that the resource use will be higher in the OVR case, but it will be interesting to determine what the difference is, and of course how much resources are used by the multiclass base network in test mode.

In figure 4.25, we can see the overall average GPU usage during testing for all applications and both network styles. The first thing that strikes us here is how even the use is across the board, where nearly all of the networks have upwards of 90% volatile GPU usage. This indicated to us that Tensorflow might be allocating resources with reasonable efficiency for both the single multiclass network case and our OVR parallel case. For most of the network styles, the reported usage is within a few percents of each other, with the OVR networks using slightly more GPU resources on average. There are a few notable exceptions though. The multiclass Inception v3 network has nearly 10% higher usage during testing than the OVR equivalent, which is odd and unexpected. It is the only network style where the multiclass usage is reported as higher.

In figure 4.26 we can see the volatile usage for Inception v3 in both styles with the test number overlaid. The culprit of the strange results becomes apparent: the binary network has a considerable period of relative inactivity on the first test. At first, we suspected that it was an anomaly of the particular test, but after reviewing the results of other tests for this style, we see the same delay across all tests, although with varying duration. Ultimately we decided to include this particular, albeit extreme, example as this is a real curiosity with using this application. We did not observe similar results with the other applications.

Otherwise, our two "high-efficiency" applications, NASNet Mobile and MobileNet both had a large increase in volatile usage during testing with the OVR network. The applications are designed to be efficient on low power hardware, and the findings thus make sense. The applications were very similar to each other in volatile usage, with only a few percent difference. We observe the actual recorded usage for NASNet Mobile in figure A.228.



(a) Multiclass all tests (b) Multiclass first test (c) Multiclass final test

(d) Binary all tests (e) Binary first test (f) Binary final test

Figure 4.26: Inception v3 GPU volatile usage during testing, with a moving window average over 40 measurements overlaid

The volatile usage is relatively stable during the entire test phase, but we can see that the stable usage is higher for the OVR case, approaching 100% use. It seems that we might be approaching the limit of what is possible given our hardware, which is a good sign as it implies that Tensorflow is efficiently allocating our resources. Here we can also see that all tests take roughly the same amount of time, and we do not see the same strange delay at the beginning of the testing. In this regard, this characteristic is similar to our other results.

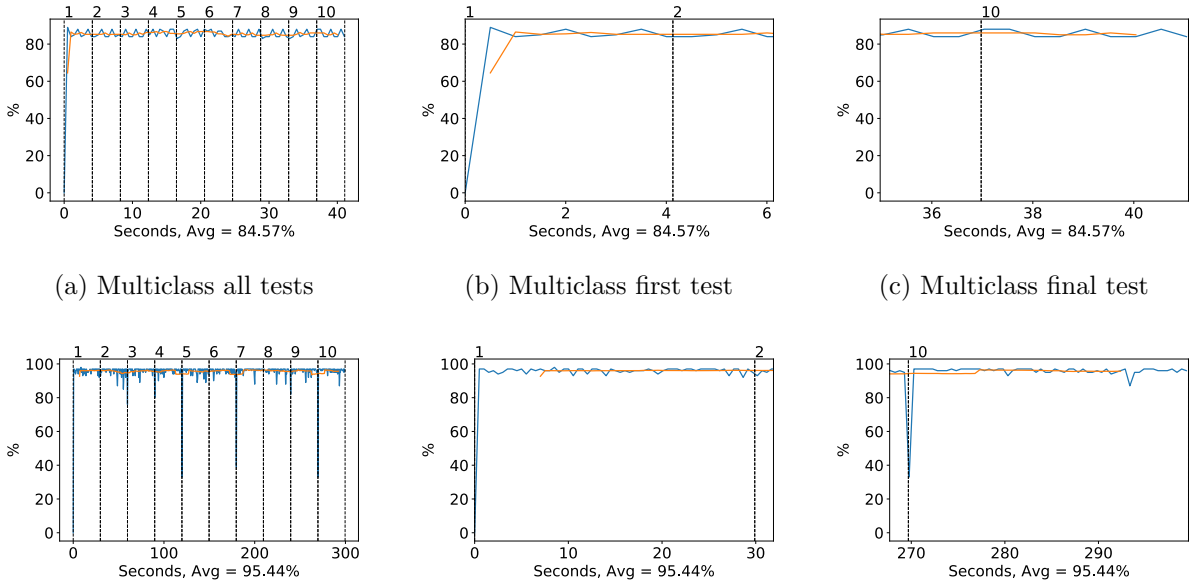
However, this might not tell the whole story. In our previous results we found that during training and fine-tuning, the amount of power used by the GPU was roughly reflective of the volatile percentage. However, if we look at figure 4.28, and compare it to the volatile use, we can see that there is, in fact, a more substantial difference than we might have expected. Our high-efficiency applications appear to use significantly less power than the other applications, even in the OVR case where the volatile usage approached 100% for both styles, the power use is about 30W shy of the TDP for this card.

The application with the highest power draw was the Xception OVR style. It would be interesting to compare the use of this application with the use of our NASNet Mobile, for example, which has within 5% of the volatile GPU usage in the OVR configuration, and yet over 40W more power usage (15% of the total TDP, or 17.5% of the above-baseline TDP[14].)

Figures 4.29 and 4.30 allow us to observe the difference here in more detail. In the Xception plots, it is immediately clear that the GPU wattage is greater than the maximum TDP specified by Nvidia for this card. This might mean that our specific card is drawing more power than the reference cards, or that the cards have this possibility in general. At any rate, we can see that the Xception OVR style is peaking at over 300 watts of power consumption during testing of the style.

In comparison, the NASNet Mobile power draw does not substantially exceed 200W, which is interesting, because it tells us that over the course of testing there might be as much as 100W of difference between these two applications in power consumption. The difference is not inconsequential, as it means that in continuous operation we could cut our power use by 20-30% by

[14]We are using the term "above-baseline TDP" here to refer to the amount of W difference between the minimum recorded consumption (20W), to the maximum TDP of the card.



(a) Multiclass all tests (b) Multiclass first test (c) Multiclass final test

(d) Binary all tests (e) Binary first test (f) Binary final test

Figure 4.27: NASNet Mobile GPU volatile usage during testing, with a moving window average over 40 measurements overlaid

switching from Xception to NASNet Mobile, given that the more efficient network has an accuracy that makes the switch justifiable.

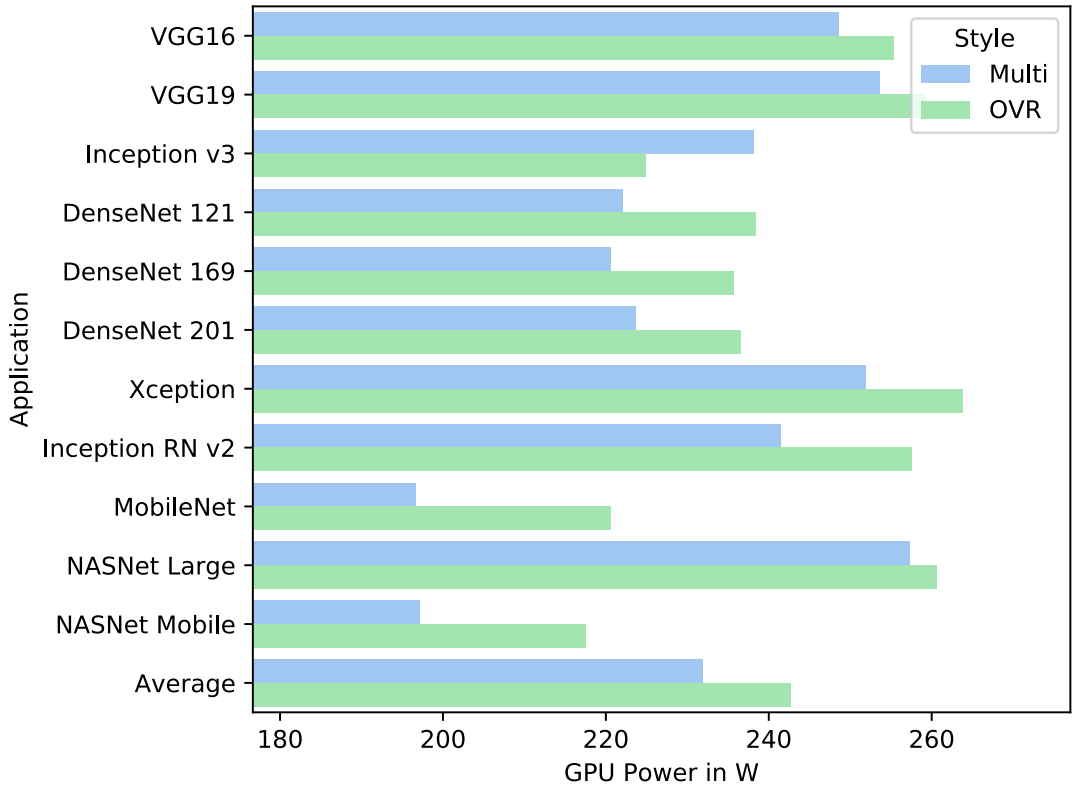
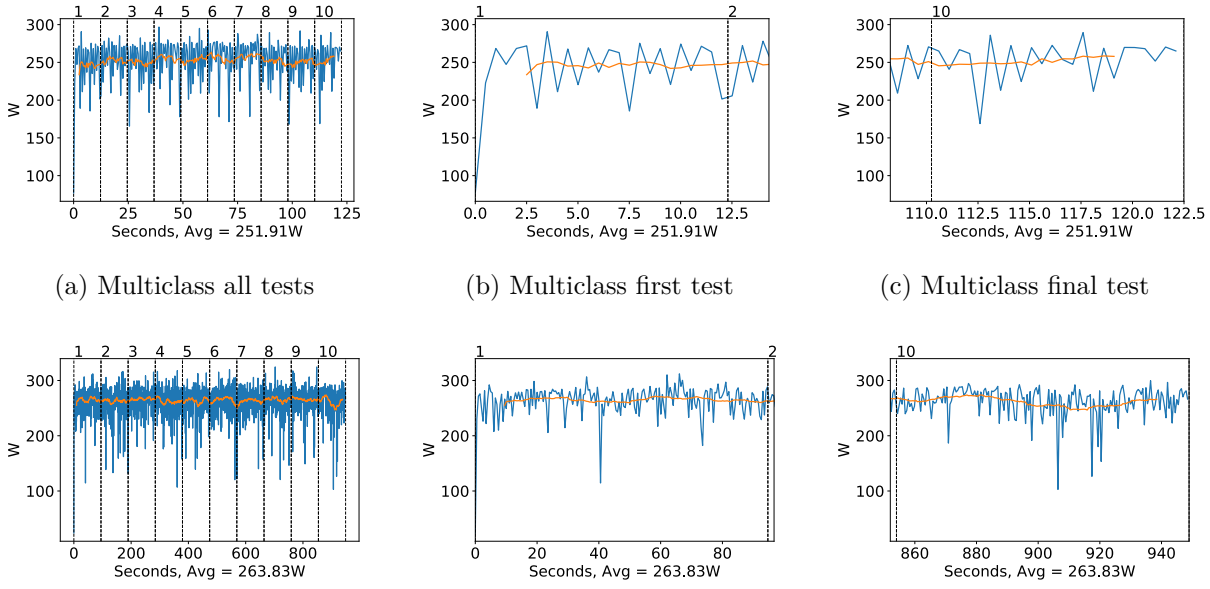


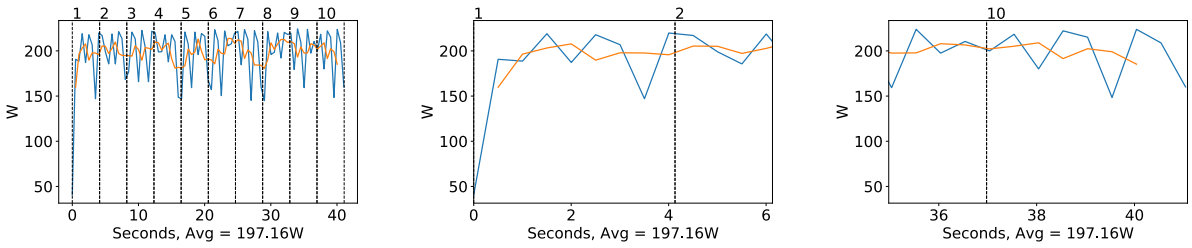
Figure 4.28: Testing: GPU Power in W, including all applications and an average value. For both binary and multiclass styles.



(a) Multiclass all tests (b) Multiclass first test (c) Multiclass final test

(d) Binary all tests (e) Binary first test (f) Binary final test

Figure 4.29: Xception GPU power usage i W during testing, with a moving window average over 40 measurements overlaid



(a) Multiclass all tests (b) Multiclass first test (c) Multiclass final test

(d) Binary all tests (e) Binary first test (f) Binary final test

Figure 4.30: NASNet Mobile GPU power usage i W during testing, with a moving window average over 40 measurements overlaid

CPU use is another metric of interest for us. Although we would expect the CPU use for most of the multiclass applications to be relatively similar, it would ostensibly have more work to do in the OVR case, as it now has eight networks running in eight threads and has to manage memory and data transfers between all of them and the GPU. Also, 8 ImageDataGenerators are working simultaneously on the test data, instead of just one. We can see the measured usage in figure 4.31. It is quite remarkable how similar the usage is for each of the multiclass applications, it appears to be almost the same.

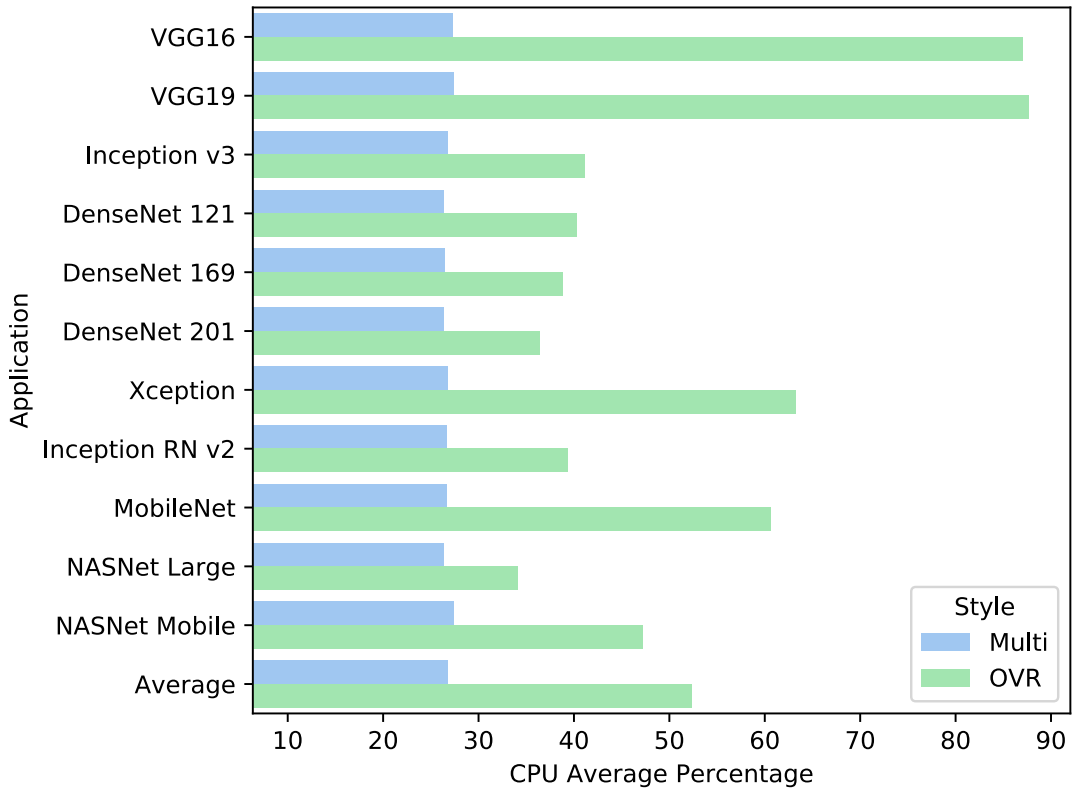


Figure 4.31: Testing: CPU Average Percentage, including all applications and an average value. For both binary and multiclass styles.

However, the OVR CPU usage is a different story. As we expected the usage is higher for all applications in the OVR case, however, there is a considerable variation in just how much higher. For example, the VGG applications seem to use almost three times as much CPU resources in the OVR configuration than in the multiclass configuration. This is the most extreme case, but it does illustrate that the difference can be quite extreme here. The difference in CPU use should be a consideration the planning process if one is going to consider an OVR approach.

Another metric of interest during testing is the amount of system memory that is used, and it is presented in figure 4.32. It would seem from the figure that the multiclass configurations all use a similar amount of system memory and that the OVR configurations can and do use significantly more, often more than double the amount of the multiclass equivalent. This is not entirely unexpected but should be considered if one is planning to apply OVR configuration in practice.

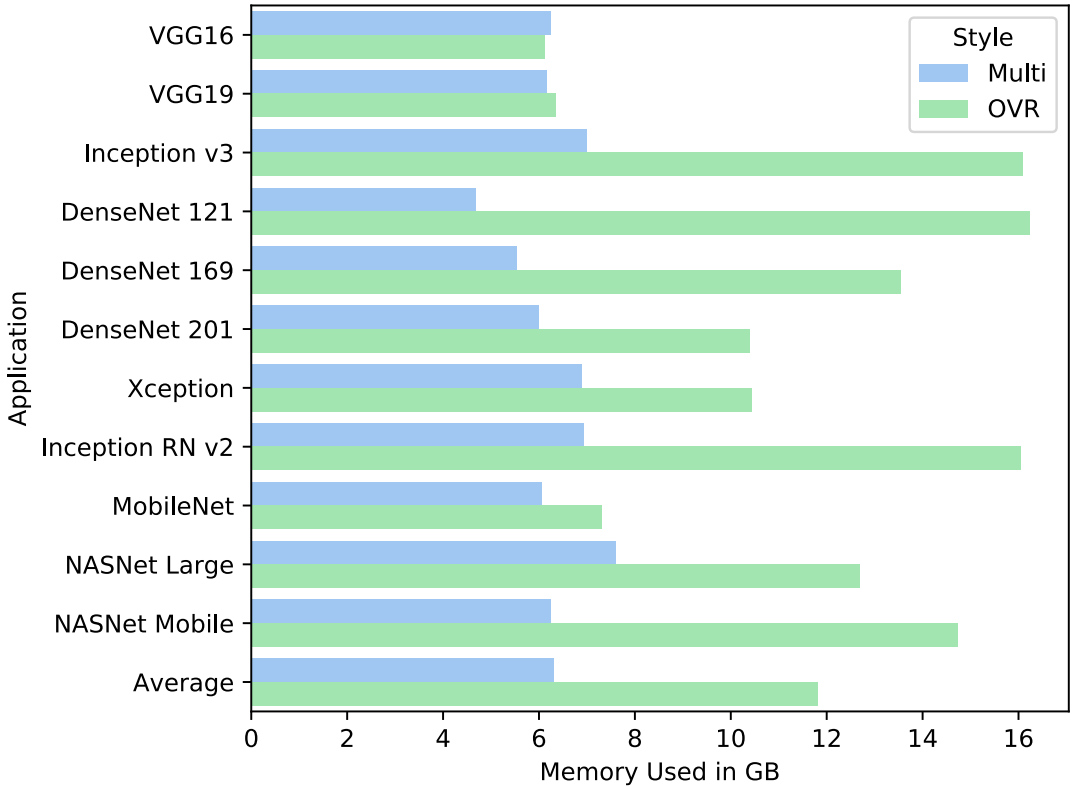


Figure 4.32: Testing: Memory Used in GB, including all applications and an average value. For both binary and multiclass styles.

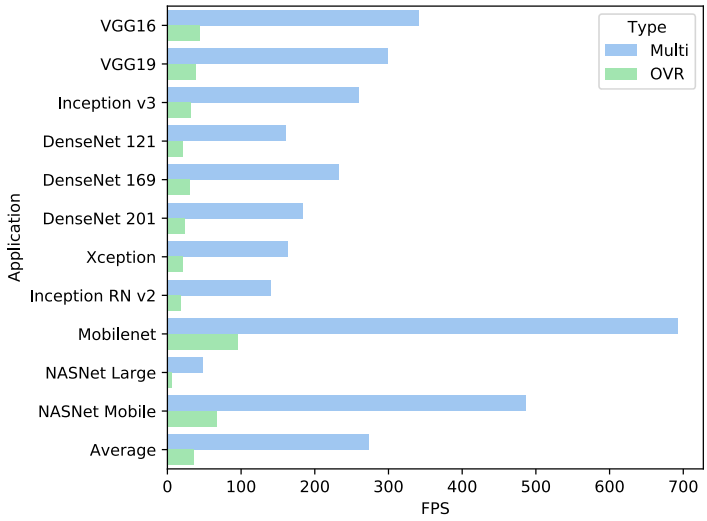
4.5 Classification

In the previous sections we have primarily examined the resource use during all phases of use for the different networks. The findings so far have given us an idea of what we might be able to expect for the classification performance. In this section we will look at how all the networks performed during classification. Of particular importance is how quickly each style and application was able to process and generate predictions on the frames. In addition, the classification accuracy will give us an idea of what effect the OVR set up has had on overall accuracy, even when the hyperparameters are not optimized for the individual binary networks.

4.5.1 Performance

To begin, we will examine the frame throughput, or FPS, of each network. We run the 2000 frames in our test set, which have not been used for training or validation, through each network's predict function. In the OVR case, it means that the frames are run through the eight networks in parallel. We observed a slight variance in FPS and decided to run the test ten times sequentially to correct for it. Effectively, we are testing our network on processing 20,000 frames quickly, which is the equivalent of ≈ 11.1 minutes of video with a framerate of 30 FPS. We did, however, log the FPS achieved for each 2000 frame test. The summary of these tests can be seen in figure 4.33.

Immediately, we see that the OVR network style has a notable impact on frame throughput. Every application is consistently faster in the multiclass configuration, but it is not surprising given the resource allocation characteristics we observed in the previous chapters. We can see that the slowdown experienced, on average, is nearly proportional to the number of classes. We interpret this to mean that our binary networks are saturating what little resources were left over from a



(a) Bar plot of FPS on the test set for both OVR and multiclass network styles, by application

Application	Multi	OVR	SDF
VGG16	341.62	44.49	7.68x
VGG19	299.49	38.52	7.77x
Inception v3	260.18	31.33	8.30x
DenseNet 121	160.89	20.67	7.78x
DenseNet 169	232.91	29.97	7.77x
DenseNet 201	183.31	23.59	7.77x
Xception	163.27	21.08	7.75x
Inception RN v2	139.86	17.72	7.89x
MobileNet	692.59	95.79	7.23x
NASNet Large	47.56	6.03	7.89x
NASNet Mobile	486.83	66.75	7.29x
Average	273.50	35.99	7.60x

(b) Raw FPS averages.

Figure 4.33: FPS Test summary for all networks, including all 10 tests and average value. SDF is the Slowdown Factor, i.e., how many times slower the OVR network is than the multiclass equivalent. From the averages we can see that using single multiclass networks is approximately 7.6 times faster than using the OVR style.

single multiclass network, and all the additional resources required are just causing the performance to decrease proportionally. This means that for a dataset with more classes, for example, ImageNet with over 20,000 classes, it would not be an appropriate strategy as the time needed to test would be too long.

Two things are interesting to note here, however. As we can see, the SDF is higher for Inception v3. This is likely because of the substantial delay for the first test that we showed in the resource use section 4.2.2. The delay reduces the average FPS rather dramatically for this application in the OVR style. Besides, we note that the SDF appears to be lessened for the two high-efficiency applications (MobileNet and NASNet Mobile). It is an unsurprising outcome, but it shows the effect of their initial resource usage not being as high as the other applications in the multiclass style. We can also see that our fastest networks, both MobileNet, experienced the least amount of relative slowdown of all our applications. This would appear to confirm that if a network is using fewer resources in a typical configuration, the consequences of using it in an OVR configuration will be lessened, at least from a performance standpoint.

We can also see from figure 4.33 that there is a rather significant difference in the applications when it comes to frame throughput. Our fastest application, MobileNet, is over ten times faster than our slowest, NASNet Large. In the tools and implementation section, we detailed what our goals were for the achieved FPS, to enable real-time performance. We set a minimum FPS of 30, and a goal of 60. From the numbers, we can see that all but one of our applications achieved 60 FPS or higher in the multiclass configuration, and even that one achieved at least 30 FPS.

However, considering OVR, we can see that the numbers are not as excellent. Only two of our chosen applications were able to achieve above 60 FPS; unsurprisingly these were the two high-efficiency applications. A further three applications were able to achieve over 30 FPS, which means that 6 of our 11 applications fell below our minimum 30 FPS goal, some by quite a substantial amount. Thus, the current networks would likely not be suitable for real-time operation. The most substantial feasibility decrease NASNet Large, which ended up with a rather paltry 6 FPS.

On the other hand, MobileNet is still able to achieve 95 FPS in the OVR configuration, which is surprisingly useful and close to exactly twice as fast as the slowest multiclass classifier. To summarize, there is a substantial performance penalty associated with choosing an OVR style. These penalties are so substantial that they will render some of our networks unable to classify in real-time. The choice of application is also critical, and a good takeaway from this section is not to choose an application which is more complex and resource hungry than what we require.

Speed and Network Complexity

To determine what effect network complexity has on performance, we can examine the same table we presented in the tools and implementation section, but with the FPS values from our tests included. In table 4.1, the basic network complexity details, and the achieved FPS are presented. However, the table cannot tell the whole story because it cannot include the issue of exactly what layer types are chosen in the application and how they are connected. However, it is still interesting to examine the relationships between the different specifications and their performance.

Network	Parameters (M)	Depth	Layers	Size	FPS (Multi)	FPS (OVR)
VGG16	138.35 [18]	21	21	115.6MB	341.62	44.49
VGG19	143.6 [18]	24	24	155.7MB	299.49	38.52
Inception v3	23.85 [18]	159	313	177.1MB	260.18	31.33
DenseNet 121	8 [18]	121	428	81.3MB	160.89	20.67
DenseNet 169	14.3 [18]	169	596	69MB	232.91	29.97
DenseNet 201	20.242 [18]	201	708	94.4MB	183.31	23.59
Xception	22.9 [18]	126	134	121.9MB	163.27	21.08
Inception RN v2	55.87 [18]	572	782	261.3MB	139.86	17.72
MobileNet	4.25 [18]	88	98	32.4MB	692.59	95.79
NASNet Large	88.9 [23]	768	1021	1002MB	47.56	6.03
NASNet Mobile	5.3 [23]	384	751	54.8MB	486.83	66.75

Table 4.1: A list of the chosen applications with specifications, depth and layers adjusted to match our actual implementation, in addition ot the achieved FPS in our tests. Size is based on actual trained models.

We can, for example, see that the VGG applications have an enormous amount of trainable parameters, by far the most of the applications we tested. However, they are also the two shallowest networks. Both VGG networks achieved above average performance (>273.5 FPS). By comparison, the DenseNet applications have far fewer trainable parameters but achieve below average performance. The two fastest networks also have the fewest trainable parameters, but it seems evident that the number of trainable parameters does not tell the whole story of performance.

Network depth does not seem to affect the achieved FPS either. Intable 4.1 we find that NASNet mobile is the third deepest network, but yet the second fastest. Conversely, DenseNet 121 is the fourth shallowest network and yet also the third slowest. In fact, after attempting to do some regression analysis on the test runs we had for each application, none of these specifications seemed to correlate with the total performance. It is more likely that the types of layers, blocks, and connections have an effect on the total performance. In future research, a more in-depth analysis of the relationship should be carried out.

4.5.2 Accuracy

While the frame throughput is important, it is of little consequence if the resulting classification accuracy does not meet our expectations. Thus, we need to determine how well, or poorly,

our networks were able to classify the test data. Using the metrics detailed in the tools and implementation section 3.4.1, we should be able to attain a satisfactory summary of the accuracy provided by both the multiclass and OVR network styles, as well as each application.

Network style	FN	FP	TN	TP	F_1	ACC	MCC	PREC	REC	SPEC
Multi, VGG16	35.38	35.38	1714.62	214.62	0.86	0.96	0.84	0.86	0.86	0.98
Multi, VGG19	35.50	35.50	1714.50	214.50	0.86	0.96	0.84	0.86	0.86	0.98
Multi, Inception v3	81.50	81.50	1668.50	168.50	0.62	0.92	0.63	0.77	0.67	0.95
Multi, DenseNet 121	113.50	113.50	1636.50	136.50	0.50	0.89	0.52	0.76	0.55	0.94
Multi, DenseNet 169	111.88	111.88	1638.12	138.12	0.51	0.89	0.51	0.69	0.55	0.94
Multi, DenseNet 201	148.12	148.12	1601.88	101.88	0.34	0.85	0.35	0.66	0.41	0.92
Multi, Xception	82.12	82.12	1667.88	167.88	0.65	0.92	0.63	0.72	0.67	0.95
Multi, Inception RN v2	79.62	79.62	1670.38	170.38	0.66	0.92	0.64	0.72	0.68	0.95
Multi, MobileNet	98.88	98.88	1651.12	151.12	0.56	0.90	0.55	0.74	0.60	0.94
Multi, NASNet Large	42.50	42.50	1707.50	207.50	0.83	0.96	0.81	0.83	0.83	0.98
Multi, NASNet Mobile	31.38	31.38	1718.62	218.62	0.87	0.97	0.86	0.88	0.87	0.98
Average	78.22	78.22	1671.78	171.78	0.66	0.92	0.65	0.77	0.69	0.96
OVR, VGG16	46.62	46.62	1703.38	203.38	0.81	0.95	0.79	0.82	0.81	0.97
OVR, VGG19	44.38	44.38	1705.62	205.62	0.82	0.96	0.80	0.83	0.82	0.97
OVR, Inception v3	58.50	58.50	1691.50	191.50	0.76	0.94	0.74	0.79	0.77	0.97
OVR, DenseNet 121	109.50	109.50	1640.50	140.50	0.53	0.89	0.53	0.69	0.56	0.94
OVR, DenseNet 169	94.12	94.12	1655.88	155.88	0.60	0.91	0.59	0.71	0.62	0.95
OVR, DenseNet 201	146.25	146.25	1603.75	103.75	0.39	0.85	0.39	0.65	0.41	0.92
OVR, Xception	91.12	91.12	1658.88	158.88	0.65	0.91	0.63	0.77	0.64	0.95
OVR, Inception RN v2	64.12	64.12	1685.88	185.88	0.74	0.94	0.72	0.79	0.74	0.96
OVR, MobileNet	45.62	45.62	1704.38	204.38	0.81	0.95	0.80	0.86	0.82	0.97
OVR, NASNet Large	33.00	33.00	1717.00	217.00	0.87	0.97	0.85	0.87	0.87	0.98
OVR, NASNet Mobile	51.25	51.25	1698.75	198.75	0.79	0.95	0.77	0.80	0.80	0.97
Average	71.32	71.32	1678.68	178.68	0.71	0.93	0.69	0.78	0.71	0.96
Average Difference	-6.90	-6.90	+6.90	+6.90	+0.05	+0.01	+0.04	+0.01	+0.03	0.00

Table 4.2: Metric averages for all network styles and applications, including all metrics.

In table 4.2 we have created an extensive summary of how well our networks were able to classify the data. In the table, we have highlighted the best results achieved for each metric, in both the multiclass and OVR styles. Of immediate interest here is that the best performing multiclass application is not the best performing OVR application. It is also interesting to note that the best performing OVR application is very similar to the best multiclass in classification performance, with only 1 TP and 1 TN difference between them on average. We can also see, with the help of the average difference fields, that the OVR classifiers perform slightly better than multiclass classifiers in total.

The table makes it difficult to see what the change was for each metric for each application, so it would be helpful to have a table with only the differences from multiclass to OVR, and it is provided in table 4.3. The new table makes the differences between OVR and multiclass applications more apparent. For example, we can see that both VGG applications performed worse in the OVR style, and by approximately the same amount. The Inception applications performed better in OVR, as did NASNet Large. Xception performed slightly worse, and NASNet Mobile performed significantly worse, showing the most significant detriment to accuracy from switching to the OVR style.

The DenseNet styles performed better on average with OVR, except when it comes to precision in the case of 121 and 201, where they performed worse. This seems odd and is discussed further below. Finally, we can see that MobileNet has had a massive improvement in performance, going from being one of the worst classifiers to one of the better classifiers, by switching from single-

network multiclass to OVR.

Network style	FN	FP	TN	TP	F_1	ACC	MCC	PREC	REC	SPEC
VGG 16	+11.25	+11.25	-11.25	-11.25	-0.05	-0.01	-0.05	-0.05	-0.05	-0.01
VGG 19	+8.88	+8.88	-8.88	-8.88	-0.04	-0.01	-0.04	-0.03	-0.04	-0.01
Inception v3	-23.00	-23.00	+23.00	+23.00	+0.14	+0.02	+0.11	+0.02	+0.09	+0.01
DenseNet 121	-4.00	-4.00	+4.00	+4.00	+0.03	0.00	+0.01	-0.07	+0.02	0.00
DenseNet 169	-17.75	-17.75	+17.75	+17.75	+0.09	+0.02	+0.08	+0.03	+0.07	+0.01
DenseNet 201	-1.88	-1.88	+1.88	+1.88	+0.05	0.00	+0.04	-0.02	+0.01	0.00
Xception	+9.00	+9.00	-9.00	-9.00	-0.00	-0.01	0.00	+0.05	-0.04	-0.01
Inception RN v2	-15.50	-15.50	+15.50	+15.50	+0.08	+0.02	+0.08	+0.07	+0.06	+0.01
MobileNet	-53.25	-53.25	+53.25	+53.25	+0.25	+0.05	+0.25	+0.11	+0.21	+0.03
NASNet Large	-9.50	-9.50	+9.50	+9.50	+0.04	+0.01	+0.04	+0.03	+0.04	+0.01
NASNet Mobile	+19.88	+19.88	-19.88	-19.88	-0.08	-0.02	-0.09	-0.08	-0.08	-0.01
Average Difference	-6.90	-6.90	+6.90	+6.90	+0.05	+0.01	+0.04	+0.01	+0.03	0.00
Average Difference (%)	-8.82	-8.82	+0.41	+4.02	+7.02	+0.79	+5.99	+1.06	+4.11	+0.38
Median Difference (%)	-28.22	-28.22	+1.38	+13.65	+16.92	+2.17	+17.46	+3.95	+14.93	+2.11

Table 4.3: Metric average differences from multiclass to OVR for all network styles and applications, including all metrics.

Specific cases

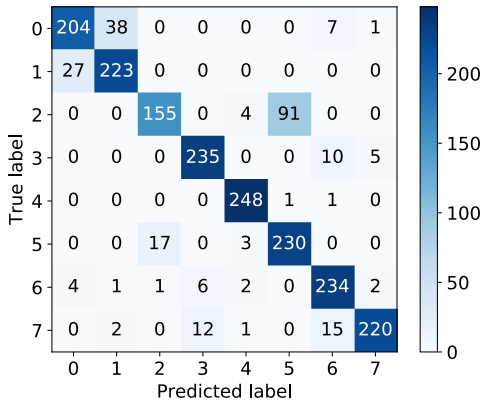
To attain a better understanding of why the differences behave as they do, we should examine a few of the mentioned cases more closely. Specifically, we should have a closer look at a case where the OVR performed worse and one where it performed better.

As we can see, NASNet Mobile had the worst result of using the OVR style. To begin, we can examine the raw figures for each class, which are seen in table 4.4 Reviewing the table, it becomes clear what is wrong: every class is classified less accurately, almost irrespectively of which metric we choose. There are a few small exceptions, such as class 5 having higher precision in the OVR style, and class 2 having higher recall.

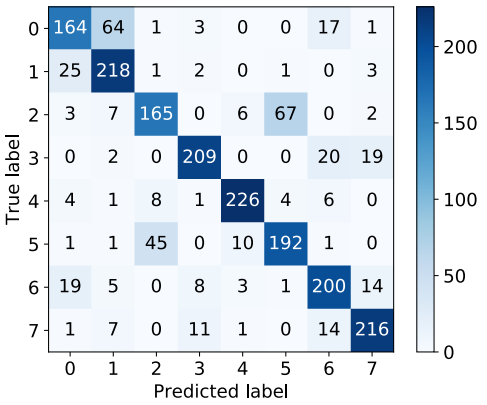
Style, class	FN	FP	TN	TP	F_1	ACC	MCC	PREC	REC	SPEC
Multi, class 0	46	31	1719	204	0.84	0.96	0.82	0.87	0.82	0.98
Multi, class 1	27	41	1709	223	0.87	0.97	0.85	0.84	0.89	0.98
Multi, class 2	95	18	1732	155	0.73	0.94	0.72	0.90	0.62	0.99
Multi, class 3	15	18	1732	235	0.93	0.98	0.92	0.93	0.94	0.99
Multi, class 4	2	10	1740	248	0.98	0.99	0.97	0.96	0.99	0.99
Multi, class 5	20	92	1658	230	0.80	0.94	0.78	0.71	0.92	0.95
Multi, class 6	16	33	1717	234	0.91	0.98	0.89	0.88	0.94	0.98
Multi, class 7	30	8	1742	220	0.92	0.98	0.91	0.96	0.88	1.00
Average	31.38	31.38	1718.62	218.62	0.87	0.97	0.86	0.88	0.87	0.98
Binary, class 0	86	53	1697	164	0.70	0.93	0.67	0.76	0.66	0.97
Binary, class 1	32	87	1663	218	0.79	0.94	0.76	0.71	0.87	0.95
Binary, class 2	85	55	1695	165	0.70	0.93	0.66	0.75	0.66	0.97
Binary, class 3	41	25	1725	209	0.86	0.97	0.85	0.89	0.84	0.99
Binary, class 4	24	20	1730	226	0.91	0.98	0.90	0.92	0.90	0.99
Binary, class 5	58	73	1677	192	0.75	0.93	0.71	0.72	0.77	0.96
Binary, class 6	50	58	1692	200	0.79	0.95	0.76	0.78	0.80	0.97
Binary, class 7	34	39	1711	216	0.86	0.96	0.83	0.85	0.86	0.98
Average	51.25	51.25	1698.75	198.75	0.79	0.95	0.77	0.80	0.80	0.97
Average Diff.	+19.88	+19.88	-19.88	-19.88	-0.08	-0.02	-0.09	-0.08	-0.08	-0.01

Table 4.4: Accuracy Test for Nasnetmobile, including all metrics and average values.

Inspecting the confusion matrix in figure 4.34, we can see the effect more clearly. Our true positives, indicated on the diagonal, are fewer for each class except class 2. Likewise, our false positives are higher for each class except class 5. We also get an indication from this plot that class 2 and 5 are often confused with each other, likewise with classes 0 and 1.



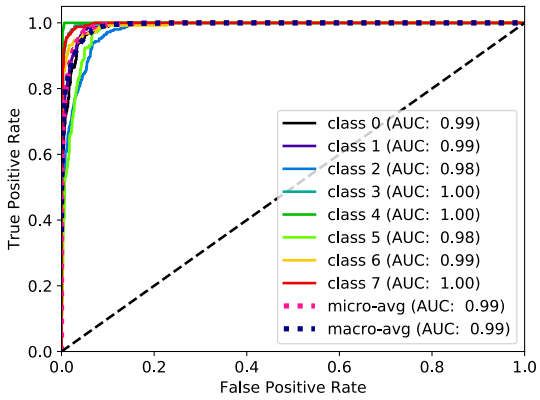
(a) Multiclass confusion matrix



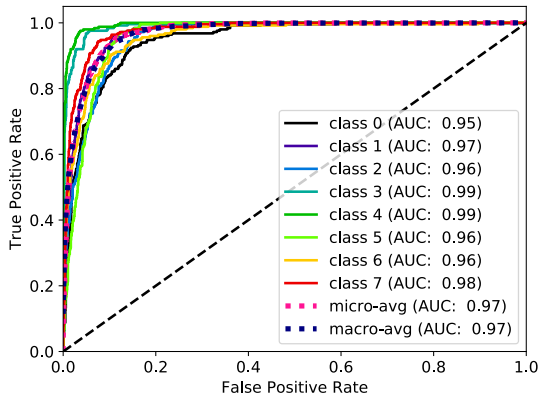
(b) Binary confusion matrix

Figure 4.34: Confusion matrices for NASNet Mobile, binary and multiclass, generated by Scikit-plots.

Looking at the ROC curves in figure 4.35, we can see that the results look very good for the multiclass case, several of the curves have an AOC ≥ 0.99. Both the micro and macro-averaged AUC values are 0.99. Unsurprisingly, this is our best classifier in the multiclass configuration. In the OVR case, however, the results have diminished slightly. Specifically, classes 0,2,5 and six have become a little worse. However, AUC values of 0.95 or higher are still usually an indication of functional classification, and we should examine the PR curves as well.



(a) Multiclass ROC curves

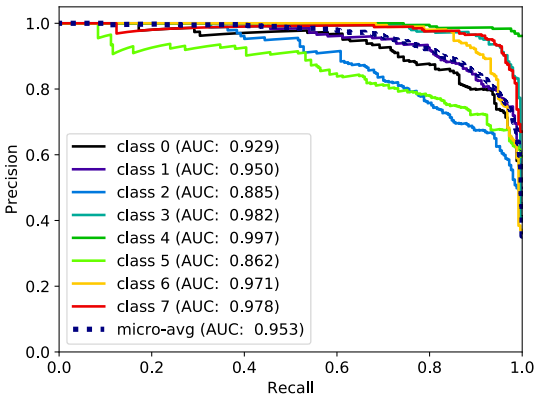


(b) Binary ROC curves

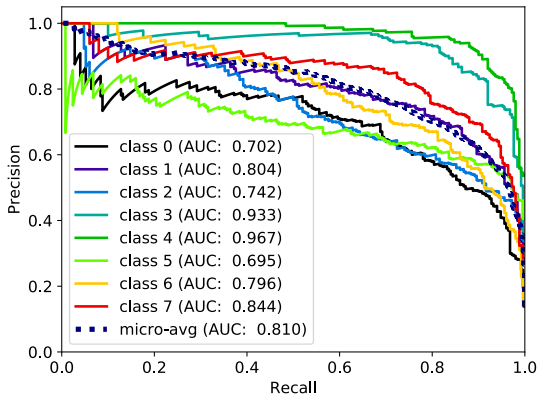
Figure 4.35: ROC curves for NASNet Mobile, binary and multiclass, generated by Scikit-plots.

From the PR curves, seen in figure 4.36, the effect of our poorest classes is much more apparent. The difference between the multiclass and OVR styles is easily visible. It also shows us a more realistic picture of how good the multiclass classifier is. The AUC is no longer almost perfect, and the differences between classes become more easily visible as well. Class 2 and 5 can be seen here reducing the average AUC, and they are a fair amount below average. Interestingly, although some of the values for classes 2 and 5 seemed to improve in the main chart, here we can see that they have also gotten worse if one takes into account both precision and recall.

In contrast to NASNet Mobile, MobileNet seems to perform much better in the OVR style. The results for the multiclass are quite weak, or certainly below our desired goals, while the OVR style meets many of our goals. We can see the overall raw figures for each class in table 4.5. We can see that in the OVR case, the numbers are better across the board, with very few exceptions.



(a) Multiclass PR curves



(b) Binary PR curves

Figure 4.36: PR curves for NASNet Mobile, binary and multiclass, generated by Scikit-plots.

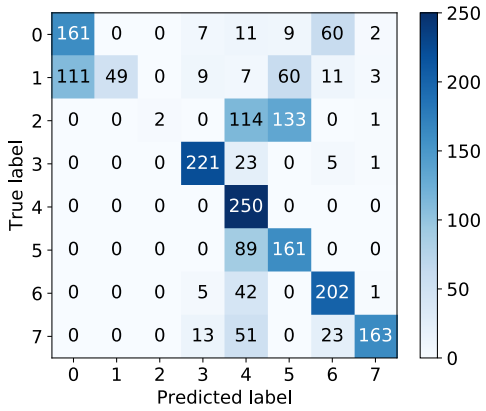
We can also see how poorly our multiclass classifier seemed to perform on the test set, with some classes having remarkably few true positives, for example, class 2 which has only two true positives. Interestingly, this class also has very few false positives, so the precision and specificity are very high for this class.

Style, class	FN	FP	TN	TP	F_1	ACC	MCC	PREC	REC	SPEC
Multi, class 0	89	111	1639	161	0.62	0.90	0.56	0.59	0.64	0.94
Multi, class 1	201	0	1750	49	0.33	0.90	0.42	1.00	0.20	1.00
Multi, class 2	248	0	1750	2	0.02	0.88	0.08	1.00	0.01	1.00
Multi, class 3	29	34	1716	221	0.88	0.97	0.86	0.87	0.88	0.98
Multi, class 4	0	337	1413	250	0.60	0.83	0.59	0.43	1.00	0.81
Multi, class 5	89	202	1548	161	0.53	0.85	0.45	0.44	0.64	0.88
Multi, class 6	48	99	1651	202	0.73	0.93	0.70	0.67	0.81	0.94
Multi, class 7	87	8	1742	163	0.77	0.95	0.77	0.95	0.65	1.00
Average	98.88	98.88	1651.12	151.12	0.56	0.90	0.55	0.74	0.60	0.94
Binary, class 0	9	104	1646	241	0.81	0.94	0.79	0.70	0.96	0.94
Binary, class 1	125	0	1750	125	0.67	0.94	0.68	1.00	0.50	1.00
Binary, class 2	128	5	1745	122	0.65	0.93	0.66	0.96	0.49	1.00
Binary, class 3	17	15	1735	233	0.94	0.98	0.93	0.94	0.93	0.99
Binary, class 4	0	117	1633	250	0.81	0.94	0.80	0.68	1.00	0.93
Binary, class 5	40	107	1643	210	0.74	0.93	0.71	0.66	0.84	0.94
Binary, class 6	26	10	1740	224	0.93	0.98	0.92	0.96	0.90	0.99
Binary, class 7	20	7	1743	230	0.94	0.99	0.94	0.97	0.92	1.00
Average	45.62	45.62	1704.38	204.38	0.81	0.95	0.80	0.86	0.82	0.97
Average Diff.	-53.25	-53.25	+53.25	+53.25	+0.25	+0.05	+0.25	+0.11	+0.21	+0.03

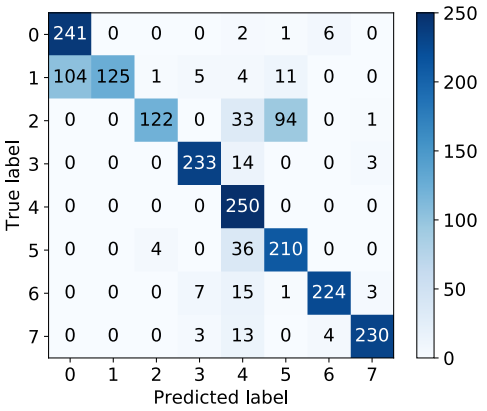
Table 4.5: Accuracy Test for Mobilenet, including all metrics and average values.

Looking at the confusion matrix 4.37, the difference is night and day. The multiclass MobileNet has unacceptable performance for almost every class, except for perhaps class 3, which has a good number of true positives and few false positives. Class 4, on the other hand, is a big problem, it has a perfect score for true positives, but unfortunately also has a huge amount of false positives.

On the other hand, the OVR classification is much more acceptable. We still have a slightly high



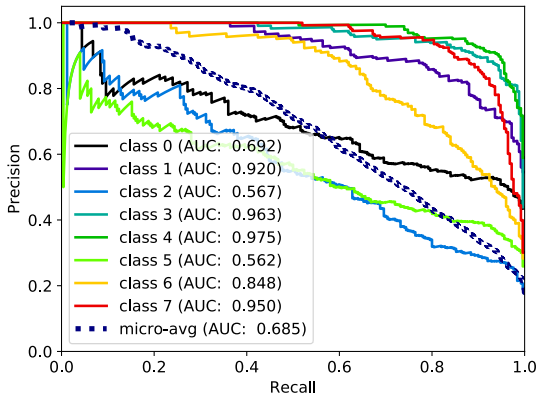
(a) Multiclass confusion matrix



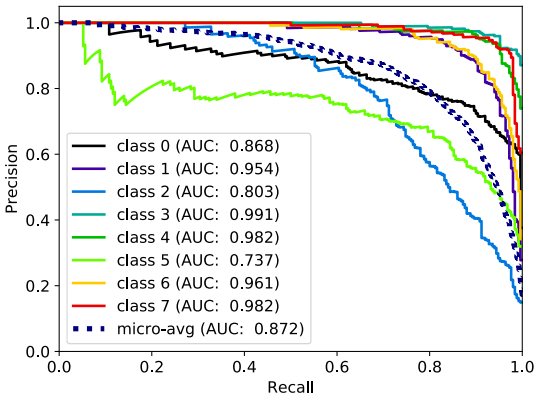
(b) Binary confusion matrix

Figure 4.37: Confusion matrices for Mobilenet, binary and multiclass, generated by Scikit-plots.

number of false positives for classes 0, 4 and 5. Classes 1 and two also have quite a few false negatives, but otherwise, the results are much more in line with what we would like to see from a classification problem like this. This effect can also be seen in the PR curves in figure 4.38. Here we see that classes 0 and two are the most difficult for our classifier. In summary, we have seen an example where the OVR style has made a great classifier worse, and one where it made a very bade classifier much better. Judging by the table of average differences, the latter case seems to be slightly more common, and in the cases where the classifier was made worse, it usually wasn't by an unacceptable amount.



(a) Multiclass PR curves



(b) Binary PR curves

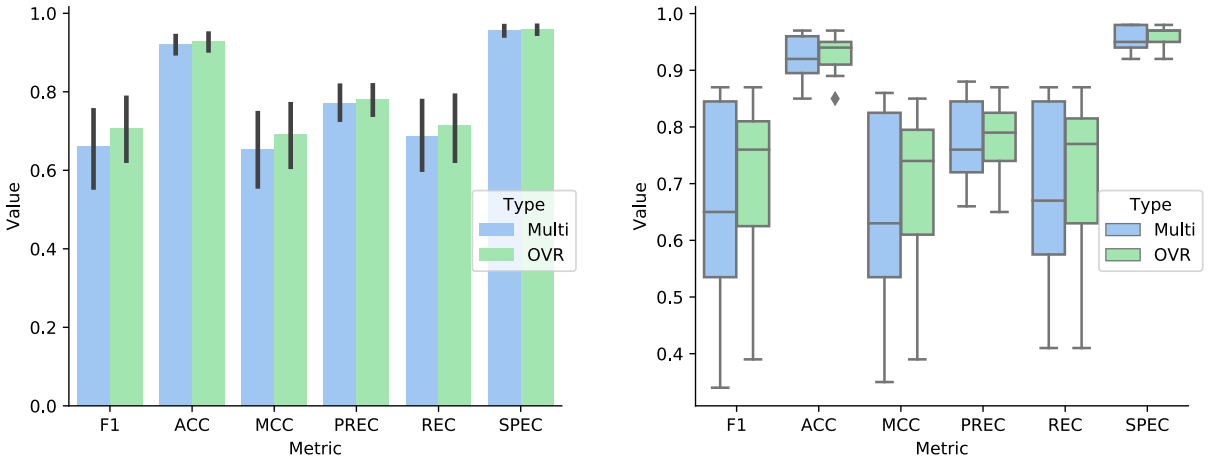
Figure 4.38: PR curves for Mobilenet, binary and multiclass, generated by Scikit-plots.

Overall effect

Looking at the raw numbers is interesting, but it is difficult to intuit how using an OVR strategy effects the accuracy metrics based on them. In figure 4.39 we have generated two plots which should help us with this. The first, a bar plot with error bars, shows us the small improvements to the average values which we saw in the previous sections. We also see that the error bars are quite large for some of these metrics, which makes sense as there is not that much data.

However, the second plot tells an exciting story. The box plot shows us an effect the action of switching to an OVR strategy has on accuracy. We can see that this changes the distribution of

the accuracies significantly. The majority of the observed values fall into a smaller range on the OVR side of the plot, both in the positive and negative sense. This means that the best performers in the multiclass style have become slightly worse, and the worst performers have become slightly better. This seems to reflect what we have previously observed about the effects. We can see that specifically the second quartile, or median, in the box plot has become much higher for many of the metrics.



(a) Bar plot of average metrics on the test set for both OVR and multiclass network styles

(b) Box plot of metrics on the test set for both OVR and multiclass network styles

Figure 4.39: Summary plots of the testing metrics

4.5.3 Goals

In the tools and implementation section, we specified a set of goals which we hoped to achieve both in regards to classification accuracy and performance. In figure 4.6 we have outlined which networks achieved what goals. The same effect is visible here as from the previous section, the best networks from multiclass now fail to achieve some of the same classification goals, but overall the number of classification goals not met has gone down, from 34 in the multiclass case to 31 in the OVR case.

A few of the cases where we only met the minimum goals have also changed to meet the desired goals in the OVR case, specifically for specificity. Unfortunately fewer applications meet our minimum goals for recall in the OVR style, with only one classifier achieving this. In our specific case, the best classifier was a multiclass style network, NASNet Mobile, which met all of our goals in addition to being extremely fast.

We aimed to have high F_1 and MCC scored in our results, and as we can see the number of applications which met our goals increased for both of these scores in the OVR style.

When it comes to speed, the effect of the OVR style becomes apparent, we have gone from almost all networks exceeding the desired goals for FPS to only 2 doing this, and six now do not even meet our requirements for real-time processing. The best classifier in the OVR configuration was NASNet Large, which was only able to achieve 6 FPS during testing. MobileNet in the multiclass style is over 100 times faster by comparison, although it is not a particularly good classifier.

Goals	F_1	ACC	MCC	PREC	REC	SPEC	FPS
VGG16, Multi	Des.	Des.	Des.	Des.	Des.	Des.	Des.
VGG19, Multi	Des.	Des.	Des.	Des.	Des.	Des.	Des.
Inception v3, Multi	N/A	N/A	N/A	Des.	No	Min.	Des.
DenseNet 121, Multi	N/A	N/A	N/A	Des.	No	Min.	Des.
DenseNet 169, Multi	N/A	N/A	N/A	N/A	No	Min.	Des.
DenseNet 201, Multi	N/A	N/A	N/A	N/A	No	Min.	Des.
Xception, Multi	N/A	N/A	N/A	N/A	No	Min.	Des.
Inception RN v2, Multi	N/A	N/A	N/A	N/A	No	Min.	Des.
Mobilenet, Multi	N/A	N/A	N/A	N/A	No	Min.	Des.
NASNet Large, Multi	Des.	Des.	Des.	Des.	No	Des.	Min.
NASNet Mobile, Multi	Des.	Des.	Des.	Des.	Des.	Des.	Des.
VGG16, OVR	Des.	N/A	Des.	Des.	No	Des.	Min.
VGG19, OVR	Des.	Des.	Des.	Des.	No	Des.	Min.
Inception v3, OVR	Des.	N/A	Des.	Des.	No	Des.	Min.
DenseNet 121, OVR	N/A	N/A	N/A	N/A	No	Min.	No
DenseNet 169, OVR	N/A	N/A	N/A	N/A	No	Min.	No
DenseNet 201, OVR	N/A	N/A	N/A	N/A	No	Min.	No
Xception, OVR	N/A	N/A	N/A	Des.	No	Min.	No
Inception RN v2, OVR	N/A	N/A	Des.	Des.	No	Min.	No
Mobilenet, OVR	Des.	N/A	Des.	Des.	No	Des.	Des.
NASNet Large, OVR	Des.	Des.	Des.	Des.	Des.	Des.	No
NASNet Mobile, OVR	Des.	N/A	Des.	Des.	No	Des.	Des.

Table 4.6: Summary of which goals were achieved for each network style and application. Green cells (Des.) meet our desired goals, yellow cells (Min.) meet our minimum goal, red cells did not meet any goal, and cells labeled N/A did not meed our desired goal, but did not have a minimum goal specified.

4.6 Summary

In this chapter, we have detailed our training and testing procedures. Besides, we have provided a breakdown of the recorded metrics during all phases. These metrics should provide us with an answer to our second research question, namely what the performance and resource usage consequences are for choosing and OVR style. In the final section, we detailed the classification performance and to what degree our networks were able to meet our performance goals. This should provide us with answers to our primary research question, whether or not neural network OVR is a viable strategy for a classification problem such as ours.

Based on our final results, we can see that the performance impact of using the OVR style is rather significant, especially in the testing phase. Here, on average, our classification slowed nearly proportionally to the number of classes. However, on the classification side of things, we saw the classification quality metrics increase on average, and that the best networks of both styles had nearly the same performance, exceeding our desired goals. These results, in total, should be enough to answer our research questions.

Chapter 5

Conclusion

We trained and ran tests on 99 networks using 11 different architectures, where 88 were binary networks set up in an OVR configuration. When comparing the networks, we measured several critical metrics of their performance, including hardware resource use (GPU, CPU and System Memory), classification speed (in FPS), and classification accuracy (F_1, Accuracy, MCC, Precision, Recall, Specificity). Our primary research goal was to determine if the OVR technique could be applied to modern neural network architectures and how this would change the metrics we set out to measure.

5.1 Summary

In summary, we saw an increase in the average classification results across all applications using the OVR method. Some applications saw vastly improved classification results, and some applications saw worsened results, but the negative findings were usually not as extensive as the improvements. All applications saw increased resource use during testing with the OVR style, and many saw increased use during training as well. In regards to frame throughput, all applications saw a severe decrease in performance using OVR, nearly proportional to the number of classes. This leads us to believe that the vast increase in complexity and the way resources are allocated on the GPU mean that this style will carry with it a severe performance penalty. The penalty likely means that our OVR is likely completely unsuitable for a dataset or classification problem with a class count greatly exceeding that of Kvasir v2.

In our introduction, we began with a set of research questions which we endeavored to answer with our research. For clarity those were:

- Can OVR produce useful results with complex binary neural networks?
- How does using an OVR style affect how we fulfill the requirements of our classifier? Will our experimental setup still perform to the desired level when it comes to classification accuracy, resource use, and classification speed?

Based on the results in the previous chapter, we would conclude that the OVR method is suitable for use with complex neural networks, as in general, it offered comparable results to the single-network method. In general, our requirements for classification accuracy are better fulfilled by the OVR style, with increases in mean and median results across the board. However, our requirements for resource use and speed go largely unmet, meaning that this style might be unsuitable for real-time classification unless some optimizations can be made which will increase the classification speed.

Thus, the OVR style might be beneficial to attempt if our initial classification results are weak, and we can tolerate the proportional (to the number of classes) decrease in classification speed. This method might be faster than attempting to optimize hyperparameters manually and is relatively easy to set up, especially using our framework.

When it comes to the clinical setting, as none of our OVR classifiers were able to meet all of our minimum design goals in this regard, the exact networks we have presented here would likely be unsuitable for this use. However, with performance optimizations and hyperparameter tuning, this might change. The fastest OVR classifier, based on MobileNet was able to achieve 95 FPS with an accuracy of .95, so it might be possible to tune these networks so that the other accuracy metrics also exceed our clinical requirements.

5.2 Contributions

We also saw that the distribution of our accuracy metrics became less spread, with the worst performing applications in multiclass becoming better in an OVR configuration, and the best performing multiclass classifiers usually becoming slightly worse. This indicates to us that the OVR technique might make the networks less sensitive to poor hyperparameter selection, so it might be a valid strategy to quickly improve abysmal classification accuracy without having to spend much time optimizing hyperparameters.

When it comes to our performance goals, the number of goals met by all of our applications increased, but the number of applications which met all of our classification goals decreased overall, especially when frame throughput is considered. Interestingly, the most substantial average increase for the OVR style was for the F_1 and MCC scores, which were essential metrics to us. In general, we can see that the classification accuracy is comparable and that the OVR structure could be

used directly in at least one case if performance is not critical. To conclude, we feel that in many cases OVR could be a viable strategy with modern neural networks, but that it will not always result in increased classification accuracy.

As mentioned in the introduction, we provide in this research a comprehensive analysis of the performance difference between our experimental OVR multi-network style, moreover, a conventional multiclass approach on our selected dataset. These performance differences include both the difference in classification performance and the difference in resource use. We have shown what effect this change in style has on the performance of several different architectures. During this analysis, we have found that, in many cases, our OVR style does provide satisfactory classification performance, but that it comes at a massive resource usage cost during final classification.

We have also created a testing and metric logging framework, TFmetrics, which allows automated training and testing of neural networks in many different configurations, and on a user's chosen dataset. TFmetrics allows the user to quickly and easily use OVR classification with select modern neural network architectures, something which is not supported by any other framework to our knowledge. This framework may be useful in future research, and we plan to develop it further to support a broader spectrum of machine learning approaches.

Also, we have created and tested multiclass networks which improve the classification speed and accuracy significantly over previous research on this dataset, ending up with three network styles which meet or exceed all of our desired classification and performance goals.

In summary, using the OVR style resulted in a 7% increase in average F_1 score, 1% increase in average accuracy, 6% increase in MCC, 1% increase in precision and finally a 4% increase in average recall. Specificity remained relatively unchanged. Also, the median values for all of these metrics increased significantly in the OVR style, with the median F_1, MCC and recall scores increasing by over 15%. The most improved network in the OVR configuration saw an increase in F_1 of 45% and an MCC increase of 40%. During testing, on average our OVR multi-network style was 7.6 times slower to classify than a single network multiclass implementation.

5.3 Future Work

Although this research was rather involved and comprehensive, there is, of course, a lot more to explore using OVR styles. In the future, we would be interested in conducting additional research in this field and exploring the possibilities offered by this style more thoroughly and in other ways that what we have done so far. There are several aspects of the OVR style we would like to explore in the future, among them are:

- For example, it is of interest to examine the performance of simpler architectures with OVR, which no longer operate well on newer or more complex datasets.
- Testing on other datasets than Kvasir v2. Our results are limited to a single dataset; it would be interesting to explore the results of many datasets and see if the performances are similar or different. Of particular interest would be datasets with fewer than eight classes, and datasets with a much higher number of classes.
- Experimenting with optimizing the hyperparameters separately for the individual binary networks in the OVR configuration, since it is possible that it could lead to better performance.
- It would be interesting to test other strategies such as One Versus One, which would halve the required number of networks. In addition, testing other methods of combining the results of the individual binary networks would be of interest. For example, we might try to place a simple network on top of the individual networks and train it on their outputs collectively.

Another strategy could be thresholding to decide whether or not a frame is positive or negative for a specific class.

Finally, there are several performance improvements should be explored, as they could make the OVR networks in our research more viable for real-time applications.

- In our research, we have used the default data structure of TensorFlow, which is NHWC as described in the background chapter. However, there could be serious performance gains from switching to an NCHW data structure instead, but still using TensorFlow [49].
- TensorRT is a network optimizer developed by NVidia specifically for use with their GPU [50]. The network optimizer was not used in our research and it would be interesting to examine what kind of performance gains could be made with this framework.
- TensorFlow itself is optimized to run on multiple GPU's [24]. It would be interesting to see if this improves the performance of the OVR style and if so, by how much.
- We have primarily focused on TensorFlow, but other machine learning libraries exist that might provide better performance. Indeed, there might be a reason to believe that TensorFlow is one of the slower frameworks [46], and it might be interesting to test the OVR style on PyTorch, Theano or Caffe.
- We would also like to examine the actual calls being made by TensorFlow during training and testing to see what exactly is happening during these phases. In future research, it might be interesting to include a visual profiler which could give us a good idea of exactly how the frameworks use resources [51].

Bibliography

[1] Robbie Allen. Nips accepted papers stats – machine learning in practice – medium. https://medium.com/machine-learning-in-practice/nips-accepted-papers-stats-26f124843aa0. (Accessed on 05/20/2018).

[2] Ryan Rifkin and Aldebaro Klautau. In defense of one-vs-all classification. *Journal of machine learning research*, 5(Jan):101–141, 2004.

[3] Stephen Marsland. *Machine Learning: An Algorithmic Perspective, Second Edition*. Chapman & Hall/CRC, 2nd edition, 2014.

[4] Irina Rish. An empirical study of the naïve bayes classifier. 3, 01 2001.

[5] Joachim Huyssen and Geoffrey Spedding. Should airplanes look like birds? engineers envision more fuel-efficient design. https://phys.org/news/2010-11-airplanes-birds-envision-fuel-efficient.html, 2010.

[6] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, Dec 1943.

[7] G. L. Shaw. Donald hebb: The organization of behavior. In Günther Palm and Ad Aertsen, editors, *Brain Theory*, pages 231–233, Berlin, Heidelberg, 1986. Springer Berlin Heidelberg.

[8] Daniel Crevier. *AI: The Tumultuous History of the Search for Artificial Intelligence*. Basic Books, Inc., New York, NY, USA, 1993.

[9] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[10] David E Rumelhart, Geoffrey E Hinton, James L McClelland, et al. A general framework for parallel distributed processing. *Parallel distributed processing: Explorations in the microstructure of cognition*, 1:45–76, 1986.

[11] Bohdan Macukow. Neural networks–state of art, brief history, basic models and architecture. In *IFIP International Conference on Computer Information Systems and Industrial Management*, pages 3–14. Springer, 2016.

[12] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861, 2017.

[13] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[14] Jin-Hyuk Hong and Sung-Bae Cho. A probabilistic multi-class strategy of one-vs.-rest support vector machines for cancer classification. *Neurocomputing*, 71(16-18):3275–3281, 2008.

[15] Jianhua Xu. An extended one-versus-rest support vector machine for multi-label classification. *Neurocomputing*, 74(17):3114–3124, 2011.

[16] Marcin Moczulski, Misha Denil, Jeremy Appleyard, and Nando de Freitas. ACDC: A structured efficient linear layer. *CoRR*, abs/1511.05946, 2015.

[17] Roi Livni, Shai Shalev-Shwartz, and Ohad Shamir. On the computational efficiency of training neural networks. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 855–863. Curran Associates, Inc., 2014.

[18] François Chollet et al. Keras. https://keras.io, 2015.

[19] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. *CoRR*, abs/1512.00567, 2015.

[20] François Chollet. Xception: Deep learning with depthwise separable convolutions. *CoRR*, abs/1610.02357, 2016.

[21] Gao Huang, Zhuang Liu, and Kilian Q. Weinberger. Densely connected convolutional networks. *CoRR*, abs/1608.06993, 2016.

[22] Christian Szegedy, Sergey Ioffe, and Vincent Vanhoucke. Inception-v4, inception-resnet and the impact of residual connections on learning. *CoRR*, abs/1602.07261, 2016.

[23] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. Learning transferable architectures for scalable image recognition. *CoRR*, abs/1707.07012, 2017.

[24] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[25] Katyanna Quach. Machine learning newbs: Tensorflow too hard? kick its ass with keras • the register. https://www.theregister.co.uk/2017/03/16/keras_new_update/. (Accessed on 05/19/2018).

[26] Aakash Nain. Tensorflow or keras? which one should i learn? https://medium.com/implodinggradients/tensorflow-or-keras-which-one-should-i-learn-5dd7fa3f9ca0. (Accessed on 05/19/2018).

[27] Sinno Jialin Pan and Qiang Yang. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359, 2010.

[28] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael S. Bernstein, Alexander C. Berg, and Fei-Fei Li. Imagenet large scale visual recognition challenge. *CoRR*, abs/1409.0575, 2014.

[29] Nvidia Corporation. Programming guide :: Cuda toolkit documentation. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#context. (Accessed on 05/09/2018).

[30] Kunio Doi. Computer-aided diagnosis in medical imaging: historical review, current status and future potential. *Computerized medical imaging and graphics*, 31(4-5):198–211, 2007.

[31] International Agency for Research on Cancer et al. Latest world cancer statistics global cancer burden rises to 14.1 million new cases in 2012: Marked increase in breast cancers must be addressed. *World Health Organization*, 12, 2013.

[32] Konstantin Pogorelov, Kristin Ranheim Randel, Carsten Griwodz, Sigrun Losada Eskeland, Thomas de Lange, Dag Johansen, Concetto Spampinato, Duc-Tien Dang-Nguyen, Mathias Lux, Peter Thelin Schmidt, Michael Riegler, and Pål Halvorsen. Kvasir: A multi-class image dataset for computer aided gastrointestinal disease detection. In *Proceedings of the 8th ACM*

on Multimedia Systems Conference, MMSys'17, pages 164–169, New York, NY, USA, 2017. ACM.

[33] Michael Riegler, Mathias Lux, Carsten Griwodz, Concetto Spampinato, Thomas de Lange, Sigrun L. Eskeland, Konstantin Pogorelov, Wallapak Tavanapong, Peter T. Schmidt, Cathal Gurrin, Dag Johansen, Håvard Johansen, and Pål Halvorsen. Multimedia and medicine: Teammates for better disease detection and survival. In *Proceedings of the 2016 ACM on Multimedia Conference*, MM '16, pages 968–977, New York, NY, USA, 2016. ACM.

[34] Michael Riegler, Konstantin Pogorelov, Sigrun Losada Eskeland, Peter Thelin Schmidt, Zeno Albisser, Dag Johansen, Carsten Griwodz, Pål Halvorsen, and Thomas De Lange. From annotation to computer-aided diagnosis: Detailed evaluation of a medical multimedia system. *ACM Trans. Multimedia Comput. Commun. Appl.*, 13(3):26:1–26:26, May 2017.

[35] Konstantin Pogorelov, Michael Riegler, Pål Halvorsen, Carsten Griwodz, Thomas de Lange, Kristin Randel, Sigrun Eskeland, Dang Nguyen, Duc Tien, Olga Ostroukhova, et al. A comparison of deep learning with global features for gastrointestinal disease detection. 2017.

[36] Nvidia Corporation. Geforce gtx 1080 ti graphics cards | nvidia geforce. https://www.nvidia.com/en-us/geforce/products/10series/geforce-gtx-1080-ti/. (Accessed on 05/07/2018).

[37] Marina Sokolova and Guy Lapalme. A systematic analysis of performance measures for classification tasks. *Information Processing & Management*, 45(4):427–437, 2009.

[38] Charles E. Metz. Basic principles of roc analysis. *Seminars in Nuclear Medicine*, 8(4):283 – 298, 1978.

[39] Jesse Davis and Mark Goadrich. The relationship between precision-recall and roc curves. In *Proceedings of the 23rd International Conference on Machine Learning*, ICML '06, pages 233–240, New York, NY, USA, 2006. ACM.

[40] Andreas Beger. Precision-recall curves. https://ssrn.com/abstract=2765419, 2018.

[41] K. Pogorelov, O. Ostroukhova, A. Petlund, P. Halvorsen, T. de Lange, H. N. Espeland, T. Kupka, C. Griwodz, and M. Riegler. Deep learning and handcrafted feature based approaches for automatic detection of angiectasia. In *2018 IEEE EMBS International Conference on Biomedical Health Informatics (BHI)*, pages 365–368, March 2018.

[42] Mirocam capsule endoscope camera offers a broder field of 170 degrees which enables a more through diagnosis of the small bowel. http://www.reyyanmedical.com/index.php?yazigoster=mirocam-endoscopic-camera&dil=en. (Accessed on 05/11/2018).

[43] Endoscopic camera | vimex endoscopy. http://vimex-endoscopy.com/aparatura-endoskopowa/kamera/?lang=en. (Accessed on 05/11/2018).

[44] Patrick Blessing Peter F. Niederer Daniel Doswald Norbert Felber Juerg Haefliger, Yves Lehareinger. High-definition digital endoscopy, 1999.

[45] M Galar, A Fernández, E Barrenechea, H Bustince, and F Herrera. Aggregation schemes for binarization techniques methods' description. *Pamplona, Spain*, 2011.

[46] Shaohuai Shi, Qiang Wang, Pengfei Xu, and Xiaowen Chu. Benchmarking state-of-the-art deep learning software tools. *CoRR*, abs/1608.07249, 2016.

[47] Giampaolo Rolada et al. psutil. https://github.com/giampaolo/psutil, 2018.

[48] Nvidia Corporation. Nvidia management library (nvml). https://developer.nvidia.com/nvidia-management-library-nvml, 2018.

[49] Google Brain. Performance guide. https://www.tensorflow.org/performance/performance_guide. (Accessed on 05/21/2018).

[50] Nvidia Corporation. Tensorrt 3: Faster tensorflow inference and volta support | nvidia developer blog. https://devblogs.nvidia.com/tensorrt-3-faster-tensorflow-inference/. (Accessed on 05/21/2018).

[51] Illarion Khlestov. Howto profile tensorflow: – towards data science. https://towardsdatascience.com/howto-profile-tensorflow-1a49fb18073d. (Accessed on 05/21/2018).

Part III

Appendices

Appendix A

Figures

A.1 Model Visualizations

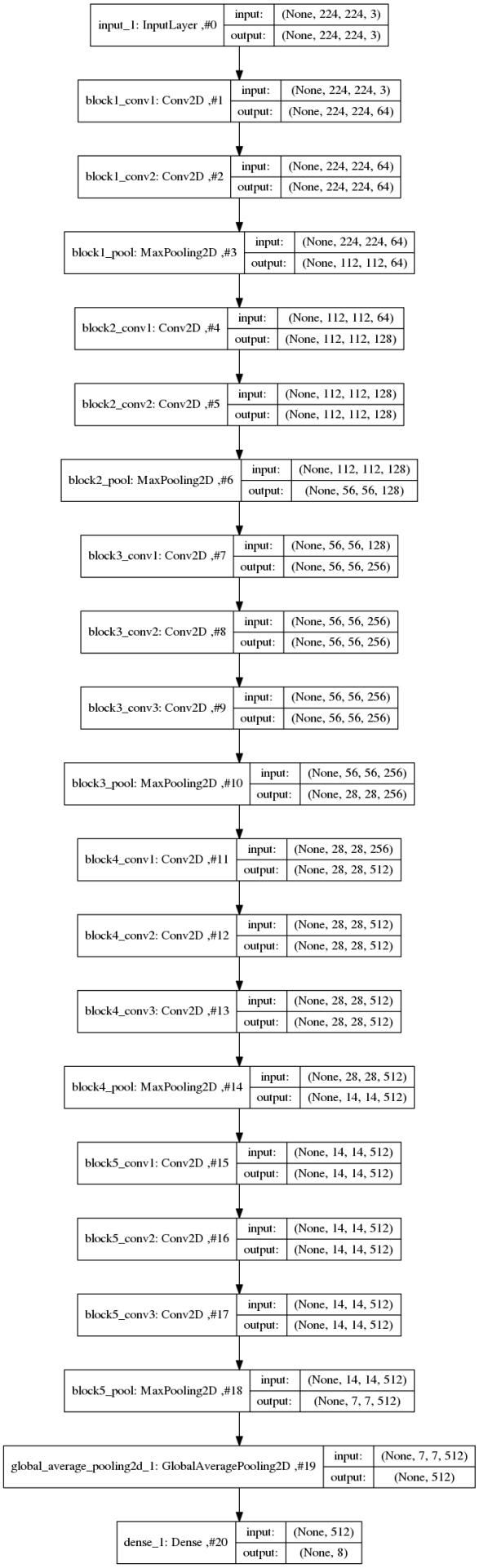


Figure A.1: VGG 16 Model Visualization

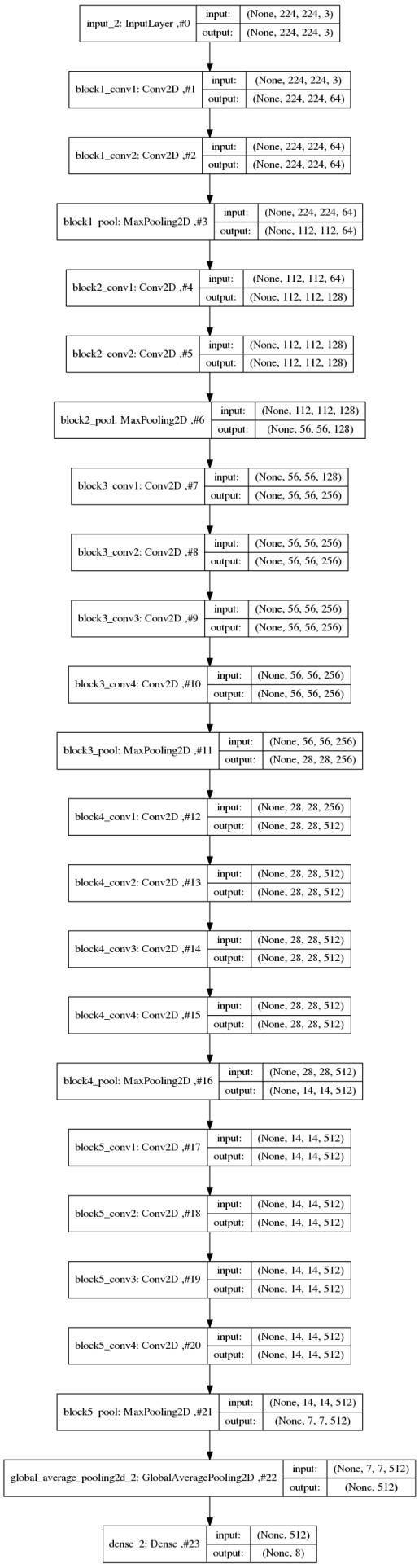


Figure A.2: VGG 19 Model Visualization

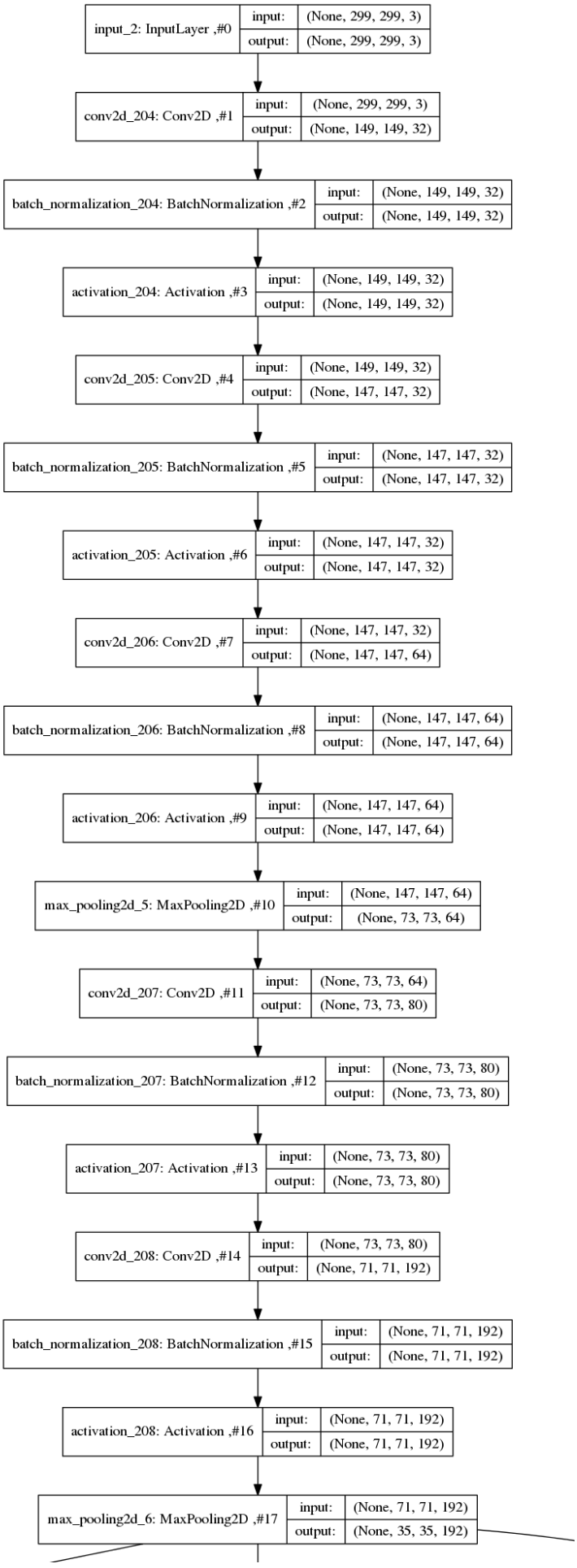


Figure A.3: Inception v3 Input Layers

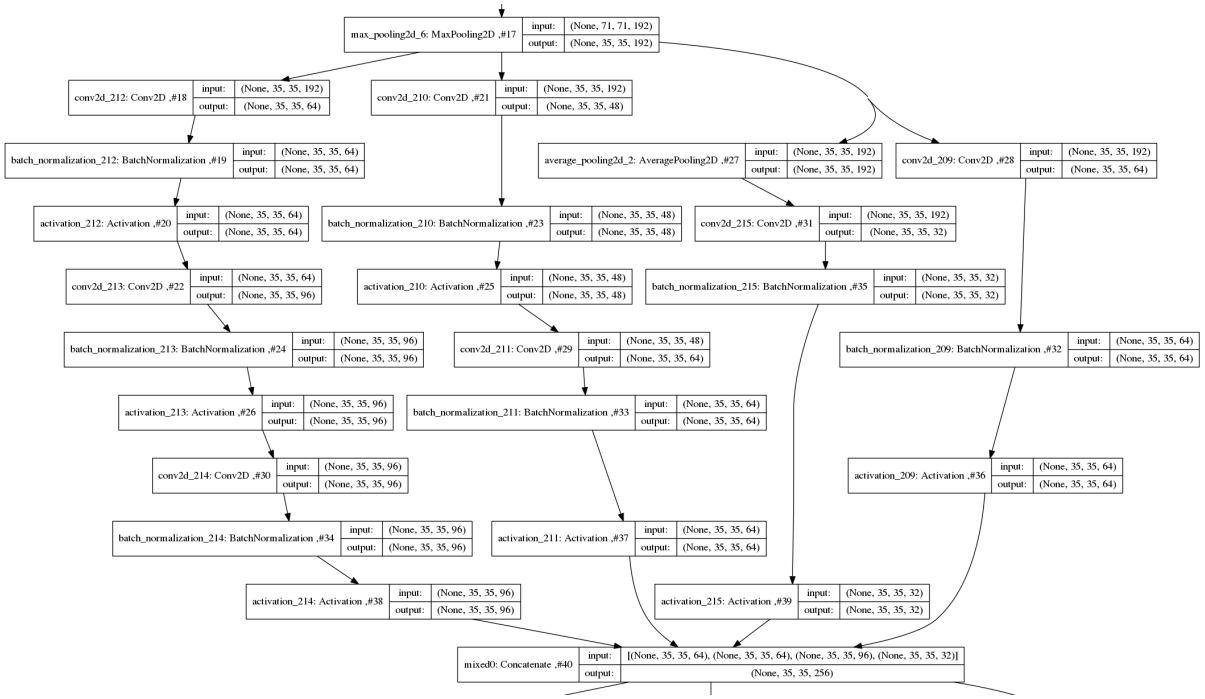


Figure A.4: Inception v3 Block

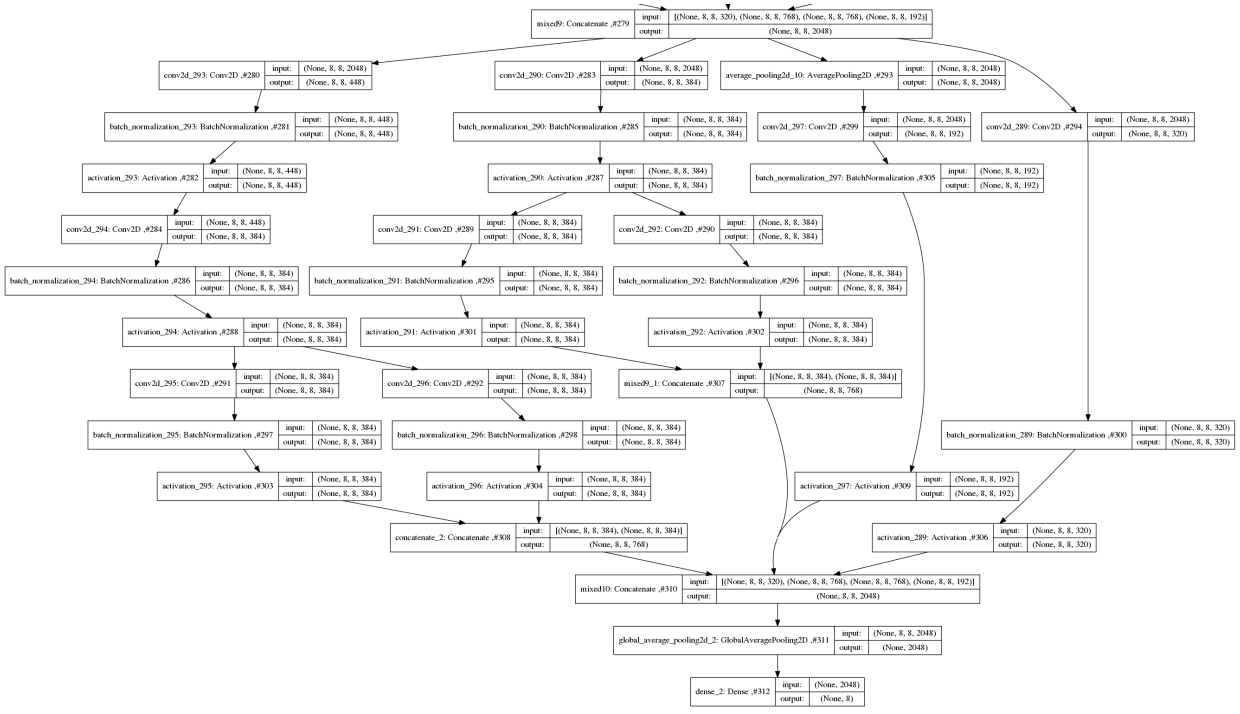


Figure A.5: Inception v3 Output Block

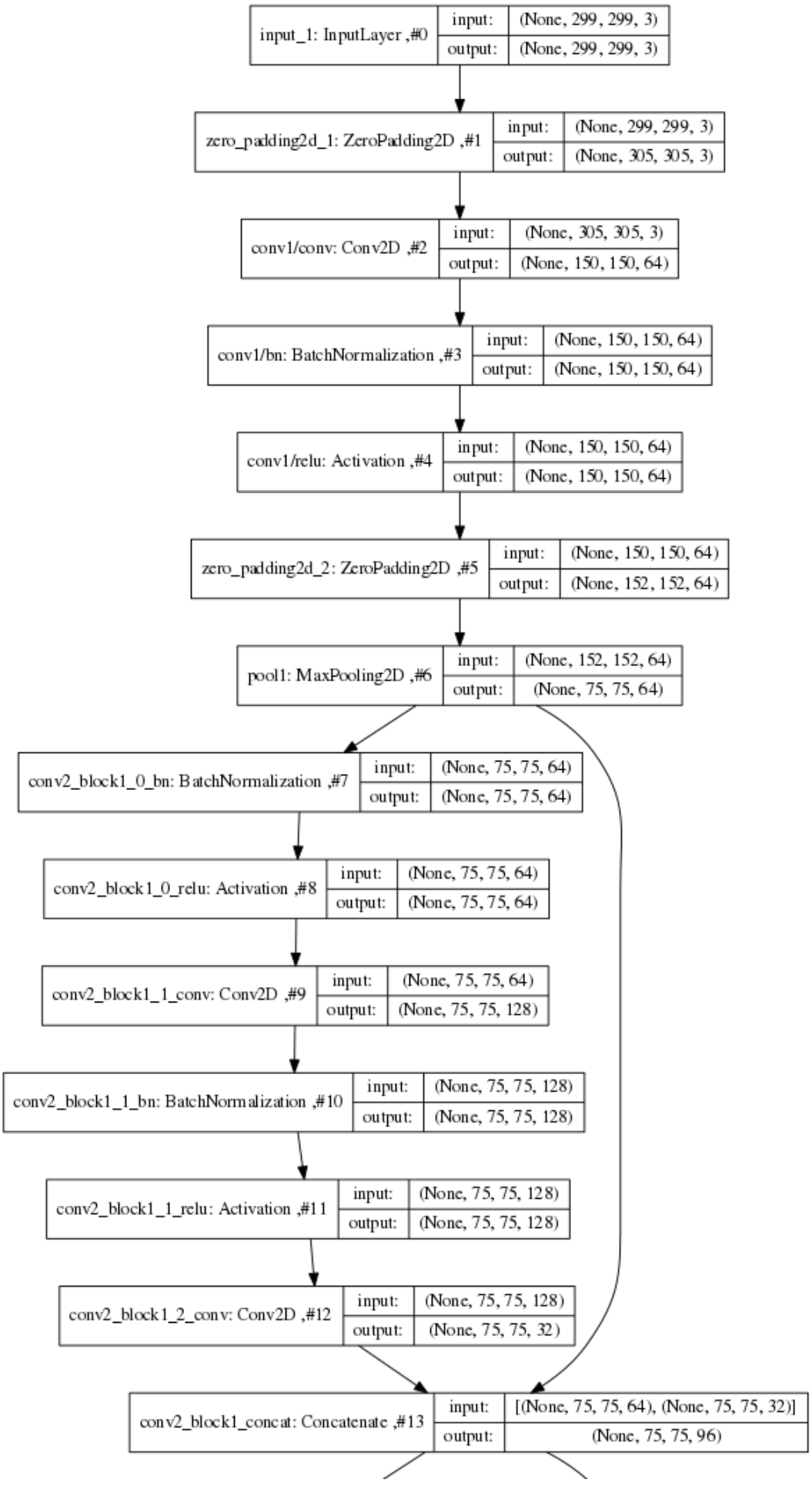


Figure A.6: DenseNet Input Block

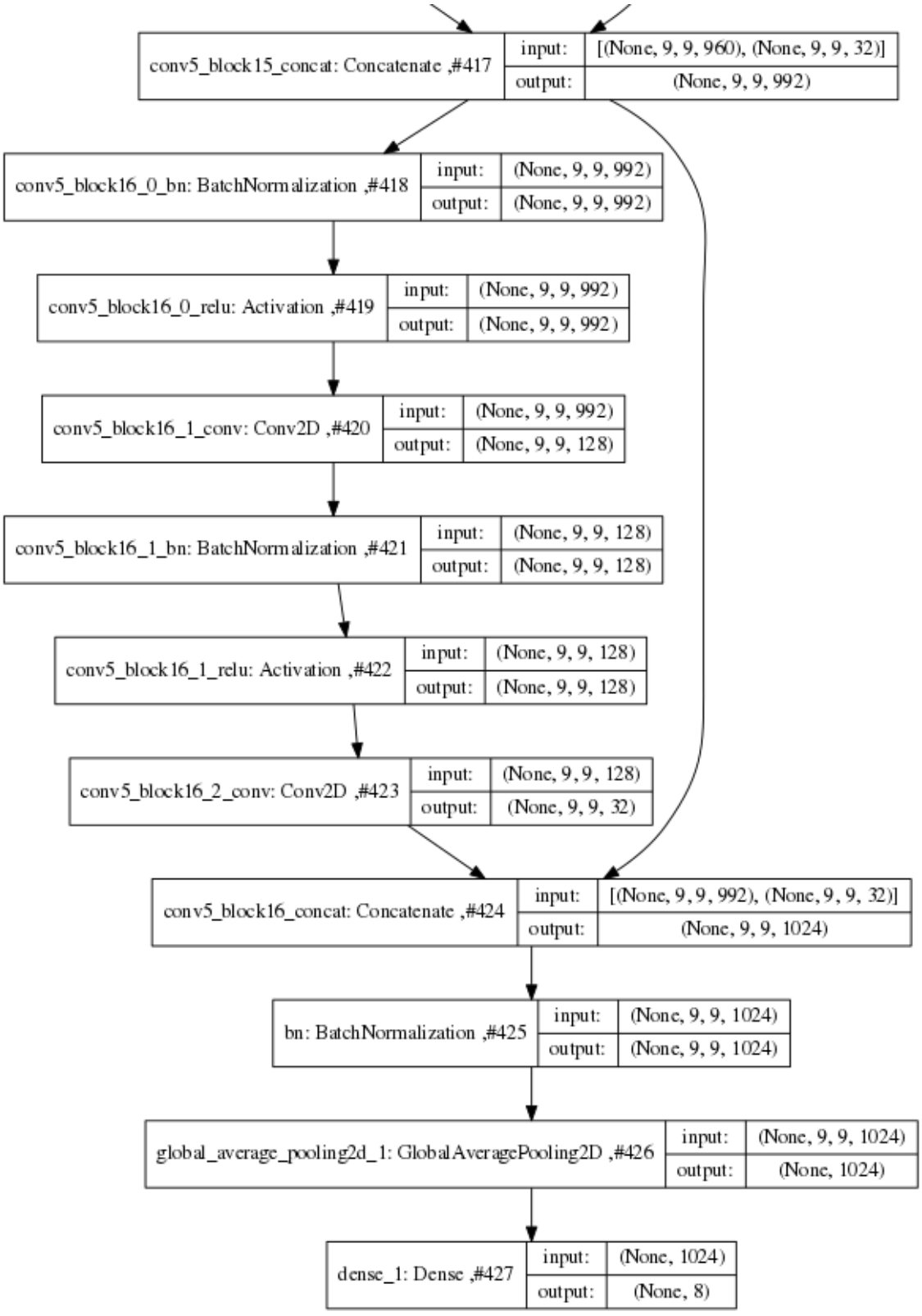


Figure A.7: DenseNet Output Block

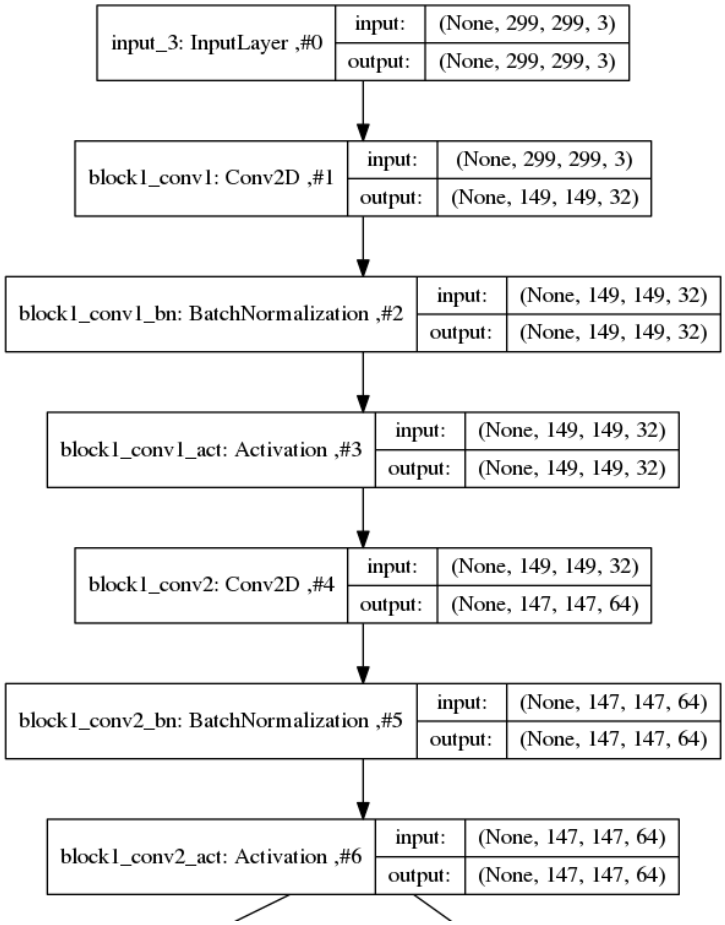


Figure A.8: Xception Input

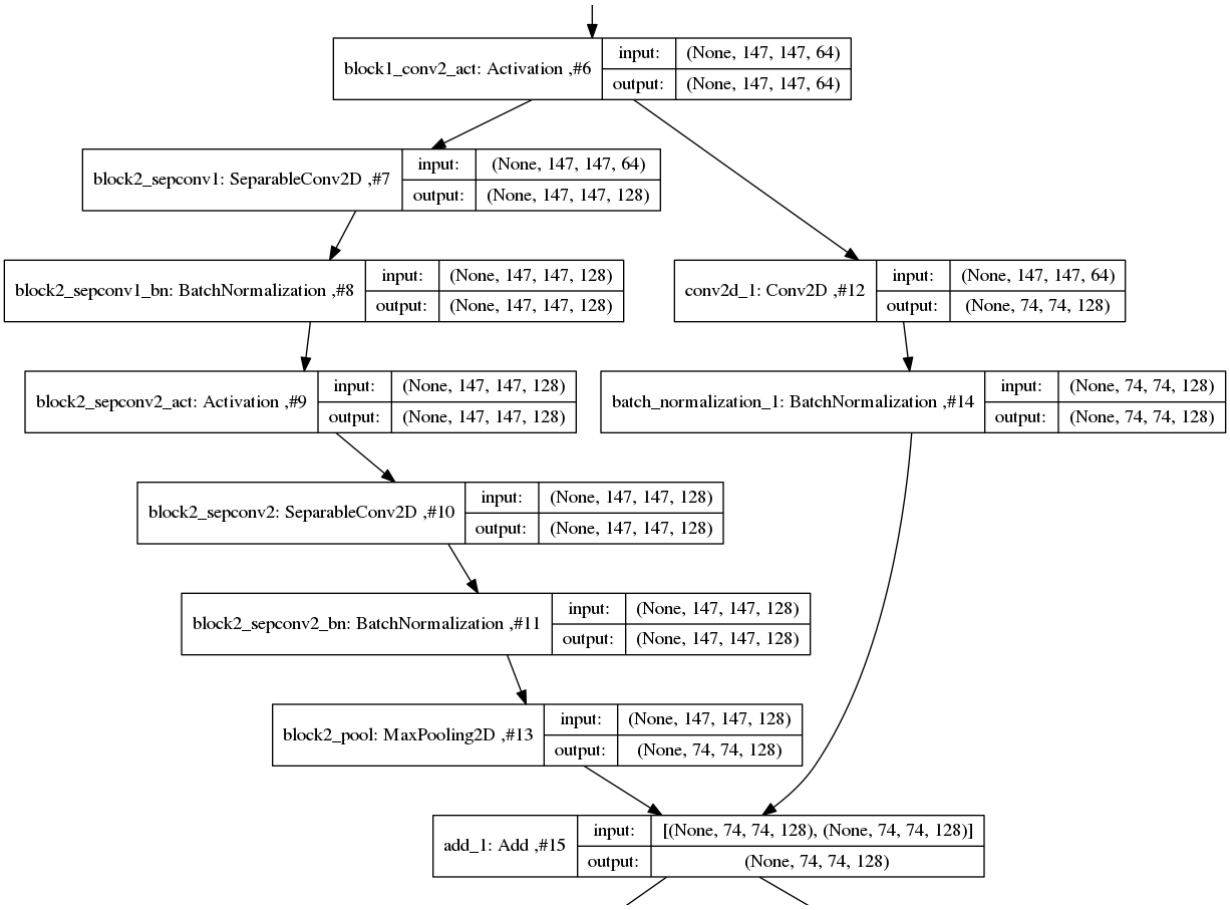


Figure A.9: Xception Block type 1

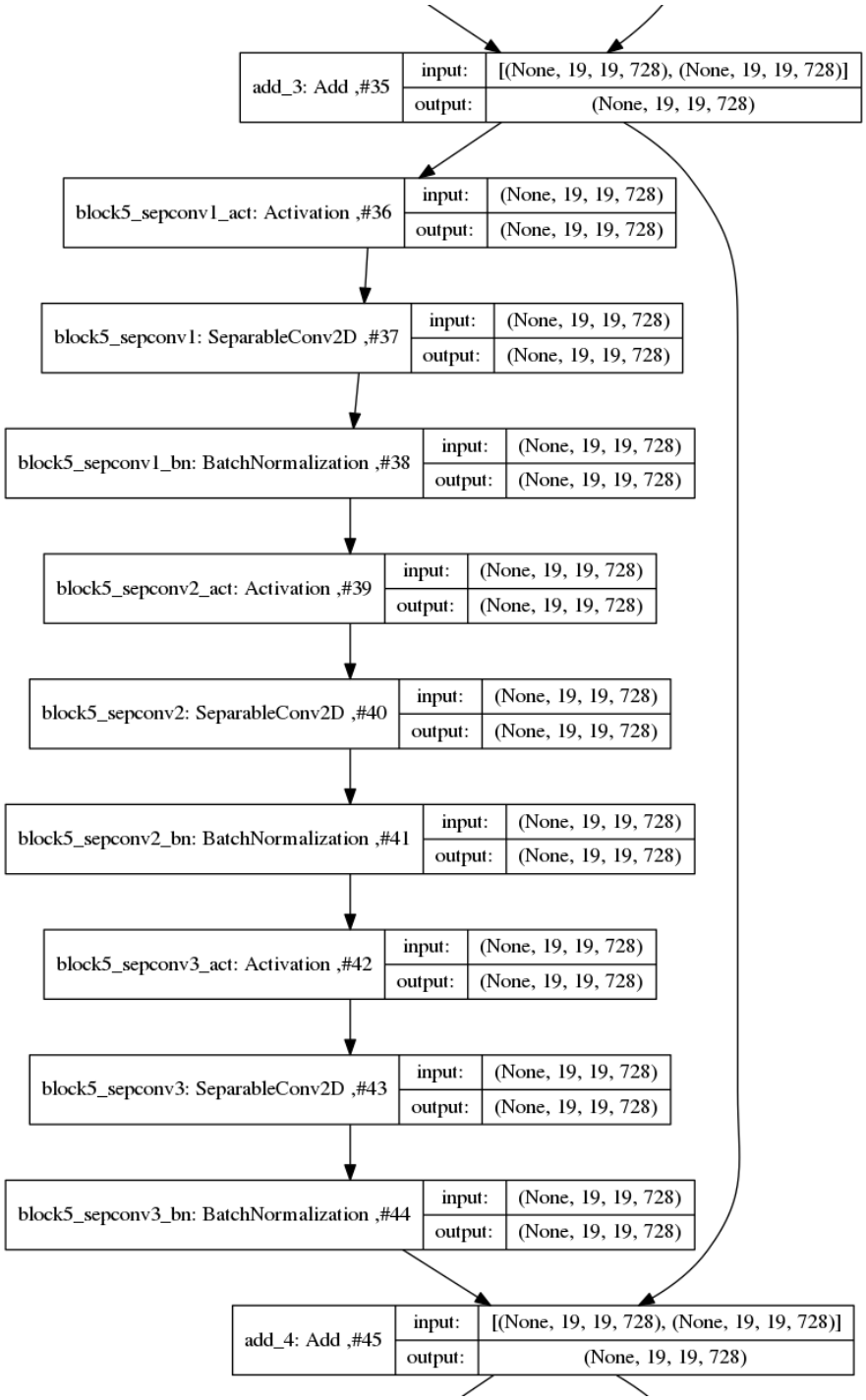


Figure A.10: Xception Block type 2

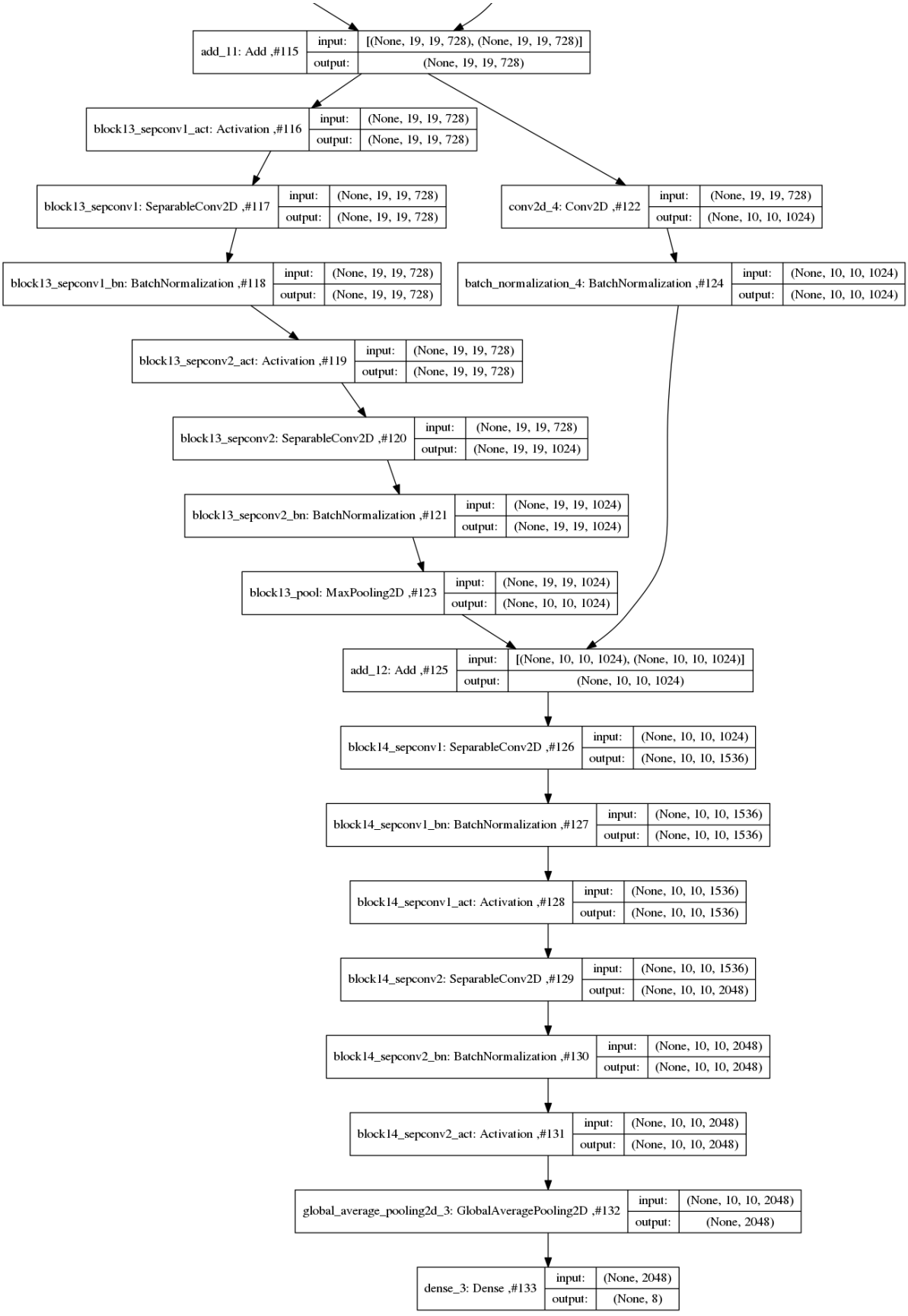


Figure A.11: Xception Output

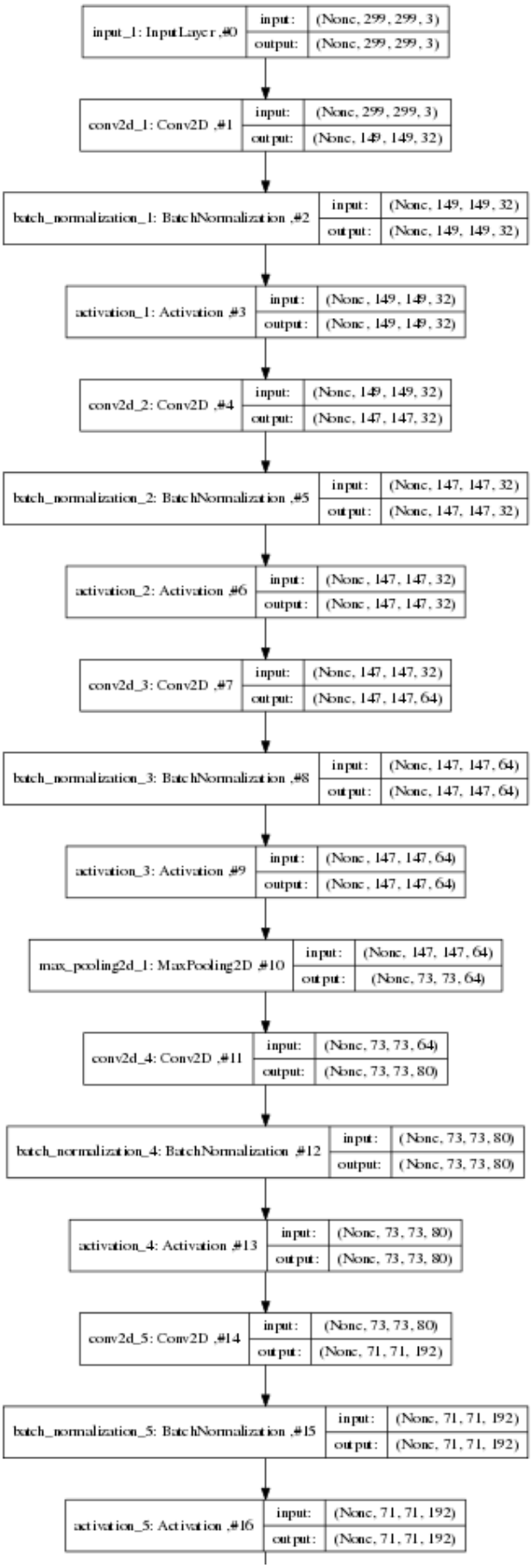


Figure A.12: Inception ResNet v2 Input

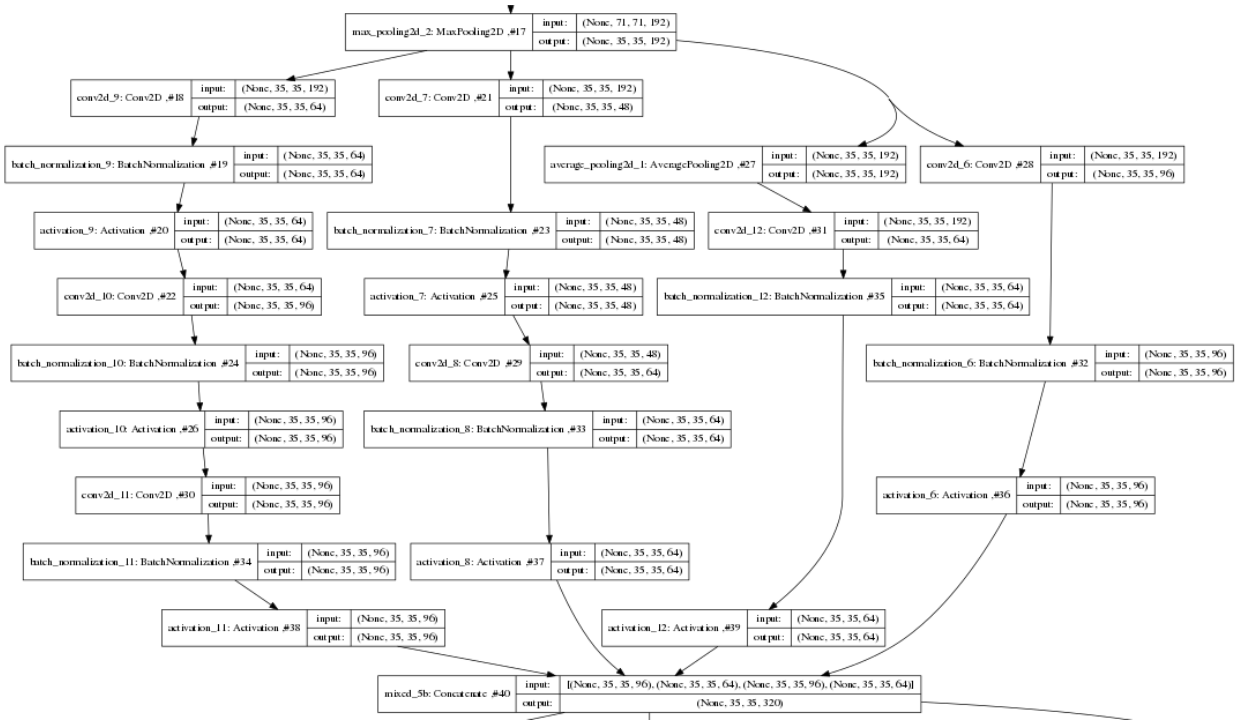


Figure A.13: Inception ResNet v2 Block type 1

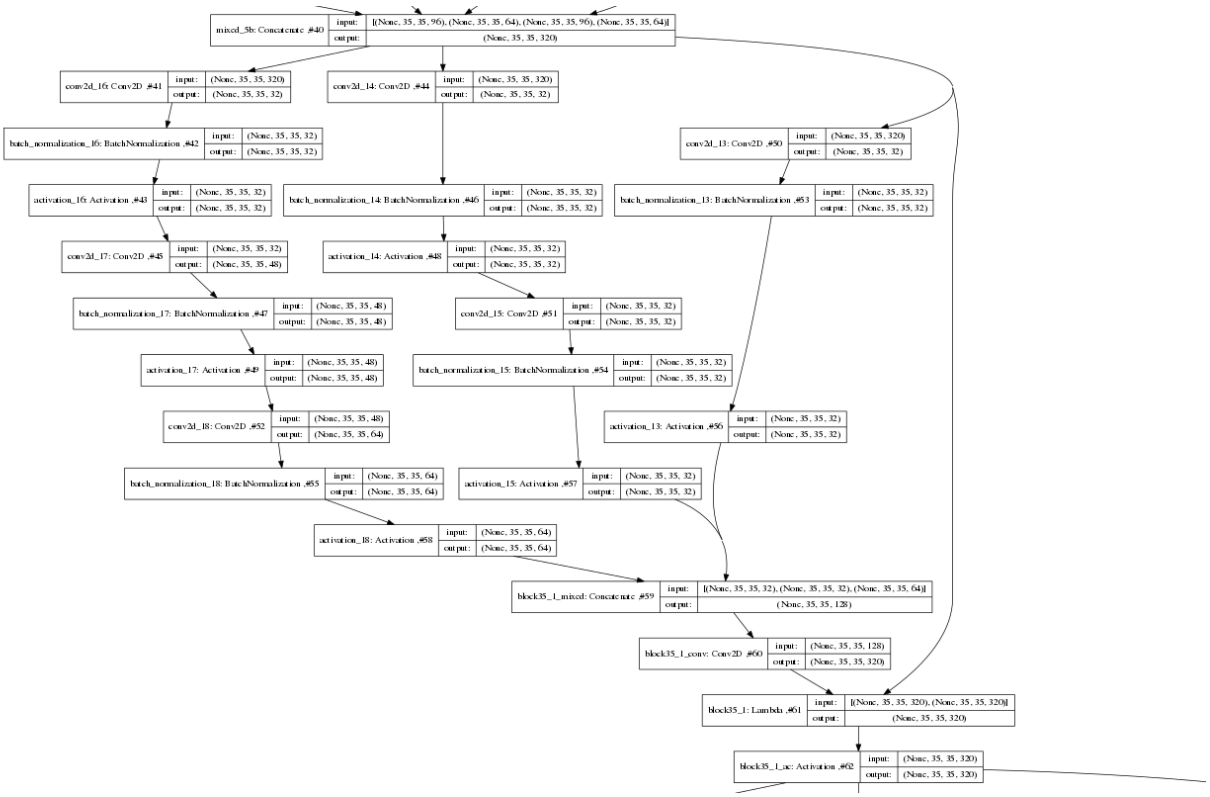


Figure A.14: Inception ResNet v2 Block type 2

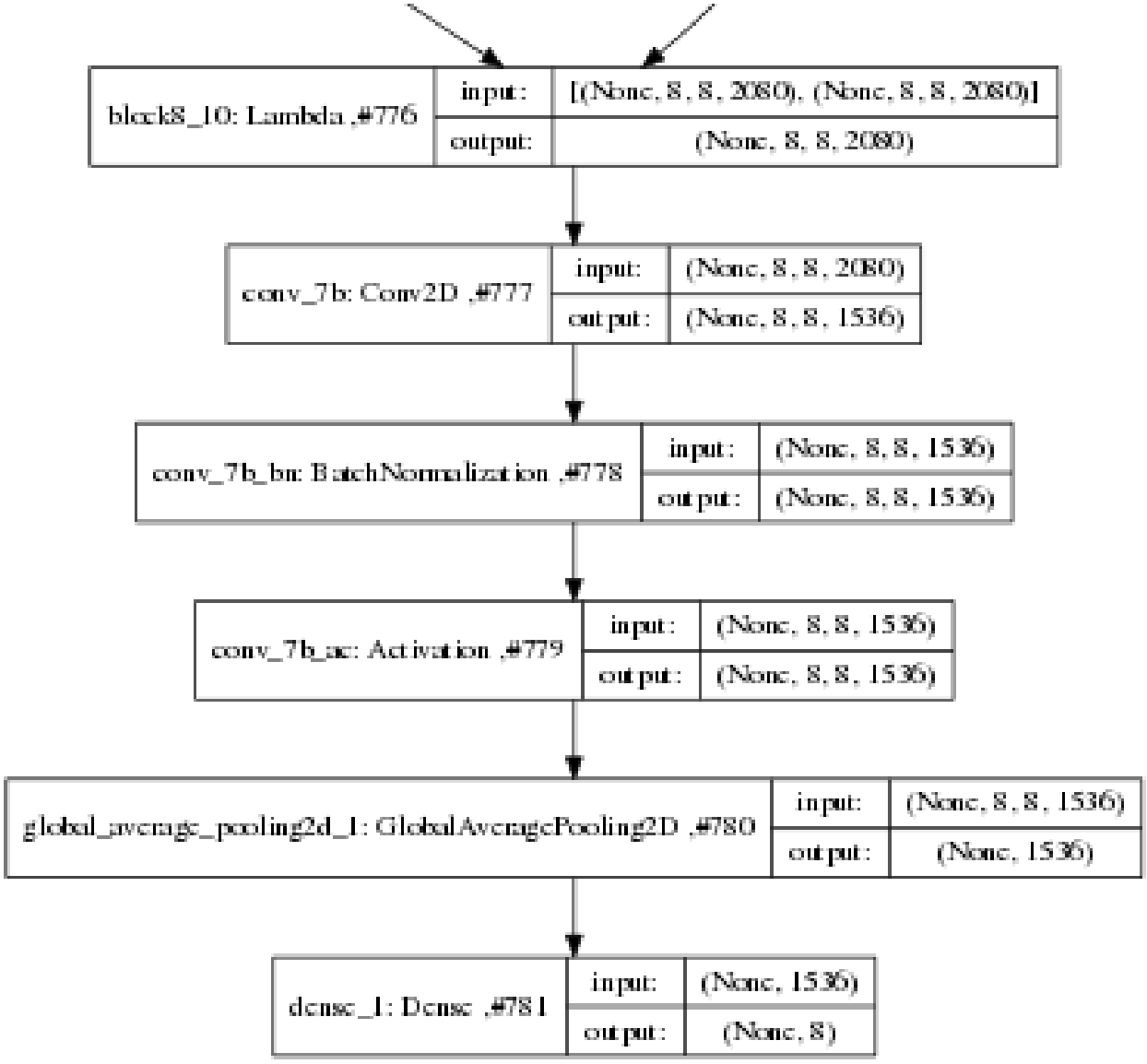


Figure A.15: Inception ResNet v2 Output

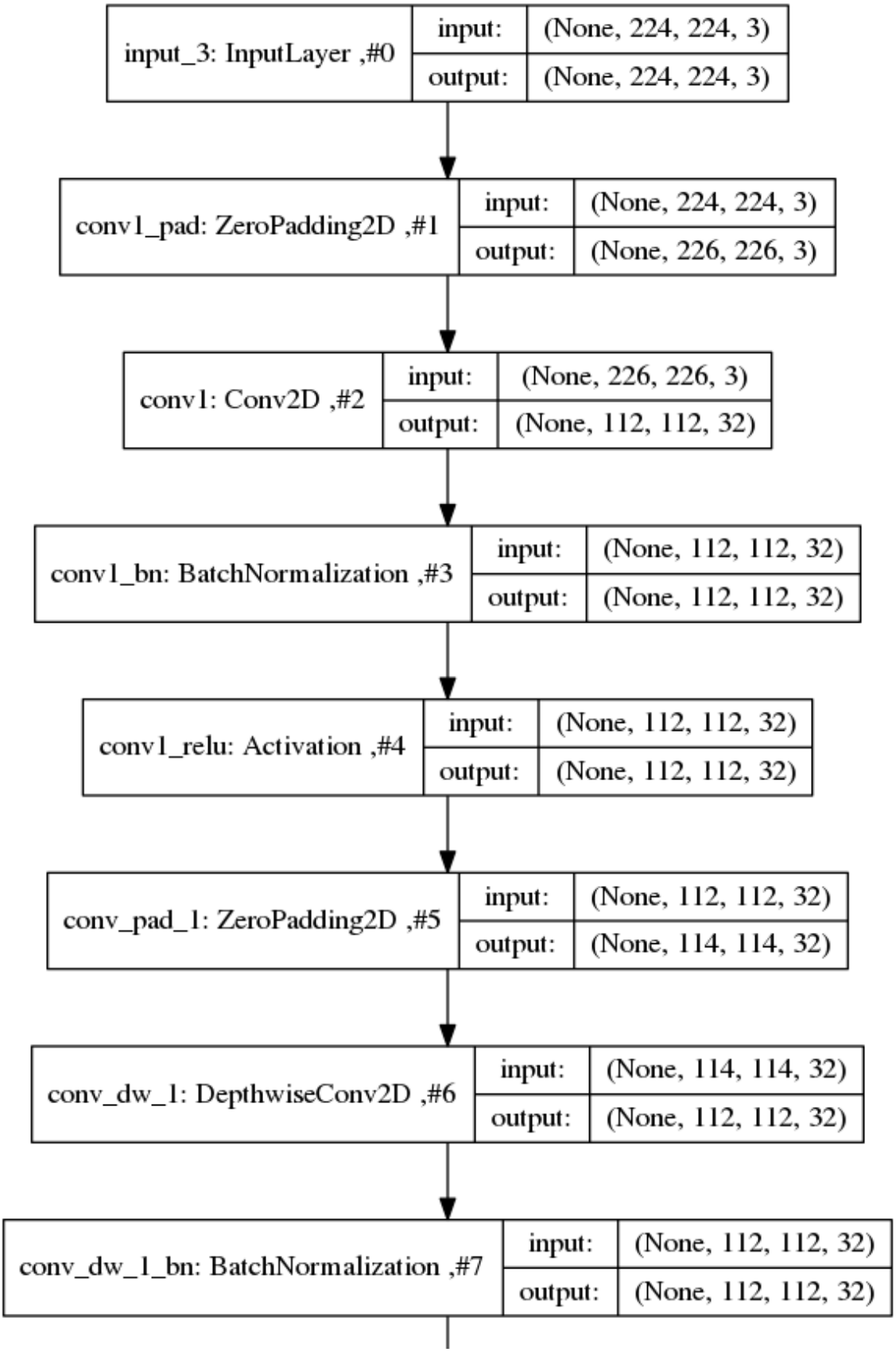


Figure A.16: MobileNet Input

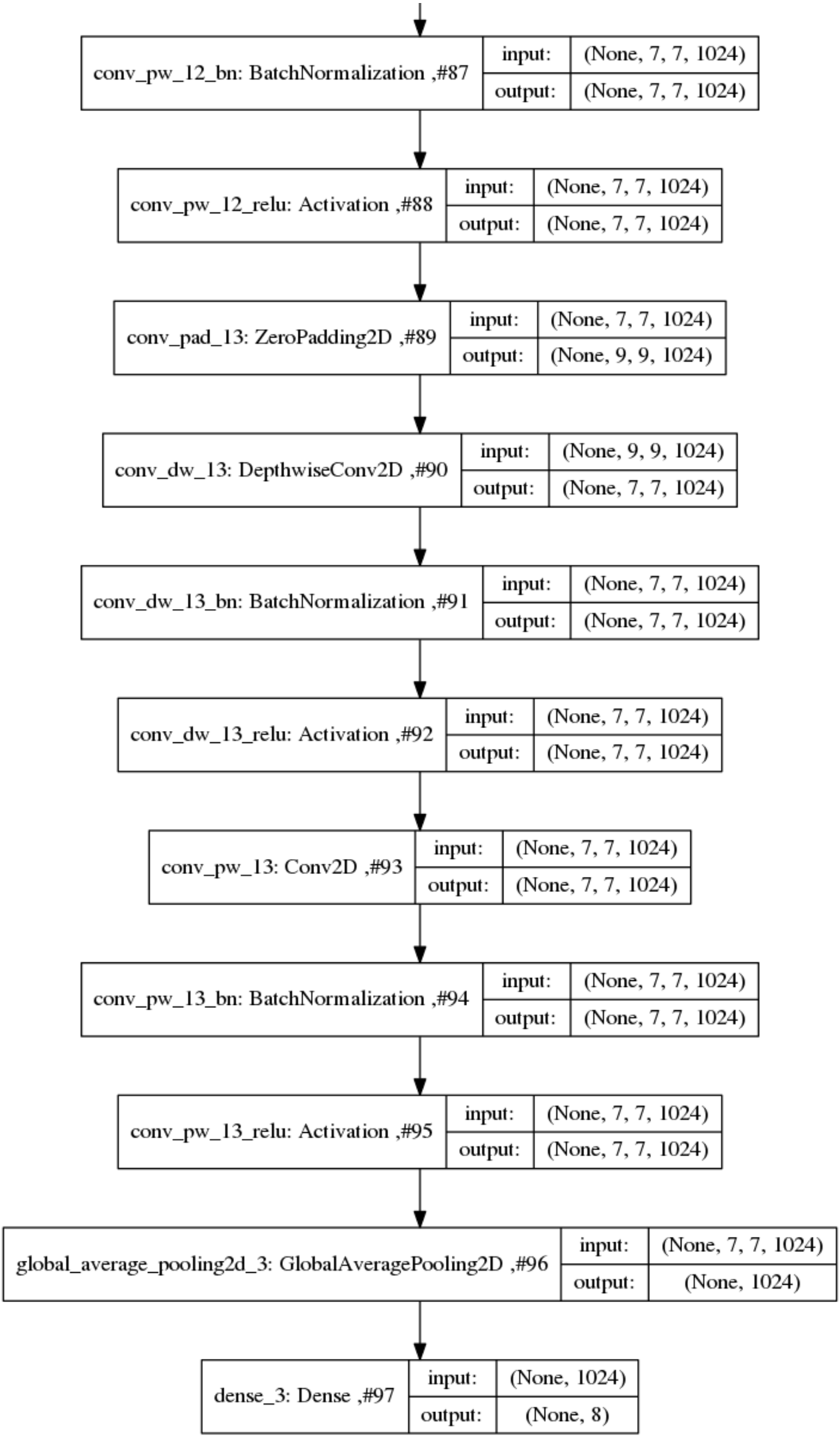


Figure A.17: MobileNet Output

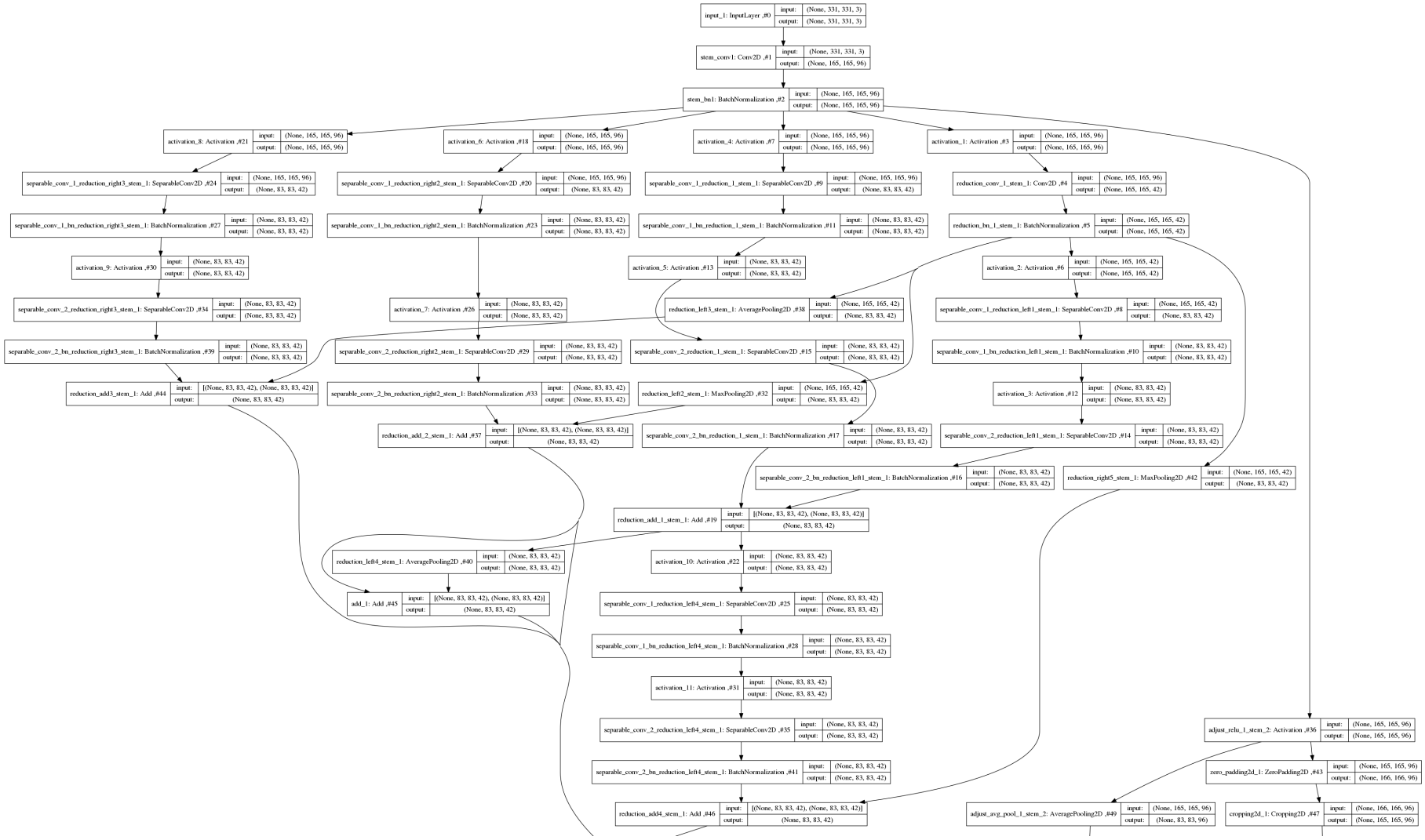


Figure A.18: NASNet Large Input 113

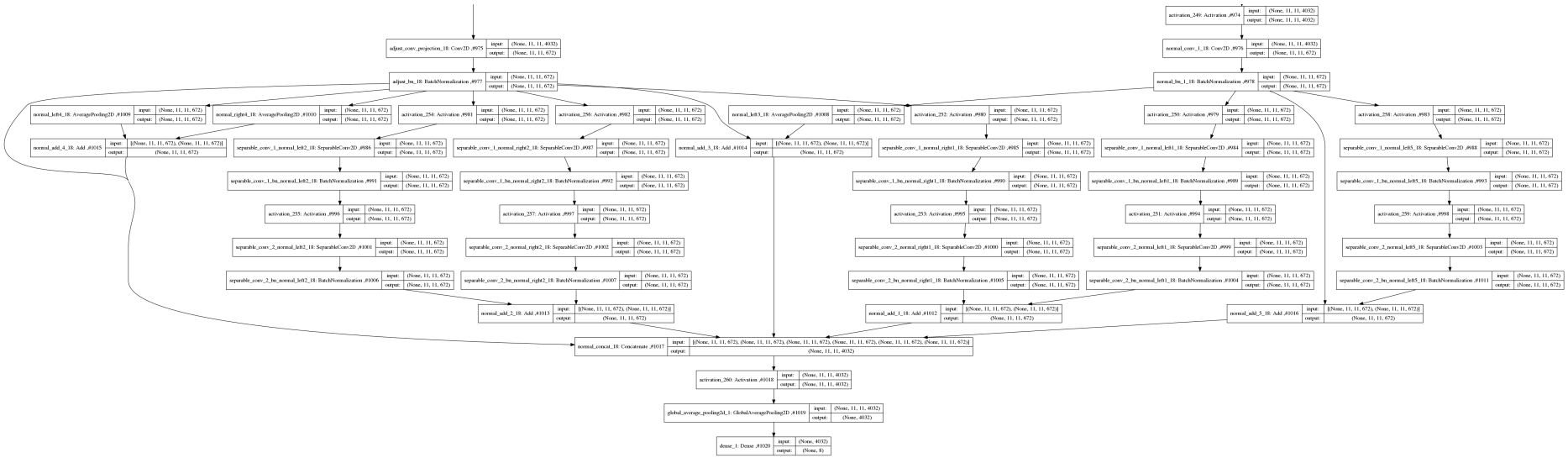


Figure A.19: NASNet Large Output

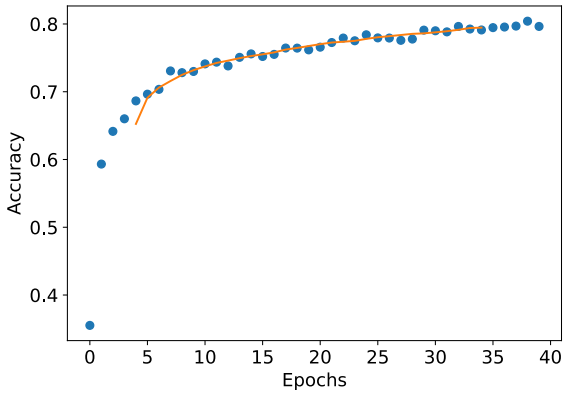
Figure A.20: NASNet Mobile Input

Figure A.21: NASNet Mobile Output

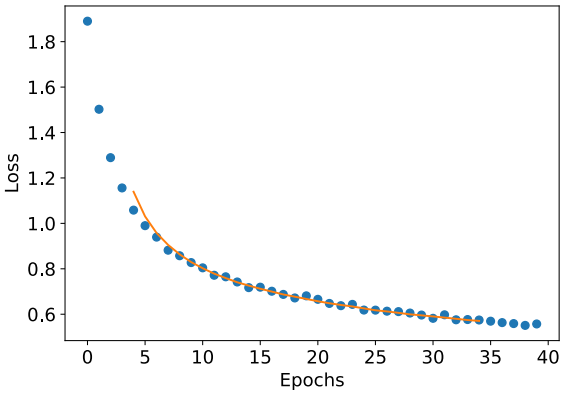
A.2 Training

A.2.1 VGG16

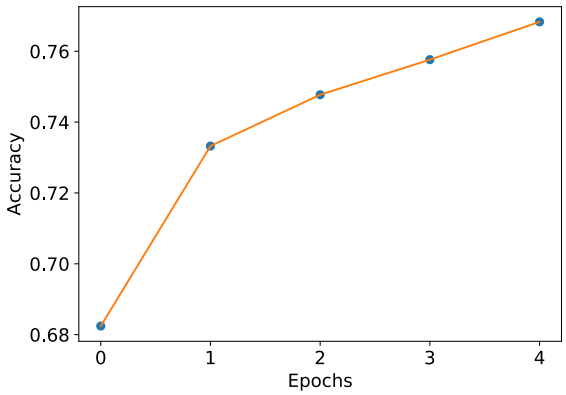
Classification Accuracy and Loss



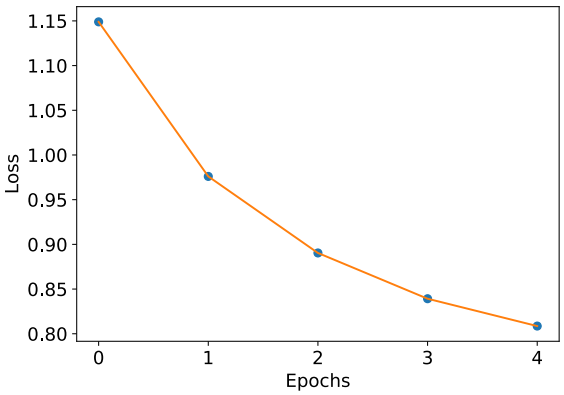
(a) Multiclass network accuracy



(b) Multiclass network loss



(c) Binary network "polyps" accuracy

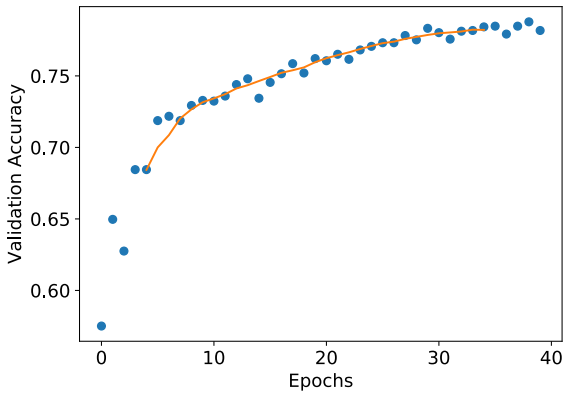


(d) Binary network "polyps" loss

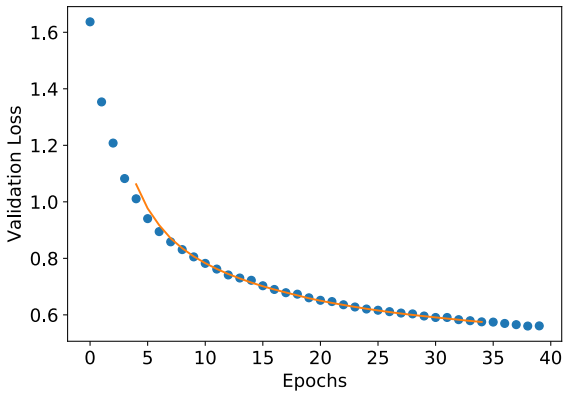
Figure A.22: VGG16 Training classification accuracy and loss history for training data

GPU Usage

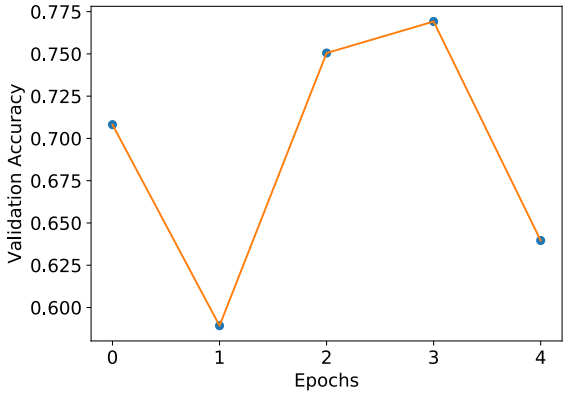
CPU and memory usage



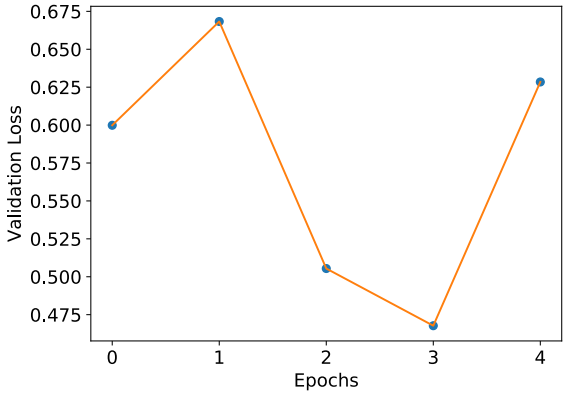
(a) Multiclass network accuracy



(b) Multiclass network loss

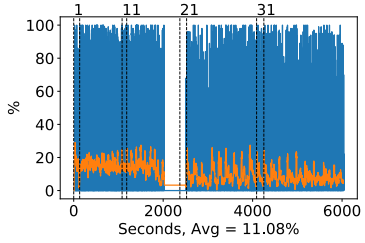


(c) Binary network "polyps" accuracy

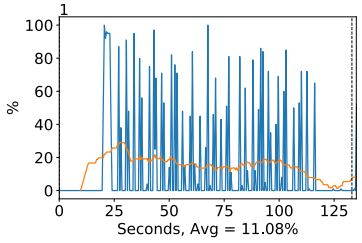


(d) Binary network "polyps" loss

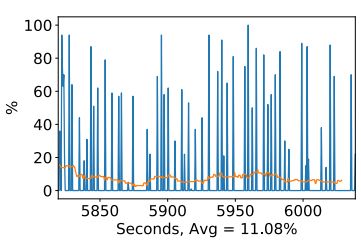
Figure A.23: VGG16 Training classification accuracy and loss history for validation data



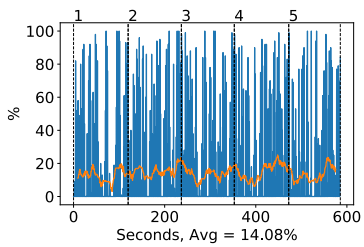
(a) Multiclass all epochs



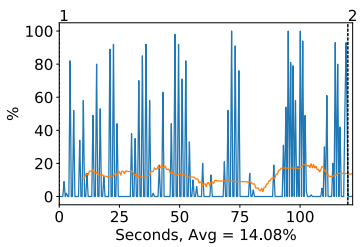
(b) Multiclass first epoch



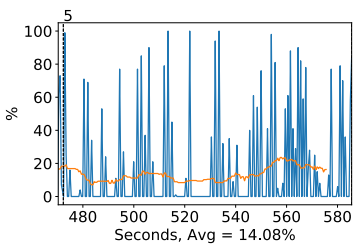
(c) Multiclass final epoch



(d) Binary all epochs

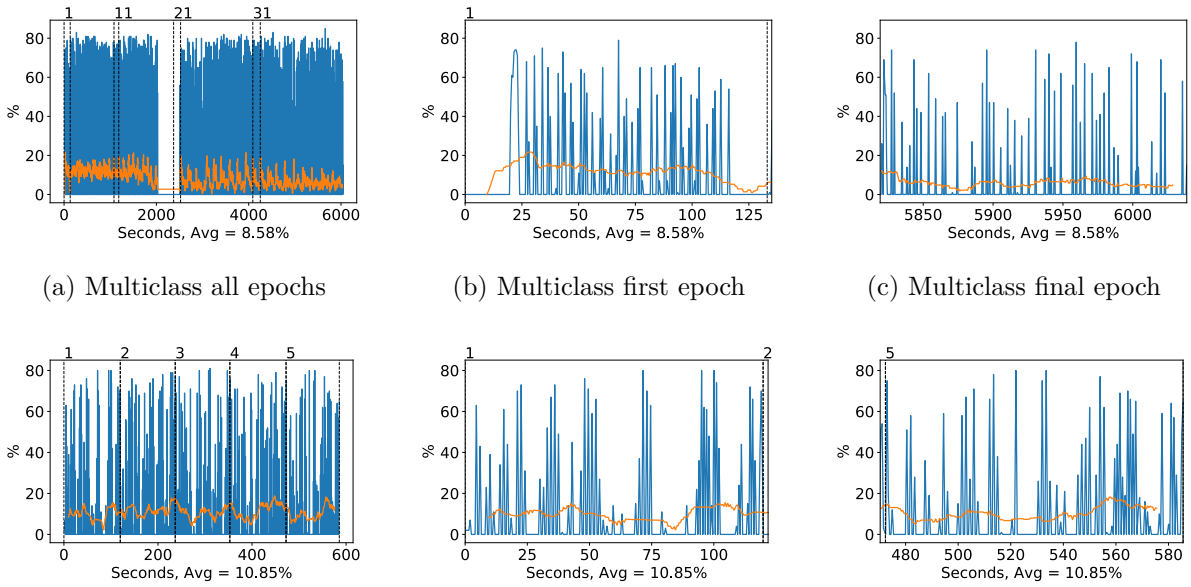


(e) Binary first epoch



(f) Binary final epoch

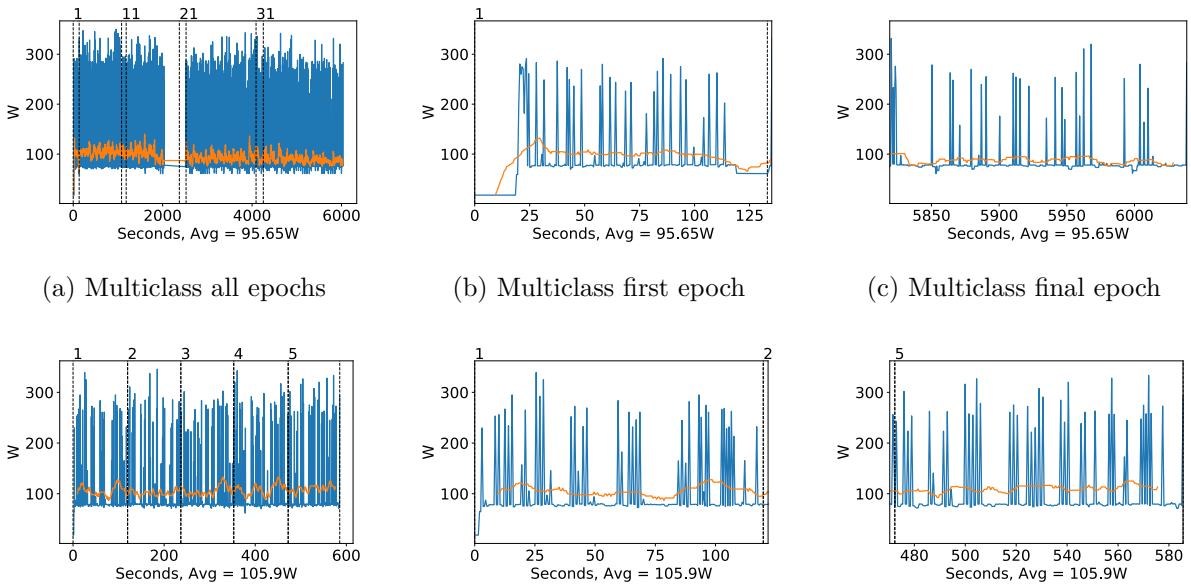
Figure A.24: VGG16 GPU volatile usage during training, with a moving window average over 40 measurements overlaid



(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

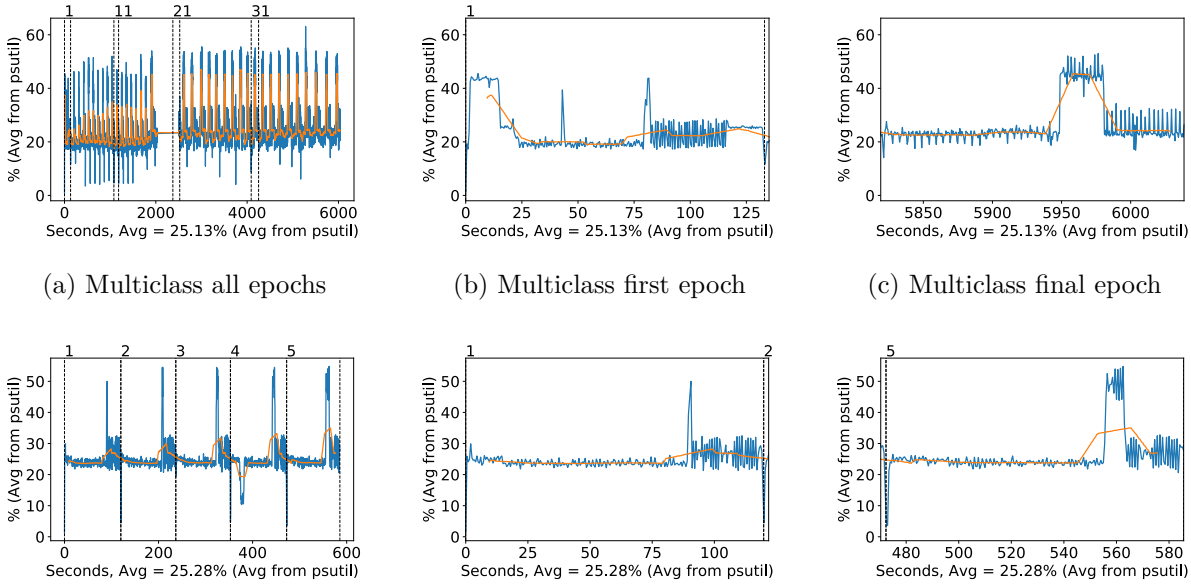
Figure A.25: VGG16 GPU memory usage during training, with a moving window average over 40 measurements overlaid



(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

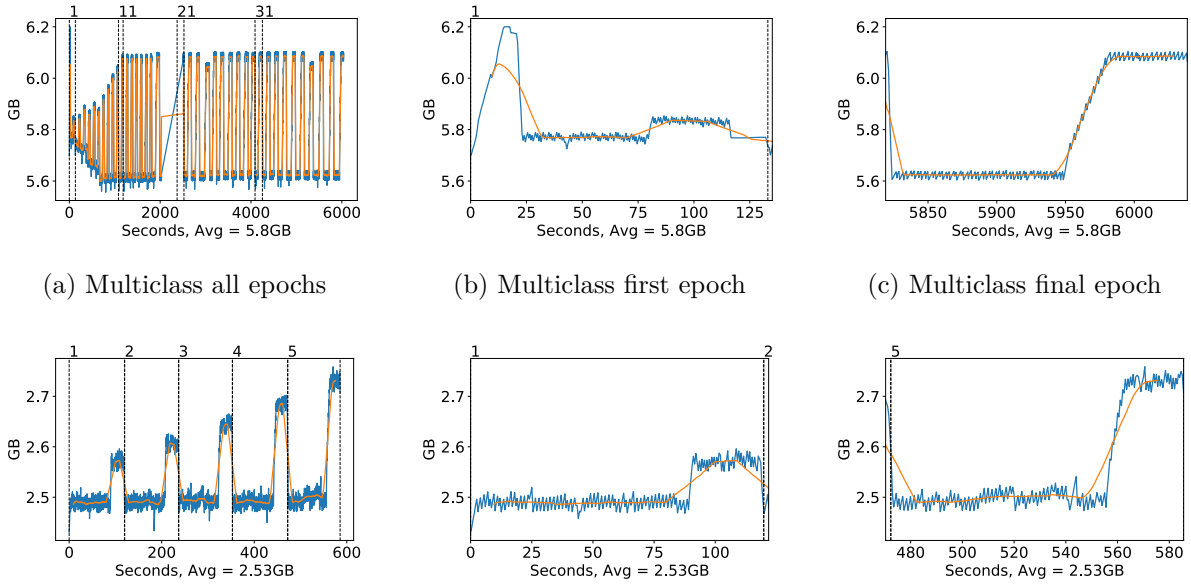
Figure A.26: VGG16 GPU power usage during training in W, with a moving window average over 40 measurements overlaid



(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

Figure A.27: VGG16 CPU usage (averaged over 4 cores) during training, with a moving window average over 40 measurements overlaid



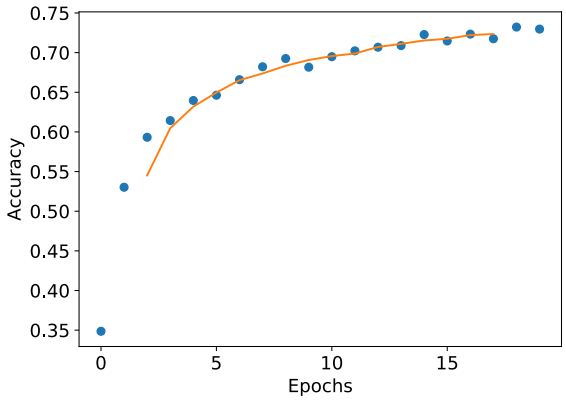
(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

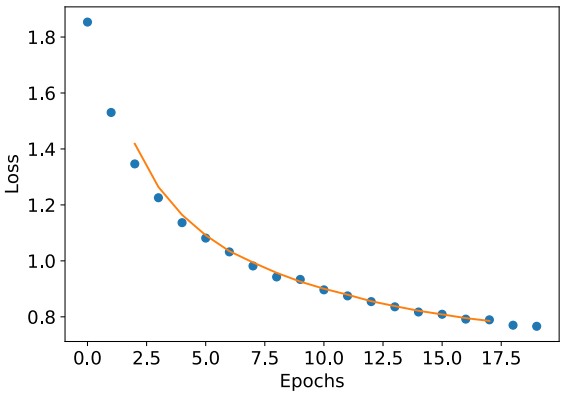
Figure A.28: VGG16 Memory usage during training, with a moving window average over 40 measurements overlaid

A.2.2 VGG19

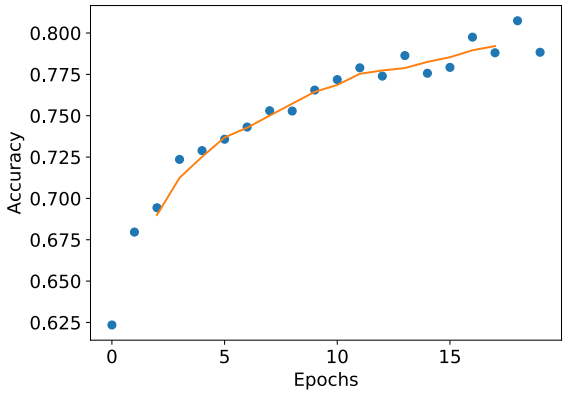
Classification Accuracy and Loss



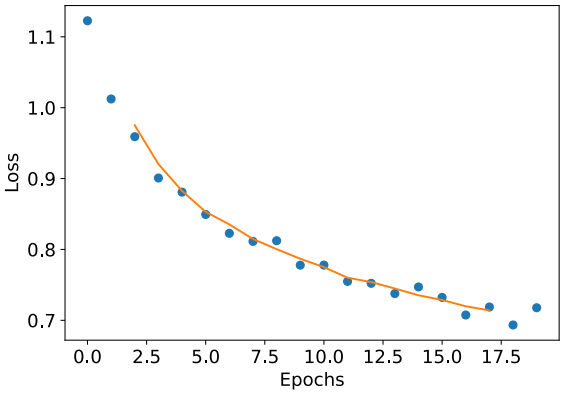
(a) Multiclass network accuracy



(b) Multiclass network loss



(c) Binary network "polyps" accuracy

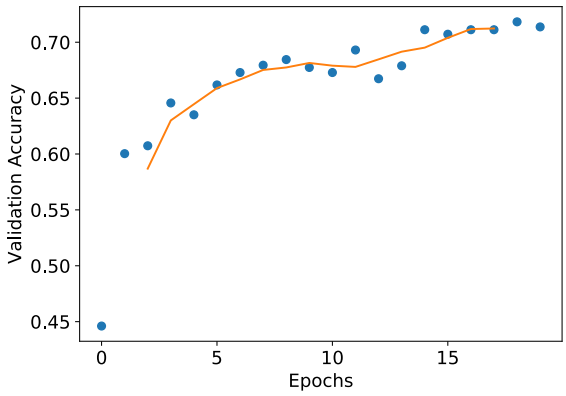


(d) Binary network "polyps" loss

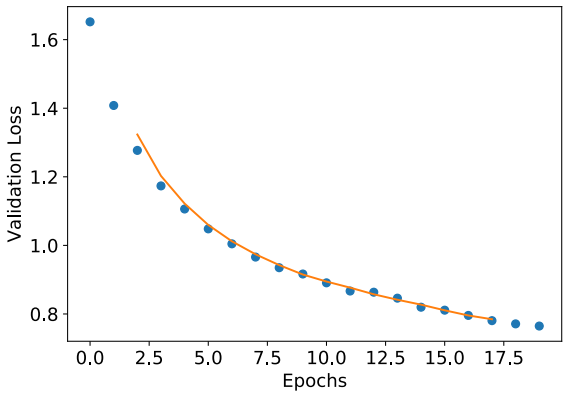
Figure A.29: VGG19 Training classification accuracy and loss history for training data

GPU Usage

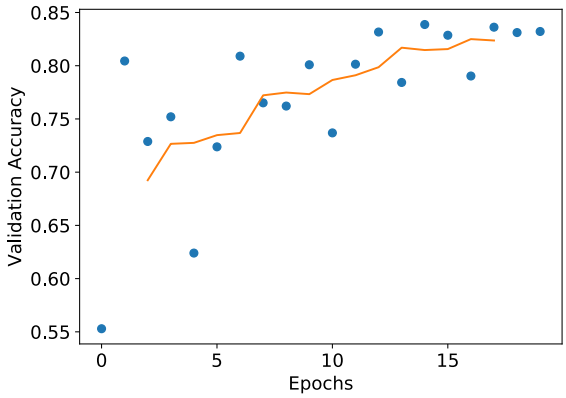
CPU and memory usage



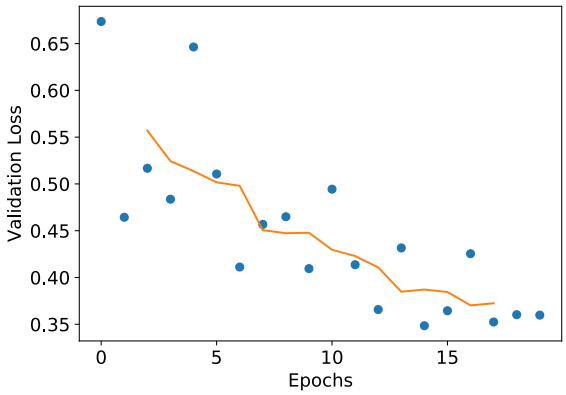
(a) Multiclass network accuracy



(b) Multiclass network loss

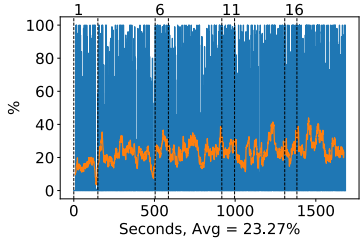


(c) Binary network "polyps" accuracy

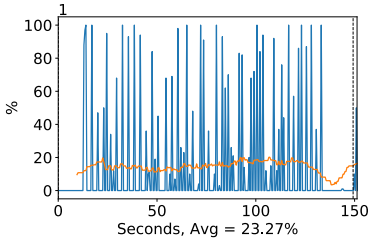


(d) Binary network "polyps" loss

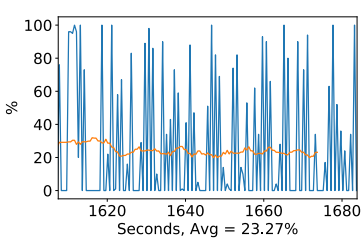
Figure A.30: VGG19 Training classification accuracy and loss history for validation data



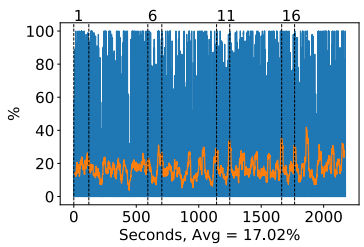
(a) Multiclass all epochs



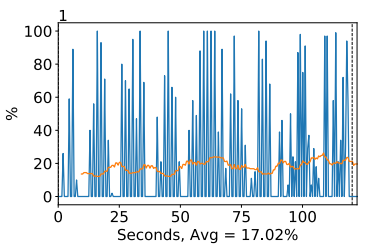
(b) Multiclass first epoch



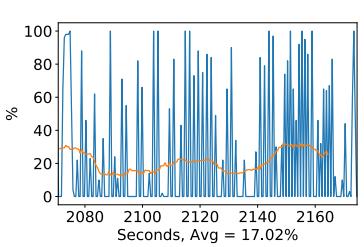
(c) Multiclass final epoch



(d) Binary all epochs

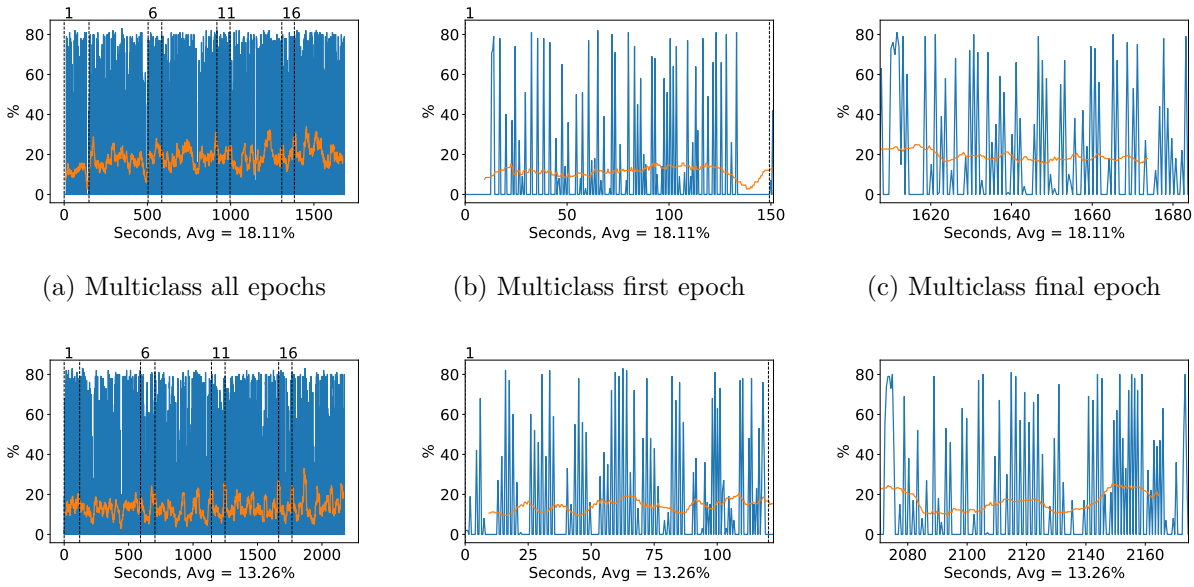


(e) Binary first epoch



(f) Binary final epoch

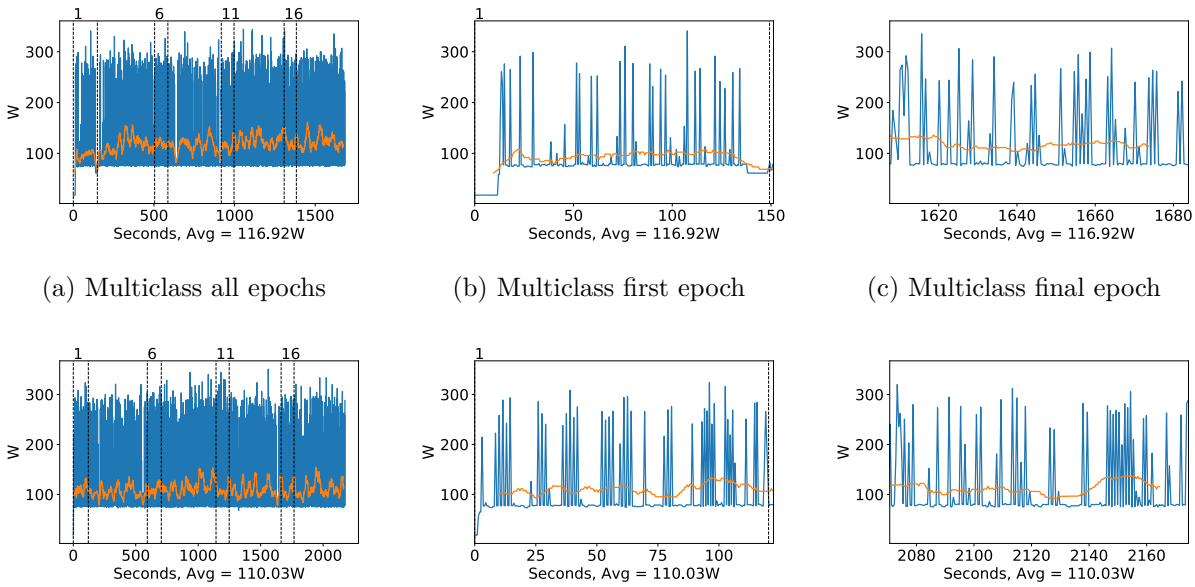
Figure A.31: VGG19 GPU volatile usage during training, with a moving window average over 40 measurements overlaid



(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

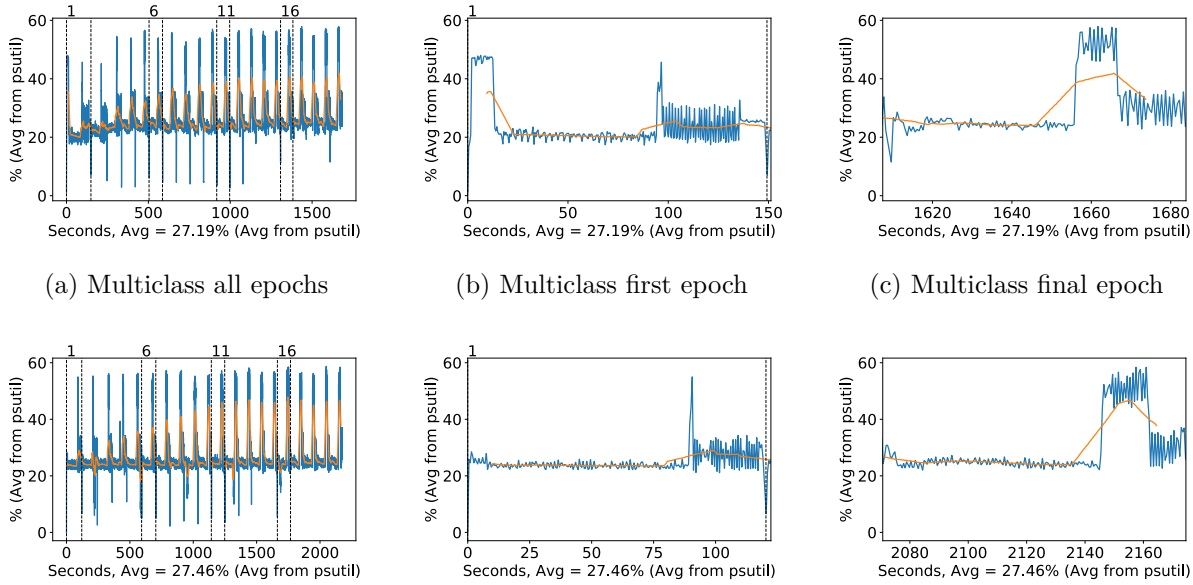
Figure A.32: VGG19 GPU memory usage during training, with a moving window average over 40 measurements overlaid



(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

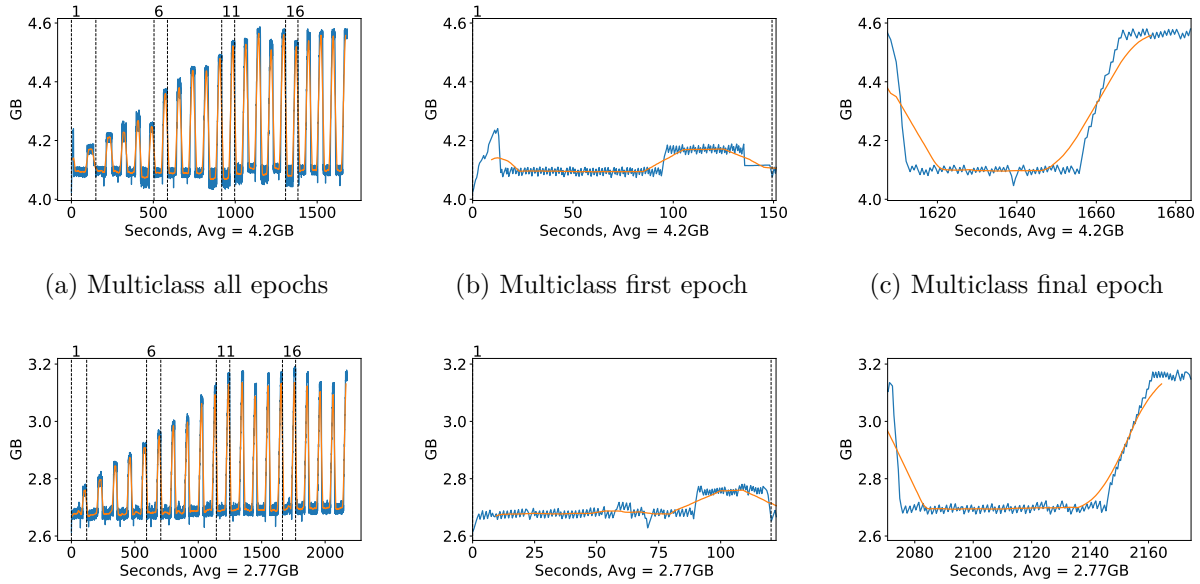
Figure A.33: VGG19 GPU power usage during training in W, with a moving window average over 40 measurements overlaid



(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

Figure A.34: VGG19 CPU usage (averaged over 4 cores) during training, with a moving window average over 40 measurements overlaid



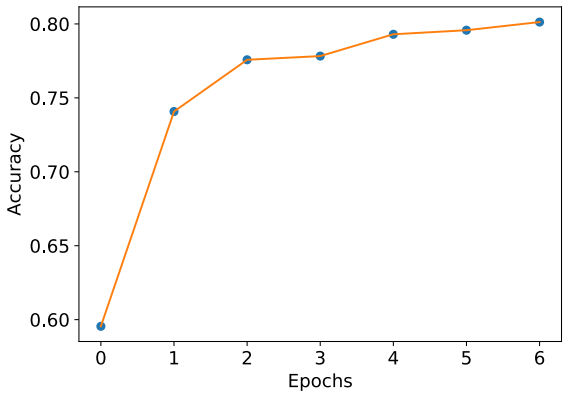
(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

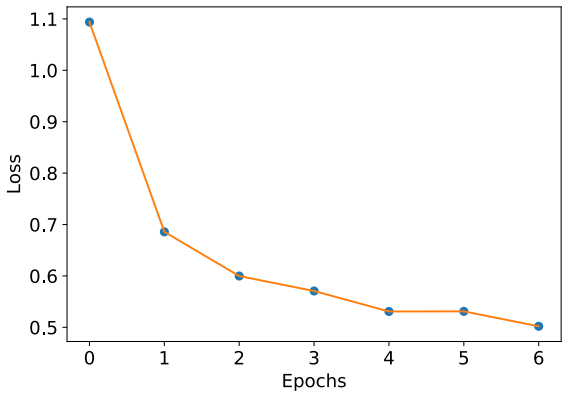
Figure A.35: VGG19 Memory usage during training, with a moving window average over 40 measurements overlaid

A.2.3 Inception v3

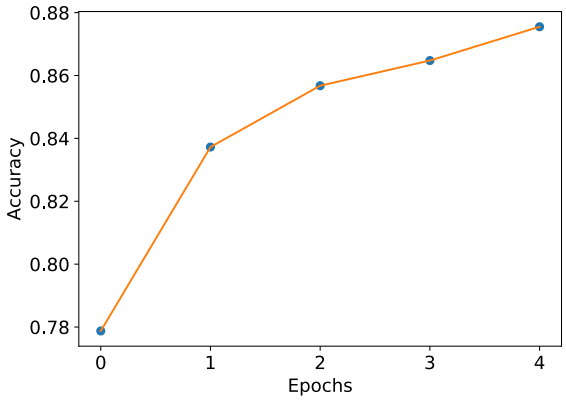
Classification Accuracy and Loss



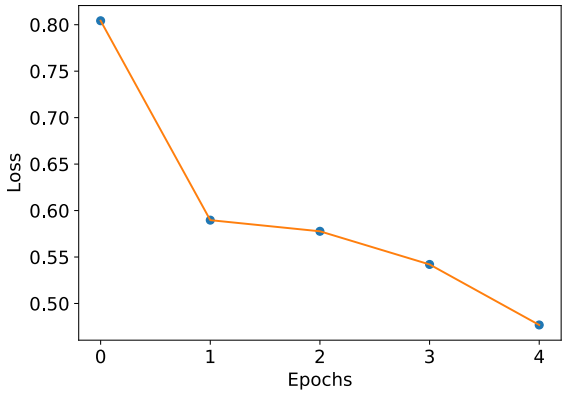
(a) Multiclass network accuracy



(b) Multiclass network loss



(c) Binary network "polyps" accuracy

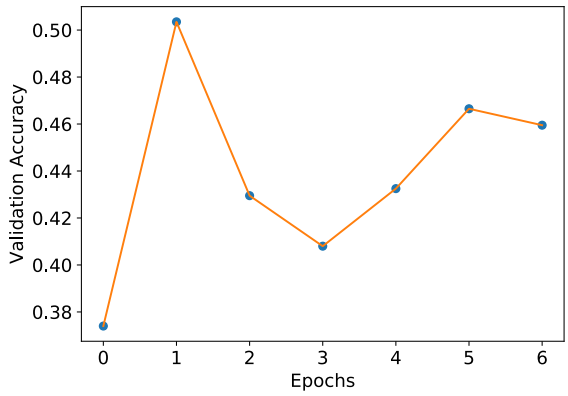


(d) Binary network "polyps" loss

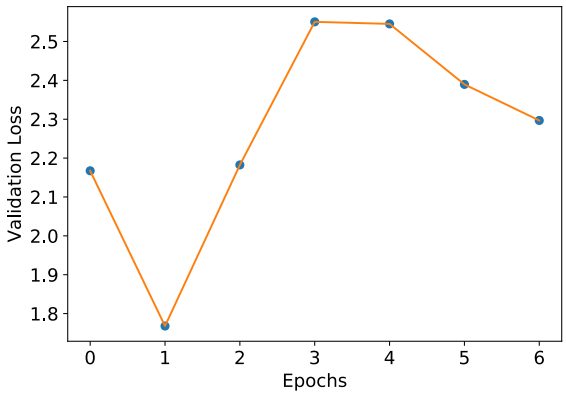
Figure A.36: Inception v3 Training classification accuracy and loss history for training data

GPU Usage

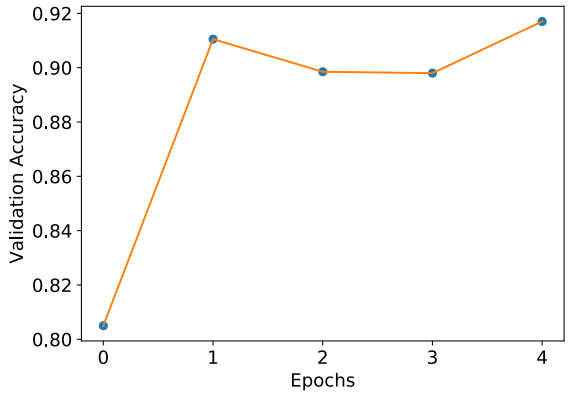
CPU and memory usage



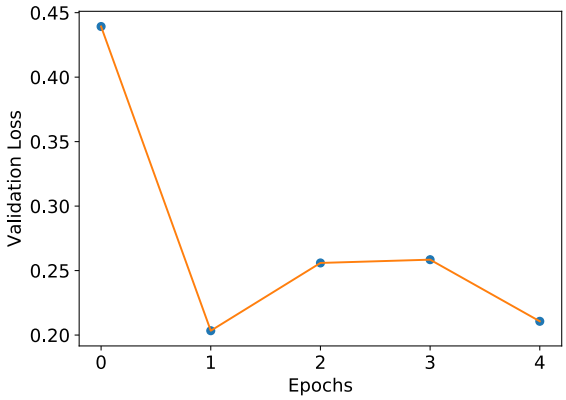
(a) Multiclass network accuracy



(b) Multiclass network loss

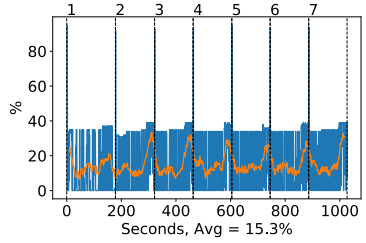


(c) Binary network "polyps" accuracy

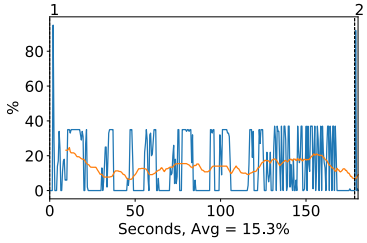


(d) Binary network "polyps" loss

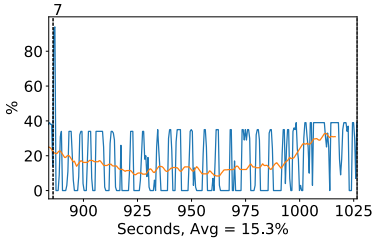
Figure A.37: Inception v3 Training classification accuracy and loss history for validation data



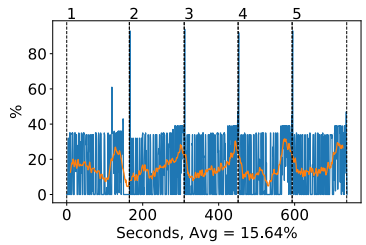
(a) Multiclass all epochs



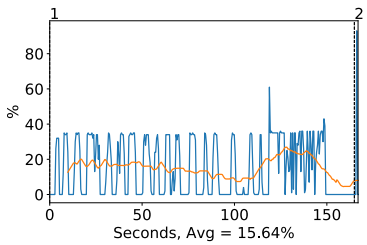
(b) Multiclass first epoch



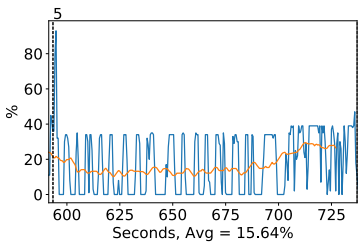
(c) Multiclass final epoch



(d) Binary all epochs

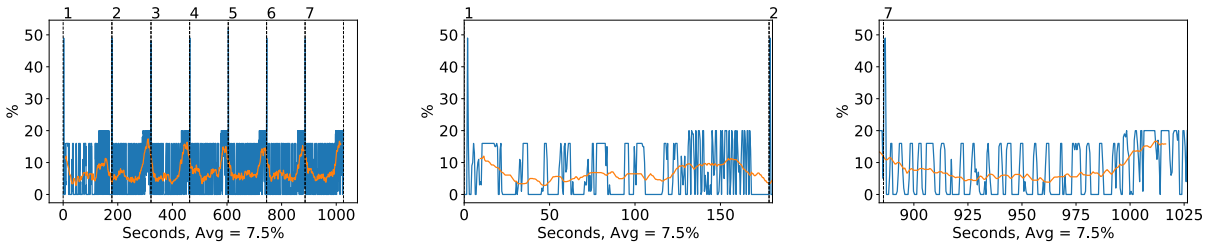


(e) Binary first epoch

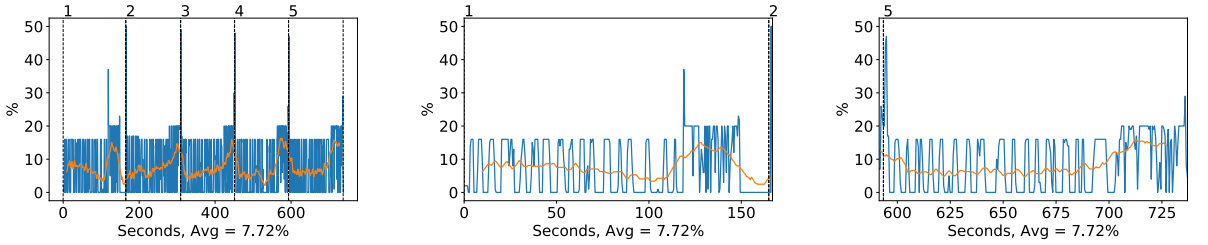


(f) Binary final epoch

Figure A.38: Inception v3 GPU volatile usage during training, with a moving window average over 40 measurements overlaid

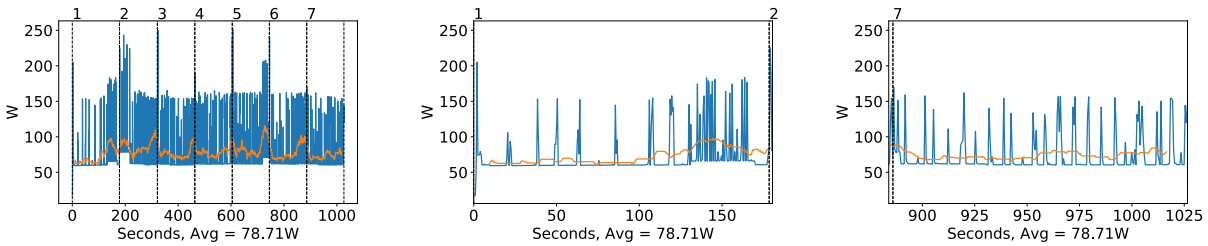


(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

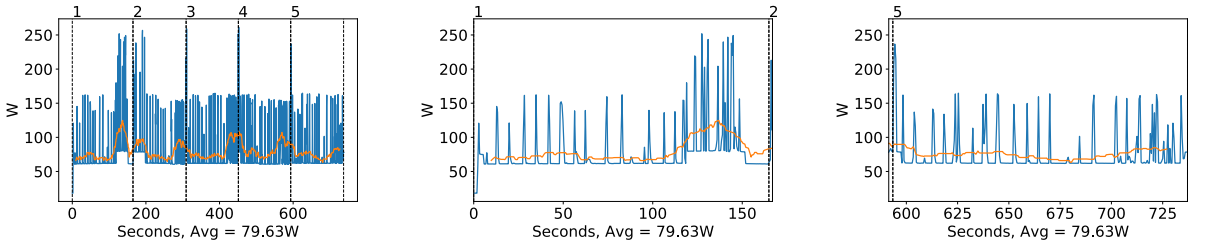


(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

Figure A.39: Inception v3 GPU memory usage during training, with a moving window average over 40 measurements overlaid

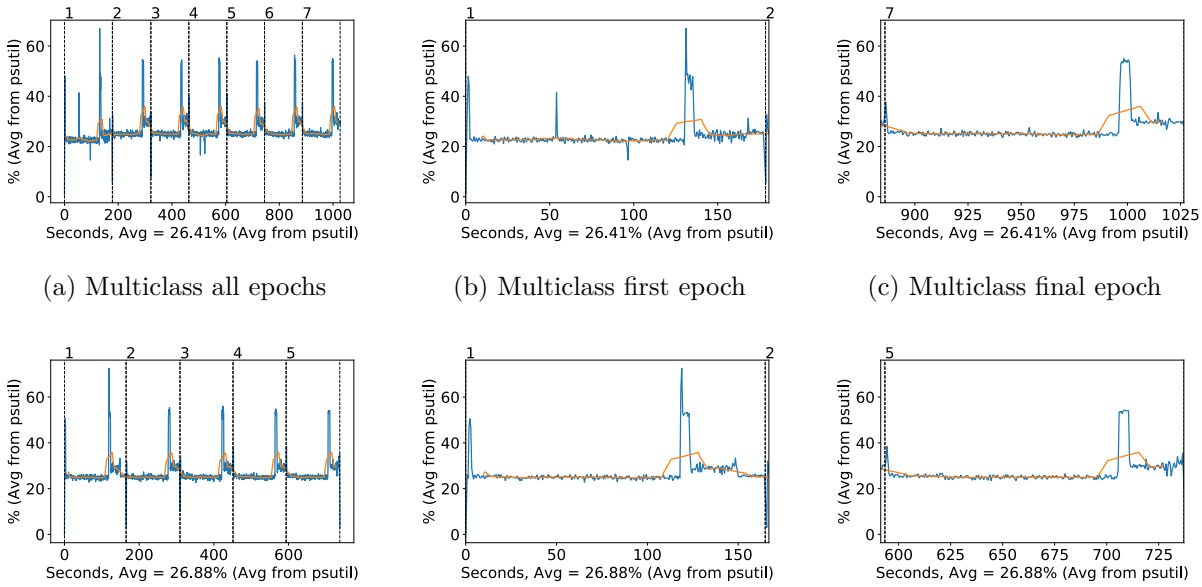


(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch



(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

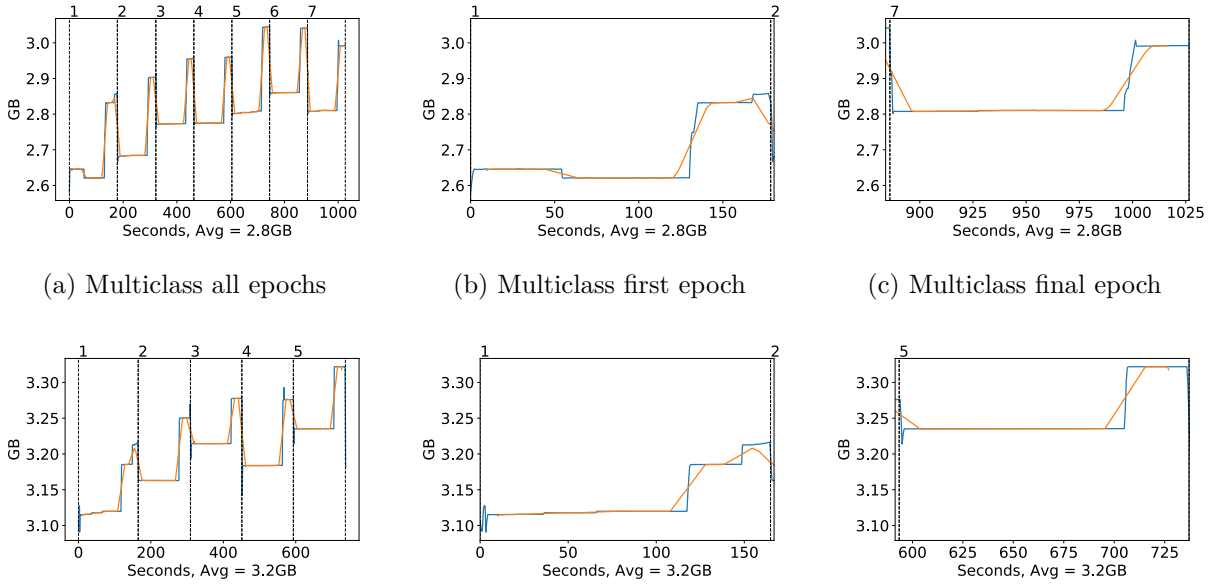
Figure A.40: Inception v3 GPU power usage during training in W, with a moving window average over 40 measurements overlaid



(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

Figure A.41: Inception v3 CPU usage (averaged over 4 cores) during training, with a moving window average over 40 measurements overlaid



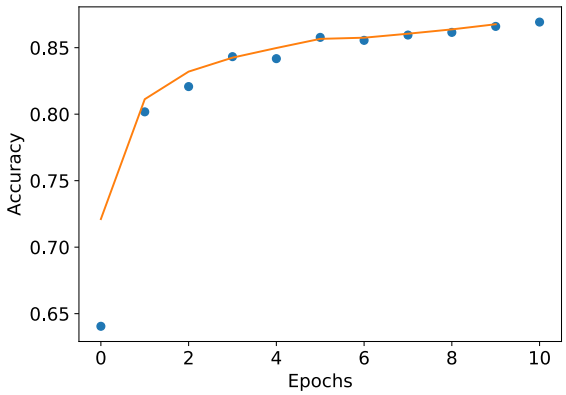
(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

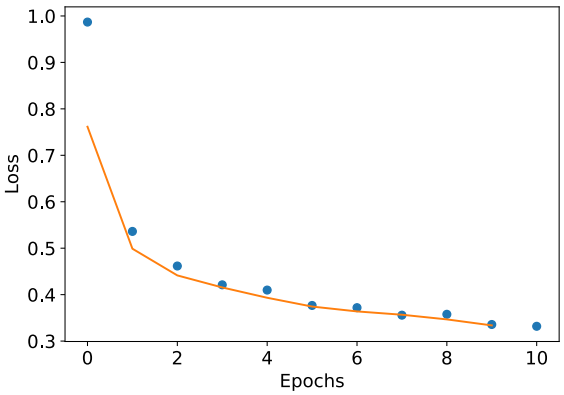
Figure A.42: Inception v3 Memory usage during training, with a moving window average over 40 measurements overlaid

A.2.4 DenseNet 121

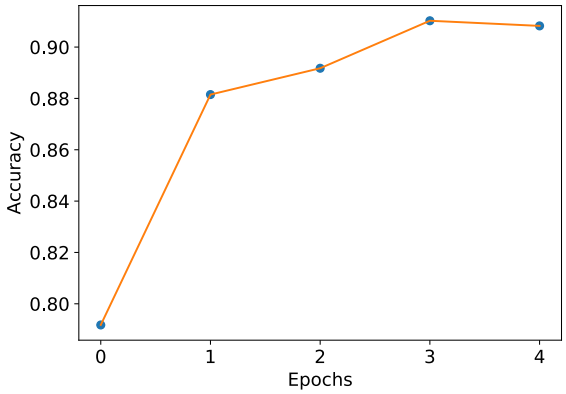
Classification Accuracy and Loss



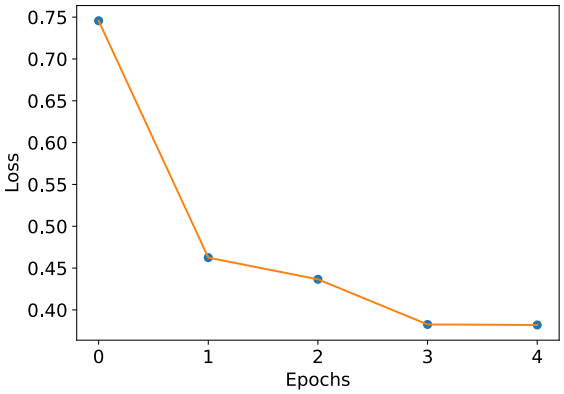
(a) Multiclass network accuracy



(b) Multiclass network loss



(c) Binary network "polyps" accuracy

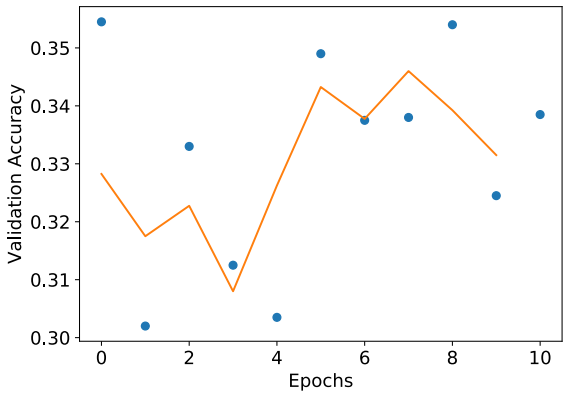


(d) Binary network "polyps" loss

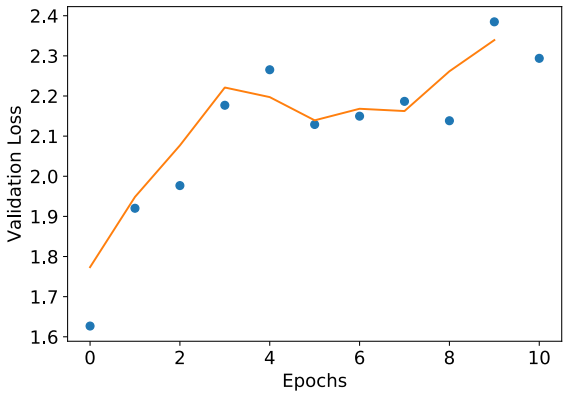
Figure A.43: DenseNet 121 Training classification accuracy and loss history for training data

GPU Usage

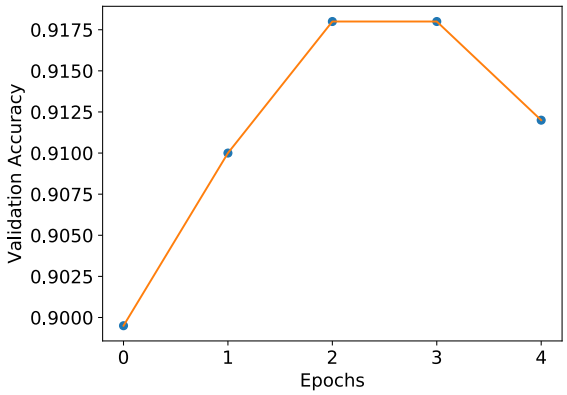
CPU and memory usage



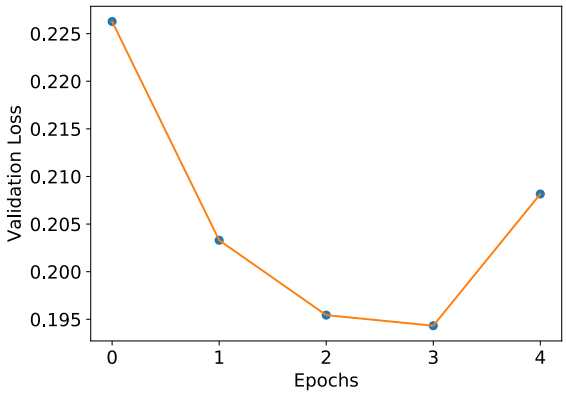
(a) Multiclass network accuracy



(b) Multiclass network loss

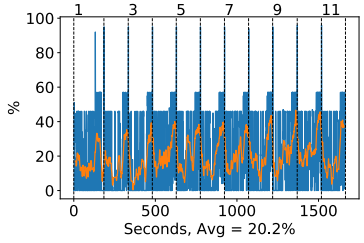


(c) Binary network "polyps" accuracy

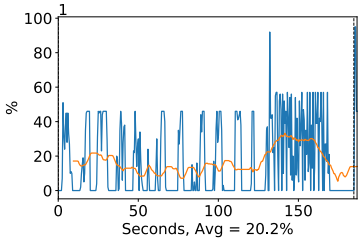


(d) Binary network "polyps" loss

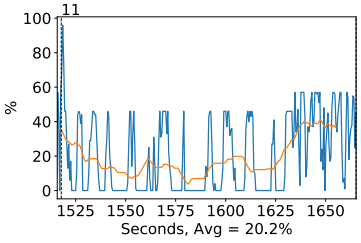
Figure A.44: DenseNet 121 Training classification accuracy and loss history for validation data



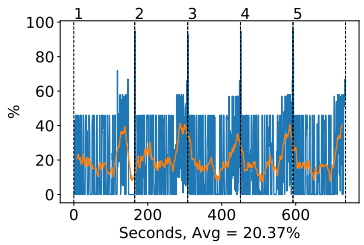
(a) Multiclass all epochs



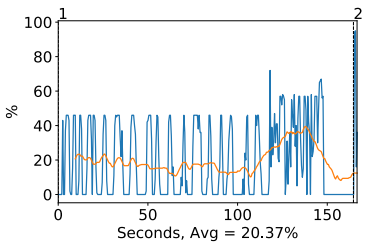
(b) Multiclass first epoch



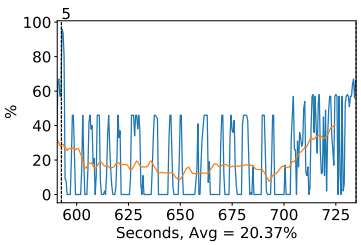
(c) Multiclass final epoch



(d) Binary all epochs

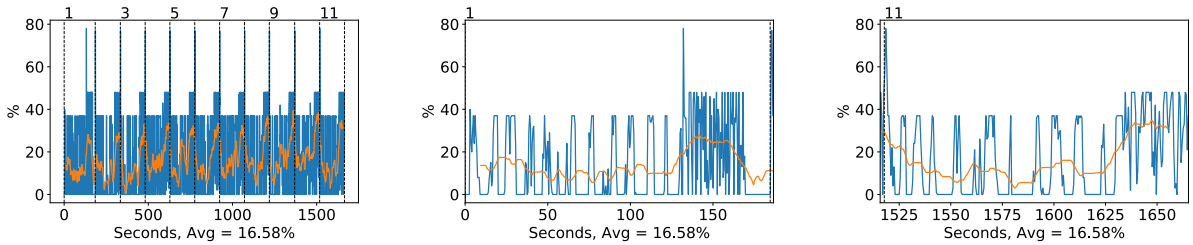


(e) Binary first epoch

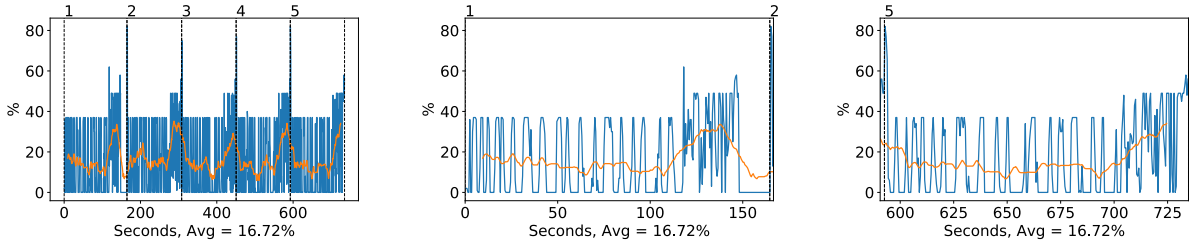


(f) Binary final epoch

Figure A.45: DenseNet 121 GPU volatile usage during training, with a moving window average over 40 measurements overlaid

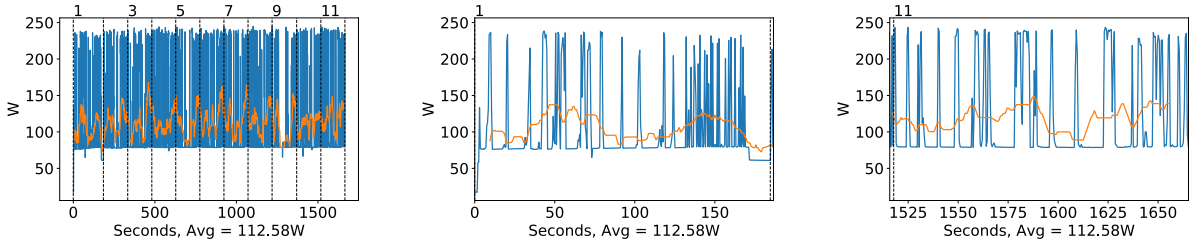


(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

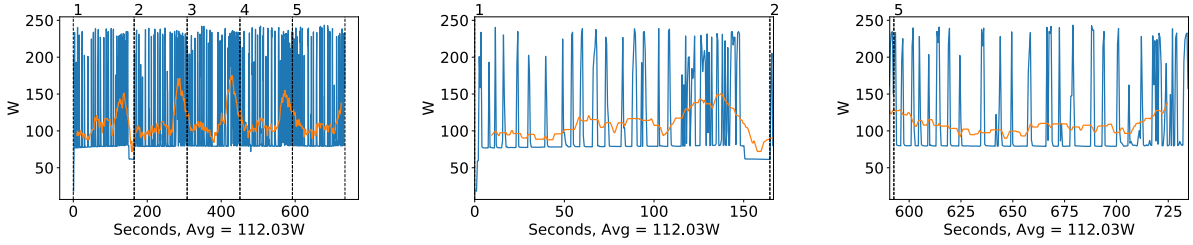


(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

Figure A.46: DenseNet 121 GPU memory usage during training, with a moving window average over 40 measurements overlaid

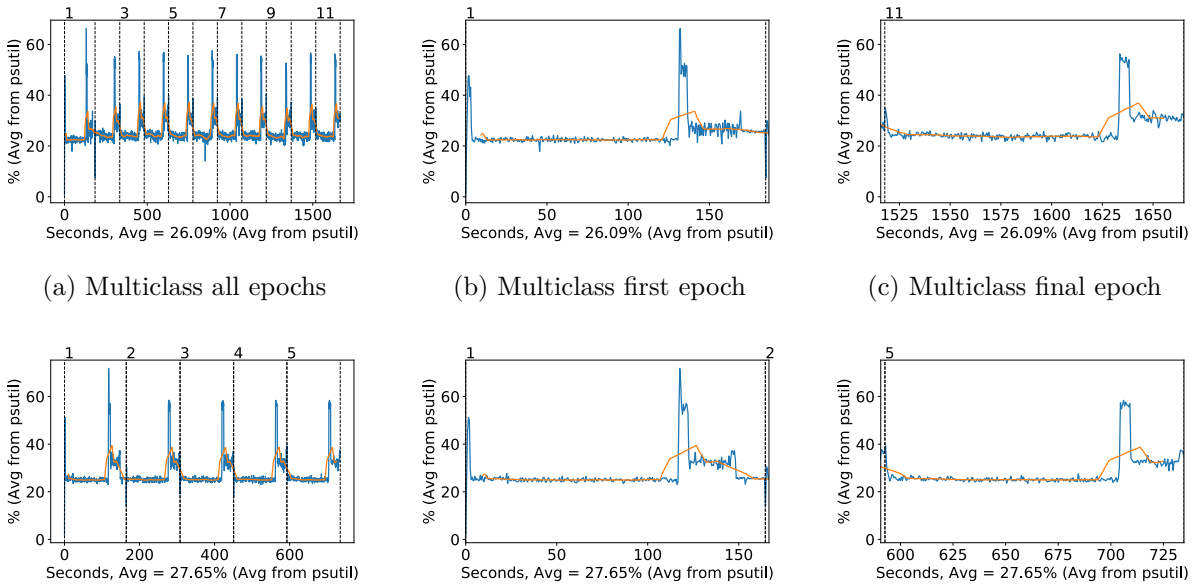


(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch



(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

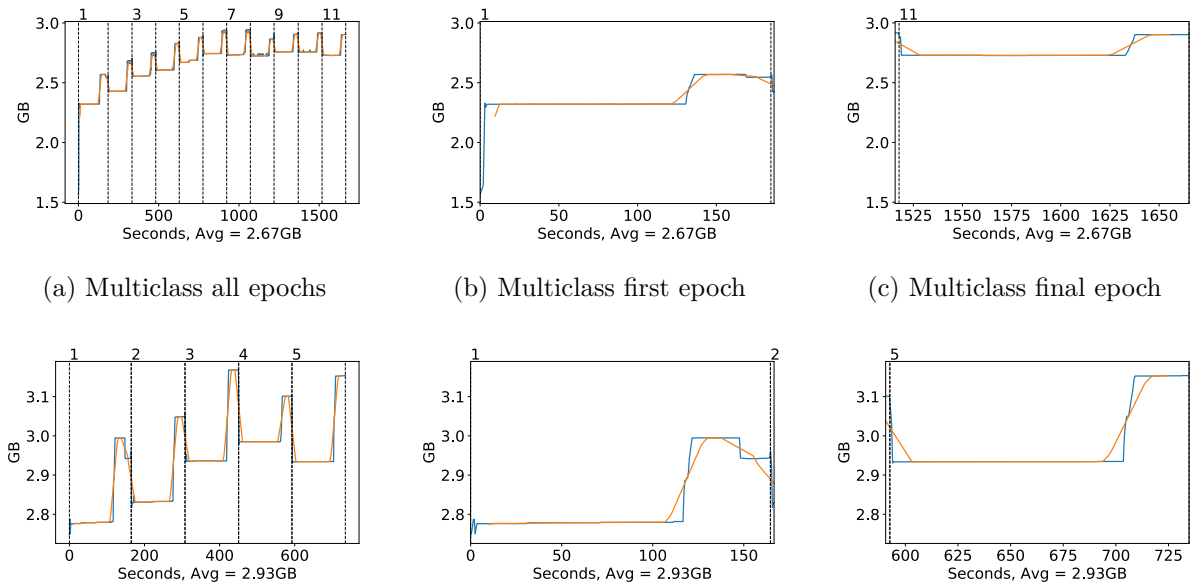
Figure A.47: DenseNet 121 GPU power usage during training in W, with a moving window average over 40 measurements overlaid



(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

Figure A.48: DenseNet 121 CPU usage (averaged over 4 cores) during training, with a moving window average over 40 measurements overlaid



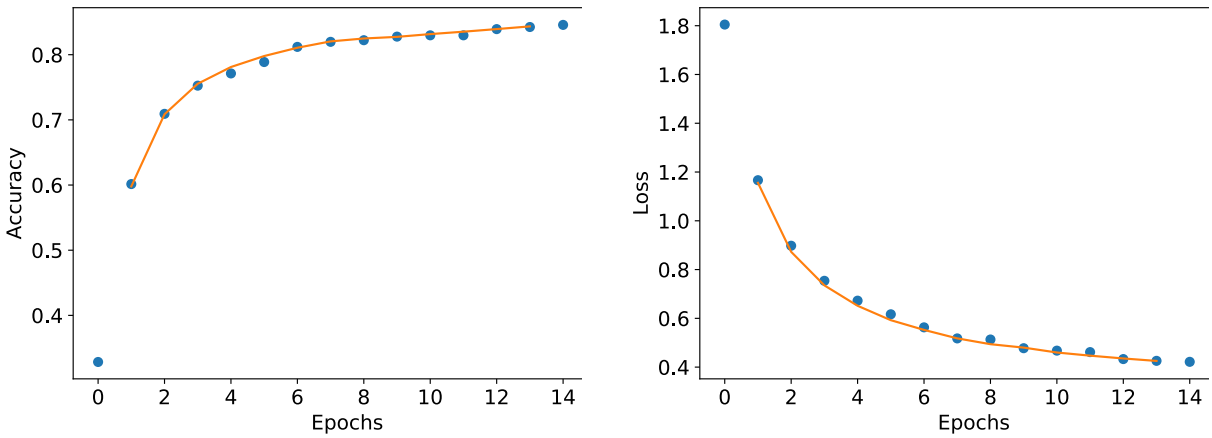
(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

Figure A.49: DenseNet 121 Memory usage during training, with a moving window average over 40 measurements overlaid

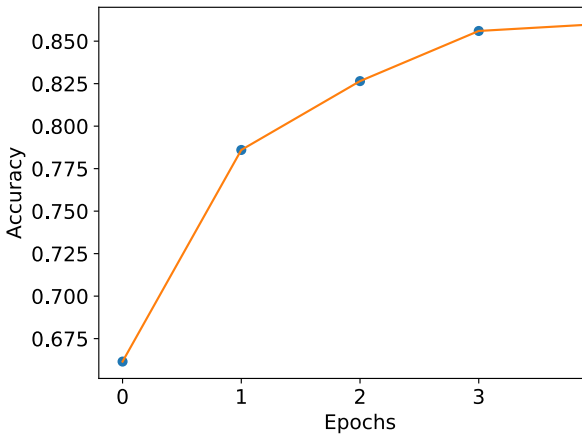
A.2.5 DenseNet 169

Classification Accuracy and Loss

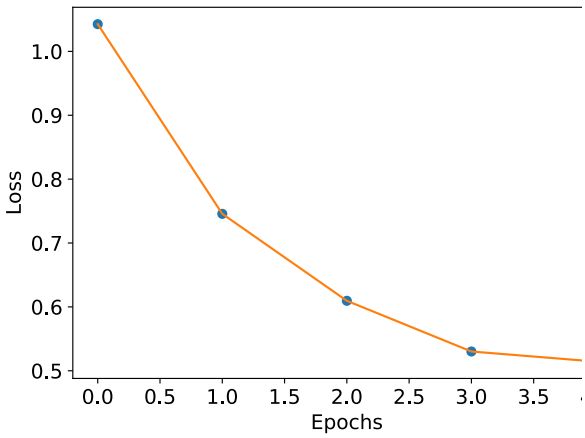


(a) Multiclass network accuracy

(b) Multiclass network loss



(c) Binary network "polyps" accuracy

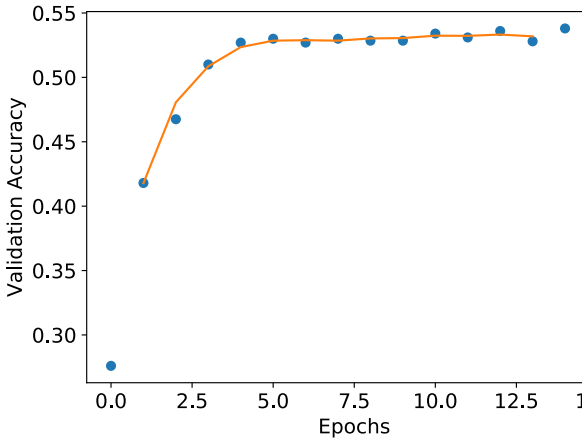


(d) Binary network "polyps" loss

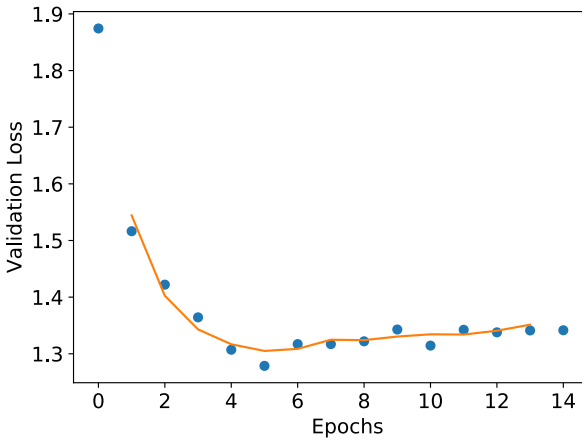
Figure A.50: DenseNet 169 Training classification accuracy and loss history for training data

GPU Usage

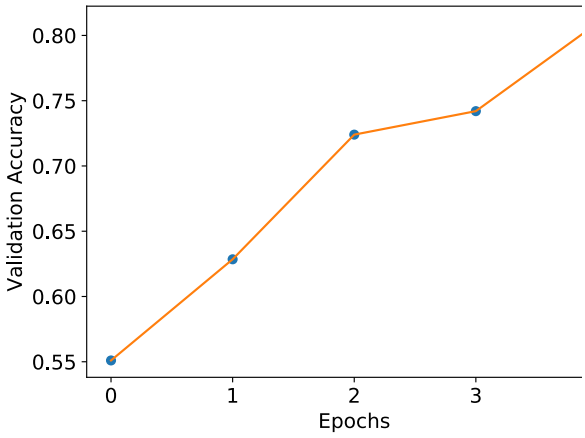
CPU and memory usage



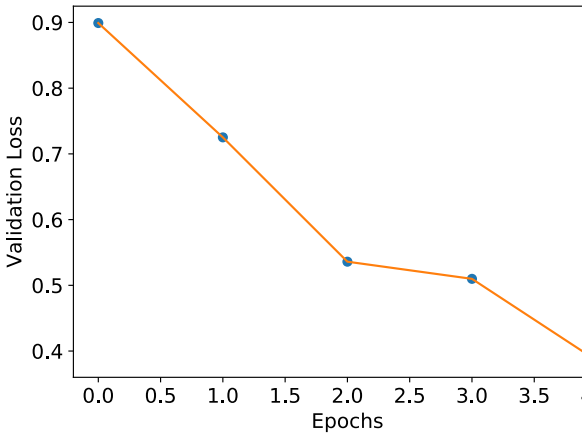
(a) Multiclass network accuracy



(b) Multiclass network loss

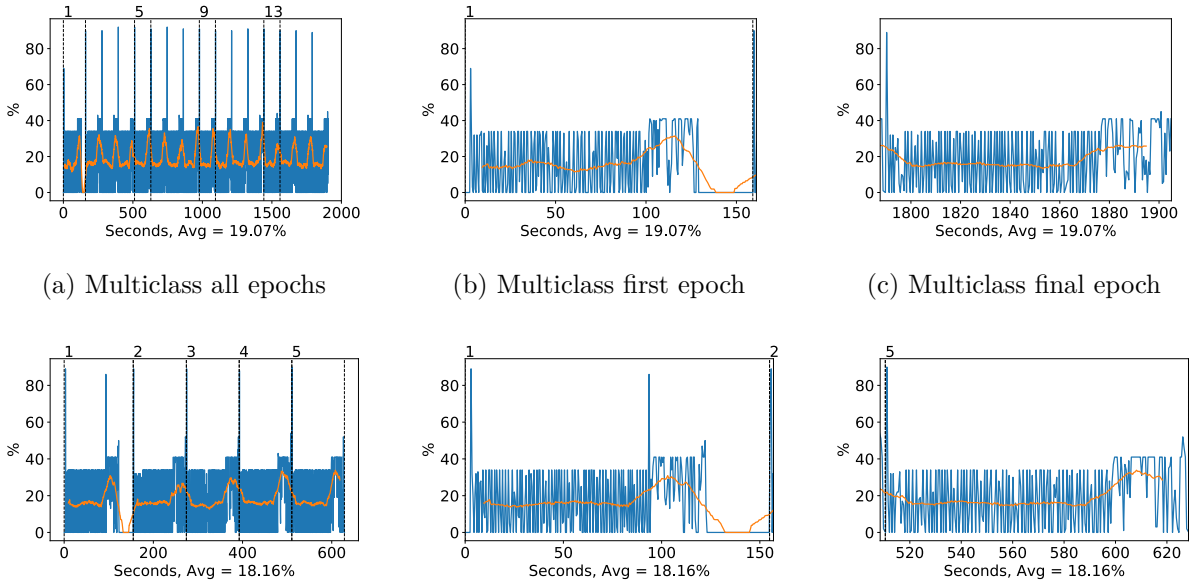


(c) Binary network "polyps" accuracy



(d) Binary network "polyps" loss

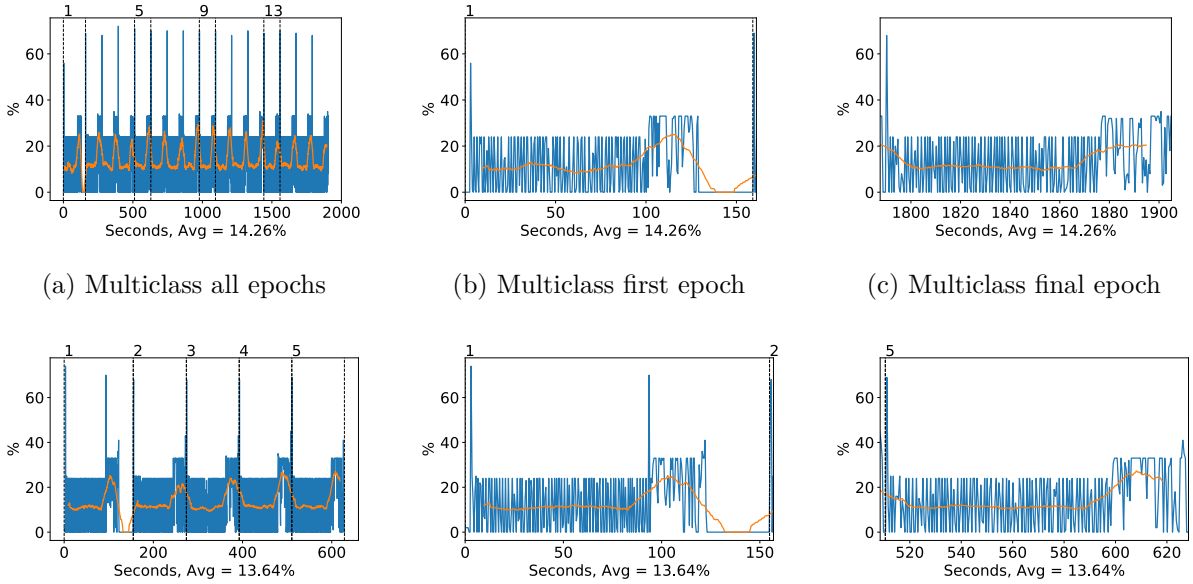
Figure A.51: DenseNet 169 Training classification accuracy and loss history for validation data



(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

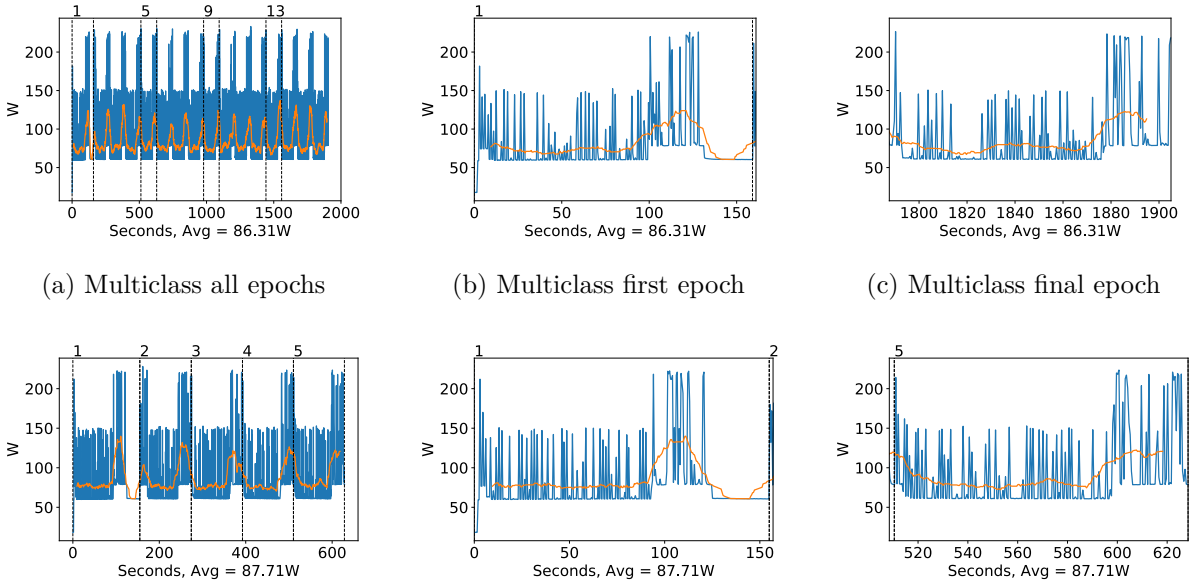
Figure A.52: DenseNet 169 GPU volatile usage during training, with a moving window average over 40 measurements overlaid



(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

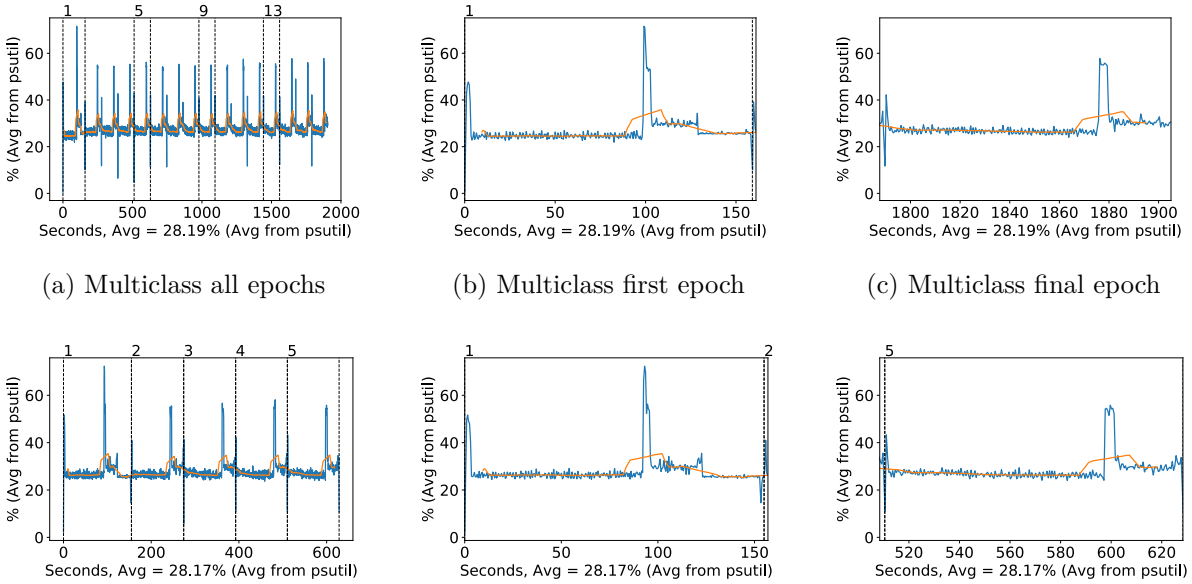
Figure A.53: DenseNet 169 GPU memory usage during training, with a moving window average over 40 measurements overlaid



(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

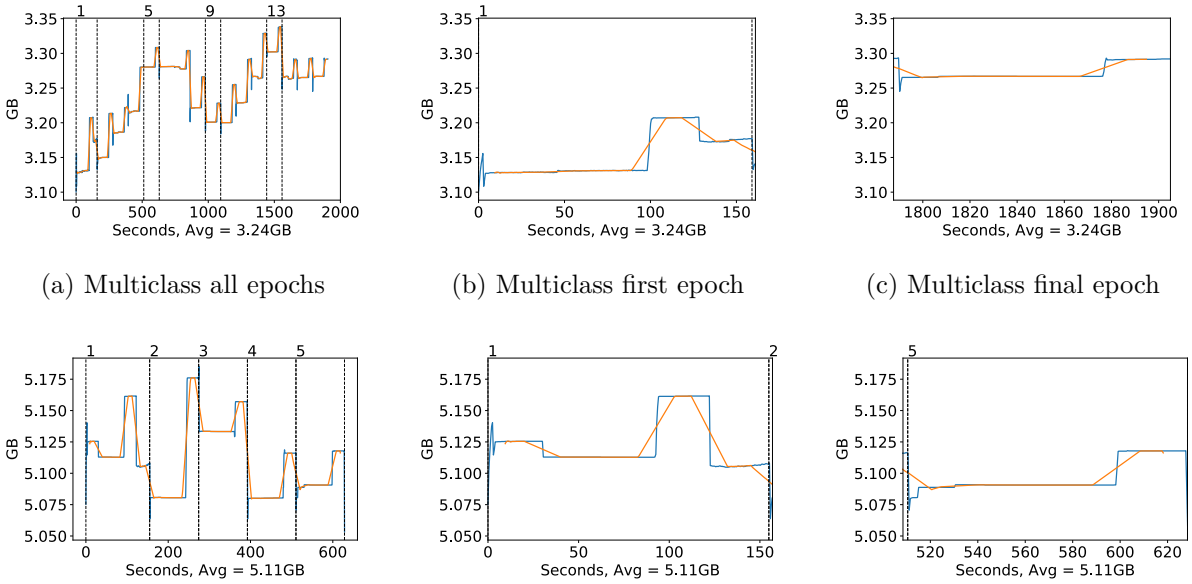
Figure A.54: DenseNet 169 GPU power usage during training in W, with a moving window average over 40 measurements overlaid



(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

Figure A.55: DenseNet 169 CPU usage (averaged over 4 cores) during training, with a moving window average over 40 measurements overlaid



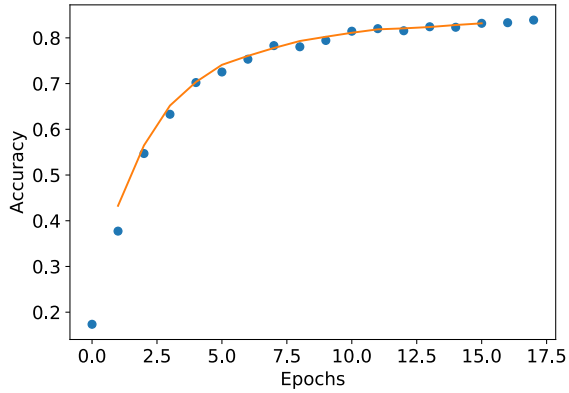
(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

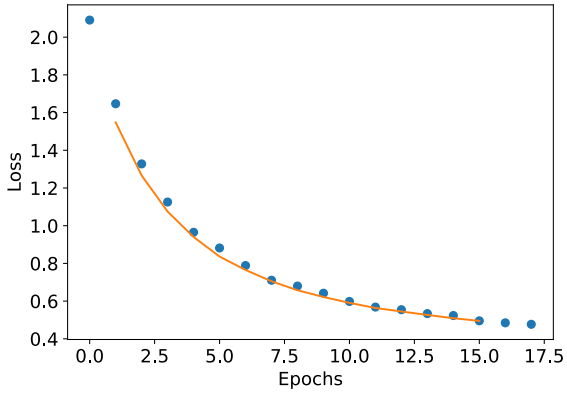
Figure A.56: DenseNet 169 Memory usage during training, with a moving window average over 40 measurements overlaid

A.2.6 DenseNet 201

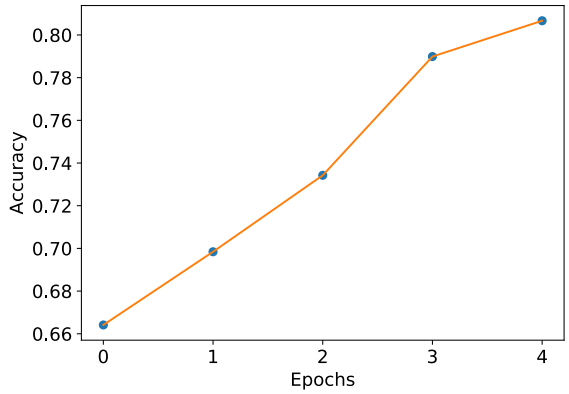
Classification Accuracy and Loss



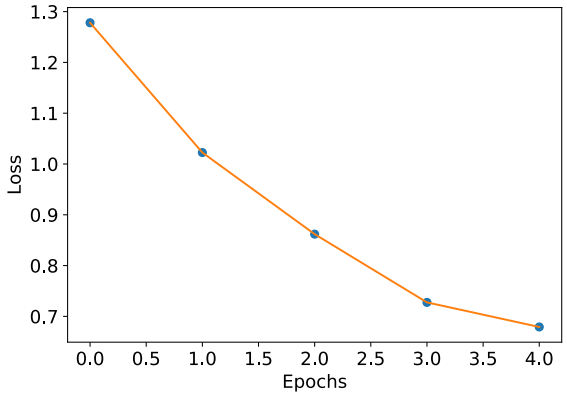
(a) Multiclass network accuracy



(b) Multiclass network loss



(c) Binary network "polyps" accuracy

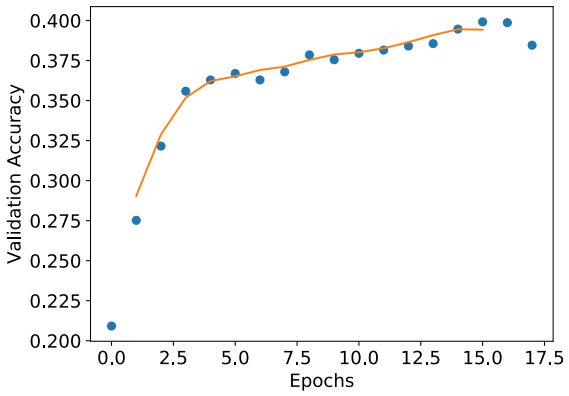


(d) Binary network "polyps" loss

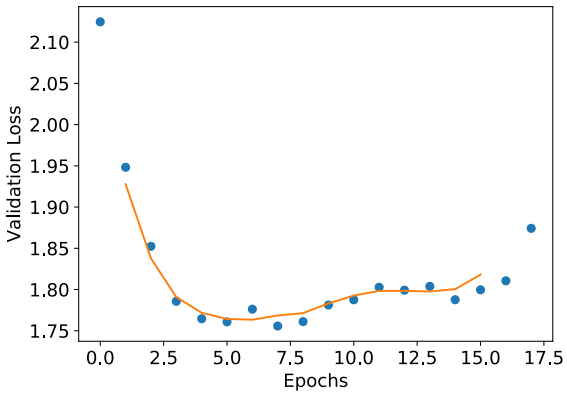
Figure A.57: DenseNet 201 Training classification accuracy and loss history for training data

GPU Usage

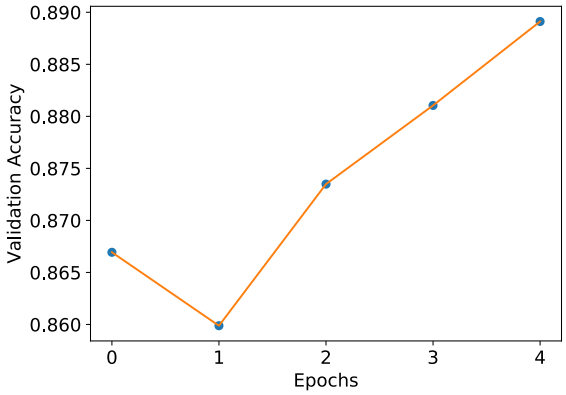
CPU and memory usage



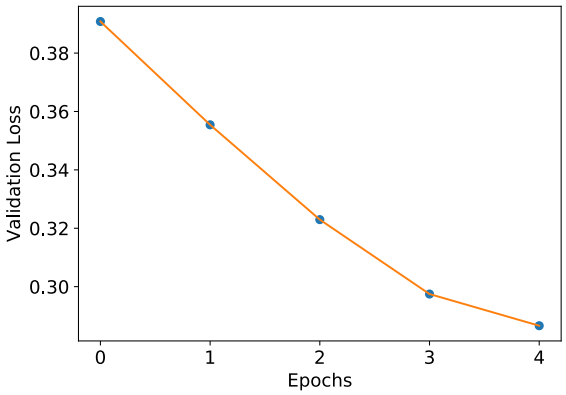
(a) Multiclass network accuracy



(b) Multiclass network loss

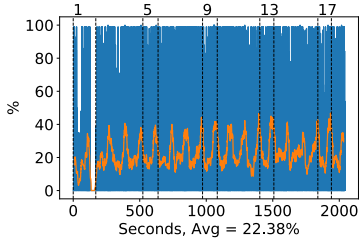


(c) Binary network "polyps" accuracy

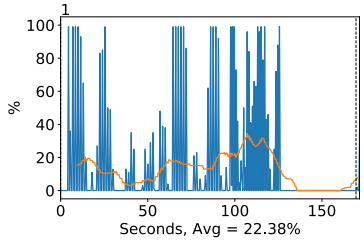


(d) Binary network "polyps" loss

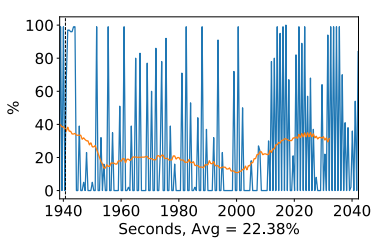
Figure A.58: DenseNet 201 Training classification accuracy and loss history for validation data



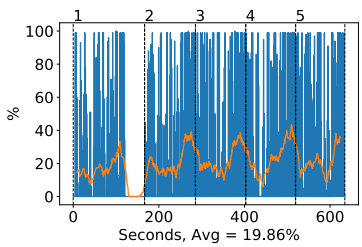
(a) Multiclass all epochs



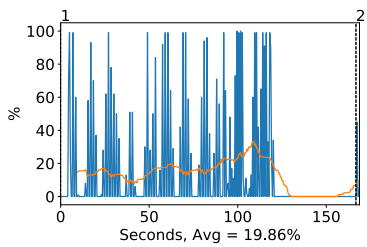
(b) Multiclass first epoch



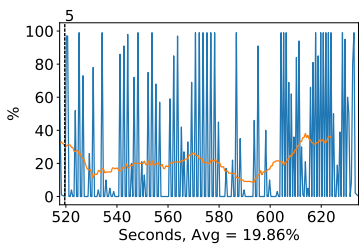
(c) Multiclass final epoch



(d) Binary all epochs

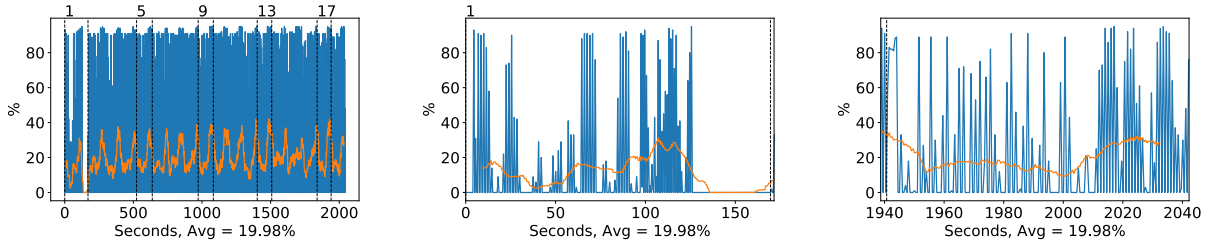


(e) Binary first epoch

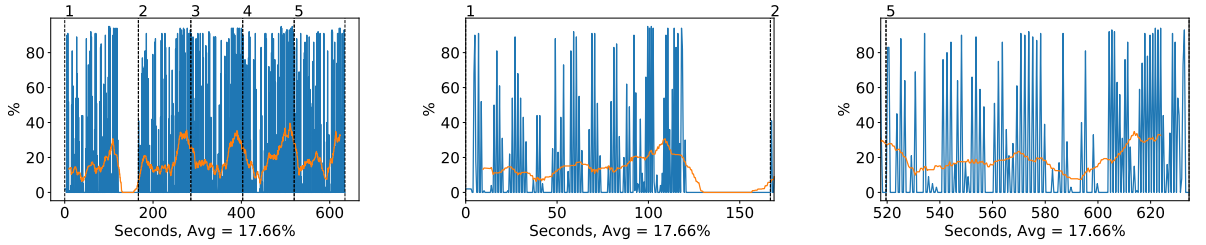


(f) Binary final epoch

Figure A.59: DenseNet 201 GPU volatile usage during training, with a moving window average over 40 measurements overlaid

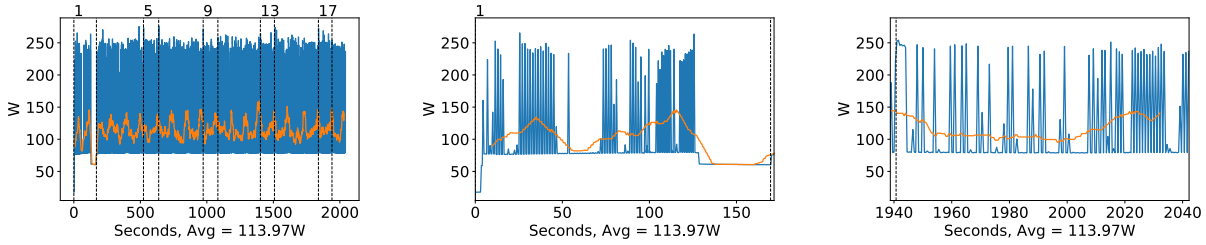


(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

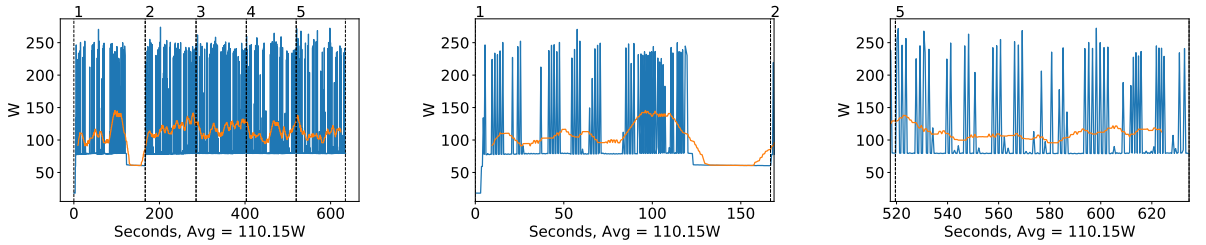


(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

Figure A.60: DenseNet 201 GPU memory usage during training, with a moving window average over 40 measurements overlaid

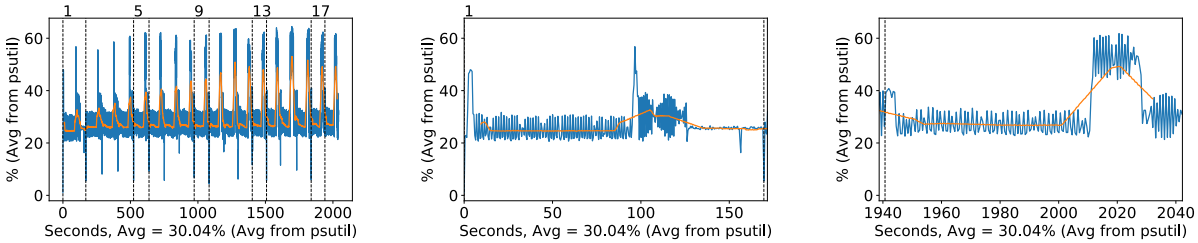


(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

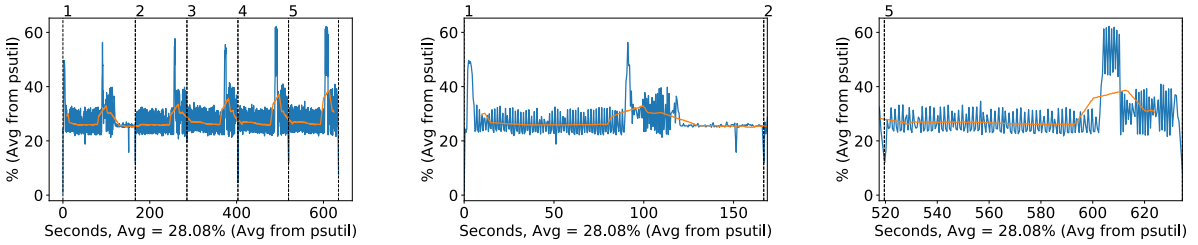


(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

Figure A.61: DenseNet 201 GPU power usage during training in W, with a moving window average over 40 measurements overlaid

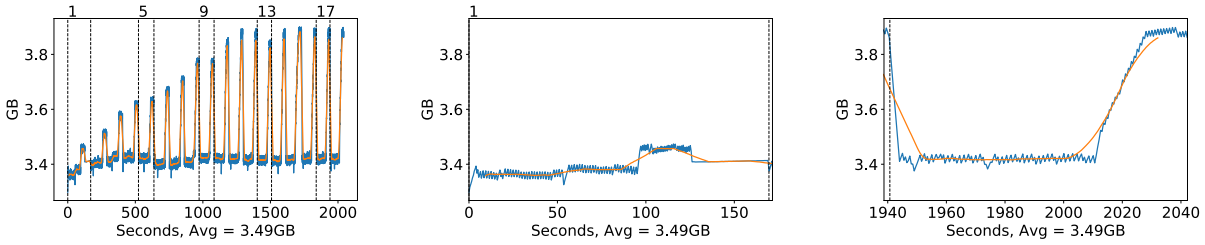


(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

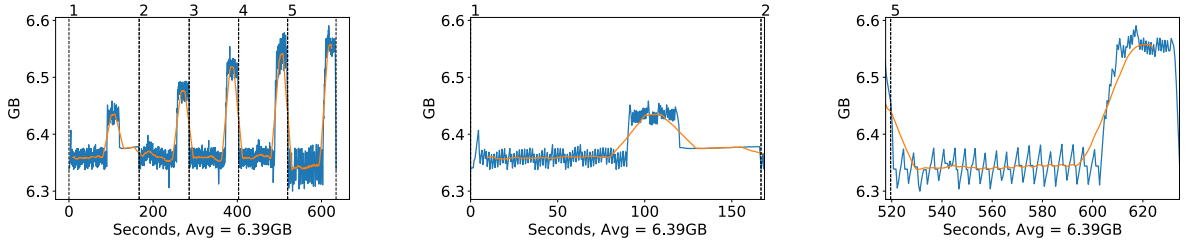


(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

Figure A.62: DenseNet 201 CPU usage (averaged over 4 cores) during training, with a moving window average over 40 measurements overlaid



(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

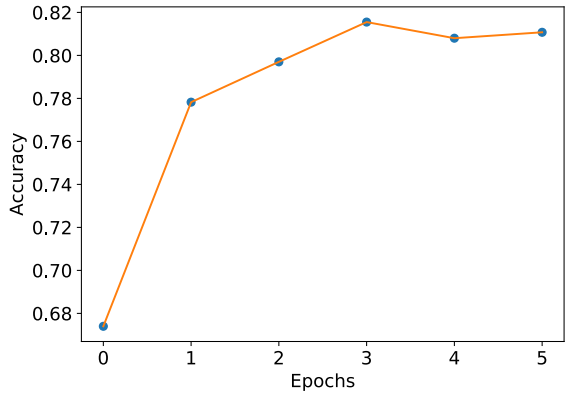


(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

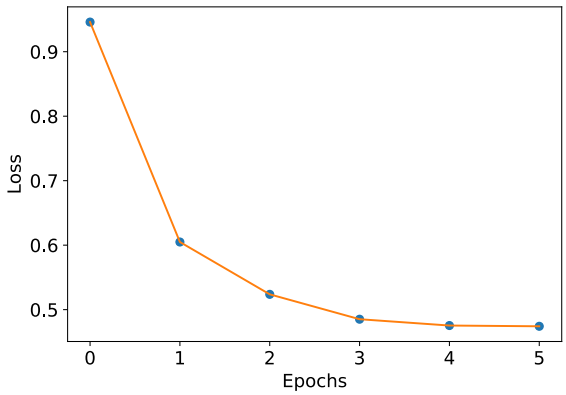
Figure A.63: DenseNet 201 Memory usage during training, with a moving window average over 40 measurements overlaid

A.2.7 Xception

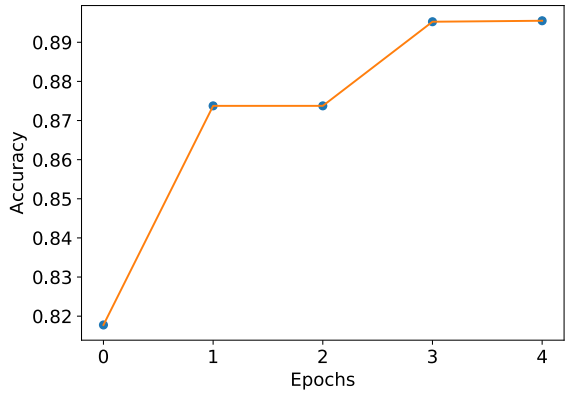
Classification Accuracy and Loss



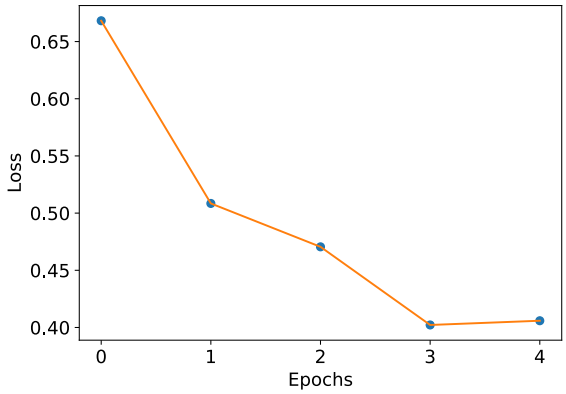
(a) Multiclass network accuracy



(b) Multiclass network loss



(c) Binary network "polyps" accuracy

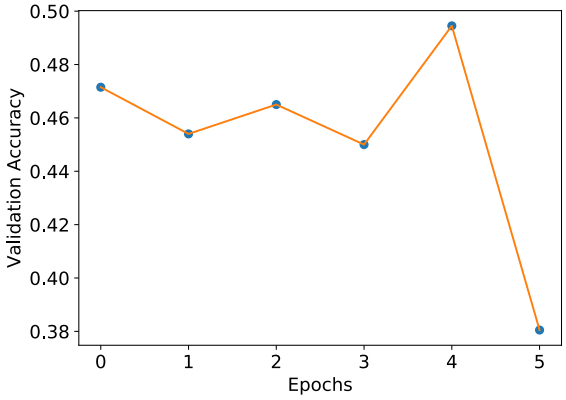


(d) Binary network "polyps" loss

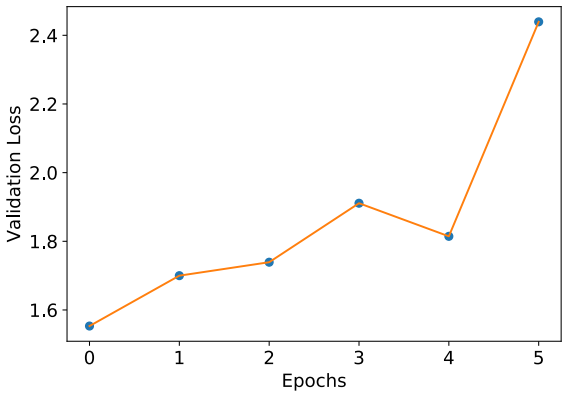
Figure A.64: Xception Training classification accuracy and loss history for training data

GPU Usage

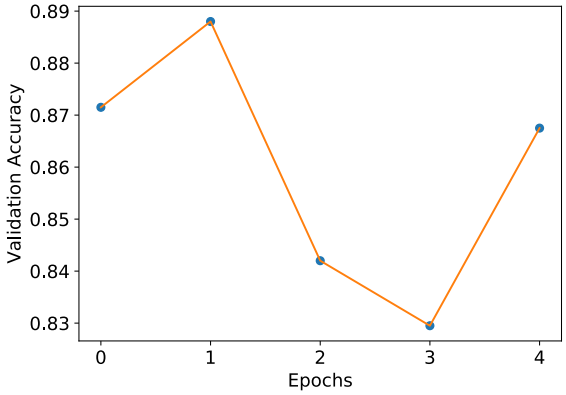
CPU and memory usage



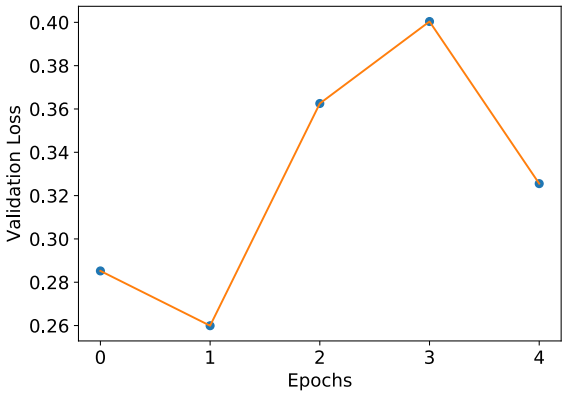
(a) Multiclass network accuracy



(b) Multiclass network loss

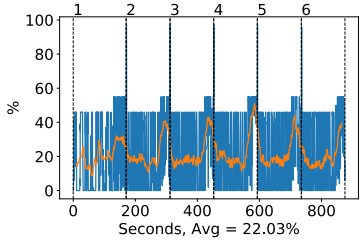


(c) Binary network "polyps" accuracy

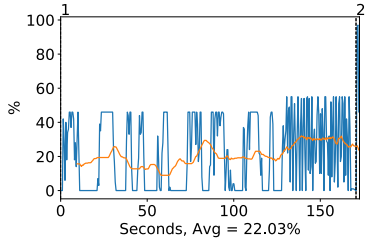


(d) Binary network "polyps" loss

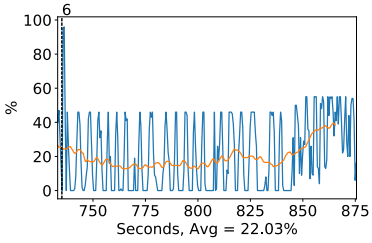
Figure A.65: Xception Training classification accuracy and loss history for validation data



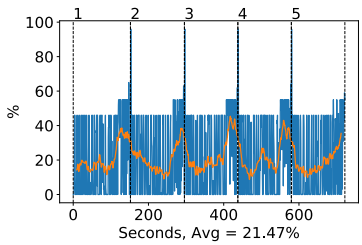
(a) Multiclass all epochs



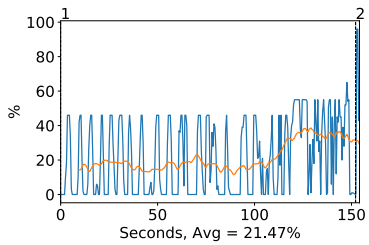
(b) Multiclass first epoch



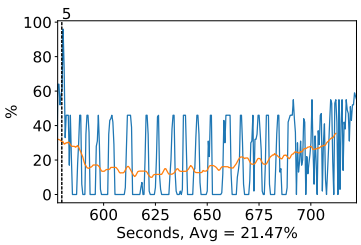
(c) Multiclass final epoch



(d) Binary all epochs

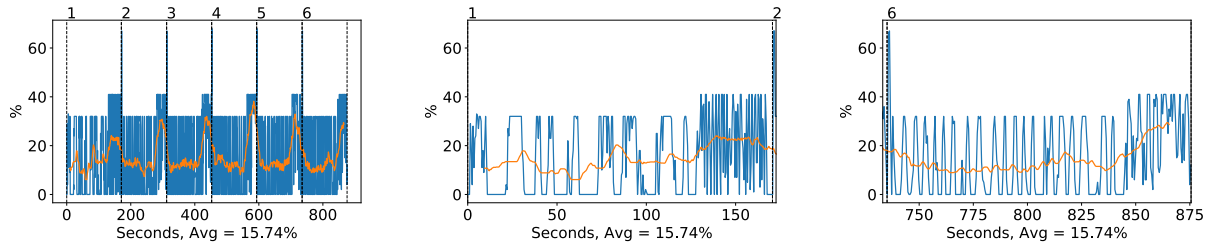


(e) Binary first epoch

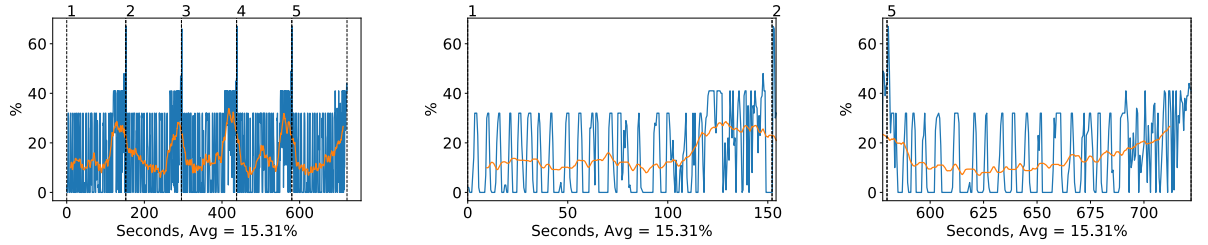


(f) Binary final epoch

Figure A.66: Xception GPU volatile usage during training, with a moving window average over 40 measurements overlaid

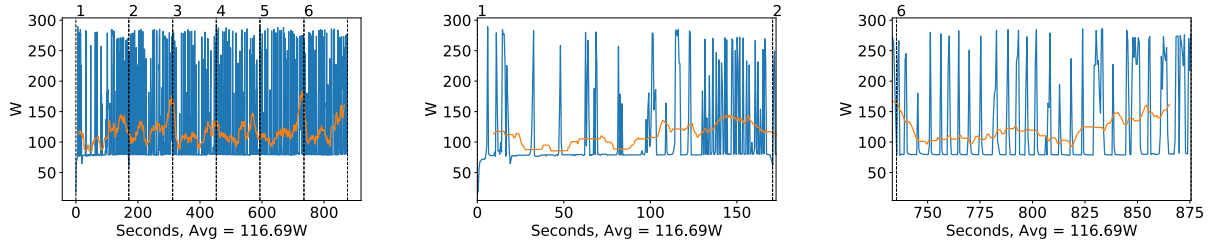


(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

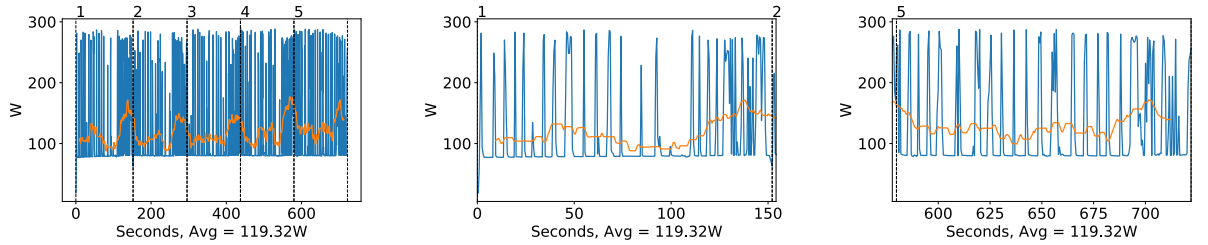


(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

Figure A.67: Xception GPU memory usage during training, with a moving window average over 40 measurements overlaid

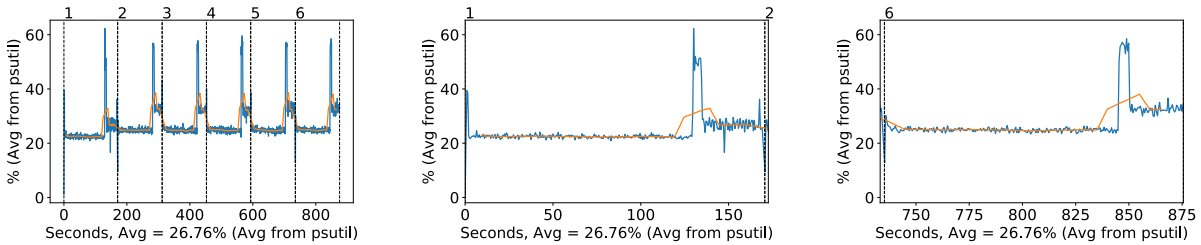


(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

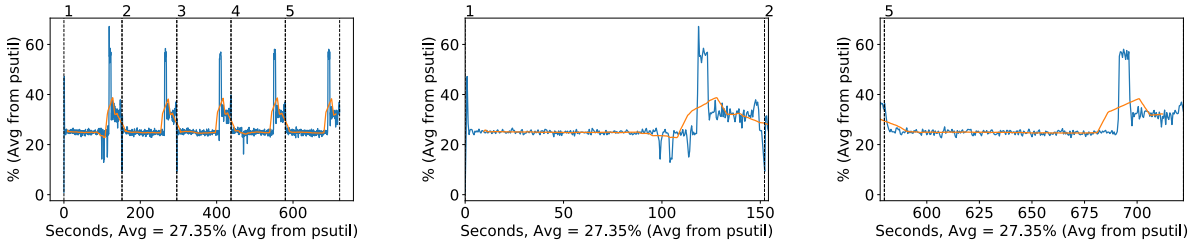


(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

Figure A.68: Xception GPU power usage during training in W, with a moving window average over 40 measurements overlaid

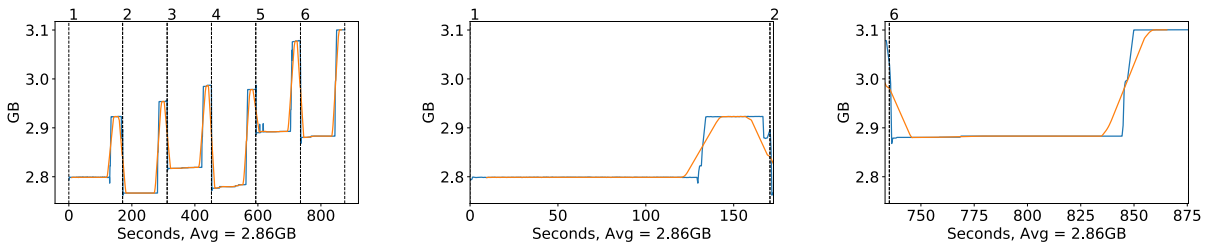


(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

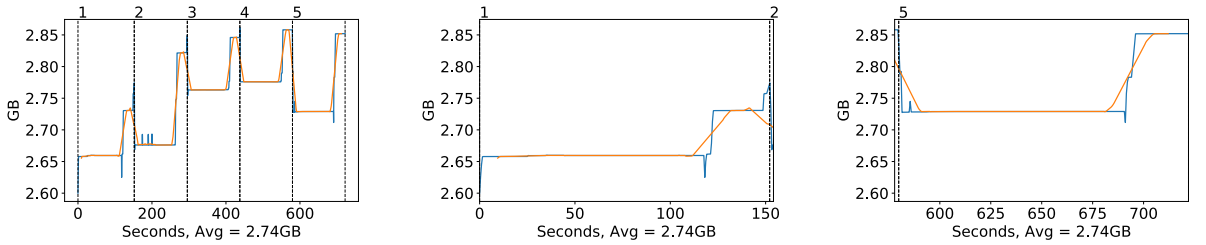


(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

Figure A.69: Xception CPU usage (averaged over 4 cores) during training, with a moving window average over 40 measurements overlaid



(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

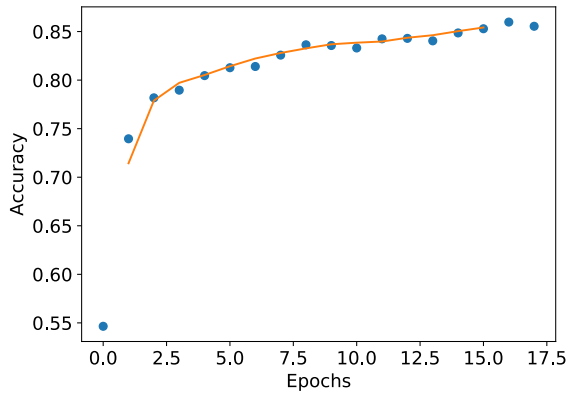


(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

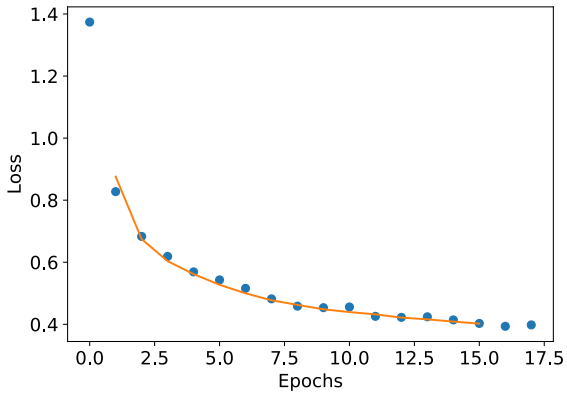
Figure A.70: Xception Memory usage during training, with a moving window average over 40 measurements overlaid

A.2.8 Inception ResNet v2

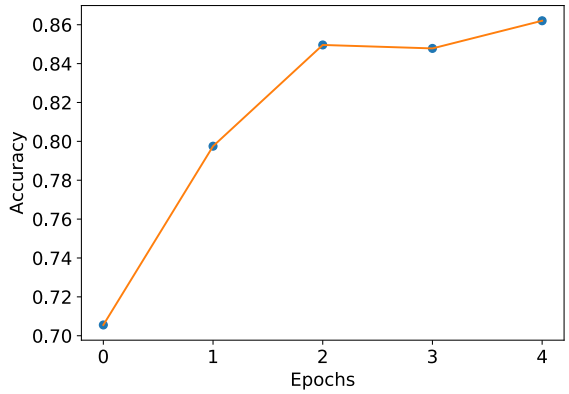
Classification Accuracy and Loss



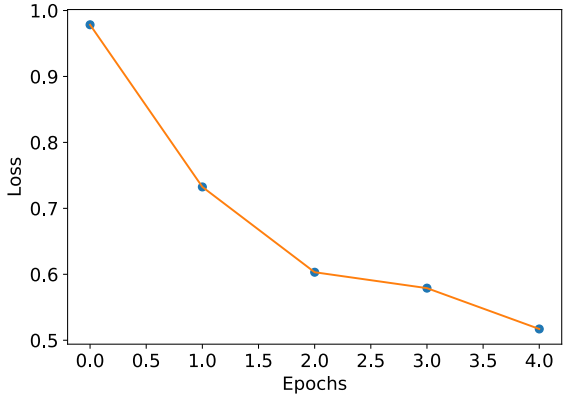
(a) Multiclass network accuracy



(b) Multiclass network loss



(c) Binary network "polyps" accuracy

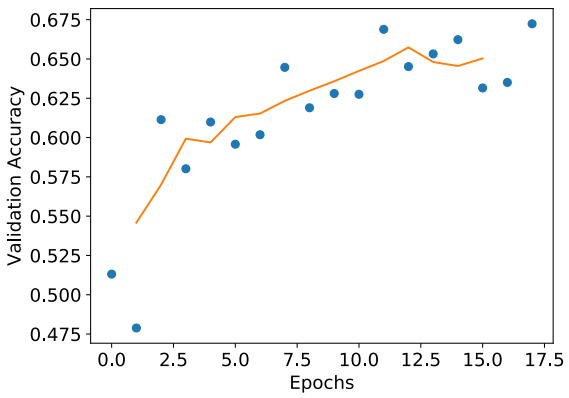


(d) Binary network "polyps" loss

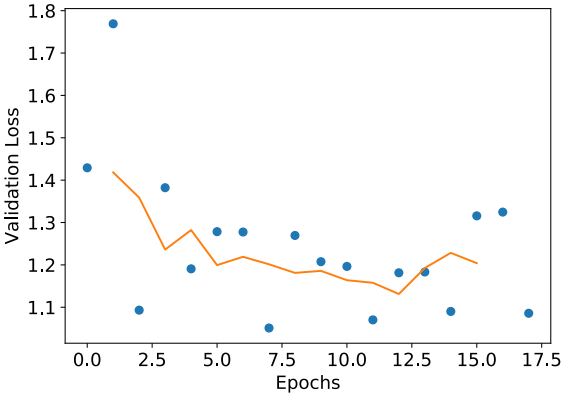
Figure A.71: Inception-ResNet-v2 Training classification accuracy and loss history for training data

GPU Usage

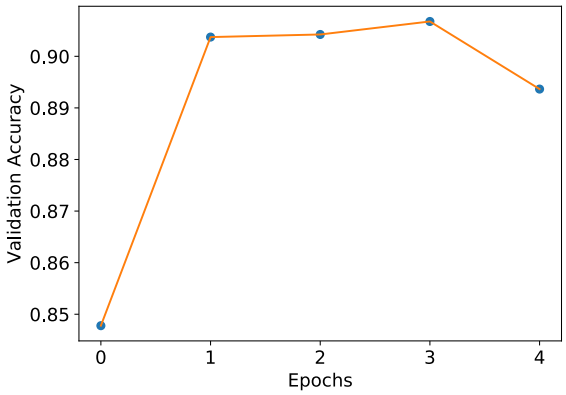
CPU and memory usage



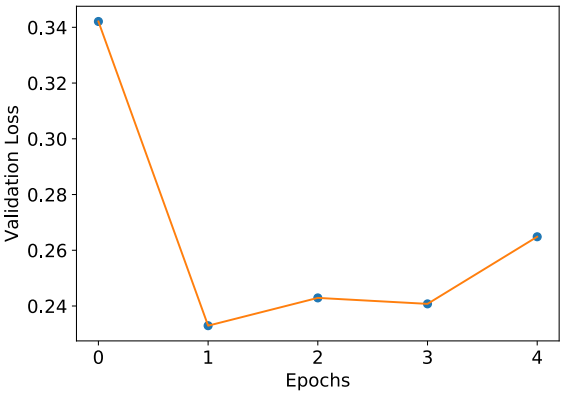
(a) Multiclass network accuracy



(b) Multiclass network loss

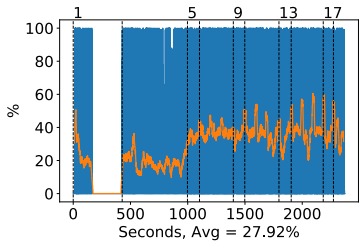


(c) Binary network "polyps" accuracy

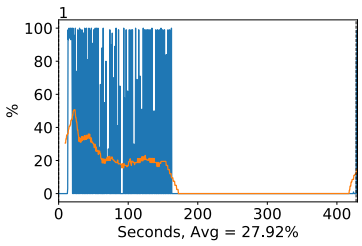


(d) Binary network "polyps" loss

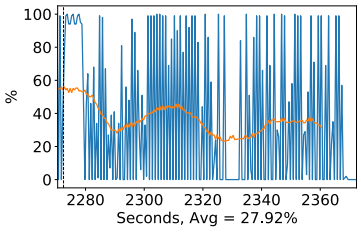
Figure A.72: Inception-ResNet-v2 Training classification accuracy and loss history for validation data



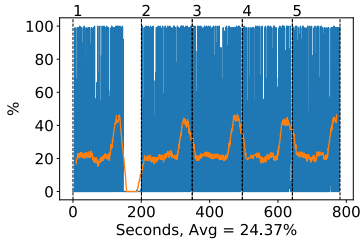
(a) Multiclass all epochs



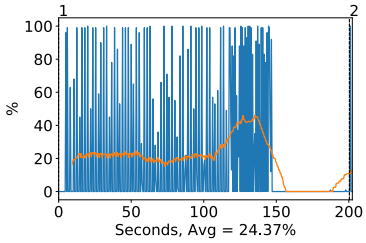
(b) Multiclass first epoch



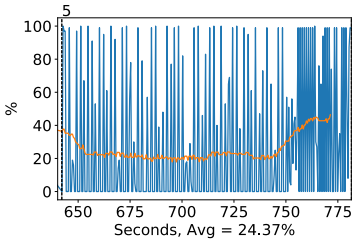
(c) Multiclass final epoch



(d) Binary all epochs

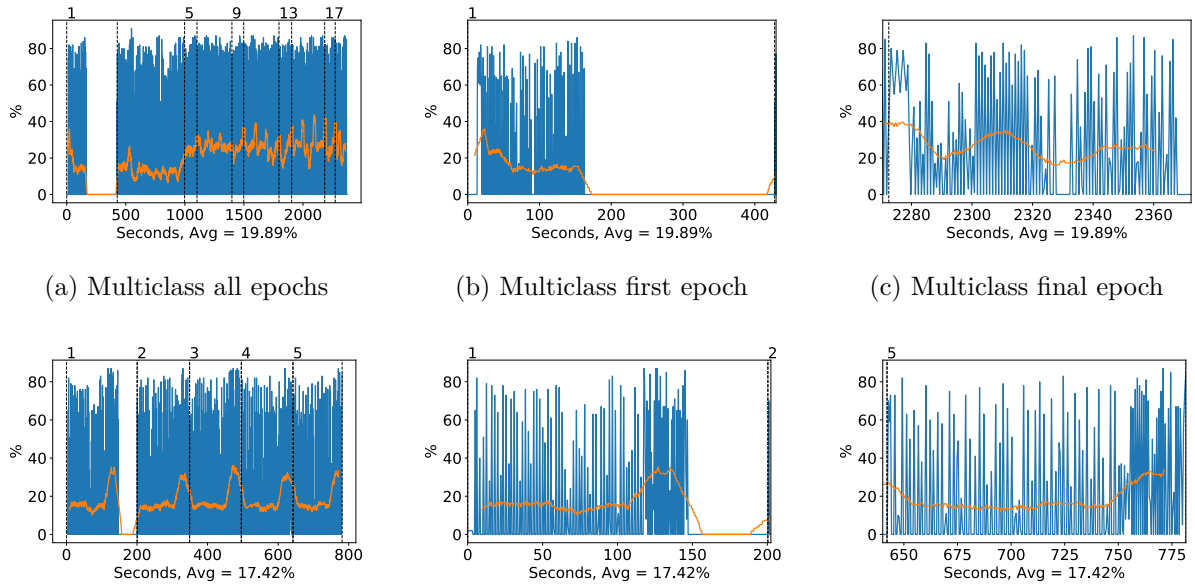


(e) Binary first epoch



(f) Binary final epoch

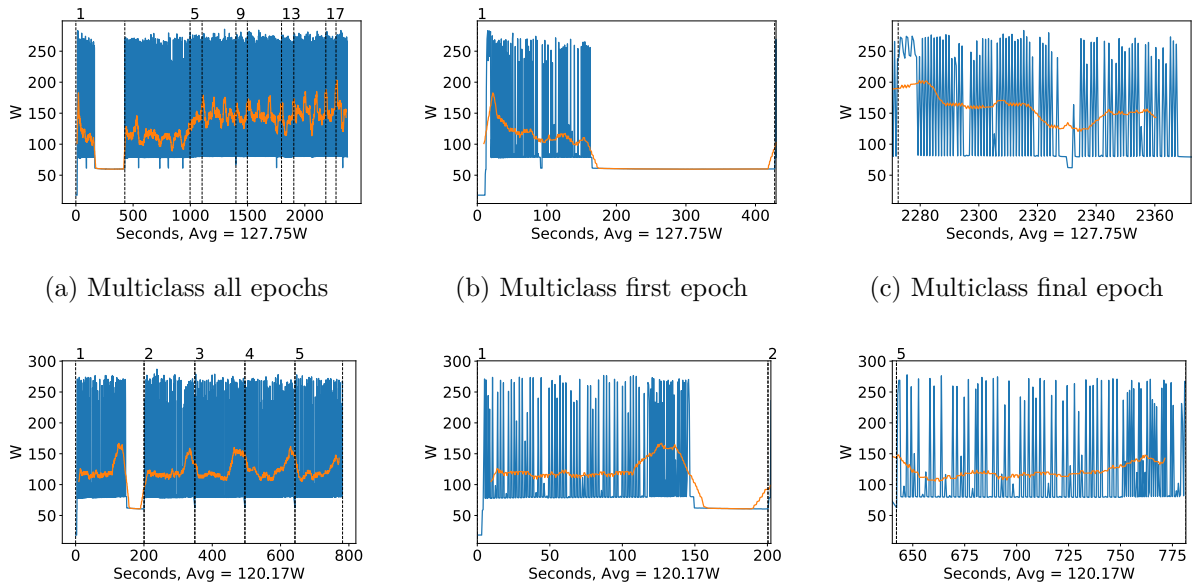
Figure A.73: Inception-ResNet-v2 GPU volatile usage during training, with a moving window average over 40 measurements overlaid



(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

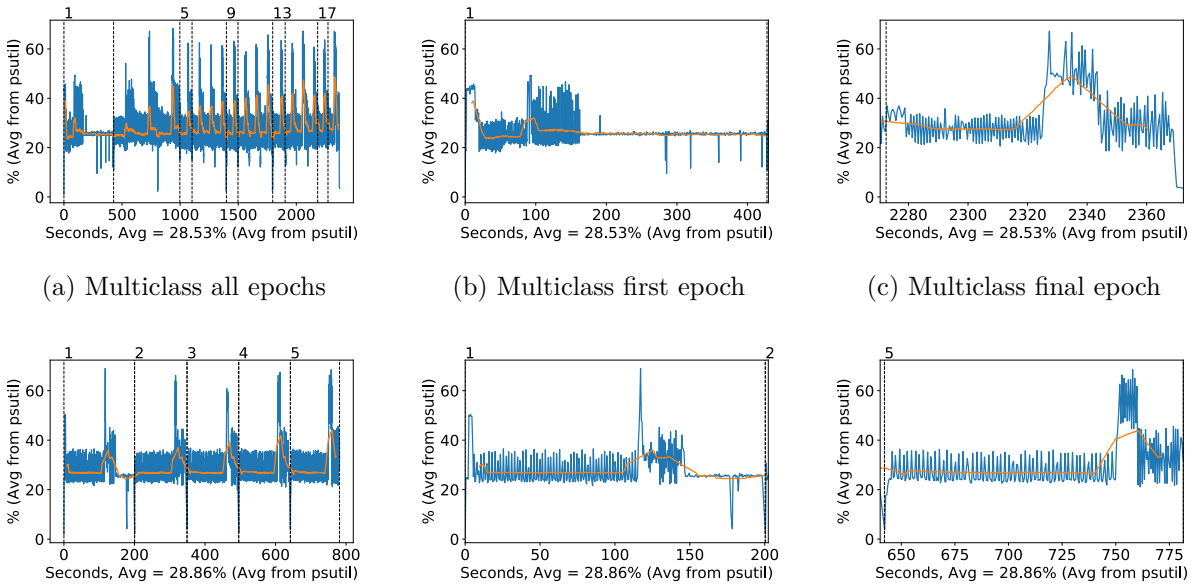
Figure A.74: Inception-ResNet-v2 GPU memory usage during training, with a moving window average over 40 measurements overlaid



(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

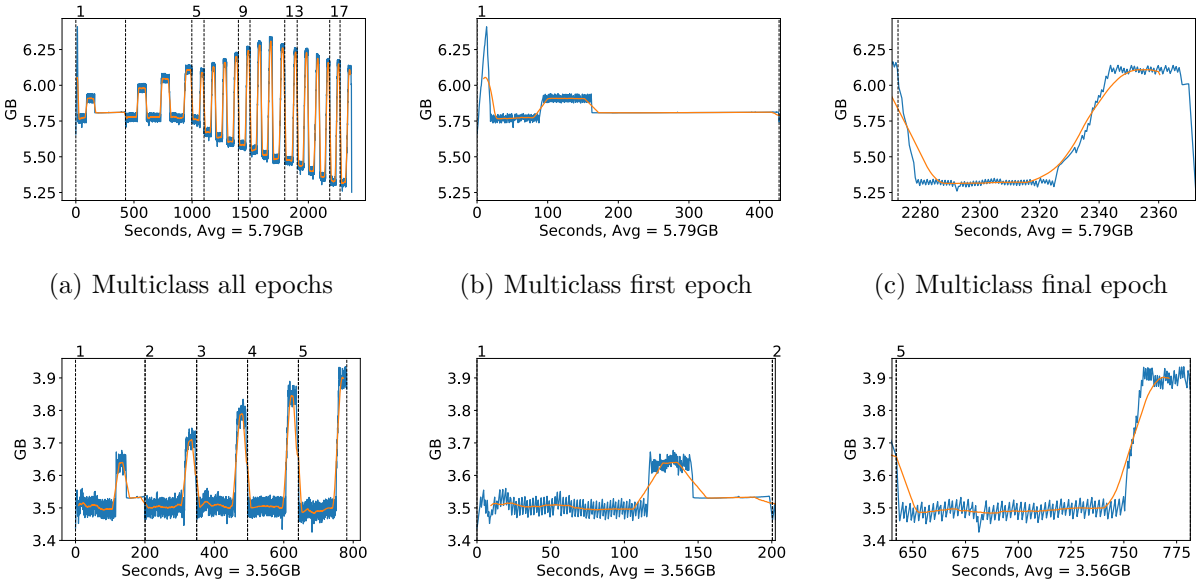
Figure A.75: Inception-ResNet-v2 GPU power usage during training in W, with a moving window average over 40 measurements overlaid



(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

Figure A.76: Inception-ResNet-v2 CPU usage (averaged over 4 cores) during training, with a moving window average over 40 measurements overlaid



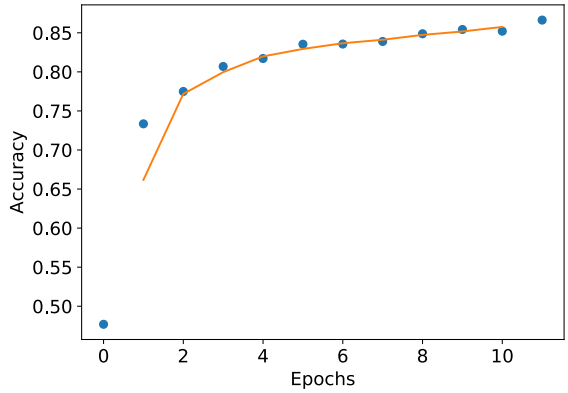
(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

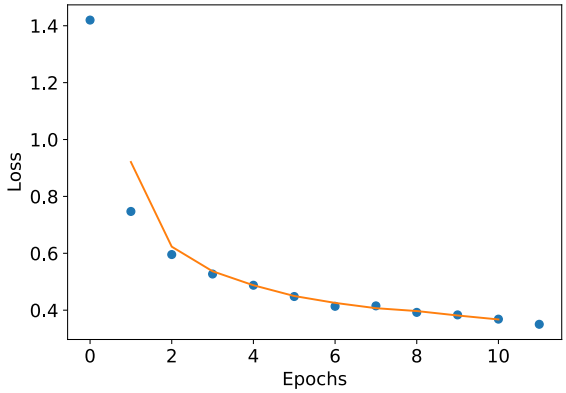
Figure A.77: Inception-ResNet-v2 Memory usage during training, with a moving window average over 40 measurements overlaid

A.2.9 Mobilenet

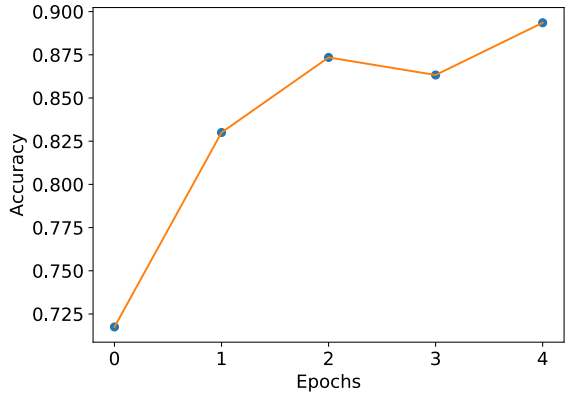
Classification Accuracy and Loss



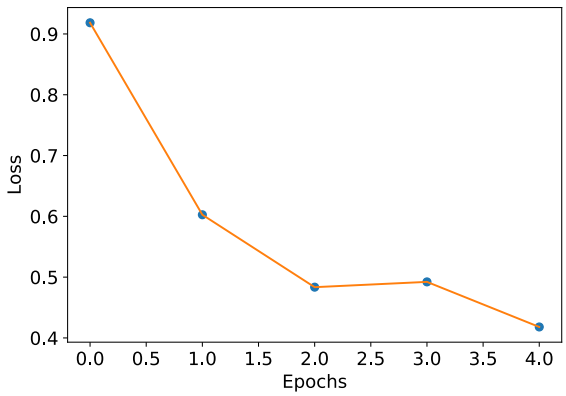
(a) Multiclass network accuracy



(b) Multiclass network loss



(c) Binary network "polyps" accuracy

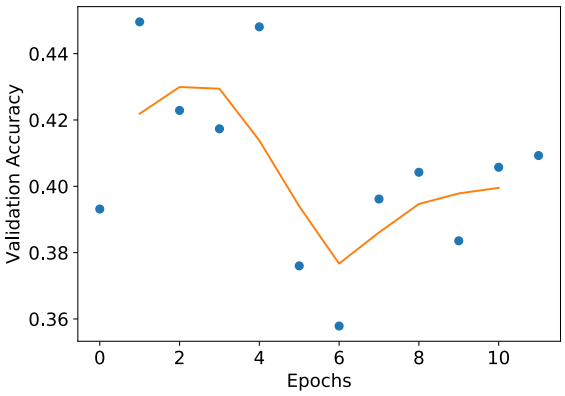


(d) Binary network "polyps" loss

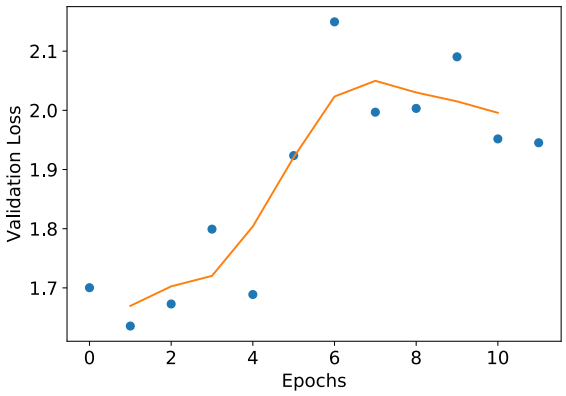
Figure A.78: Mobilenet Training classification accuracy and loss history for training data

GPU Usage

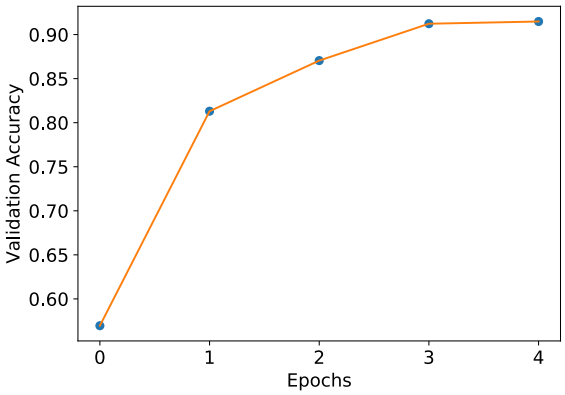
CPU and memory usage



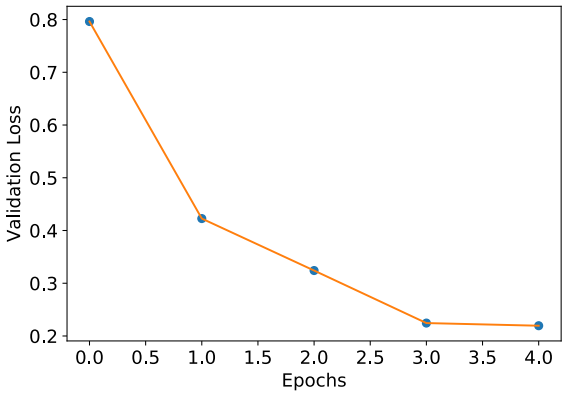
(a) Multiclass network accuracy



(b) Multiclass network loss

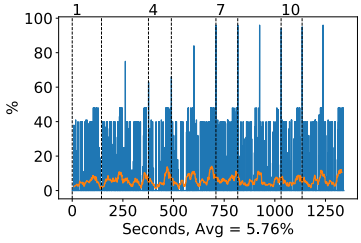


(c) Binary network "polyps" accuracy

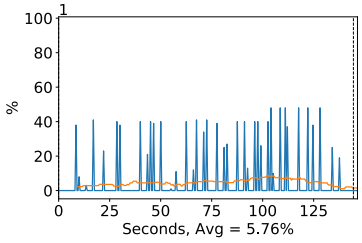


(d) Binary network "polyps" loss

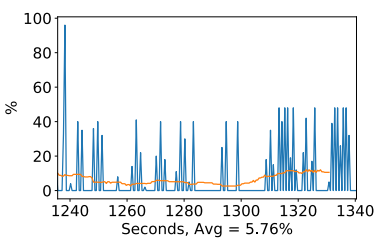
Figure A.79: Mobilenet Training classification accuracy and loss history for validation data



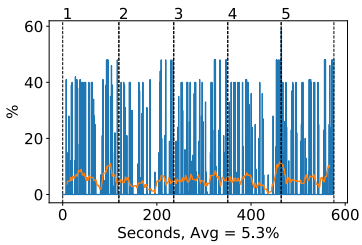
(a) Multiclass all epochs



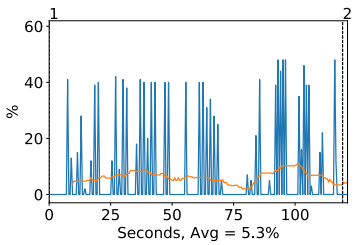
(b) Multiclass first epoch



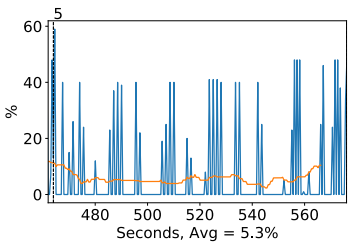
(c) Multiclass final epoch



(d) Binary all epochs

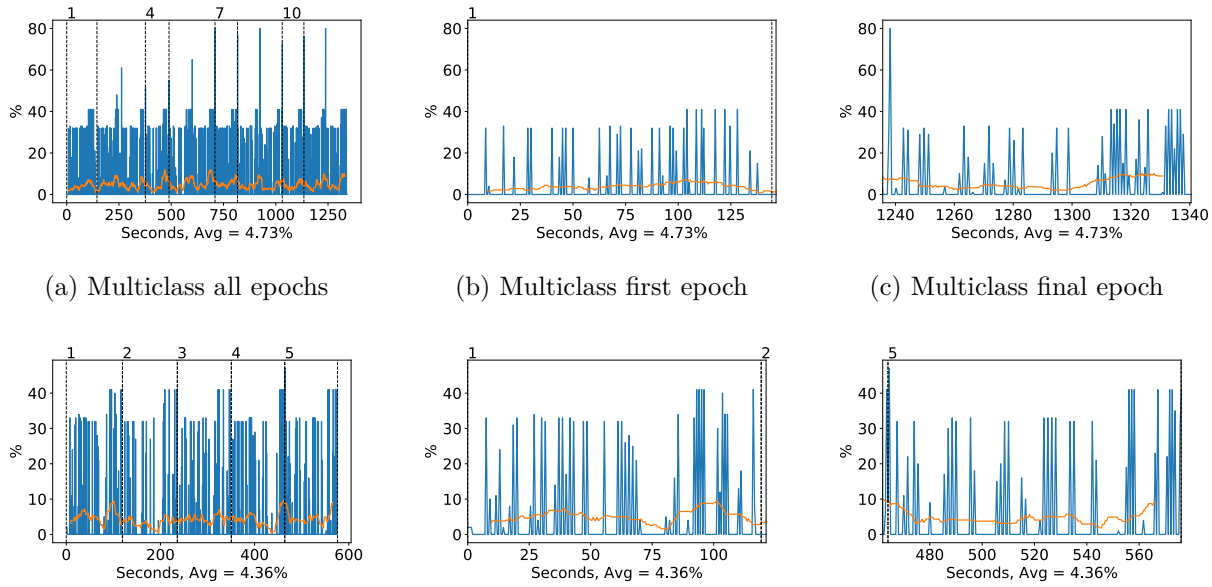


(e) Binary first epoch



(f) Binary final epoch

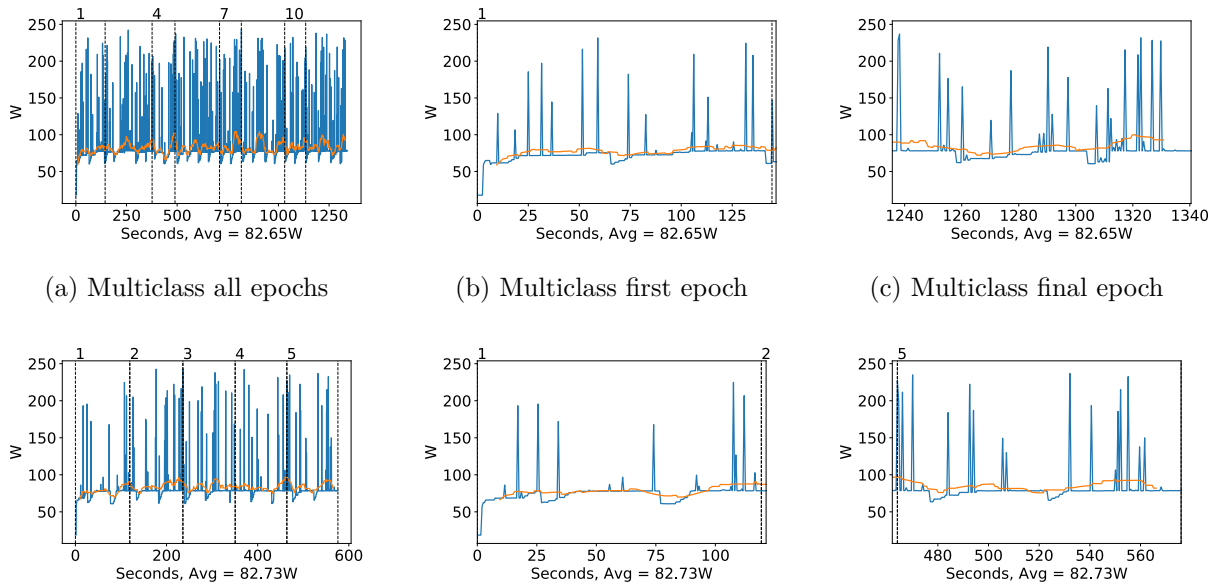
Figure A.80: Mobilenet GPU volatile usage during training, with a moving window average over 40 measurements overlaid



(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

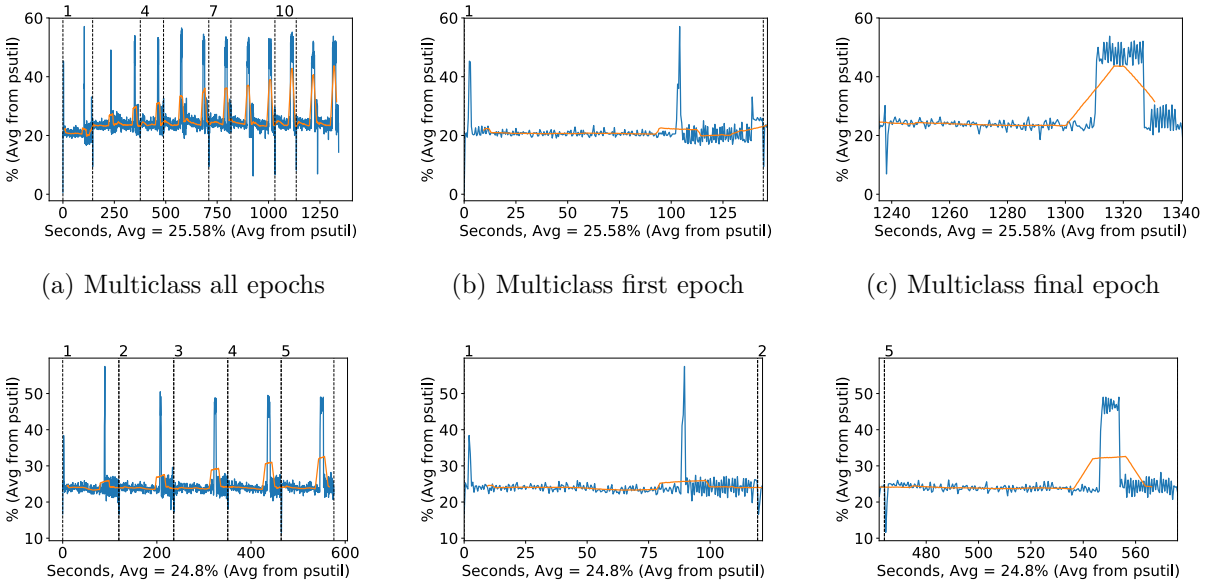
Figure A.81: Mobilenet GPU memory usage during training, with a moving window average over 40 measurements overlaid



(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

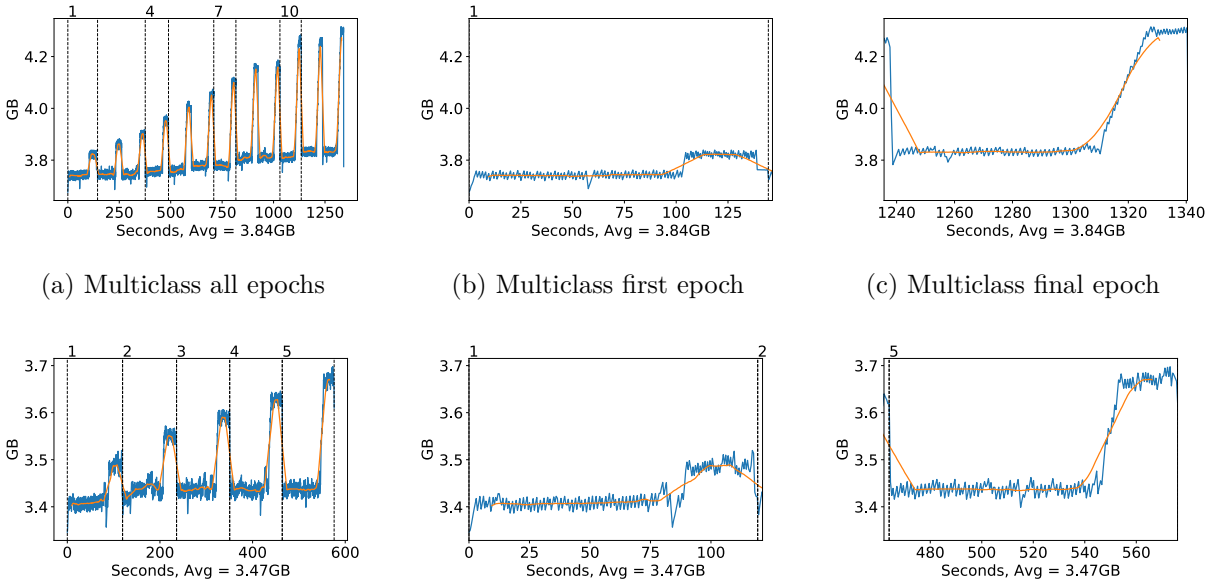
Figure A.82: Mobilenet GPU power usage during training in W, with a moving window average over 40 measurements overlaid



(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

Figure A.83: Mobilenet CPU usage (averaged over 4 cores) during training, with a moving window average over 40 measurements overlaid



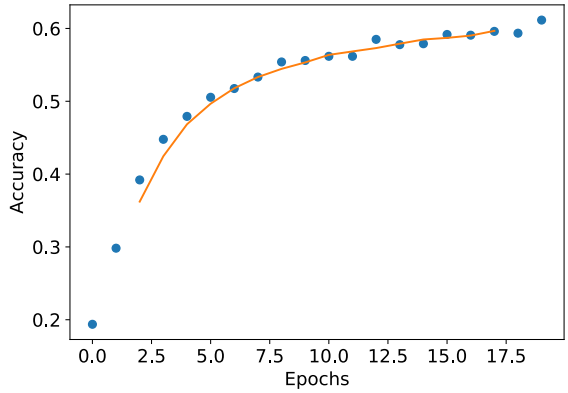
(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

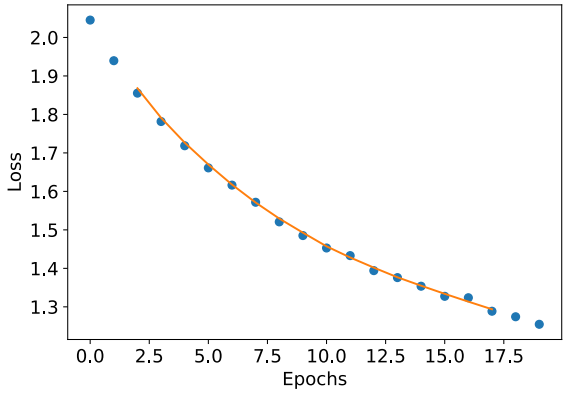
Figure A.84: Mobilenet Memory usage during training, with a moving window average over 40 measurements overlaid

A.2.10 NasNet Large

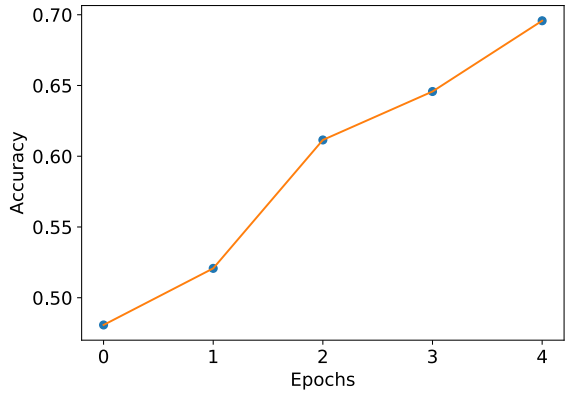
Classification Accuracy and Loss



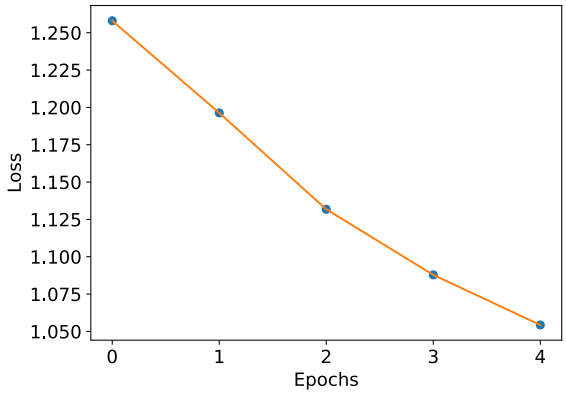
(a) Multiclass network accuracy



(b) Multiclass network loss



(c) Binary network "polyps" accuracy

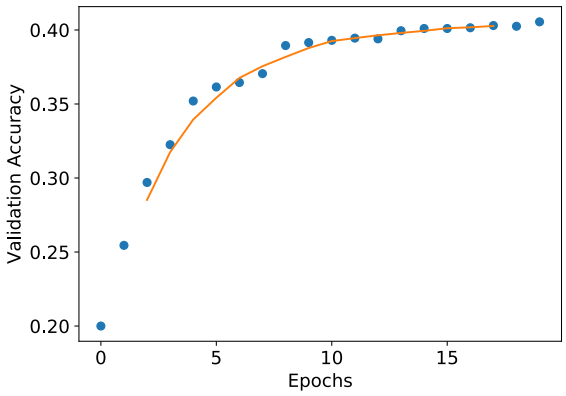


(d) Binary network "polyps" loss

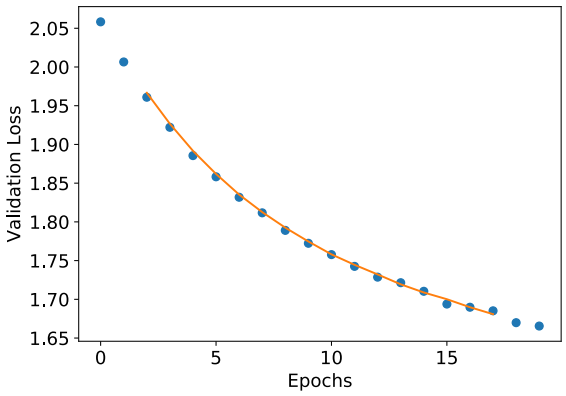
Figure A.85: NASNet Large Training classification accuracy and loss history for training data

GPU Usage

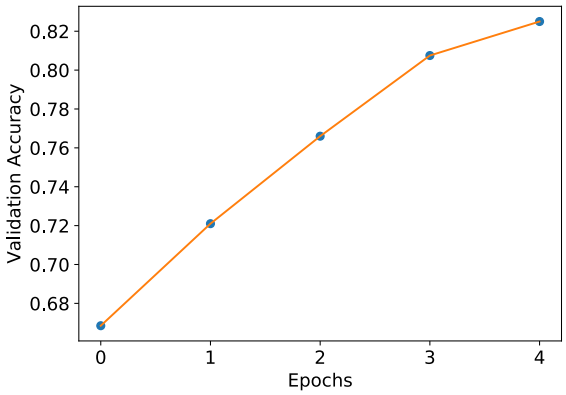
CPU and memory usage



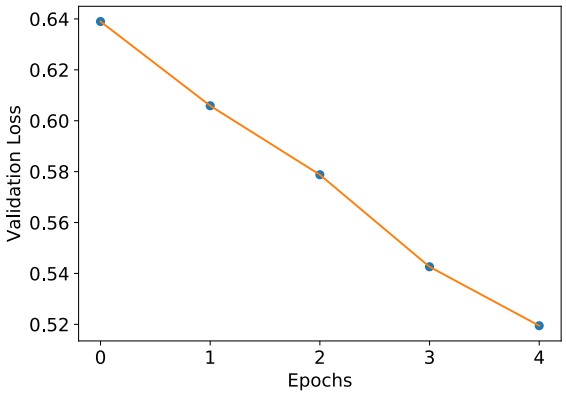
(a) Multiclass network accuracy



(b) Multiclass network loss

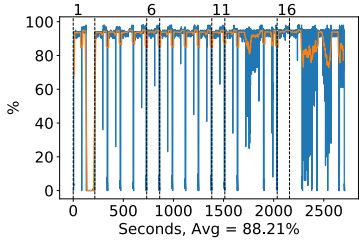


(c) Binary network "polyps" accuracy

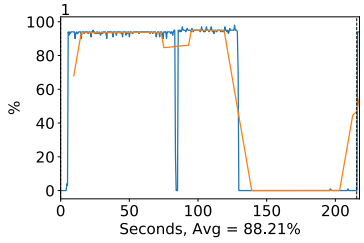


(d) Binary network "polyps" loss

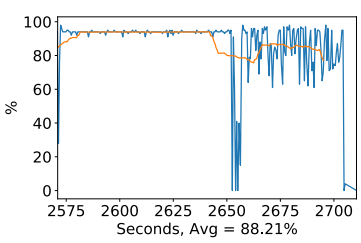
Figure A.86: NASNet Large Training classification accuracy and loss history for validation data



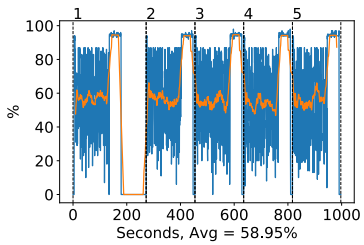
(a) Multiclass all epochs



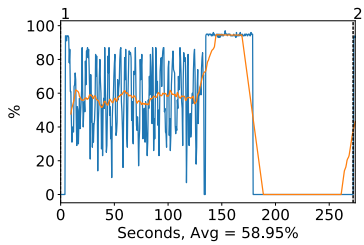
(b) Multiclass first epoch



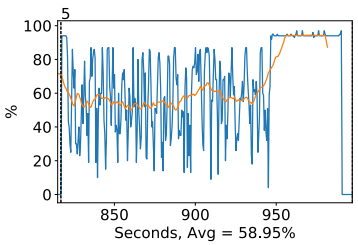
(c) Multiclass final epoch



(d) Binary all epochs

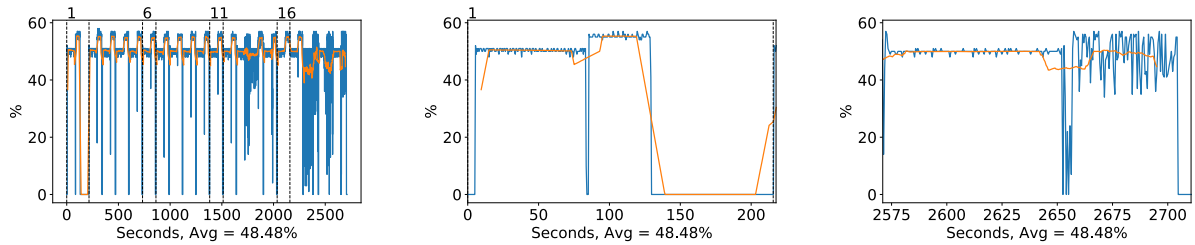


(e) Binary first epoch

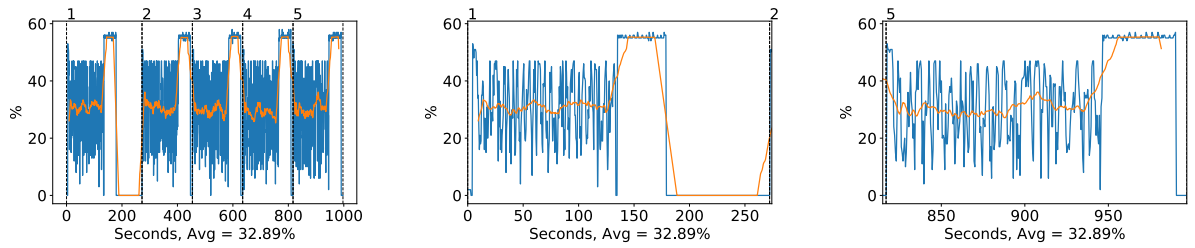


(f) Binary final epoch

Figure A.87: NASNet Large GPU volatile usage during training, with a moving window average over 40 measurements overlaid

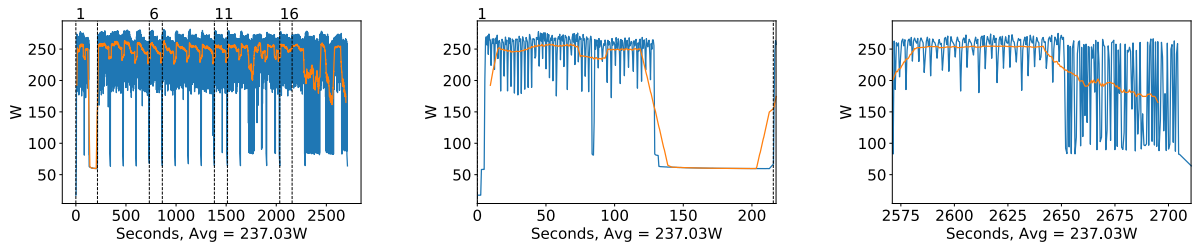


(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

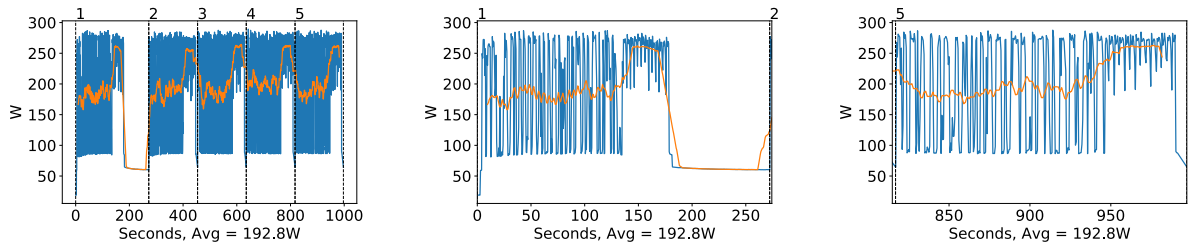


(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

Figure A.88: NASNet Large GPU memory usage during training, with a moving window average over 40 measurements overlaid

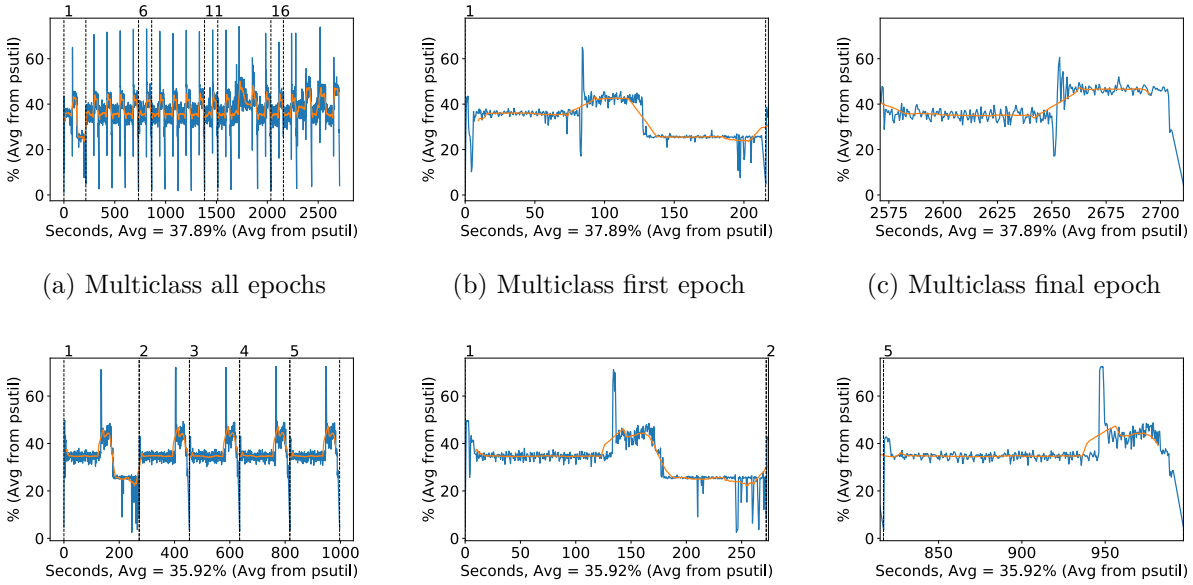


(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch



(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

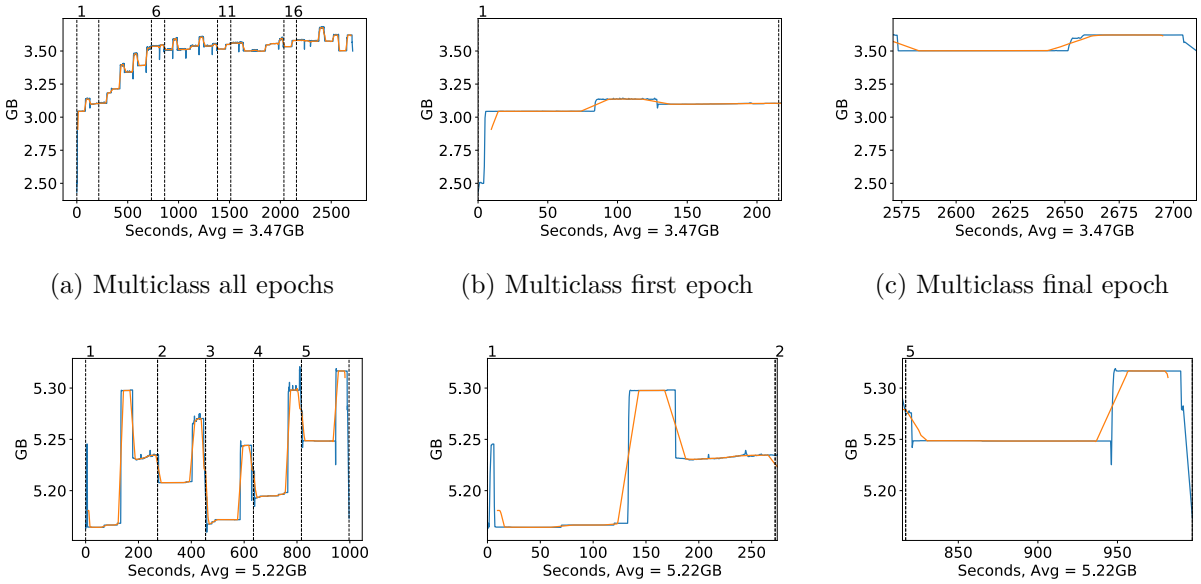
Figure A.89: NASNet Large GPU power usage during training in W, with a moving window average over 40 measurements overlaid



(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

Figure A.90: NASNet Large CPU usage (averaged over 4 cores) during training, with a moving window average over 40 measurements overlaid



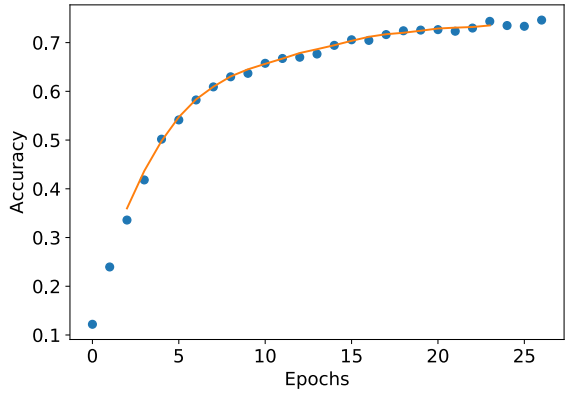
(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

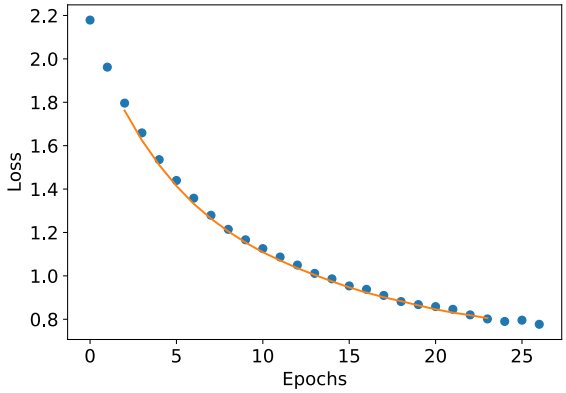
Figure A.91: NASNet Large Memory usage during training, with a moving window average over 40 measurements overlaid

A.2.11 NasNet Mobile

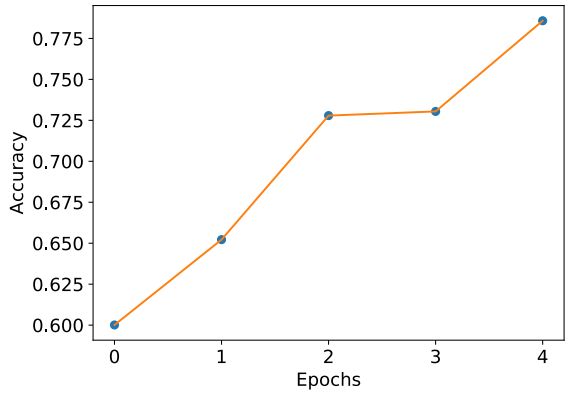
Classification Accuracy and Loss



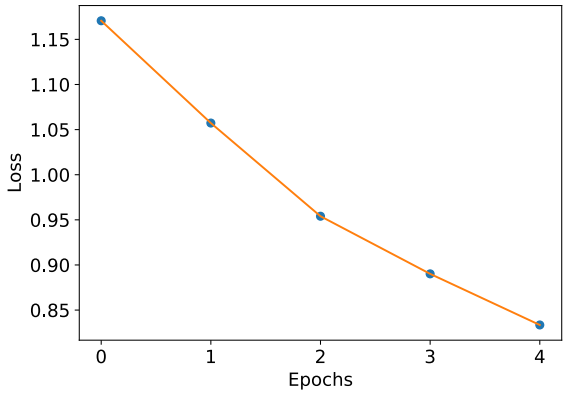
(a) Multiclass network accuracy



(b) Multiclass network loss



(c) Binary network "polyps" accuracy

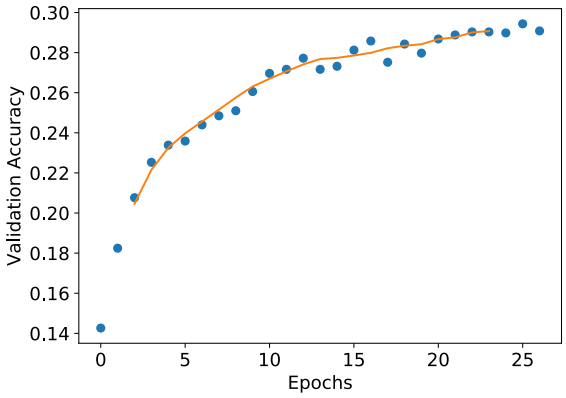


(d) Binary network "polyps" loss

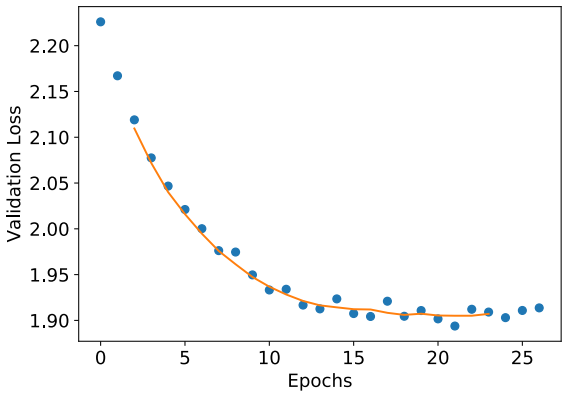
Figure A.92: NASNet Mobile Training classification accuracy and loss history for training data

GPU Usage

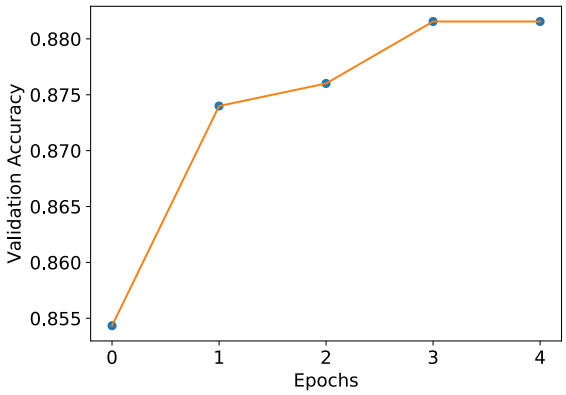
CPU and memory usage



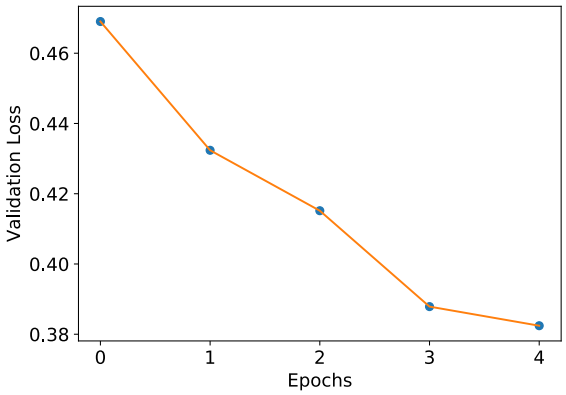
(a) Multiclass network accuracy



(b) Multiclass network loss

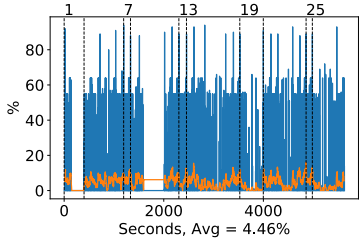


(c) Binary network "polyps" accuracy

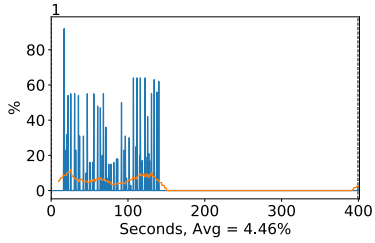


(d) Binary network "polyps" loss

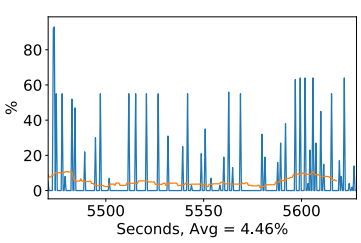
Figure A.93: NASNet Mobile Training classification accuracy and loss history for validation data



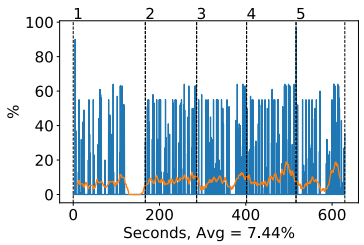
(a) Multiclass all epochs



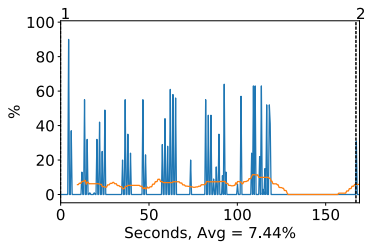
(b) Multiclass first epoch



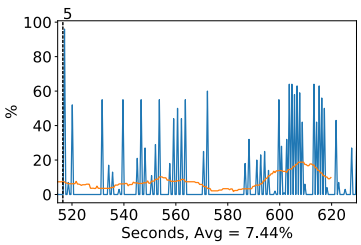
(c) Multiclass final epoch



(d) Binary all epochs

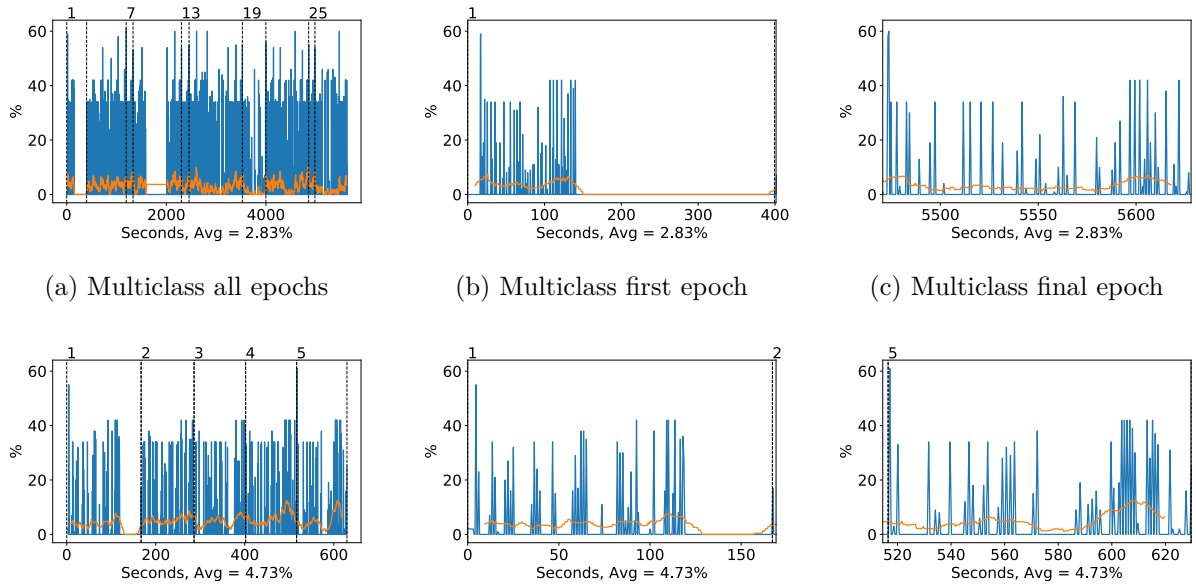


(e) Binary first epoch



(f) Binary final epoch

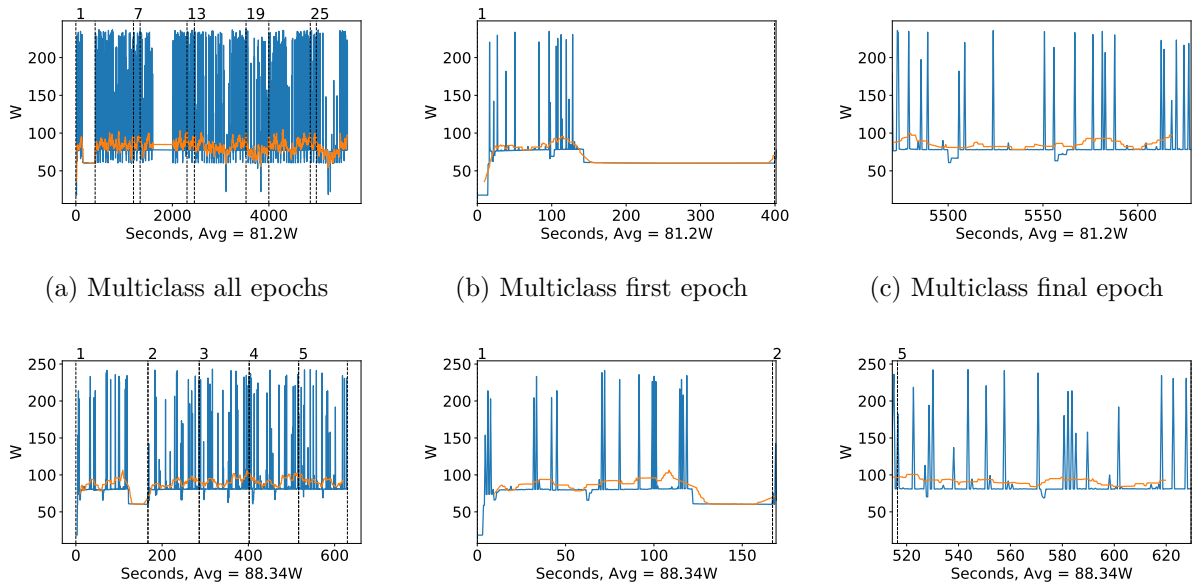
Figure A.94: NASNet Mobile GPU volatile usage during training, with a moving window average over 40 measurements overlaid



(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

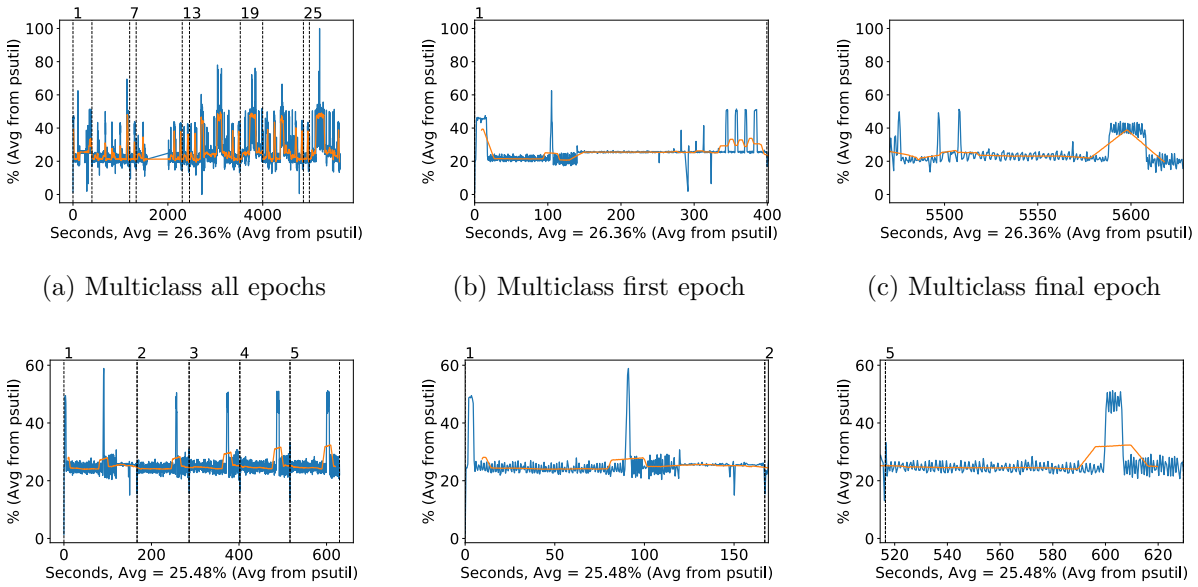
Figure A.95: NASNet Mobile GPU memory usage during training, with a moving window average over 40 measurements overlaid



(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

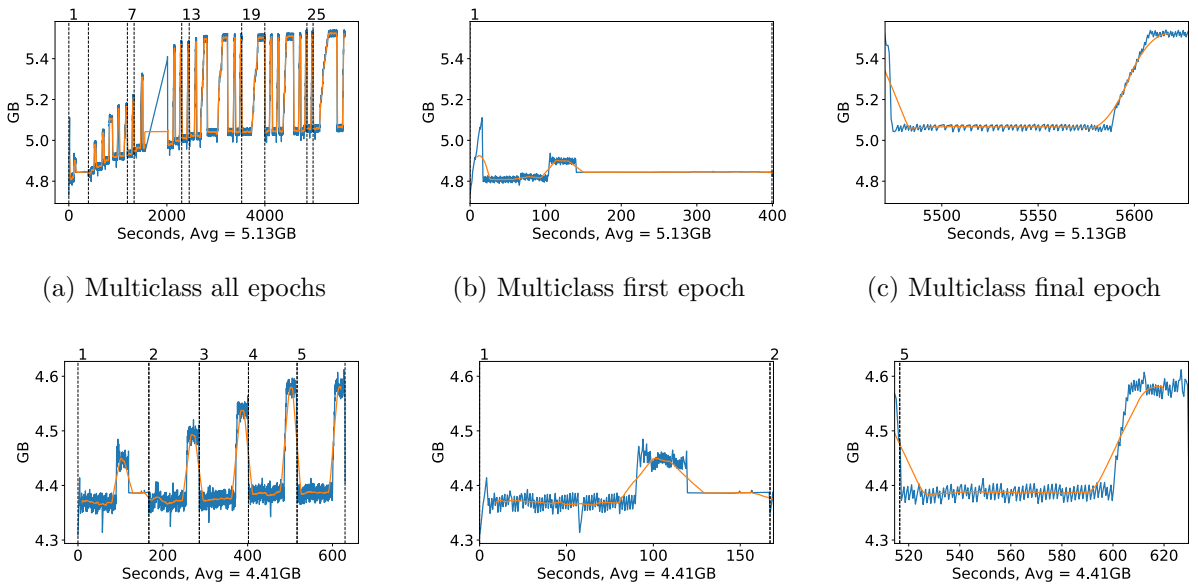
Figure A.96: NASNet Mobile GPU power usage during training in W, with a moving window average over 40 measurements overlaid



(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

Figure A.97: NASNet Mobile CPU usage (averaged over 4 cores) during training, with a moving window average over 40 measurements overlaid



(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

Figure A.98: NASNet Mobile Memory usage during training, with a moving window average over 40 measurements overlaid

A.3 Fine-tuning

A.3.1 Average Power Use

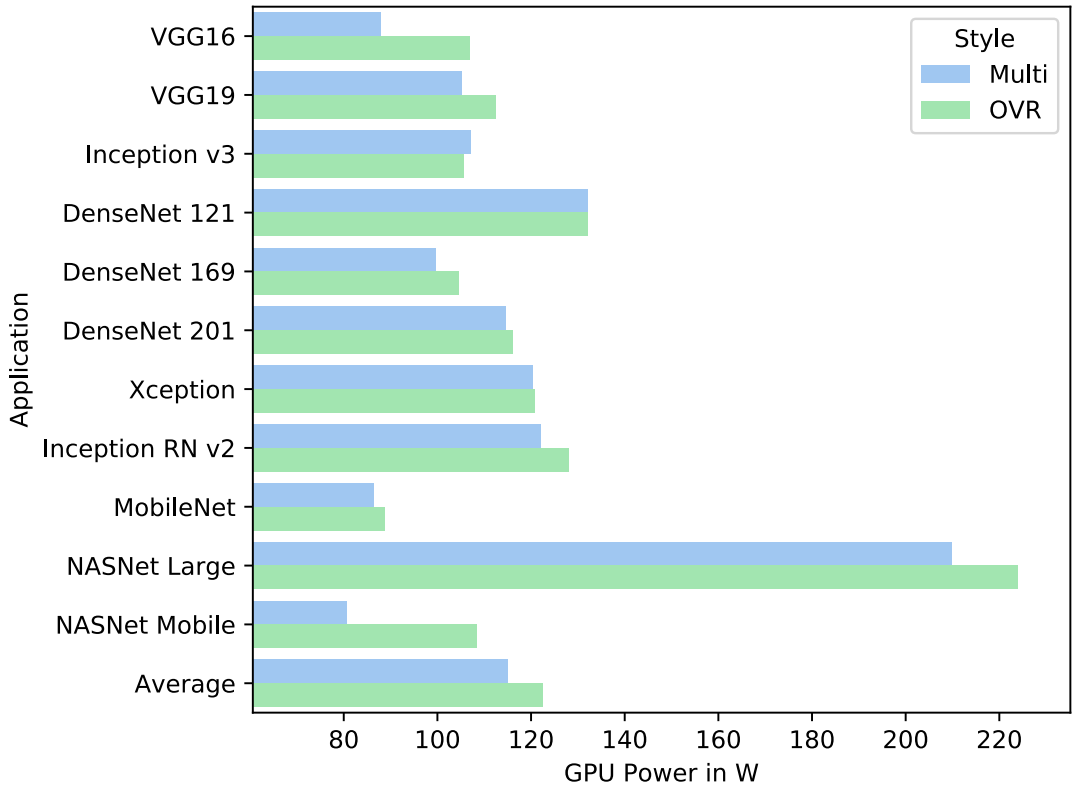


Figure A.99: Fine-tuning: GPU Power in W, including all applications and an average value. For both binary and multiclass styles.

A.3.2 Memory Use

A.3.3 VGG16

Classification Accuracy and Loss

GPU Usage

CPU and memory usage

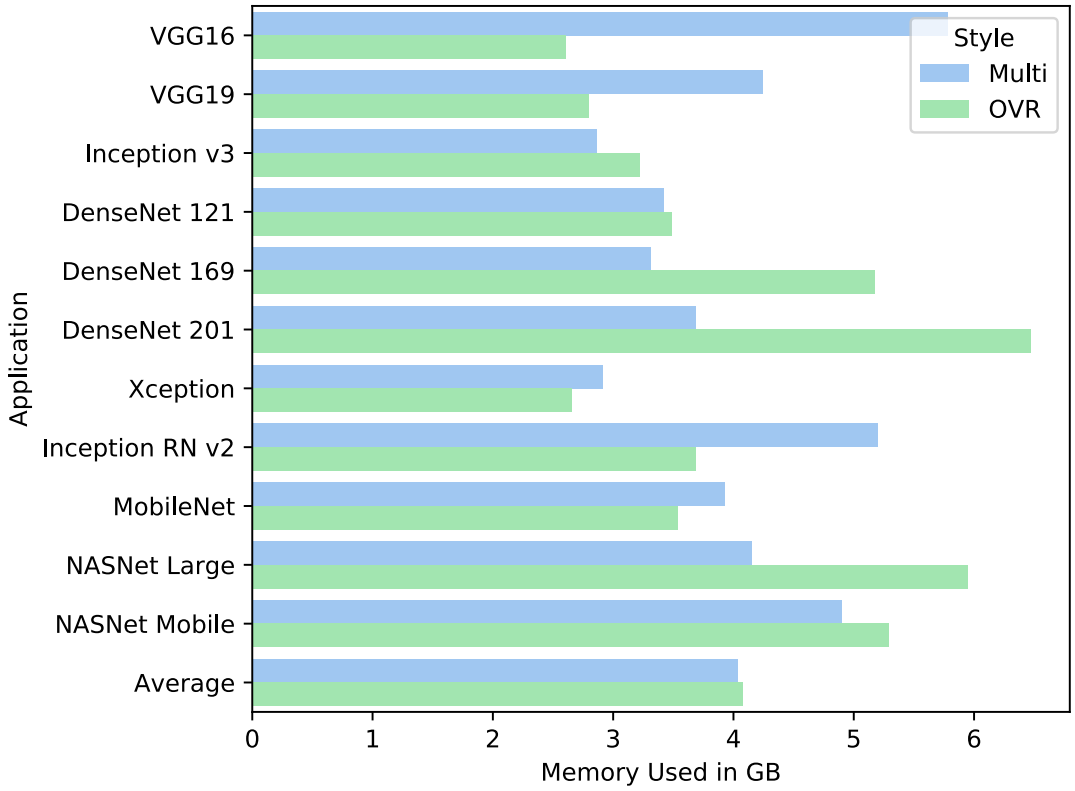
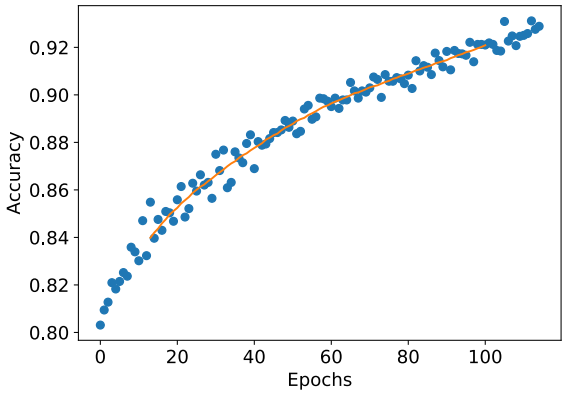
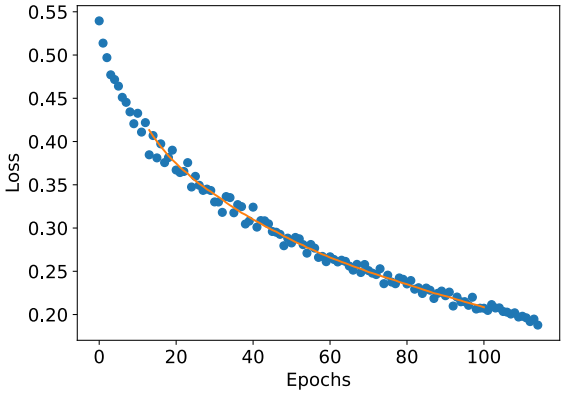


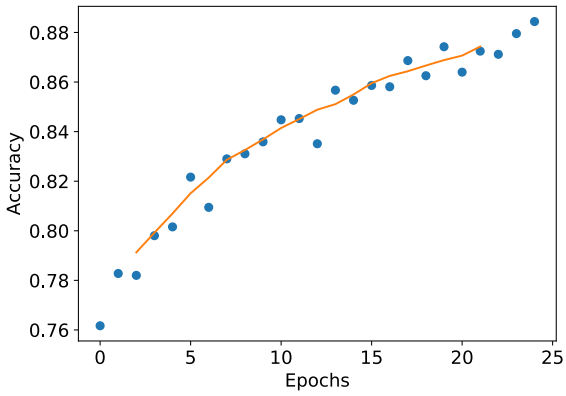
Figure A.100: Fine-tuning: Memory Used in GB, including all applications and an average value. For both binary and multiclass styles.



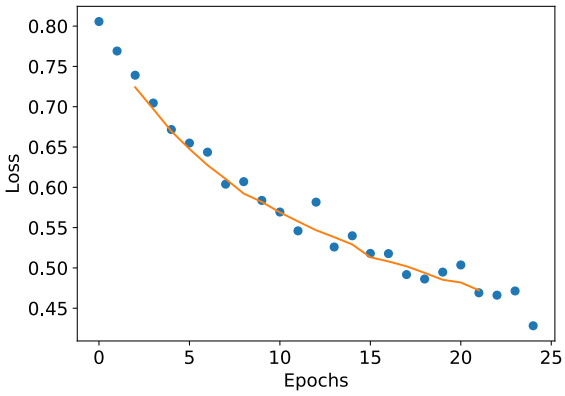
(a) Multiclass network accuracy



(b) Multiclass network loss

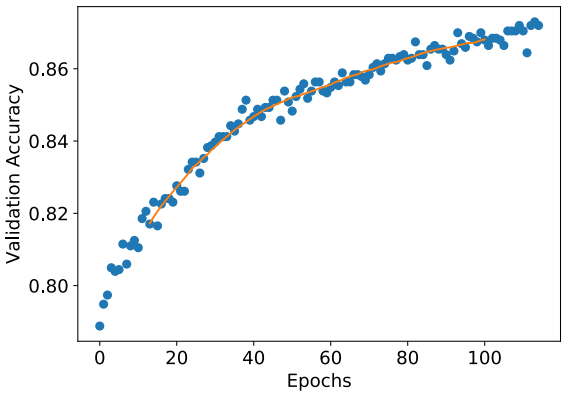


(c) Binary network "polyps" accuracy

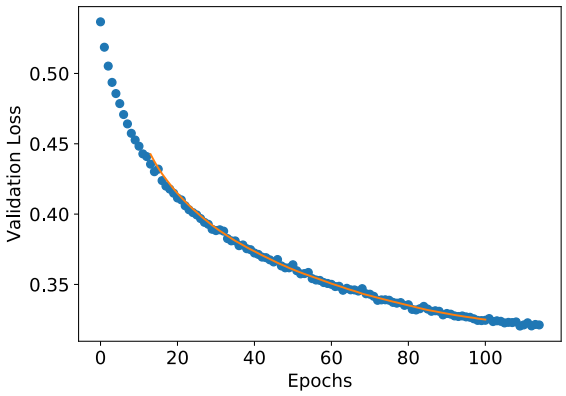


(d) Binary network "polyps" loss

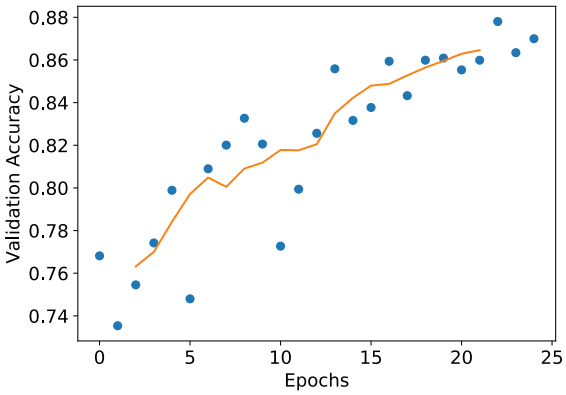
Figure A.101: VGG16 Fine-tuning classification accuracy and loss history for training data



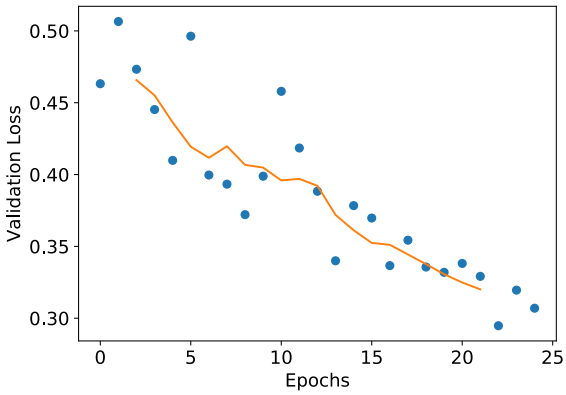
(a) Multiclass network accuracy



(b) Multiclass network loss

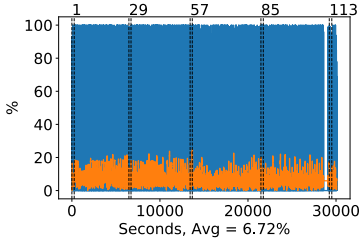


(c) Binary network "polyps" accuracy

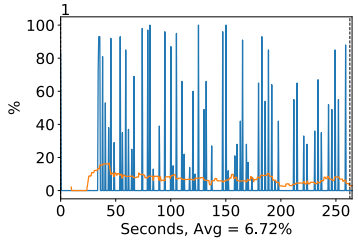


(d) Binary network "polyps" loss

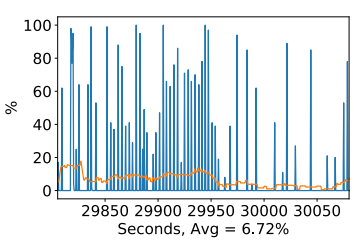
Figure A.102: VGG16 Fine tuning classification accuracy and loss history for validation data



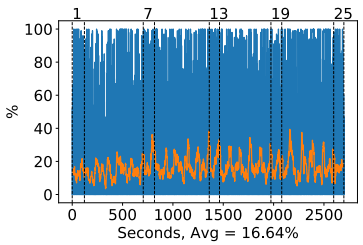
(a) Multiclass all epochs



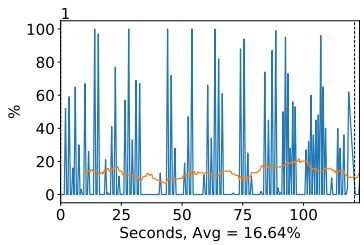
(b) Multiclass first epoch



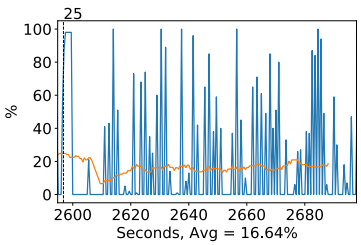
(c) Multiclass final epoch



(d) Binary all epochs

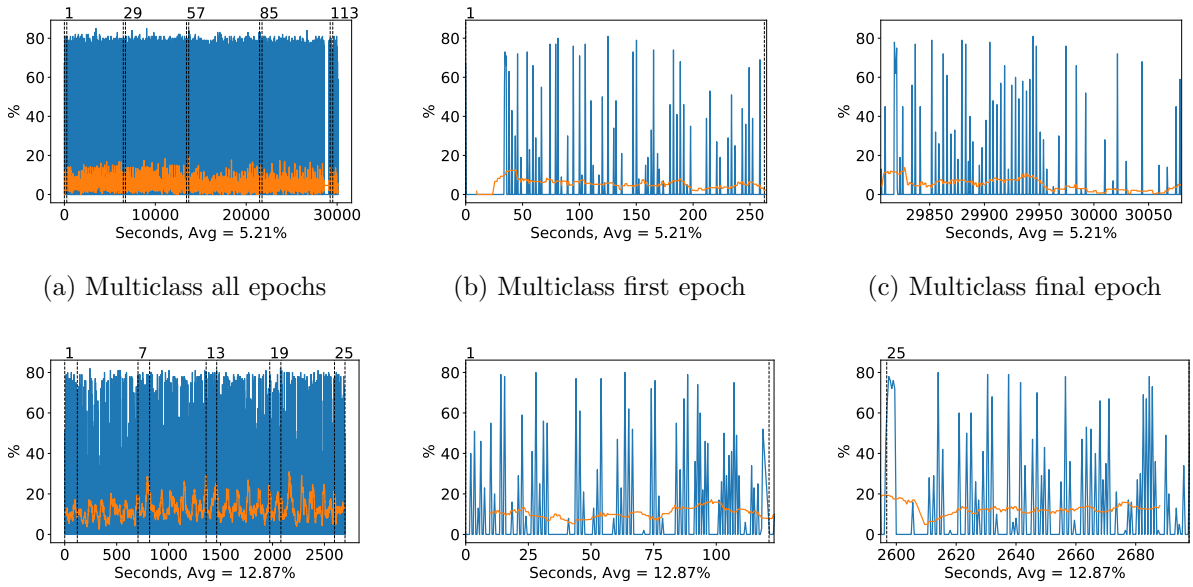


(e) Binary first epoch



(f) Binary final epoch

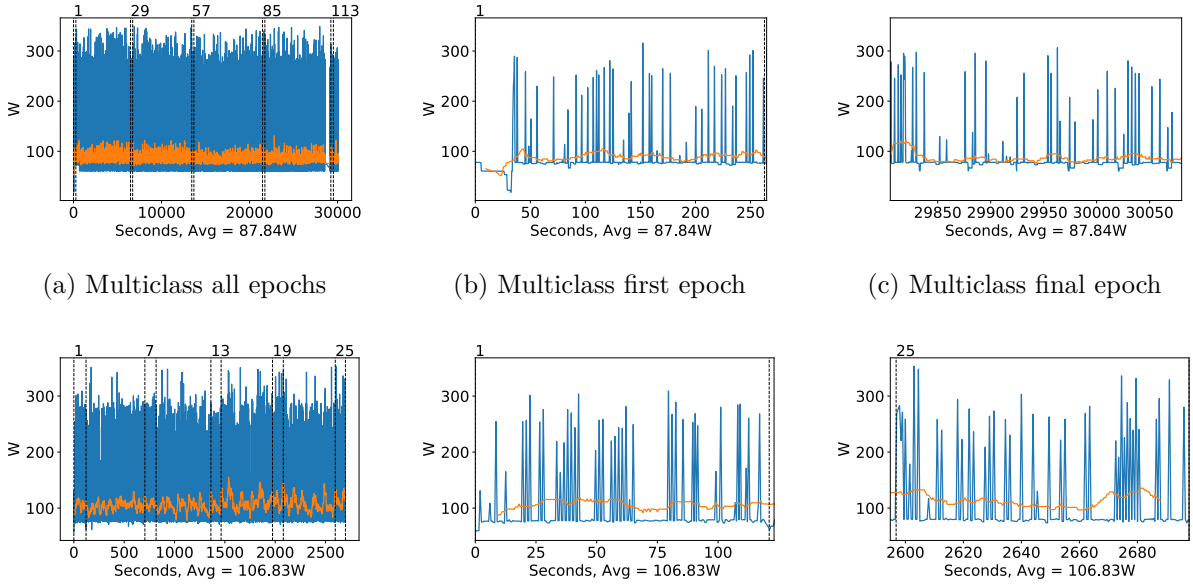
Figure A.103: VGG16 GPU volatile usage during fine tuning, with a moving window average over 40 measurements overlaid



(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

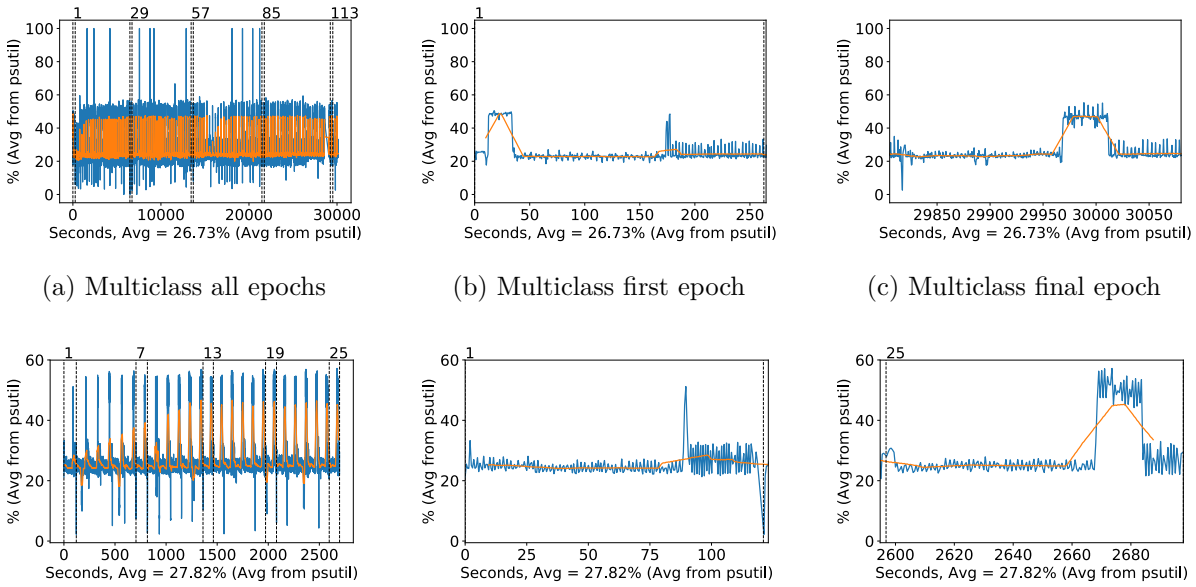
Figure A.104: VGG16 GPU memory usage during fine tuning, with a moving window average over 40 measurements overlaid



(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

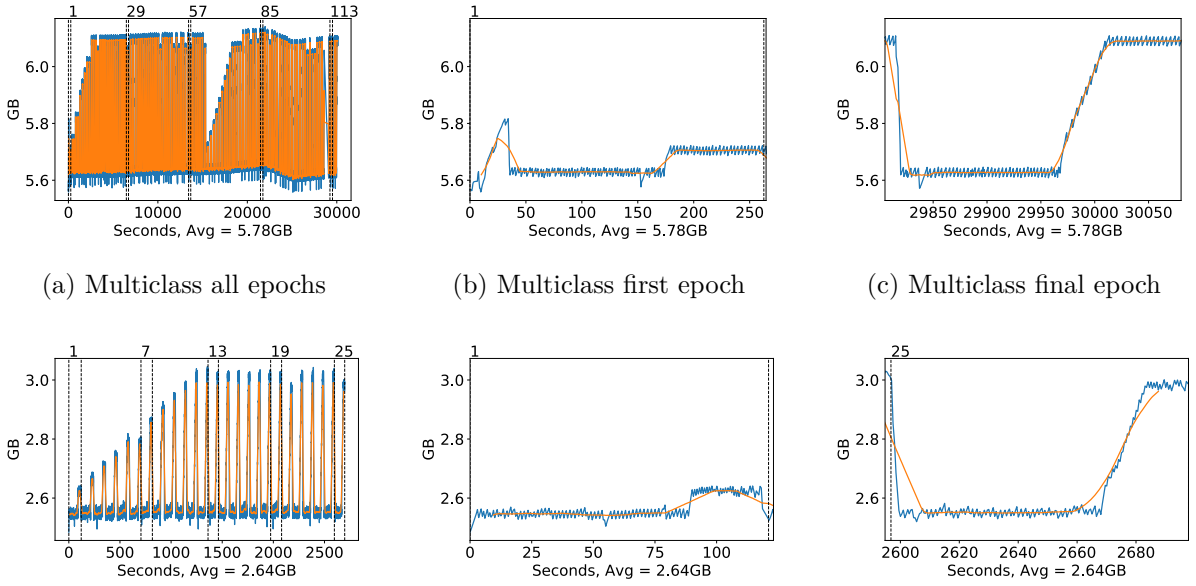
Figure A.105: VGG16 GPU power usage during fine tuning in W, with a moving window average over 40 measurements overlaid



(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

Figure A.106: VGG16 CPU usage (averaged over 4 cores) during fine tuning, with a moving window average over 40 measurements overlaid



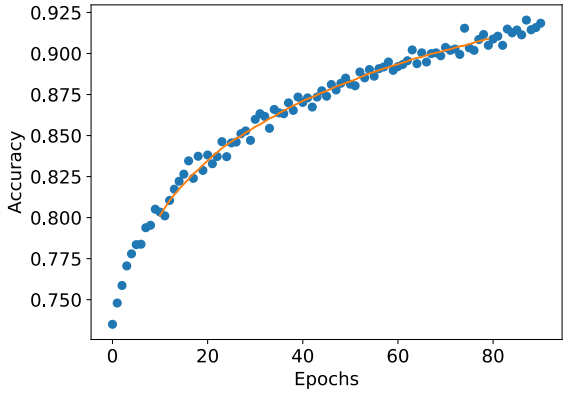
(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

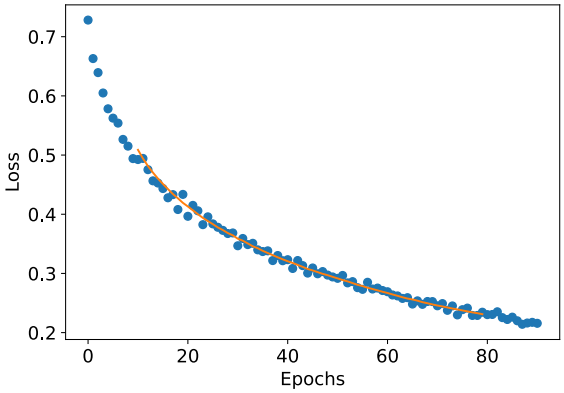
Figure A.107: VGG16 Memory usage during fine tuning, with a moving window average over 40 measurements overlaid

A.3.4 VGG19

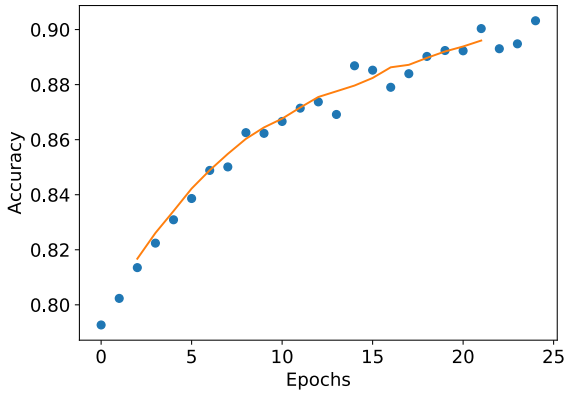
Classification Accuracy and Loss



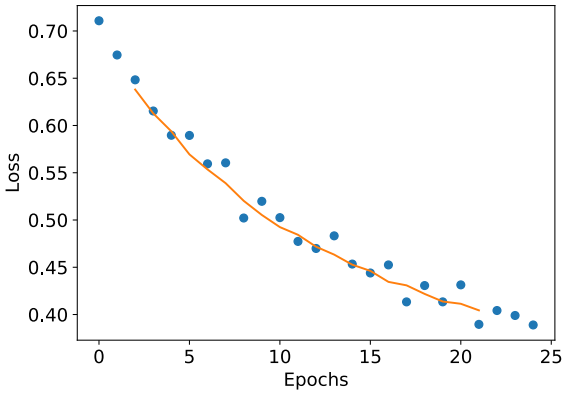
(a) Multiclass network accuracy



(b) Multiclass network loss



(c) Binary network "polyps" accuracy

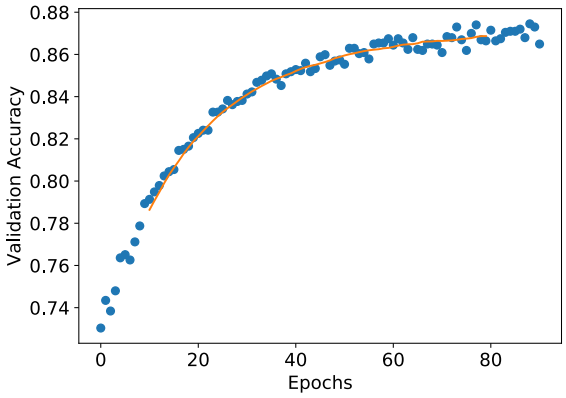


(d) Binary network "polyps" loss

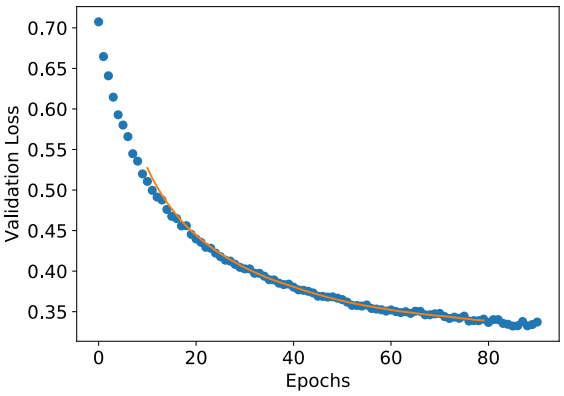
Figure A.108: VGG19 Fine-tuning classification accuracy and loss history for training data

GPU Usage

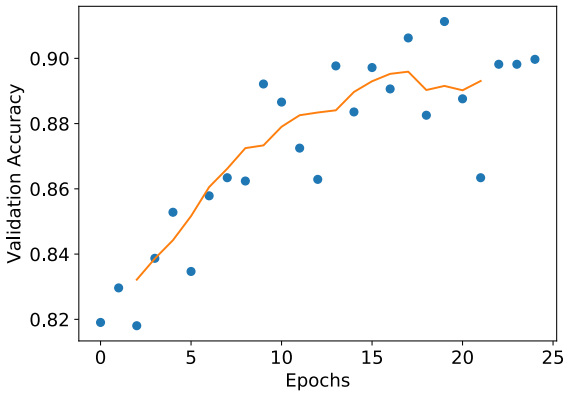
CPU and memory usage



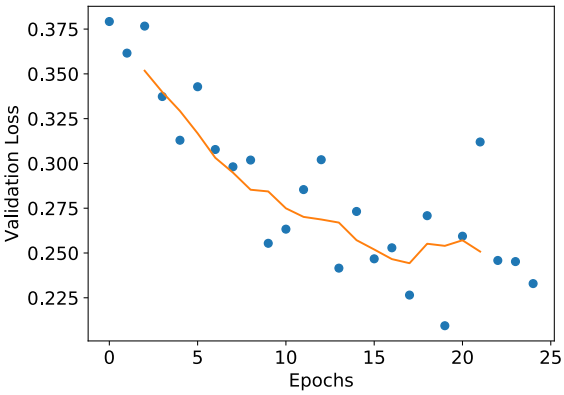
(a) Multiclass network accuracy



(b) Multiclass network loss

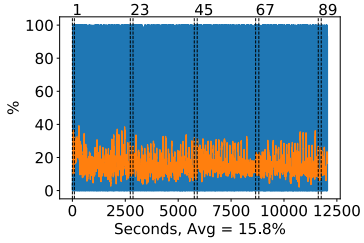


(c) Binary network "polyps" accuracy

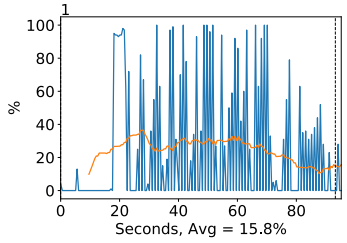


(d) Binary network "polyps" loss

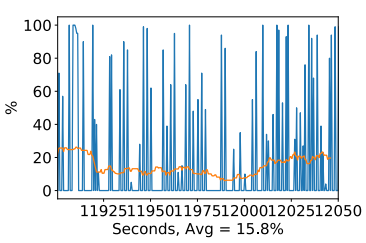
Figure A.109: VGG19 Fine tuning classification accuracy and loss history for validation data



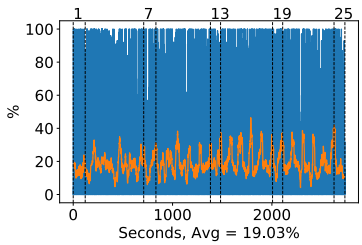
(a) Multiclass all epochs



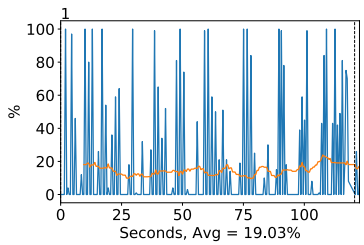
(b) Multiclass first epoch



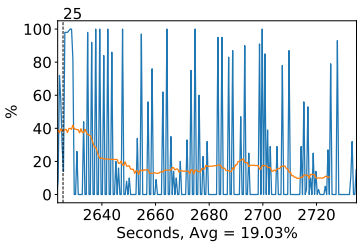
(c) Multiclass final epoch



(d) Binary all epochs

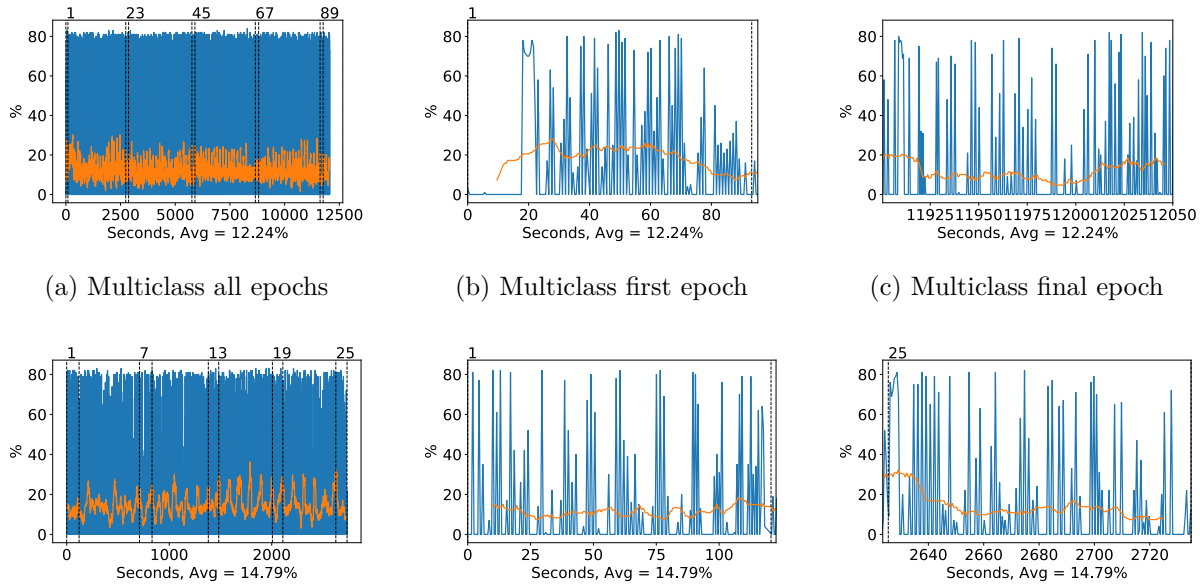


(e) Binary first epoch



(f) Binary final epoch

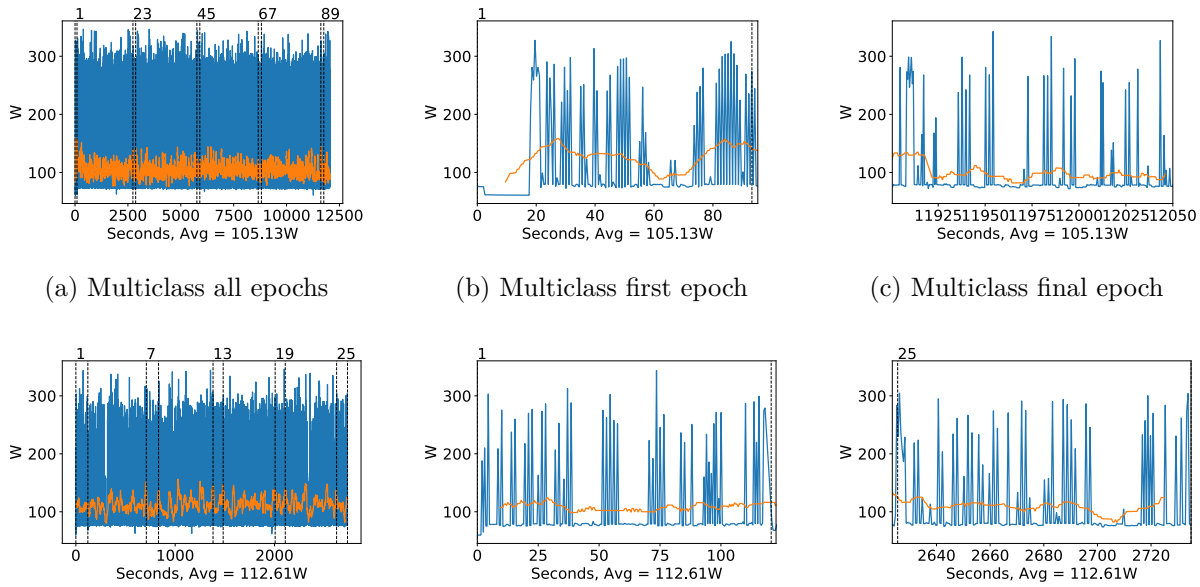
Figure A.110: VGG19 GPU volatile usage during fine tuning, with a moving window average over 40 measurements overlaid



(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

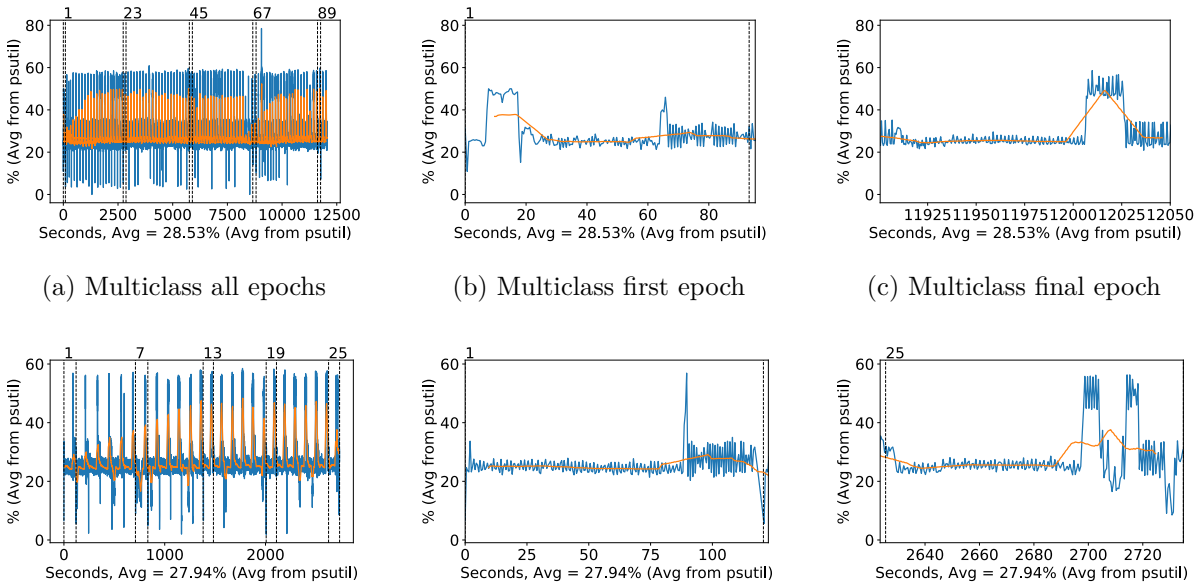
Figure A.111: VGG19 GPU memory usage during fine tuning, with a moving window average over 40 measurements overlaid



(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

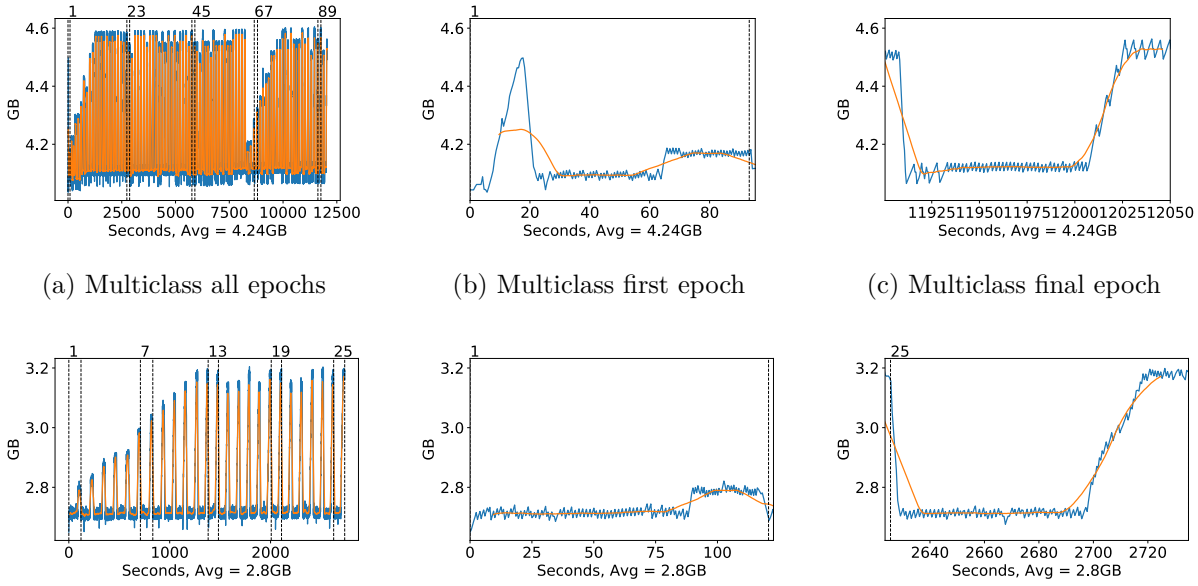
Figure A.112: VGG19 GPU power usage during fine tuning in W, with a moving window average over 40 measurements overlaid



(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

Figure A.113: VGG19 CPU usage (averaged over 4 cores) during fine tuning, with a moving window average over 40 measurements overlaid



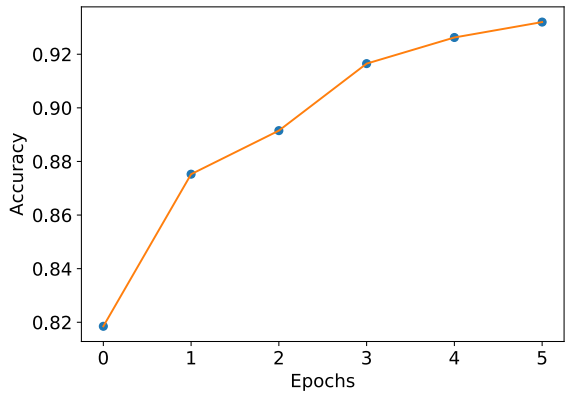
(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

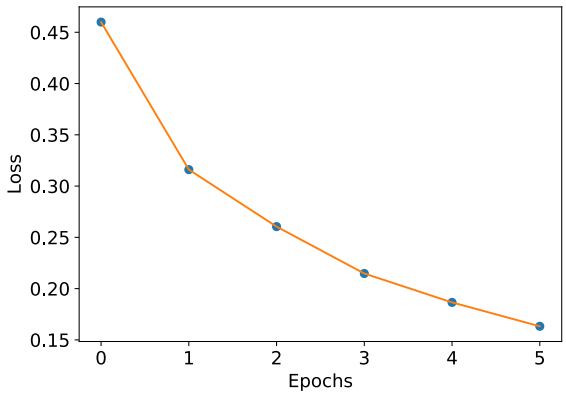
Figure A.114: VGG19 Memory usage during fine tuning, with a moving window average over 40 measurements overlaid

A.3.5 Inception v3

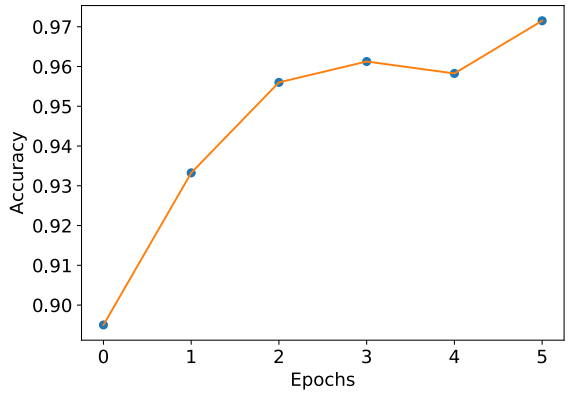
Classification Accuracy and Loss



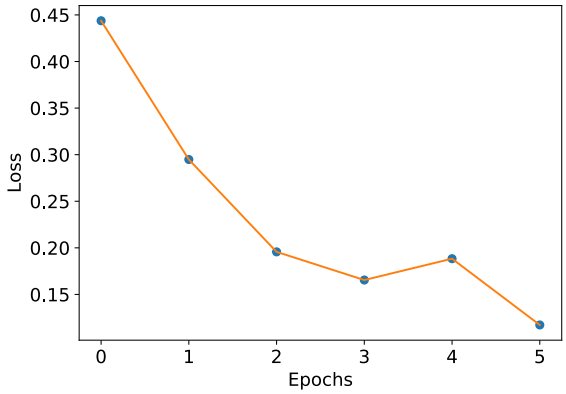
(a) Multiclass network accuracy



(b) Multiclass network loss



(c) Binary network "polyps" accuracy

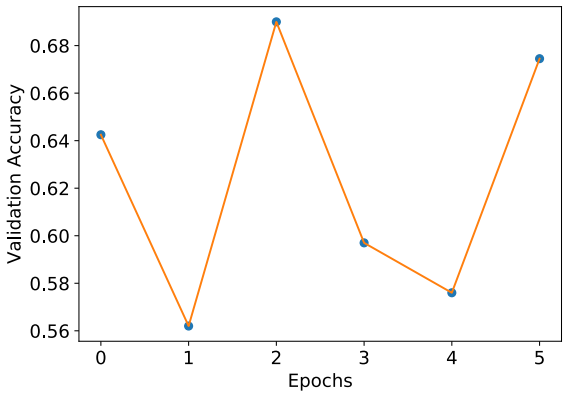


(d) Binary network "polyps" loss

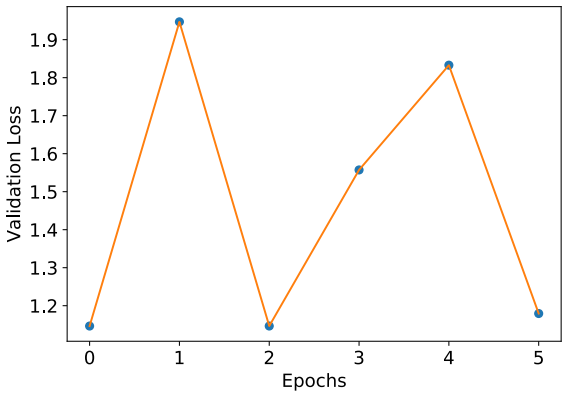
Figure A.115: Inception v3 Fine-tuning classification accuracy and loss history for training data

GPU Usage

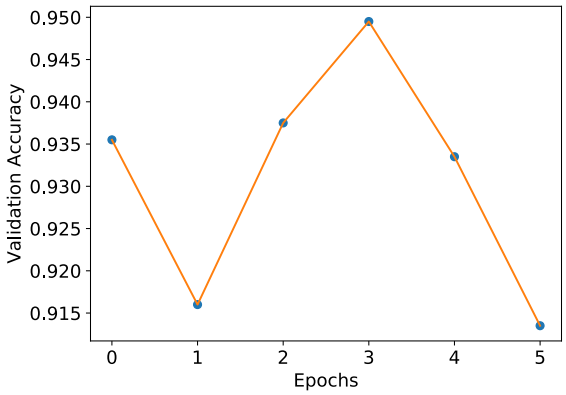
CPU and memory usage



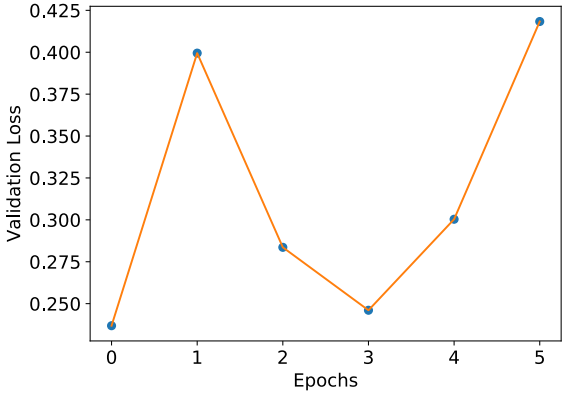
(a) Multiclass network accuracy



(b) Multiclass network loss

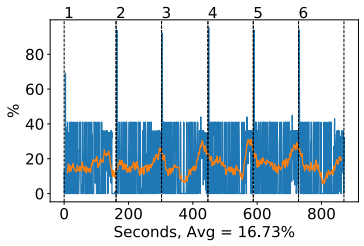


(c) Binary network "polyps" accuracy

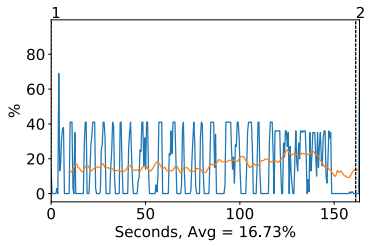


(d) Binary network "polyps" loss

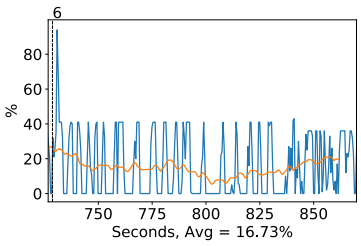
Figure A.116: Inception v3 Fine tuning classification accuracy and loss history for validation data



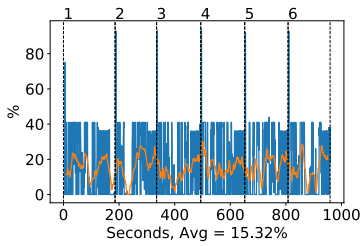
(a) Multiclass all epochs



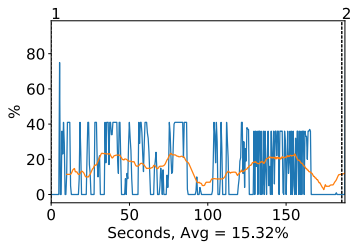
(b) Multiclass first epoch



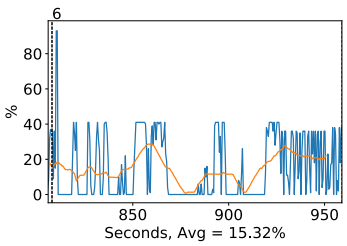
(c) Multiclass final epoch



(d) Binary all epochs

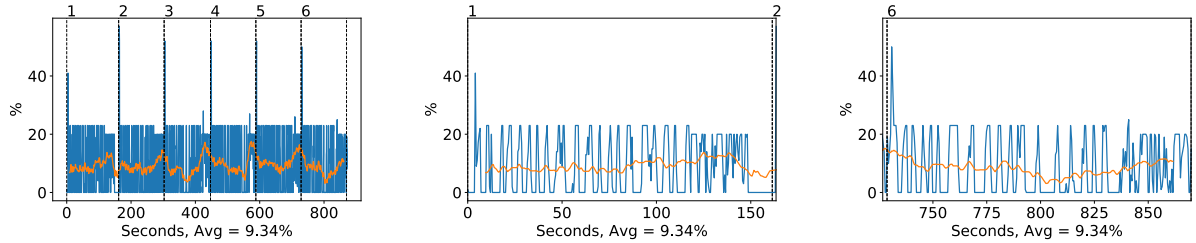


(e) Binary first epoch

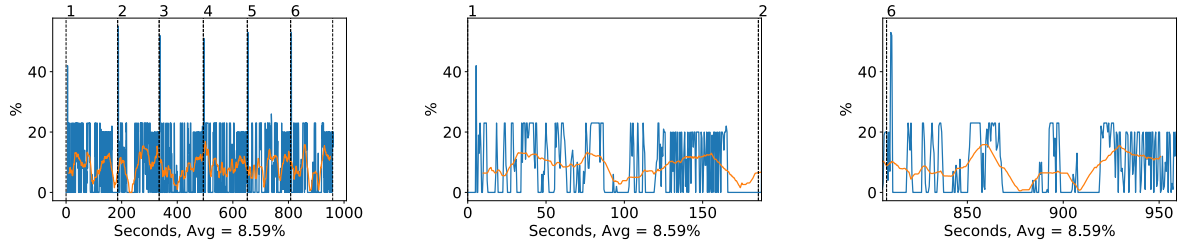


(f) Binary final epoch

Figure A.117: Inception v3 GPU volatile usage during fine tuning, with a moving window average over 40 measurements overlaid

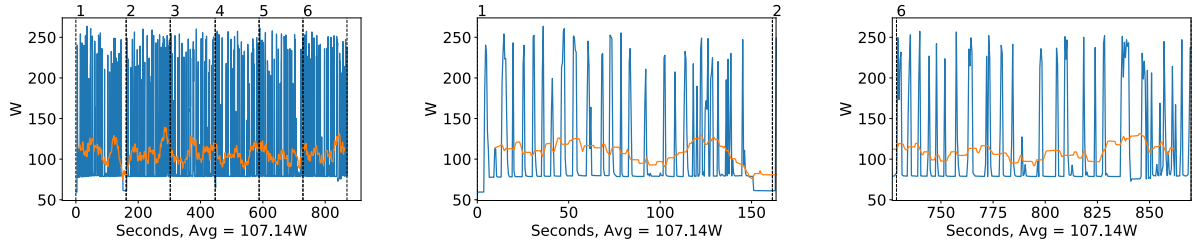


(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

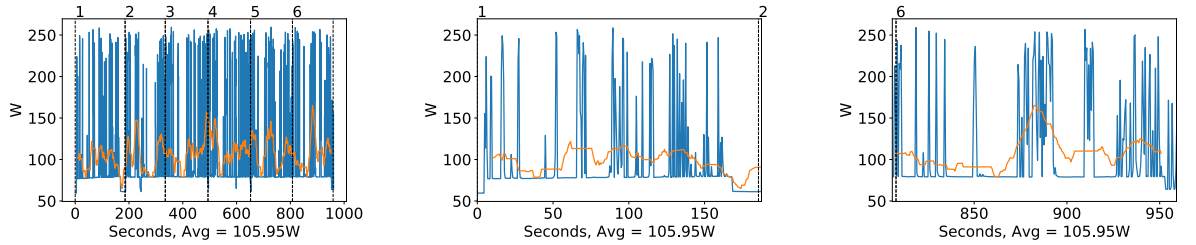


(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

Figure A.118: Inception v3 GPU memory usage during fine tuning, with a moving window average over 40 measurements overlaid

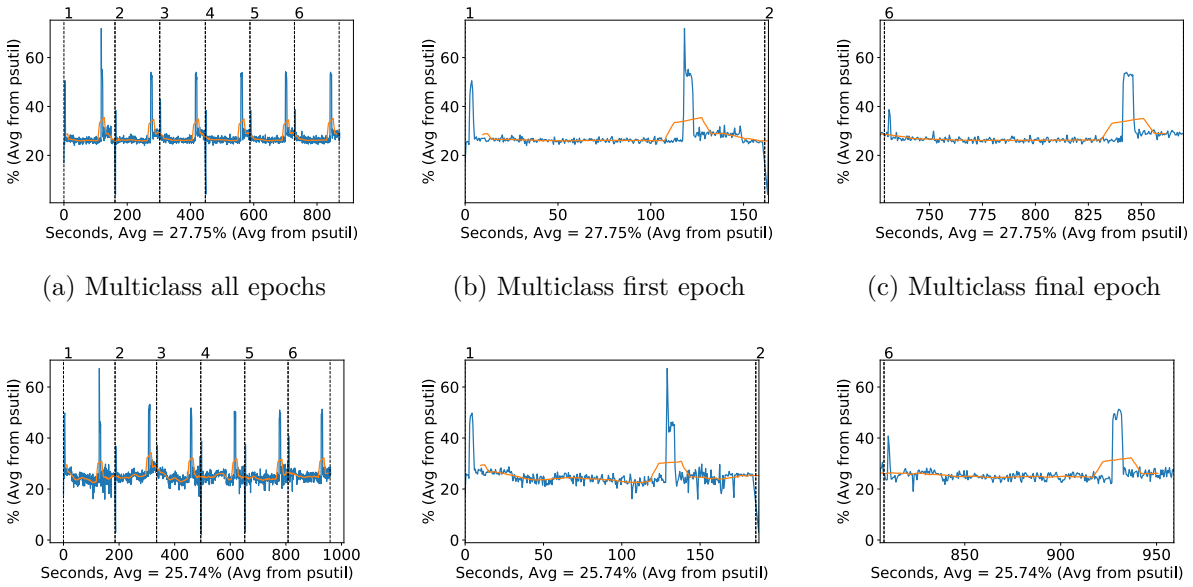


(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch



(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

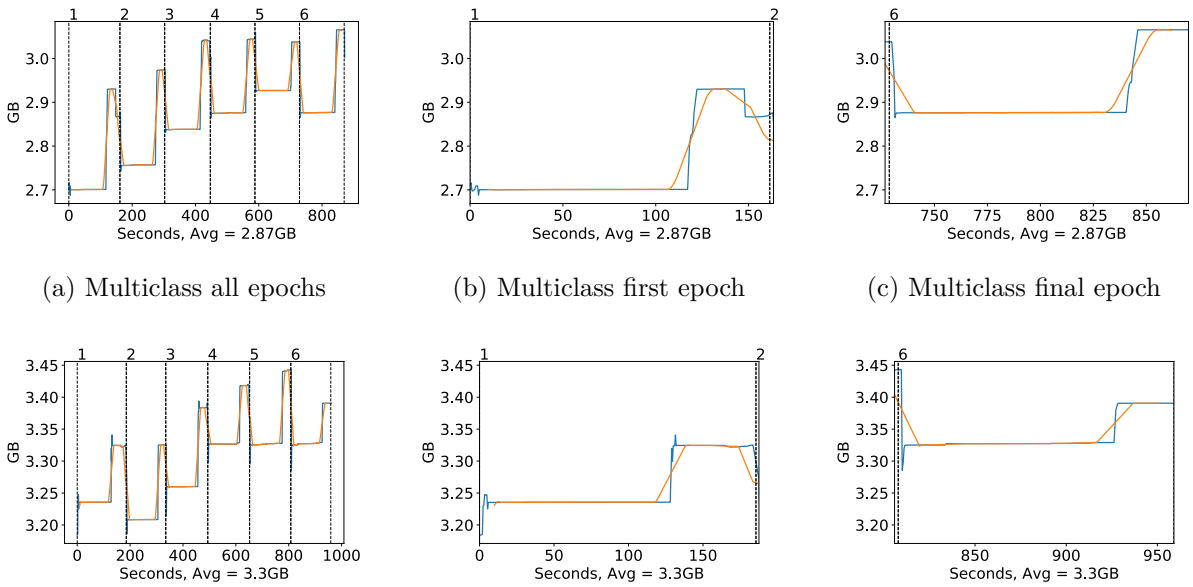
Figure A.119: Inception v3 GPU power usage during fine tuning in W, with a moving window average over 40 measurements overlaid



(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

Figure A.120: Inception v3 CPU usage (averaged over 4 cores) during fine tuning, with a moving window average over 40 measurements overlaid



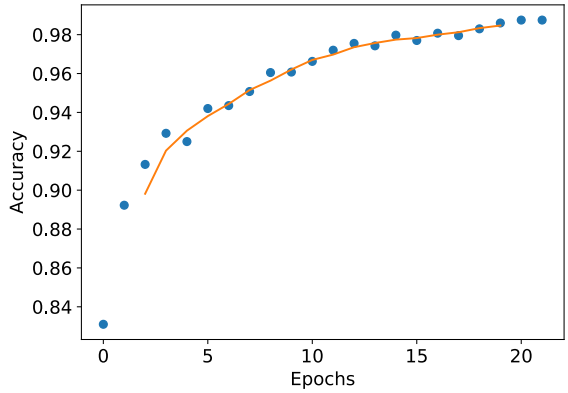
(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

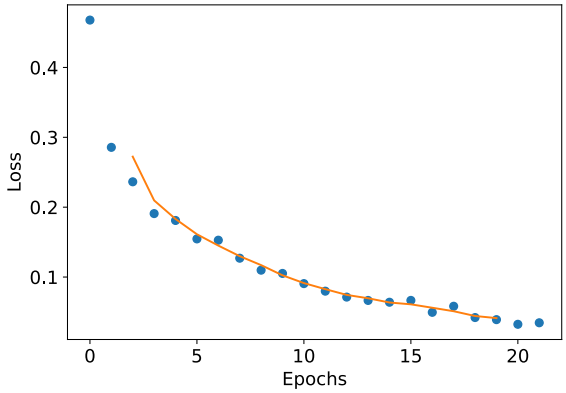
Figure A.121: Inception v3 Memory usage during fine tuning, with a moving window average over 40 measurements overlaid

A.3.6 DenseNet 121

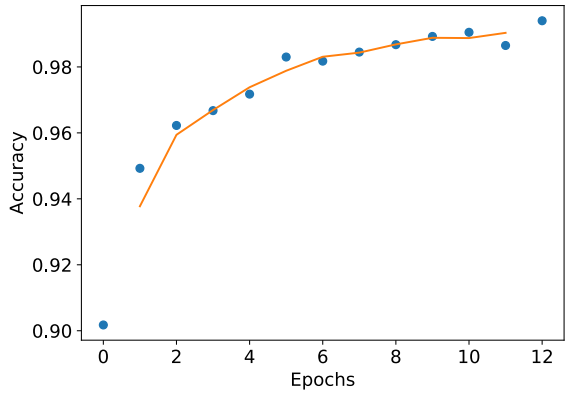
Classification Accuracy and Loss



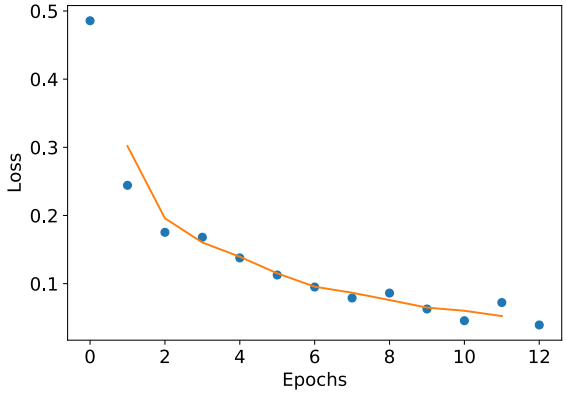
(a) Multiclass network accuracy



(b) Multiclass network loss



(c) Binary network "polyps" accuracy

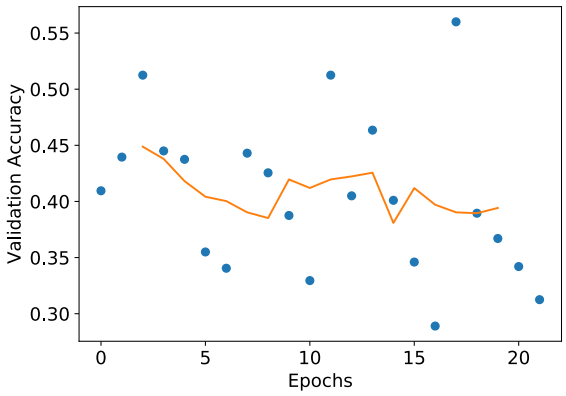


(d) Binary network "polyps" loss

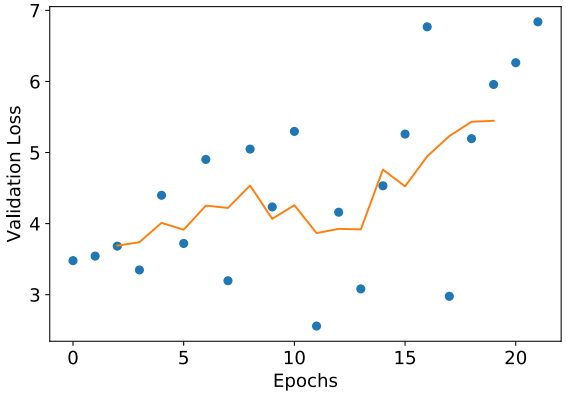
Figure A.122: DenseNet 121 Fine-tuning classification accuracy and loss history for training data

GPU Usage

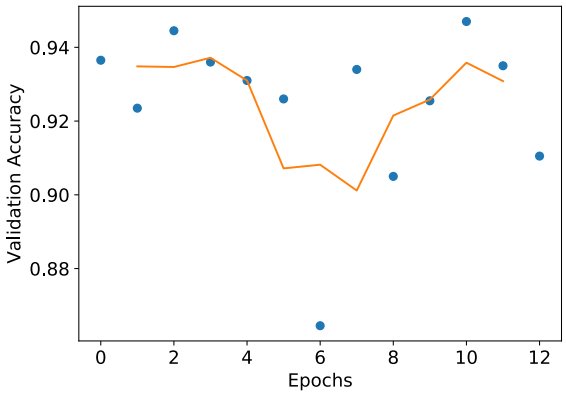
CPU and memory usage



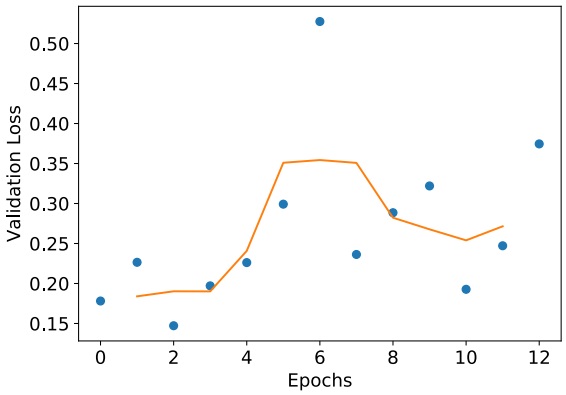
(a) Multiclass network accuracy



(b) Multiclass network loss

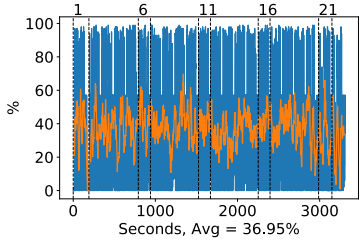


(c) Binary network "polyps" accuracy

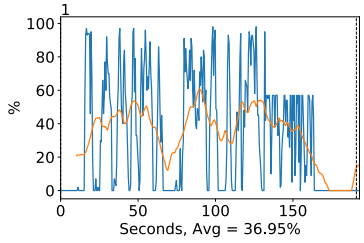


(d) Binary network "polyps" loss

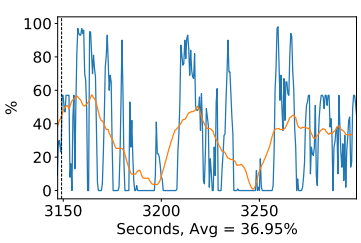
Figure A.123: DenseNet 121 Fine tuning classification accuracy and loss history for validation data



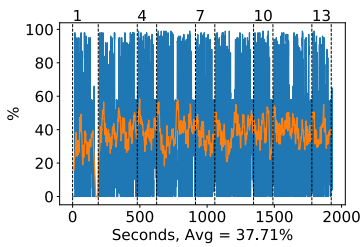
(a) Multiclass all epochs



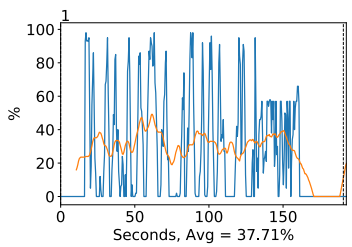
(b) Multiclass first epoch



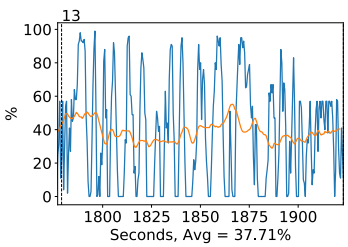
(c) Multiclass final epoch



(d) Binary all epochs

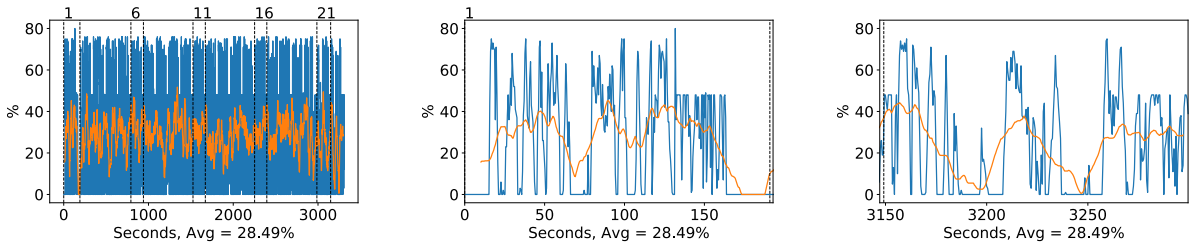


(e) Binary first epoch

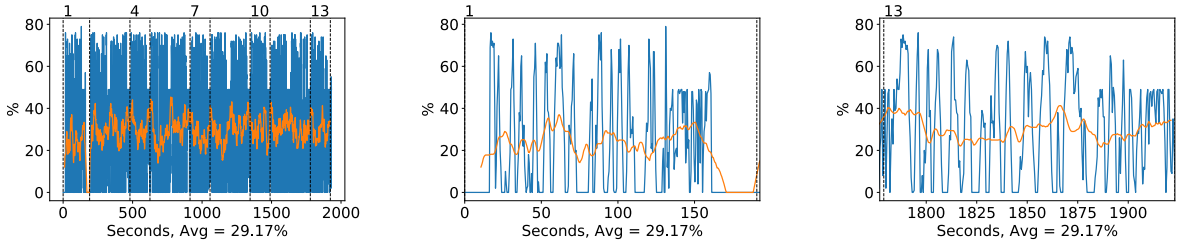


(f) Binary final epoch

Figure A.124: DenseNet 121 GPU volatile usage during fine tuning, with a moving window average over 40 measurements overlaid

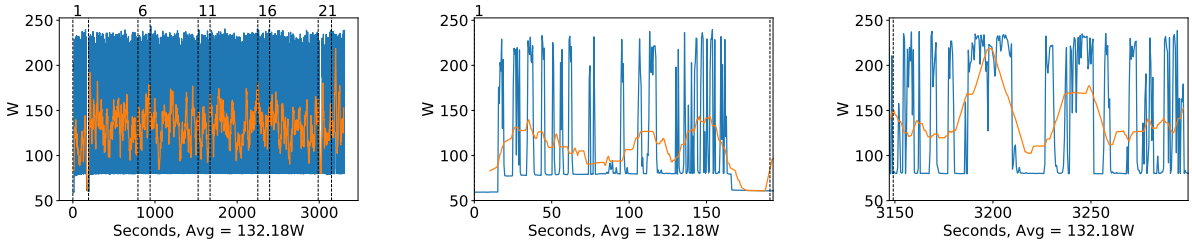


(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

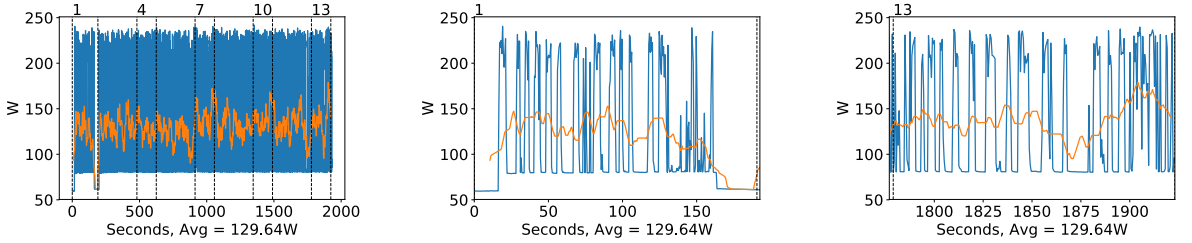


(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

Figure A.125: DenseNet 121 GPU memory usage during fine tuning, with a moving window average over 40 measurements overlaid

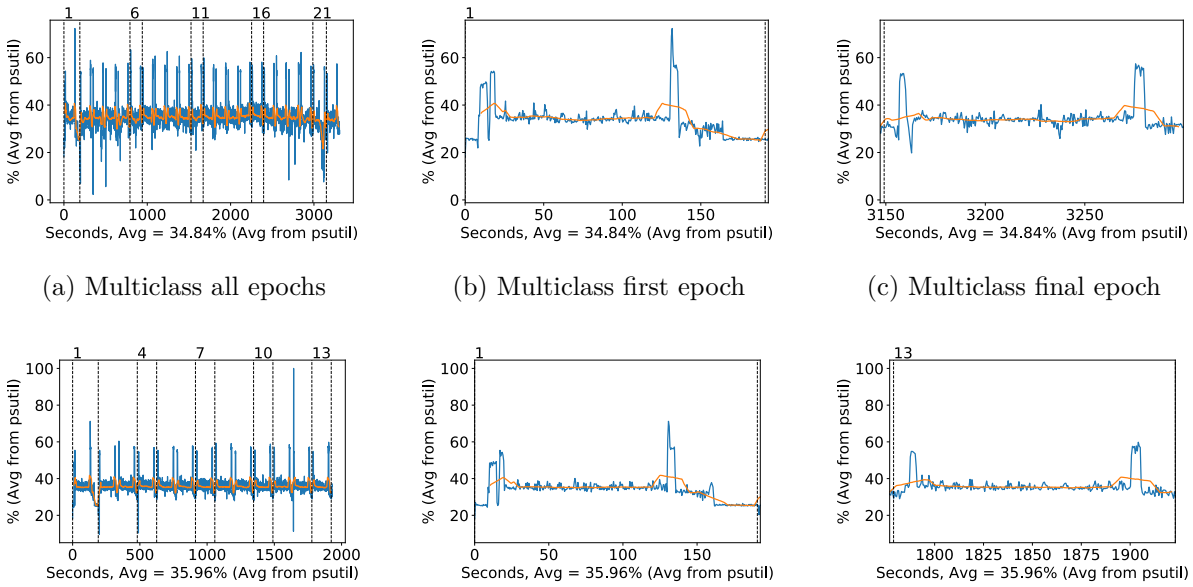


(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch



(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

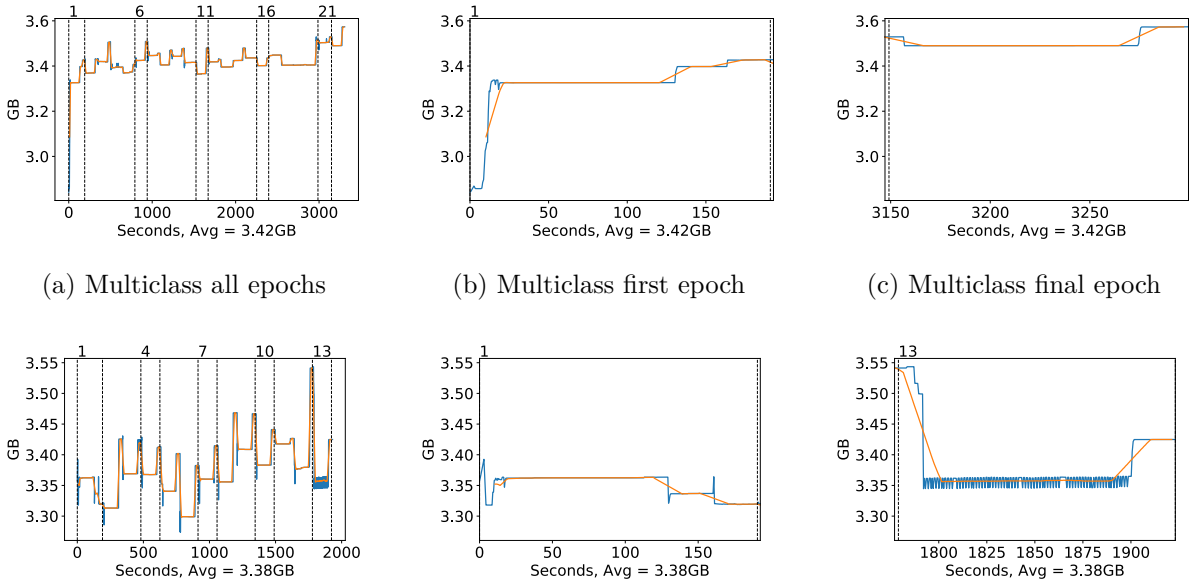
Figure A.126: DenseNet 121 GPU power usage during fine tuning in W, with a moving window average over 40 measurements overlaid



(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

Figure A.127: DenseNet 121 CPU usage (averaged over 4 cores) during fine tuning, with a moving window average over 40 measurements overlaid



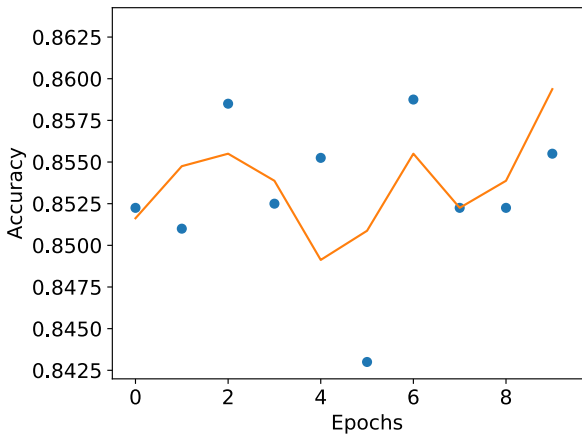
(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

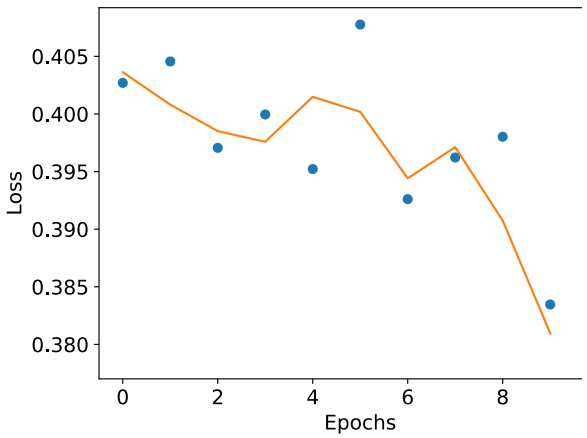
Figure A.128: DenseNet 121 Memory usage during fine tuning, with a moving window average over 40 measurements overlaid

A.3.7 DenseNet 169

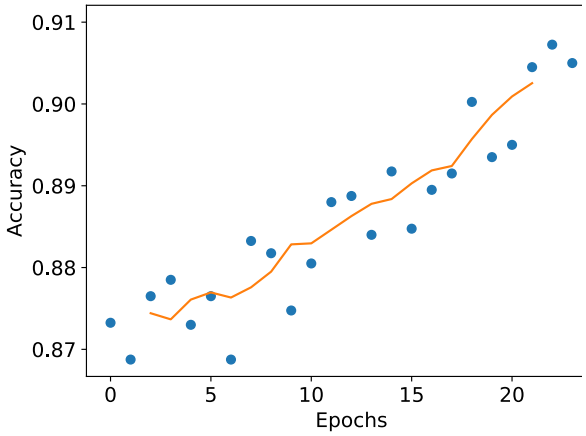
Classification Accuracy and Loss



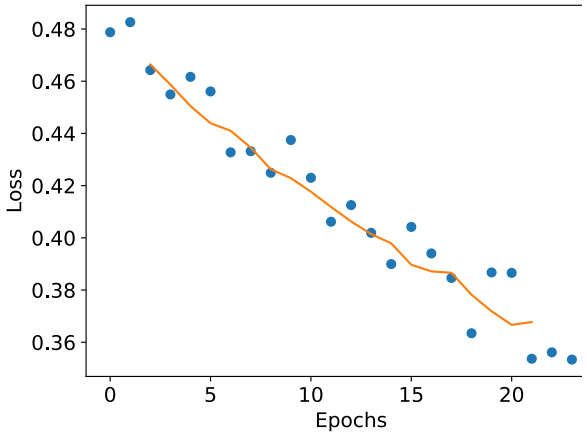
(a) Multiclass network accuracy



(b) Multiclass network loss



(c) Binary network "polyps" accuracy

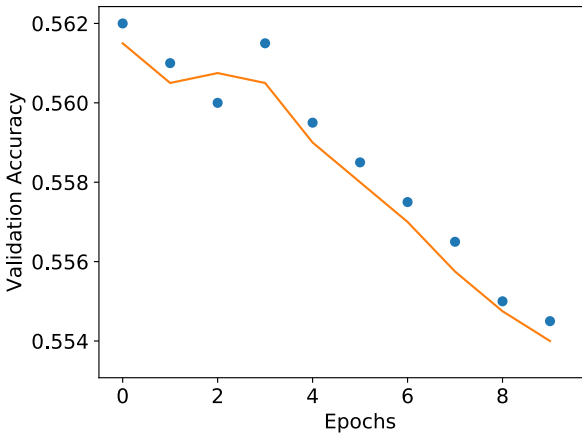


(d) Binary network "polyps" loss

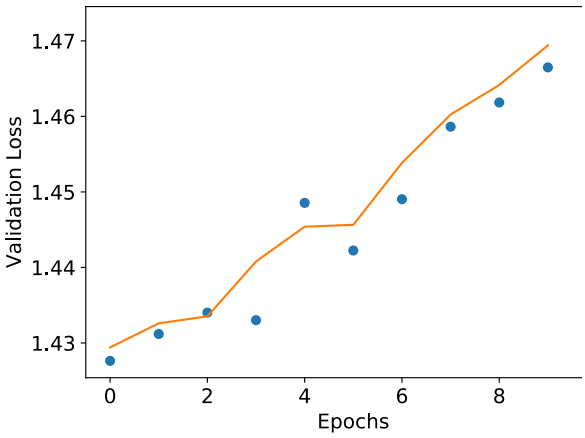
Figure A.129: DenseNet 169 Fine-tuning classification accuracy and loss history for training data

GPU Usage

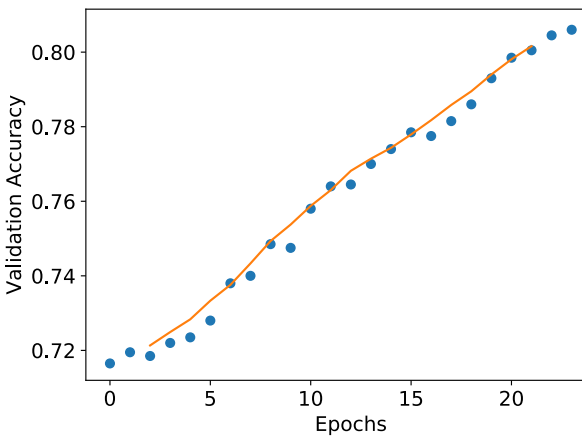
CPU and memory usage



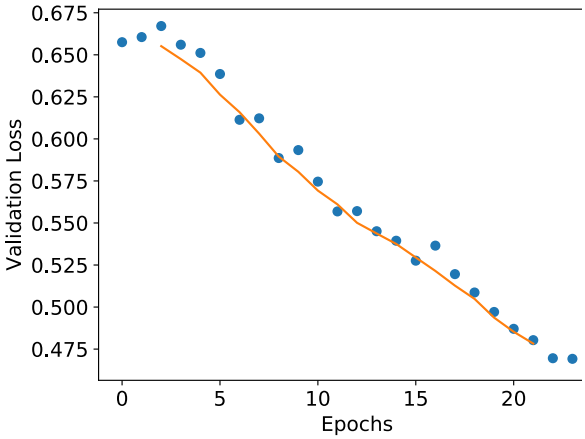
(a) Multiclass network accuracy



(b) Multiclass network loss

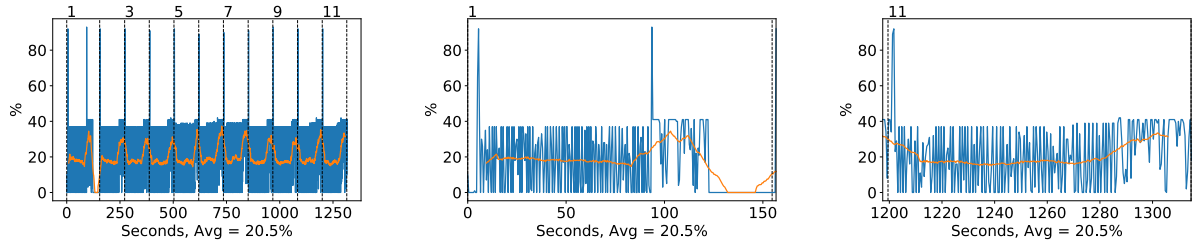


(c) Binary network "polyps" accuracy

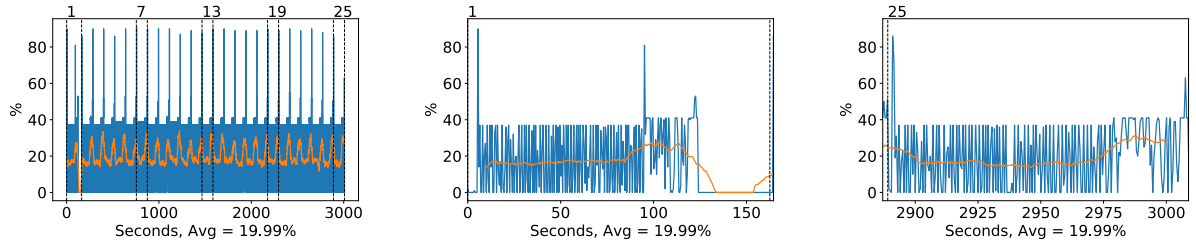


(d) Binary network "polyps" loss

Figure A.130: DenseNet 169 Fine tuning classification accuracy and loss history for validation data

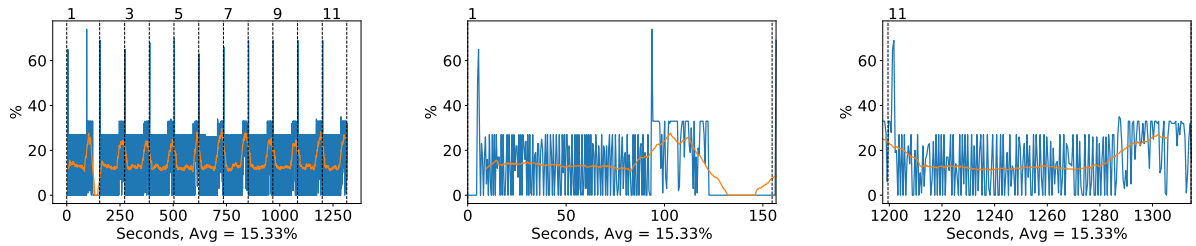


(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

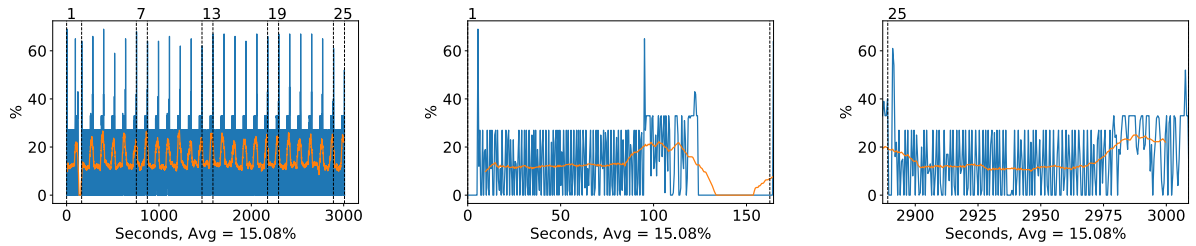


(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

Figure A.131: DenseNet 169 GPU volatile usage during fine tuning, with a moving window average over 40 measurements overlaid

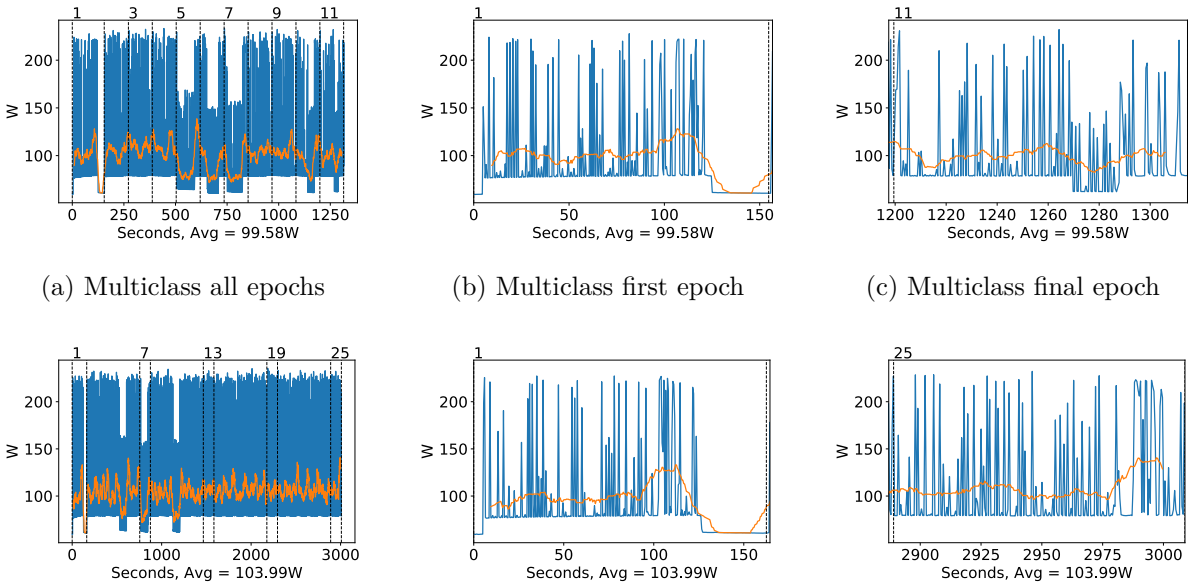


(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch



(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

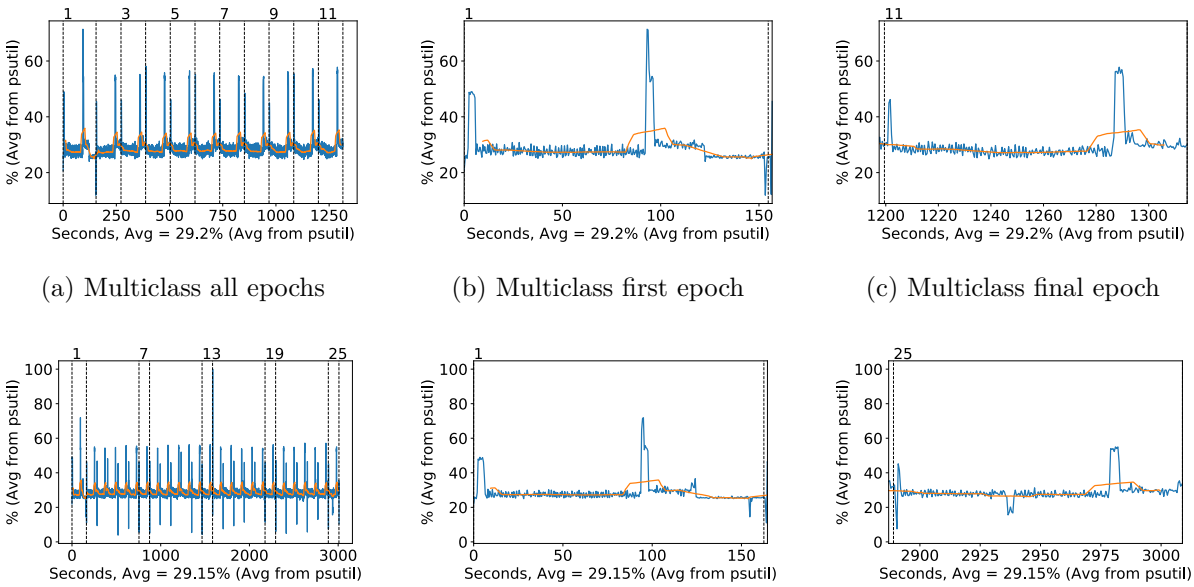
Figure A.132: DenseNet 169 GPU memory usage during fine tuning, with a moving window average over 40 measurements overlaid



(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

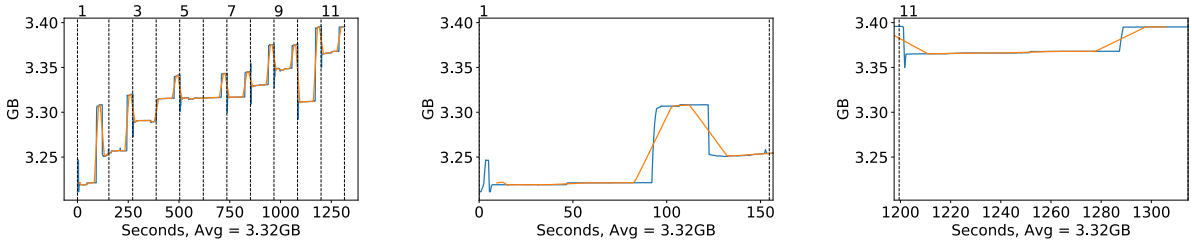
Figure A.133: DenseNet 169 GPU power usage during fine tuning in W, with a moving window average over 40 measurements overlaid



(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

Figure A.134: DenseNet 169 CPU usage (averaged over 4 cores) during fine tuning, with a moving window average over 40 measurements overlaid



(a) Multiclass all epochs

(b) Multiclass first epoch

(c) Multiclass final epoch

(d) Binary all epochs

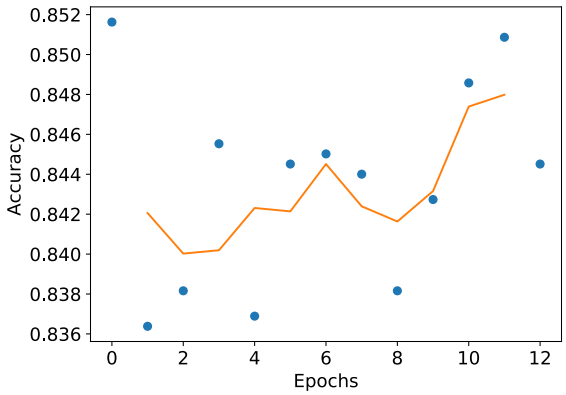
(e) Binary first epoch

(f) Binary final epoch

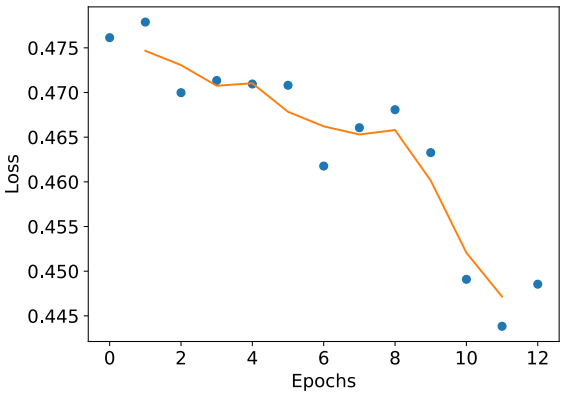
Figure A.135: DenseNet 169 Memory usage during fine tuning, with a moving window average over 40 measurements overlaid

A.3.8 DenseNet 201

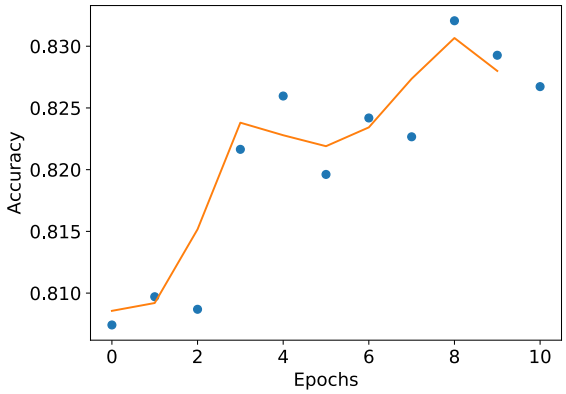
Classification Accuracy and Loss



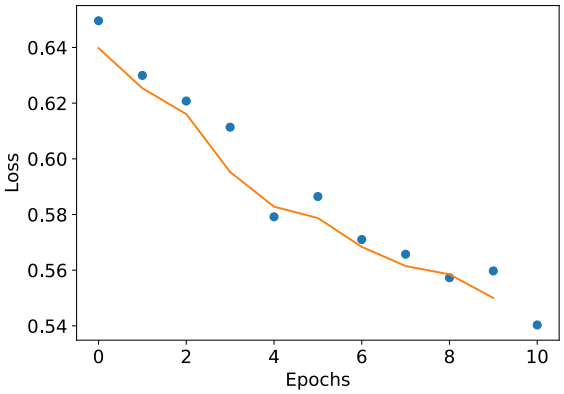
(a) Multiclass network accuracy



(b) Multiclass network loss



(c) Binary network "polyps" accuracy

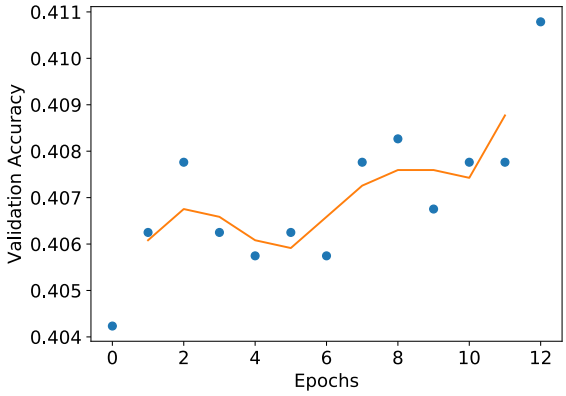


(d) Binary network "polyps" loss

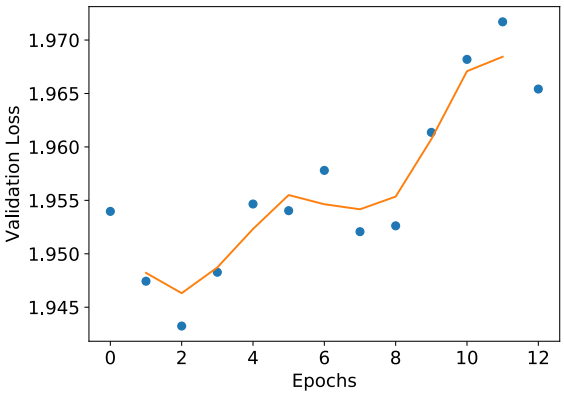
Figure A.136: DenseNet 201 Fine-tuning classification accuracy and loss history for training data

GPU Usage

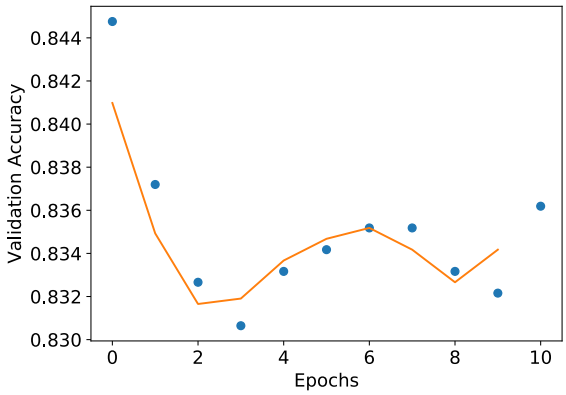
CPU and memory usage



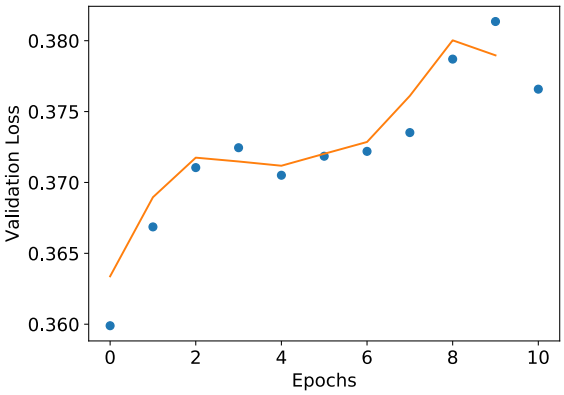
(a) Multiclass network accuracy



(b) Multiclass network loss

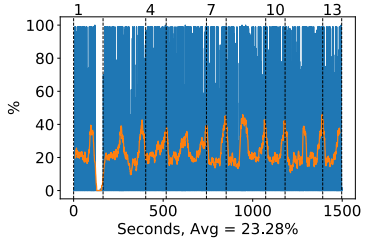


(c) Binary network "polyps" accuracy

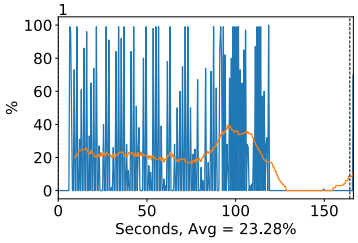


(d) Binary network "polyps" loss

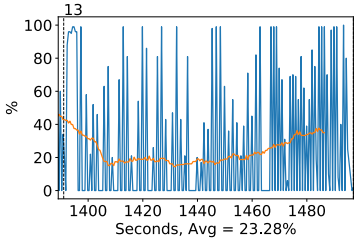
Figure A.137: DenseNet 201 Fine tuning classification accuracy and loss history for validation data



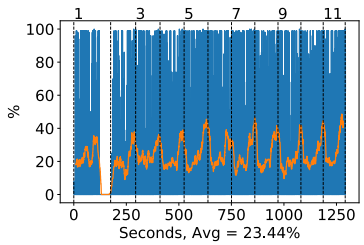
(a) Multiclass all epochs



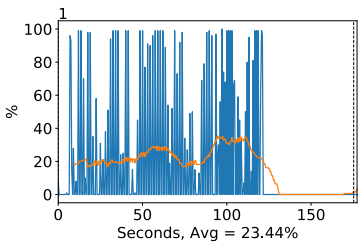
(b) Multiclass first epoch



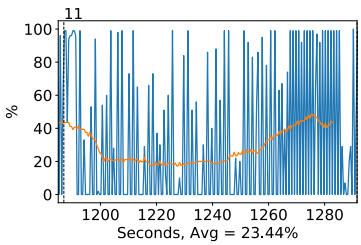
(c) Multiclass final epoch



(d) Binary all epochs

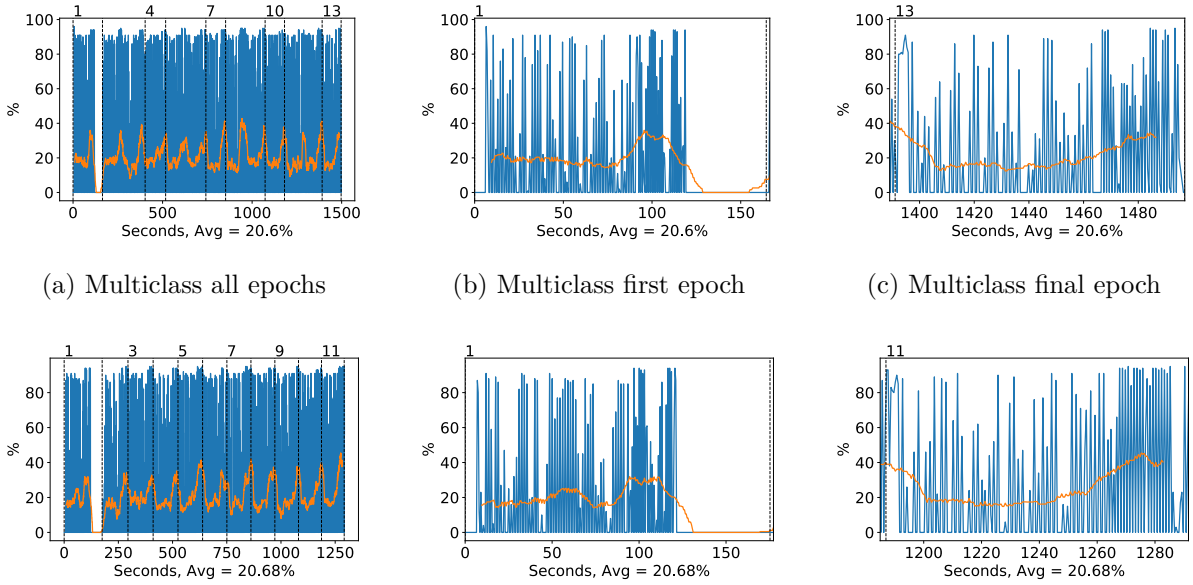


(e) Binary first epoch



(f) Binary final epoch

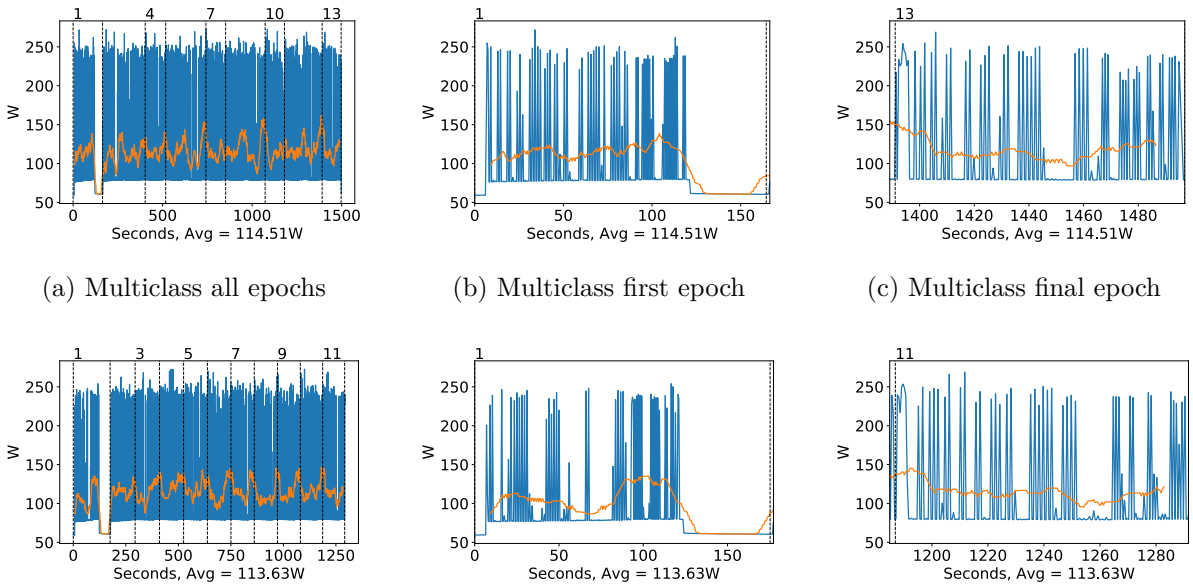
Figure A.138: DenseNet 201 GPU volatile usage during fine tuning, with a moving window average over 40 measurements overlaid



(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

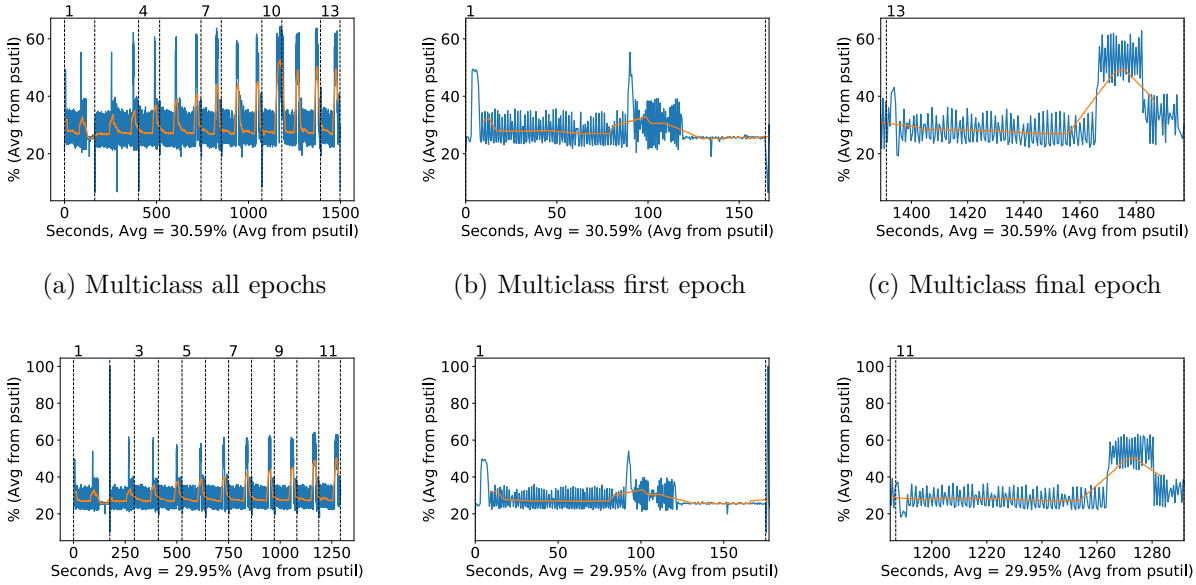
Figure A.139: DenseNet 201 GPU memory usage during fine tuning, with a moving window average over 40 measurements overlaid



(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

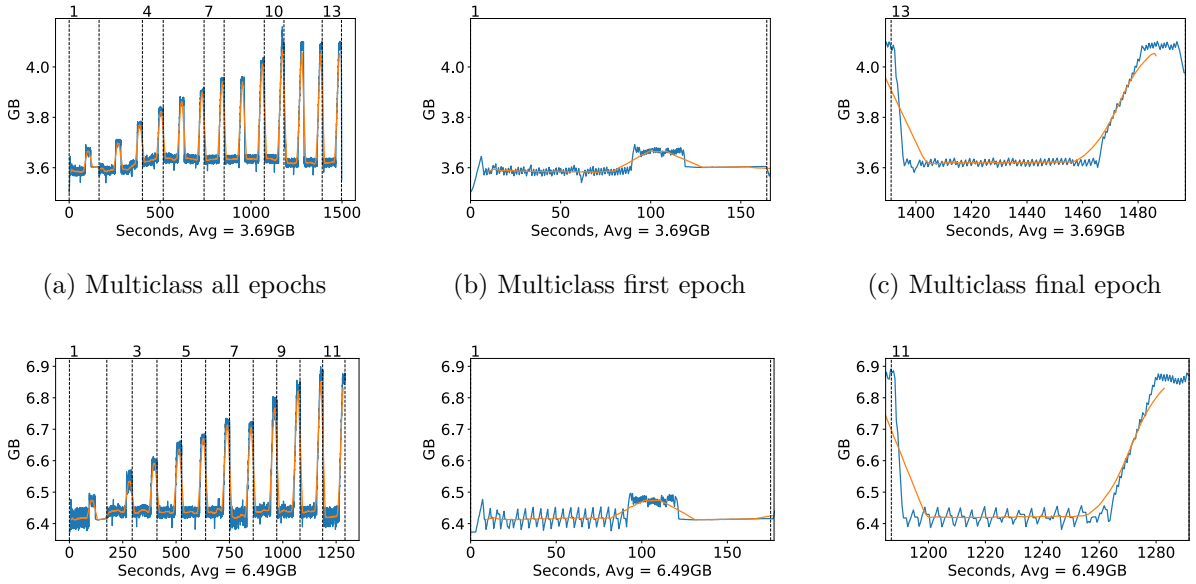
Figure A.140: DenseNet 201 GPU power usage during fine tuning in W, with a moving window average over 40 measurements overlaid



(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

Figure A.141: DenseNet 201 CPU usage (averaged over 4 cores) during fine tuning, with a moving window average over 40 measurements overlaid



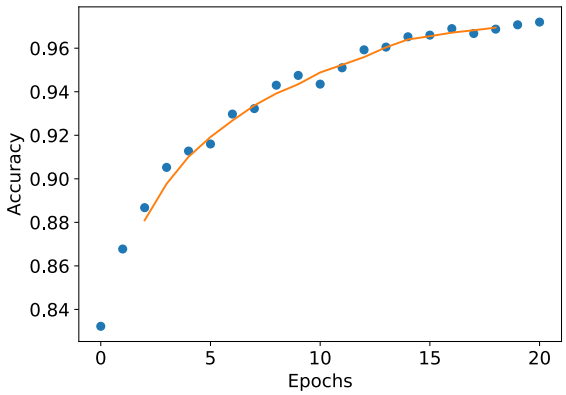
(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

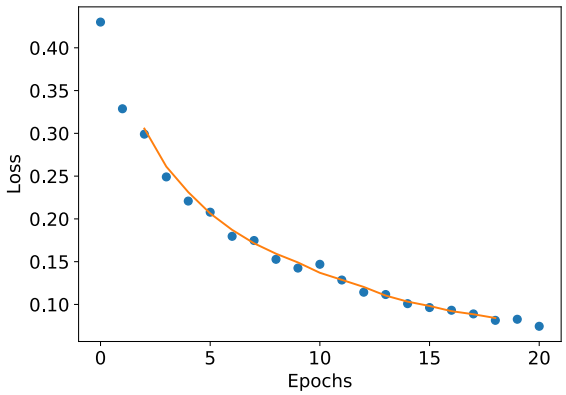
Figure A.142: DenseNet 201 Memory usage during fine tuning, with a moving window average over 40 measurements overlaid

A.3.9 Xception

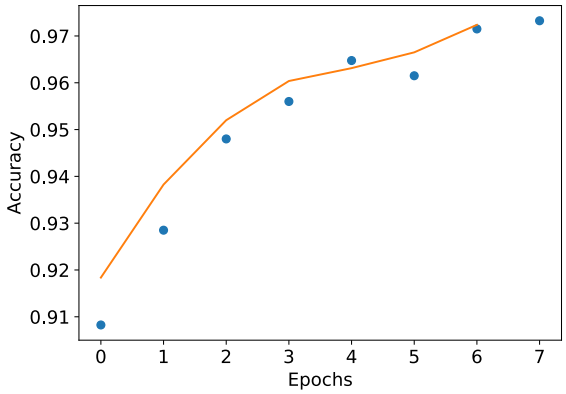
Classification Accuracy and Loss



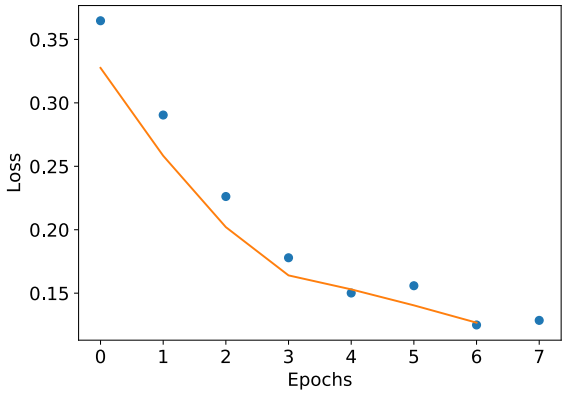
(a) Multiclass network accuracy



(b) Multiclass network loss



(c) Binary network "polyps" accuracy

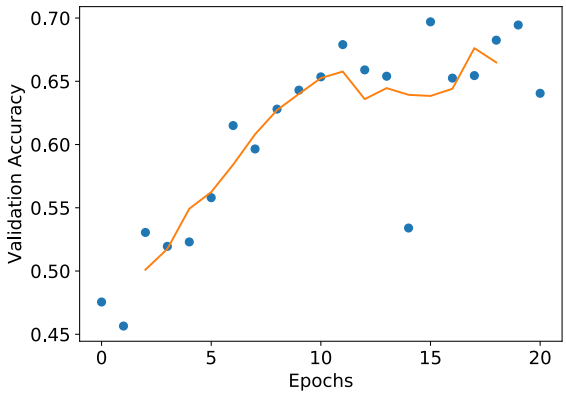


(d) Binary network "polyps" loss

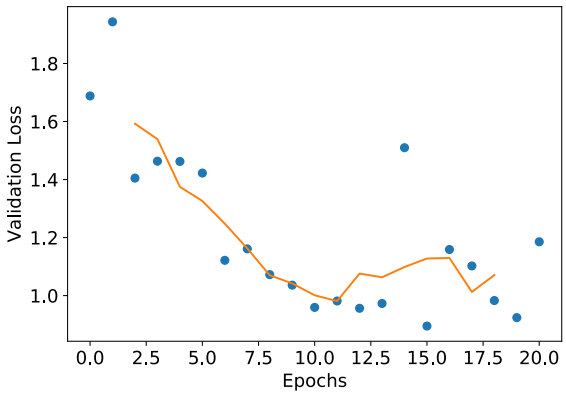
Figure A.143: Xception Fine-tuning classification accuracy and loss history for training data

GPU Usage

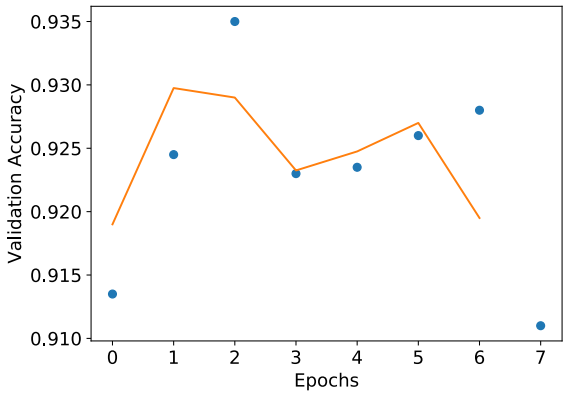
CPU and memory usage



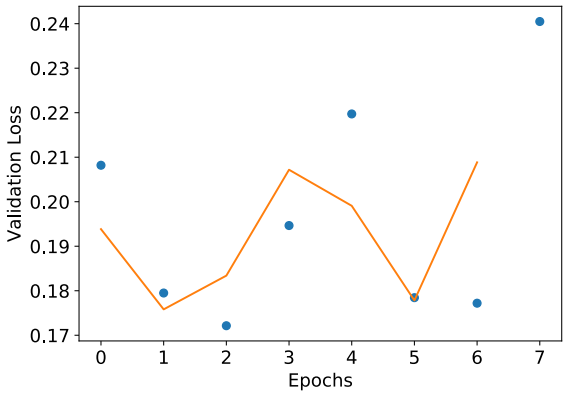
(a) Multiclass network accuracy



(b) Multiclass network loss

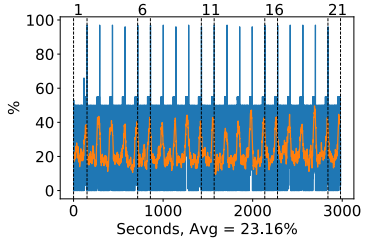


(c) Binary network "polyps" accuracy

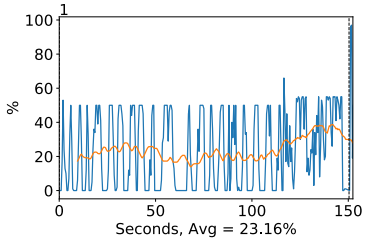


(d) Binary network "polyps" loss

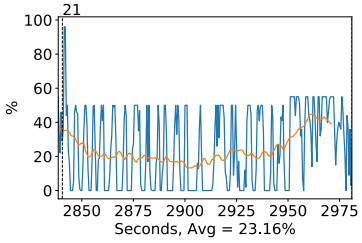
Figure A.144: Xception Fine tuning classification accuracy and loss history for validation data



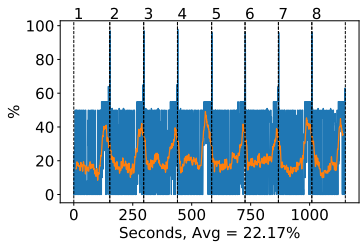
(a) Multiclass all epochs



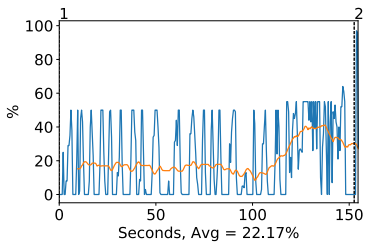
(b) Multiclass first epoch



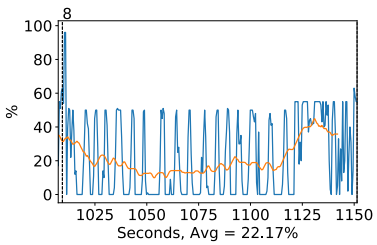
(c) Multiclass final epoch



(d) Binary all epochs

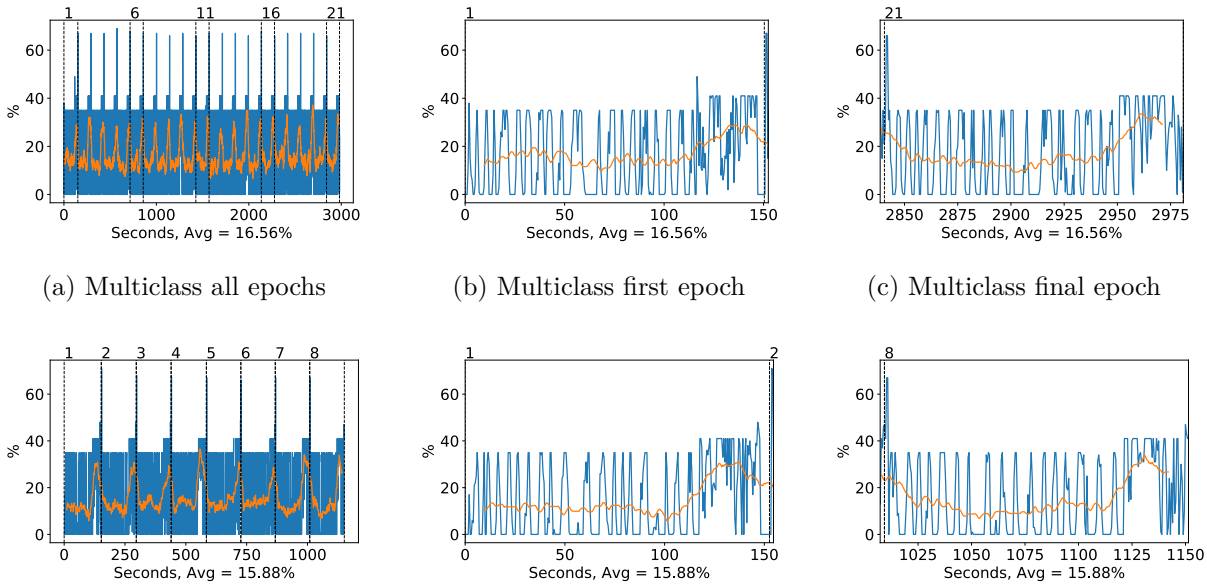


(e) Binary first epoch



(f) Binary final epoch

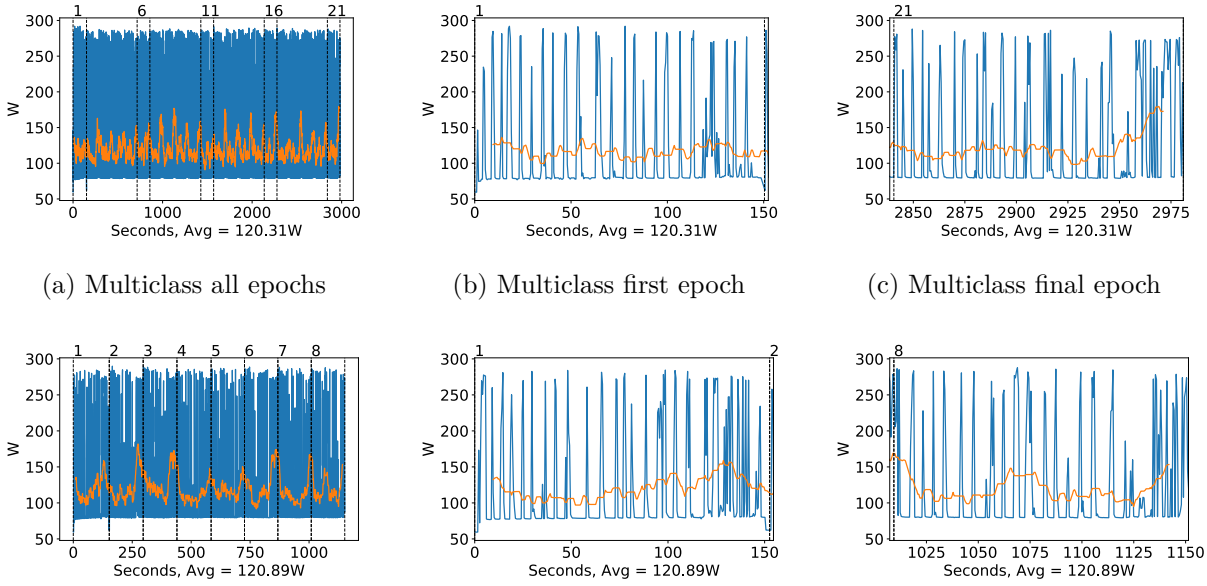
Figure A.145: Xception GPU volatile usage during fine tuning, with a moving window average over 40 measurements overlaid



(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

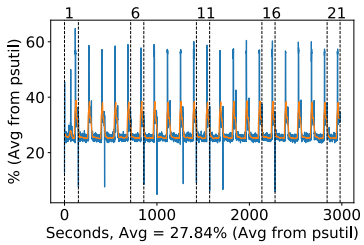
Figure A.146: Xception GPU memory usage during fine tuning, with a moving window average over 40 measurements overlaid



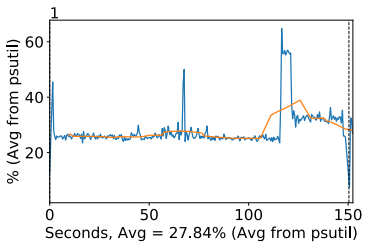
(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

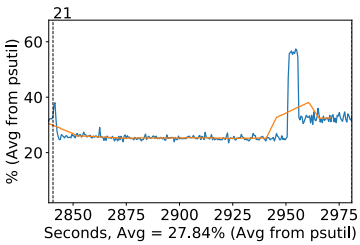
Figure A.147: Xception GPU power usage during fine tuning in W, with a moving window average over 40 measurements overlaid



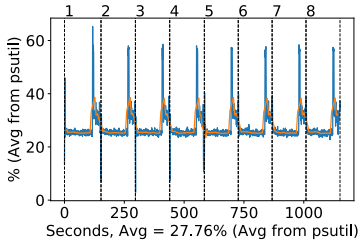
(a) Multiclass all epochs



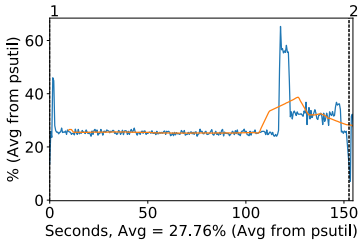
(b) Multiclass first epoch



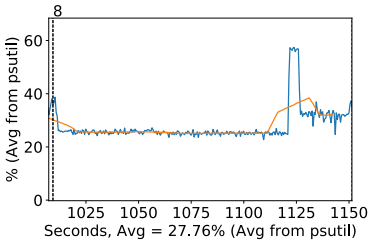
(c) Multiclass final epoch



(d) Binary all epochs

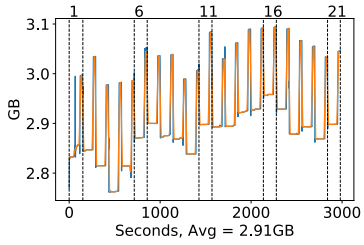


(e) Binary first epoch

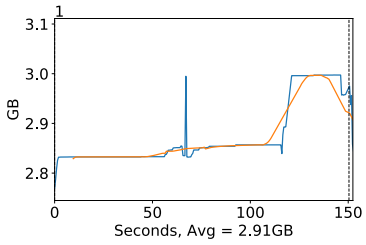


(f) Binary final epoch

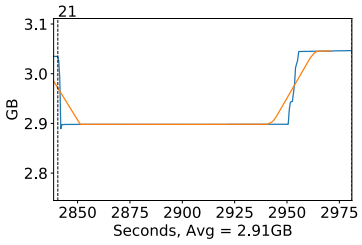
Figure A.148: Xception CPU usage (averaged over 4 cores) during fine tuning, with a moving window average over 40 measurements overlaid



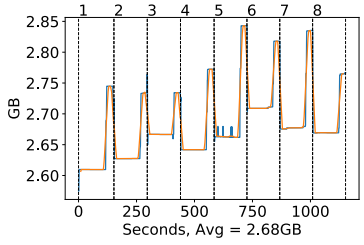
(a) Multiclass all epochs



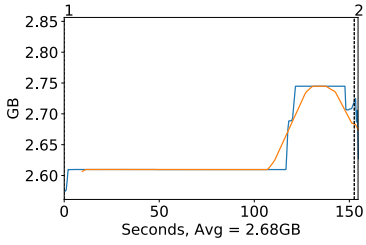
(b) Multiclass first epoch



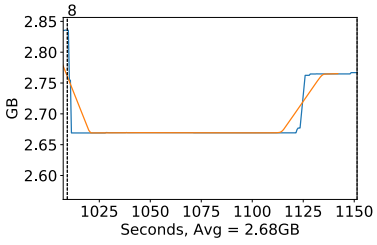
(c) Multiclass final epoch



(d) Binary all epochs



(e) Binary first epoch

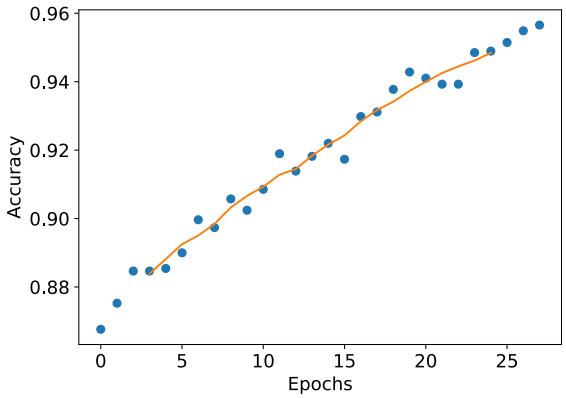


(f) Binary final epoch

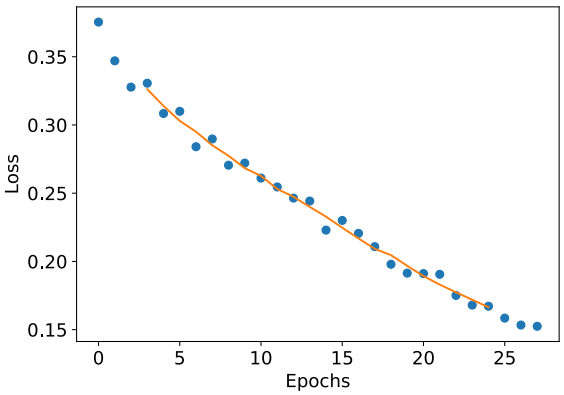
Figure A.149: Xception Memory usage during fine tuning, with a moving window average over 40 measurements overlaid

A.3.10 Inception ResNet v2

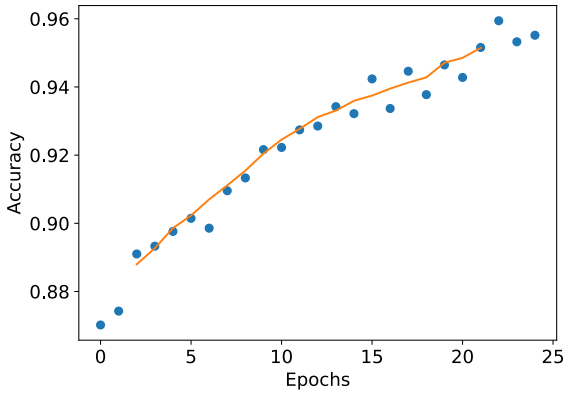
Classification Accuracy and Loss



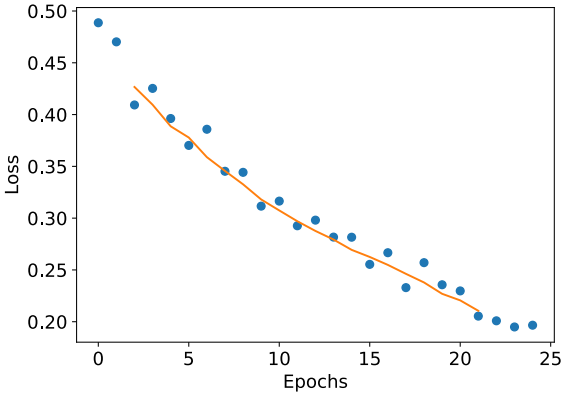
(a) Multiclass network accuracy



(b) Multiclass network loss



(c) Binary network "polyps" accuracy

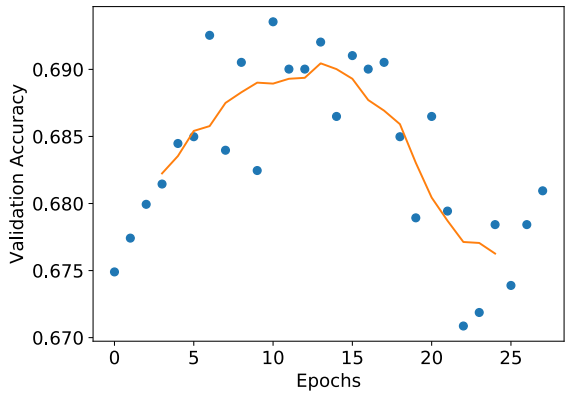


(d) Binary network "polyps" loss

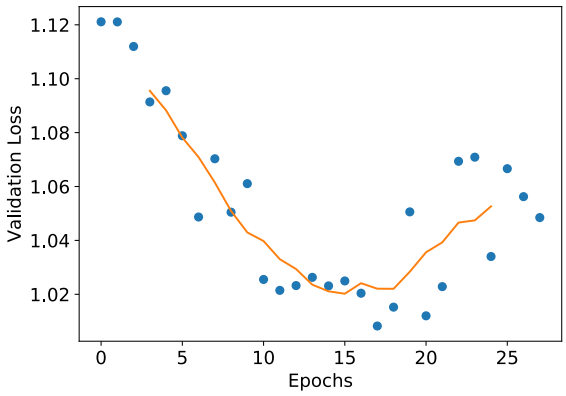
Figure A.150: Inception-ResNet-v2 Fine-tuning classification accuracy and loss history for training data

GPU Usage

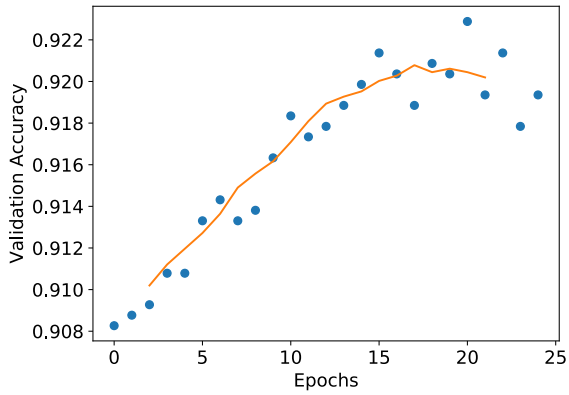
CPU and memory usage



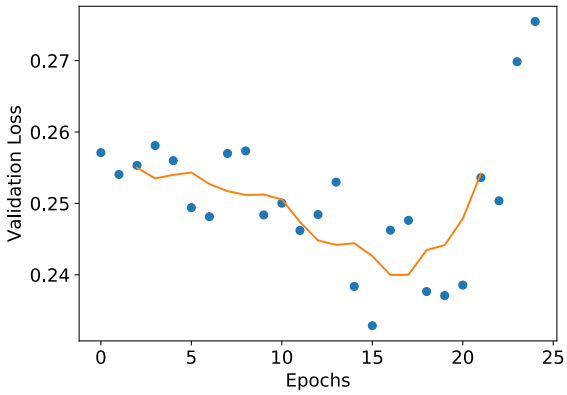
(a) Multiclass network accuracy



(b) Multiclass network loss

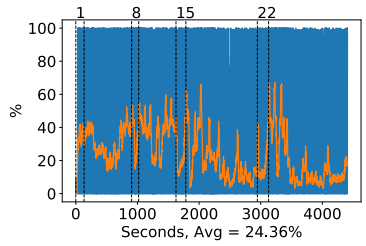


(c) Binary network "polyps" accuracy

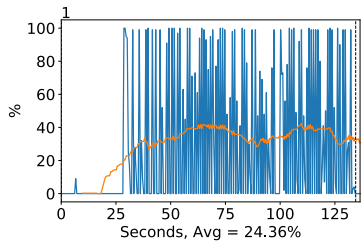


(d) Binary network "polyps" loss

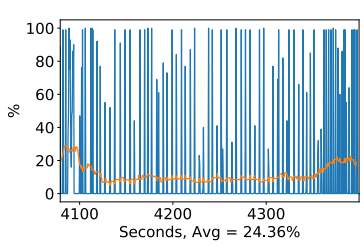
Figure A.151: Inception-ResNet-v2 Fine tuning classification accuracy and loss history for validation data



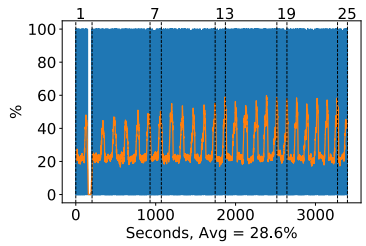
(a) Multiclass all epochs



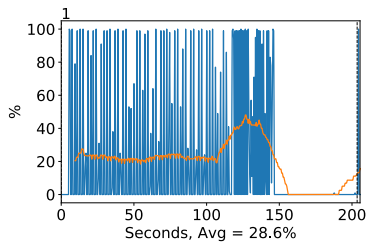
(b) Multiclass first epoch



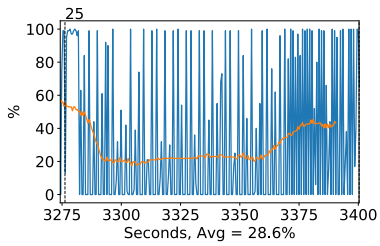
(c) Multiclass final epoch



(d) Binary all epochs

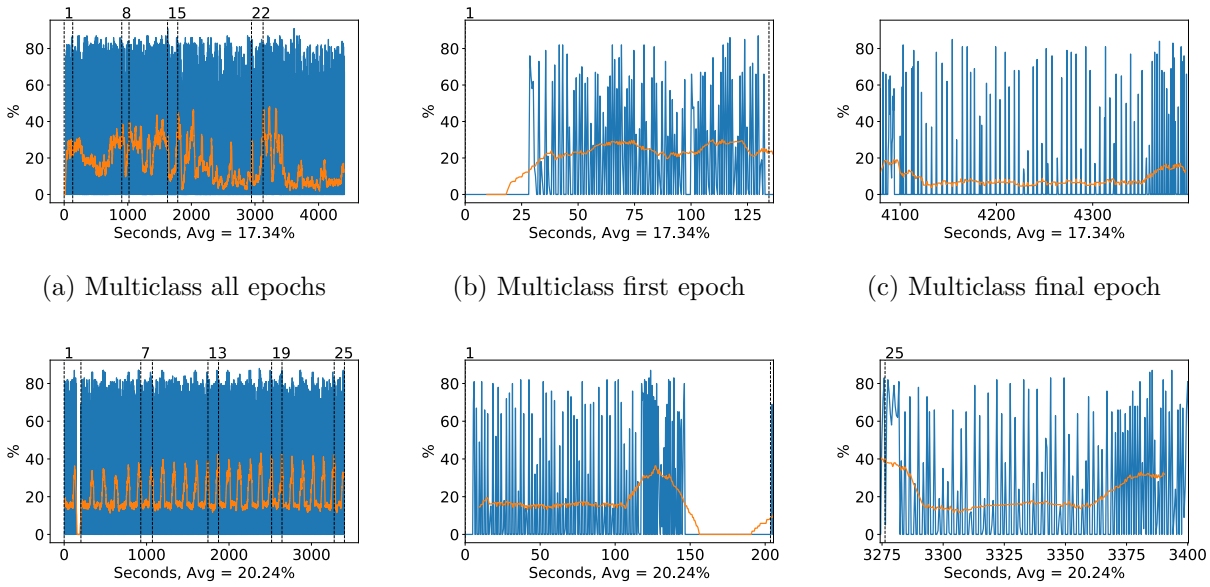


(e) Binary first epoch



(f) Binary final epoch

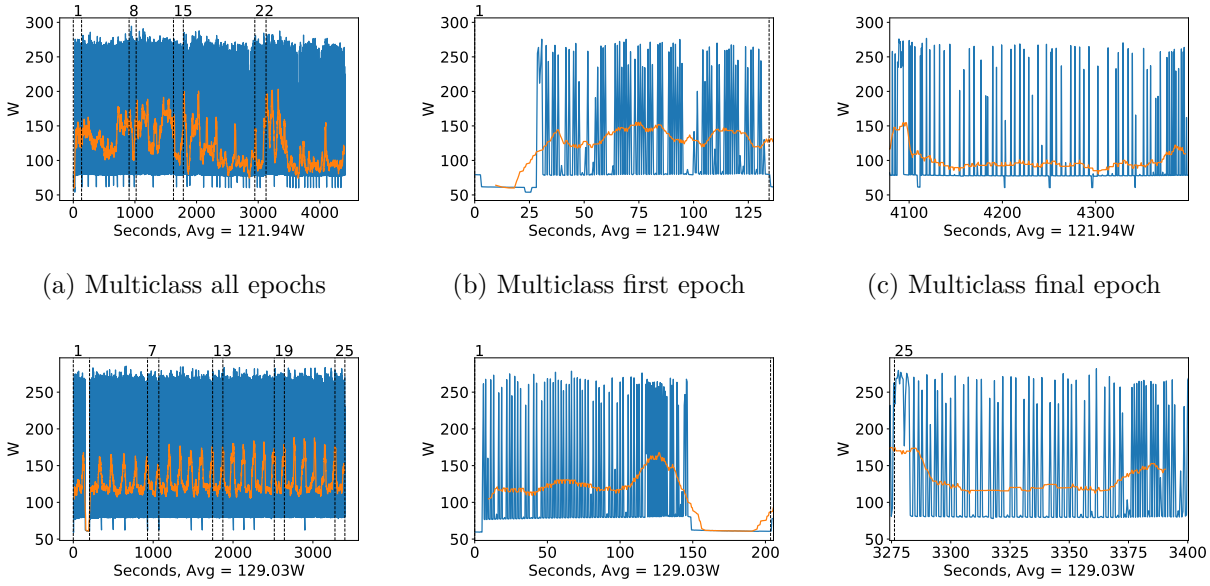
Figure A.152: Inception-ResNet-v2 GPU volatile usage during fine tuning, with a moving window average over 40 measurements overlaid



(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

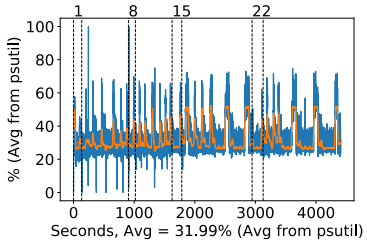
Figure A.153: Inception-ResNet-v2 GPU memory usage during fine tuning, with a moving window average over 40 measurements overlaid



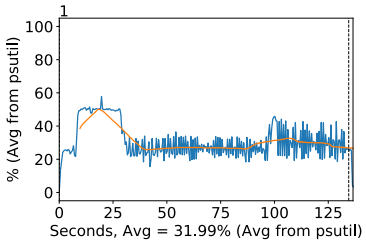
(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

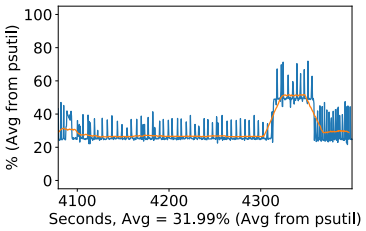
Figure A.154: Inception-ResNet-v2 GPU power usage during fine tuning in W, with a moving window average over 40 measurements overlaid



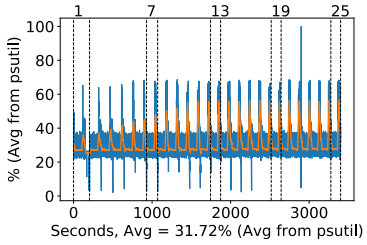
(a) Multiclass all epochs



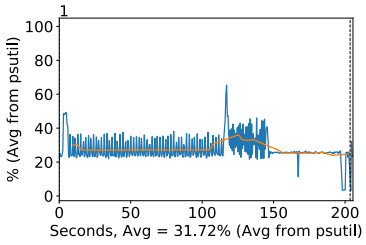
(b) Multiclass first epoch



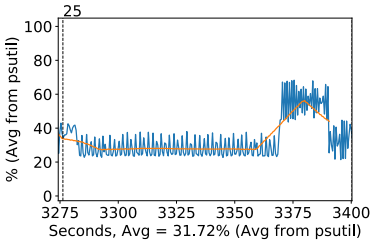
(c) Multiclass final epoch



(d) Binary all epochs

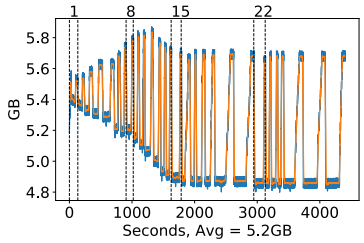


(e) Binary first epoch

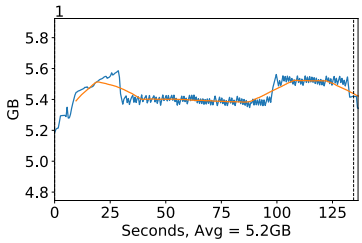


(f) Binary final epoch

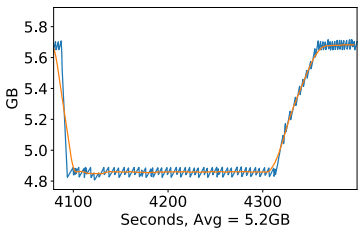
Figure A.155: Inception-ResNet-v2 CPU usage (averaged over 4 cores) during fine tuning, with a moving window average over 40 measurements overlaid



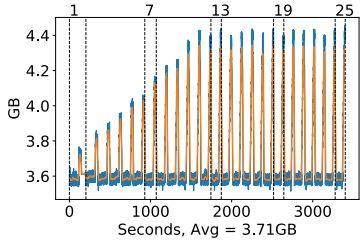
(a) Multiclass all epochs



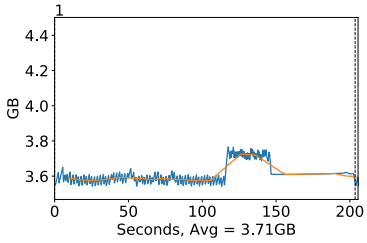
(b) Multiclass first epoch



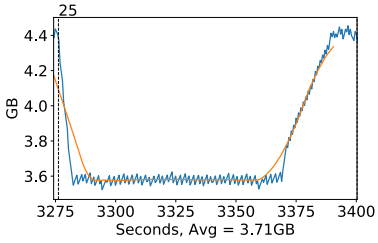
(c) Multiclass final epoch



(d) Binary all epochs



(e) Binary first epoch

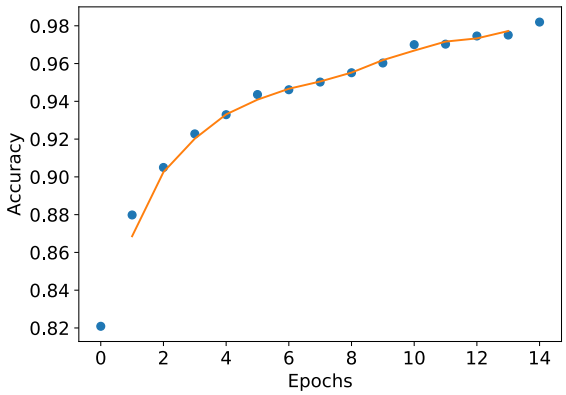


(f) Binary final epoch

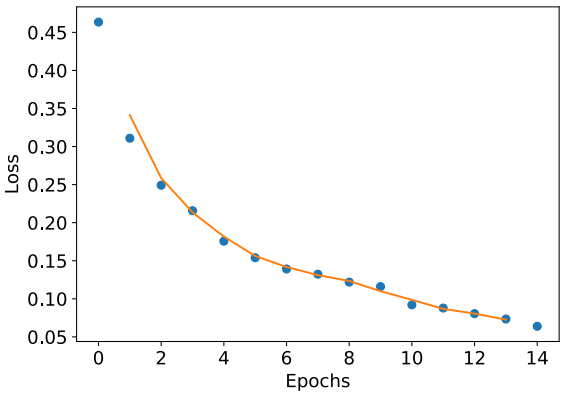
Figure A.156: Inception-ResNet-v2 Memory usage during fine tuning, with a moving window average over 40 measurements overlaid

A.3.11 Mobilenet

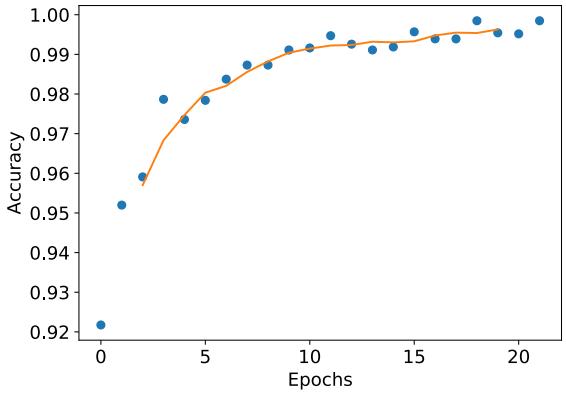
Classification Accuracy and Loss



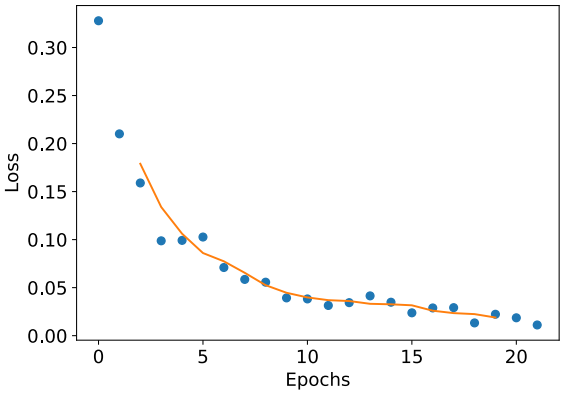
(a) Multiclass network accuracy



(b) Multiclass network loss



(c) Binary network "polyps" accuracy

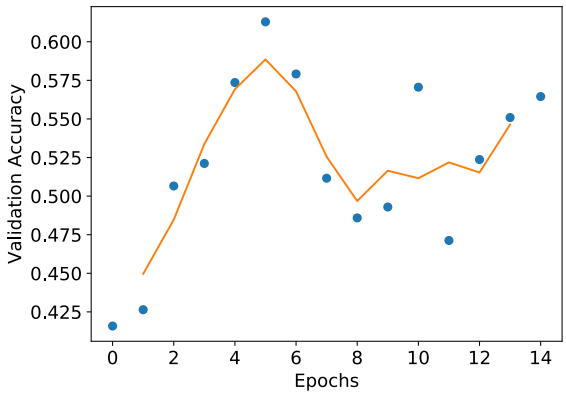


(d) Binary network "polyps" loss

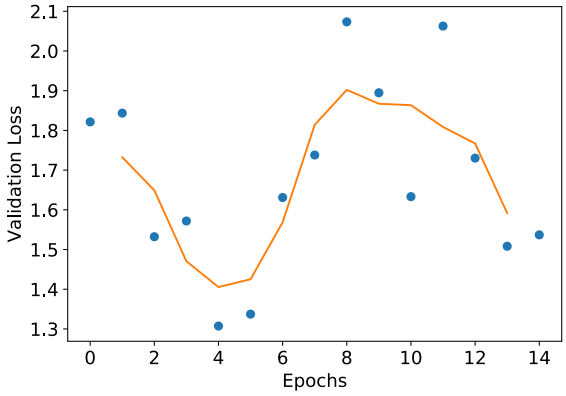
Figure A.157: Mobilenet Fine-tuning classification accuracy and loss history for training data

GPU Usage

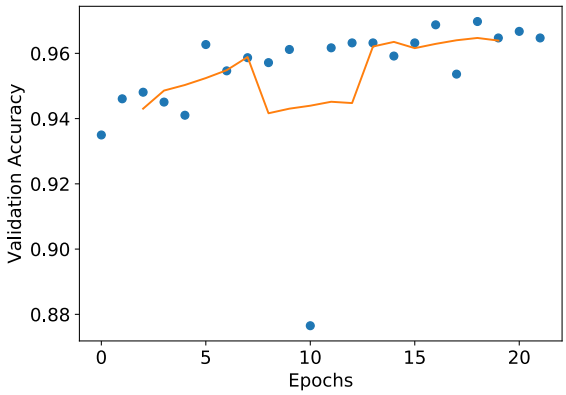
CPU and memory usage



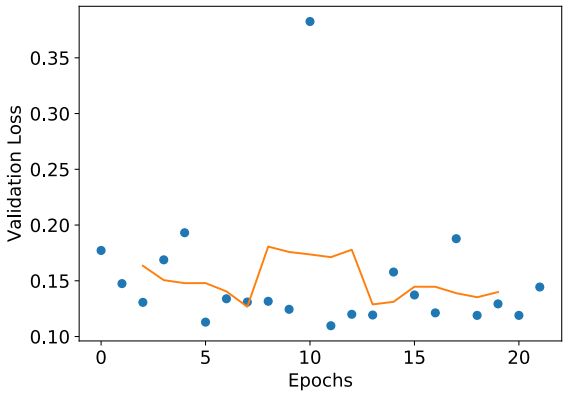
(a) Multiclass network accuracy



(b) Multiclass network loss

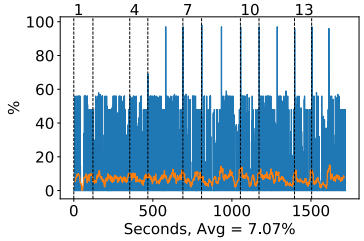


(c) Binary network "polyps" accuracy

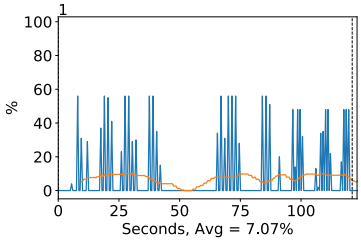


(d) Binary network "polyps" loss

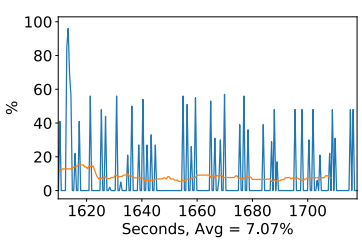
Figure A.158: Mobilenet Fine tuning classification accuracy and loss history for validation data



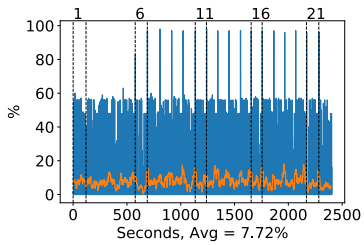
(a) Multiclass all epochs



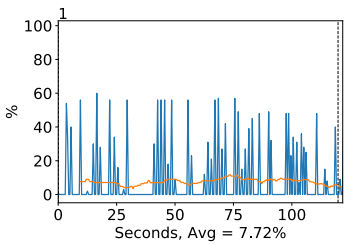
(b) Multiclass first epoch



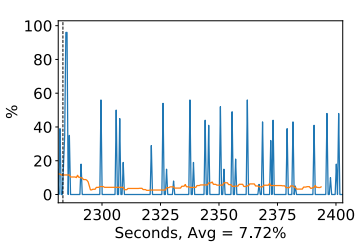
(c) Multiclass final epoch



(d) Binary all epochs

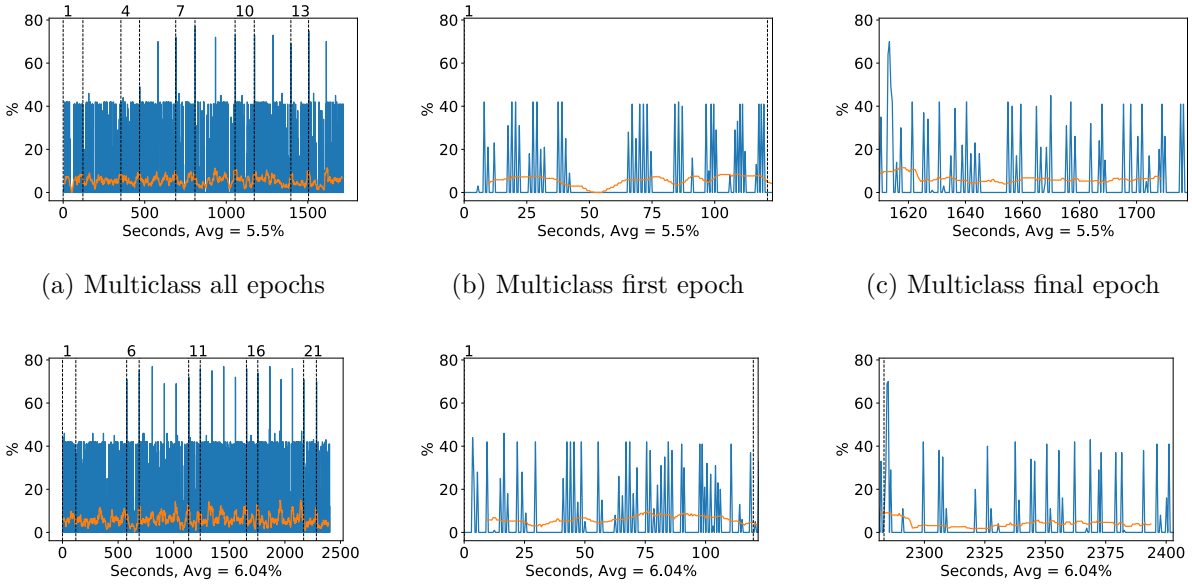


(e) Binary first epoch



(f) Binary final epoch

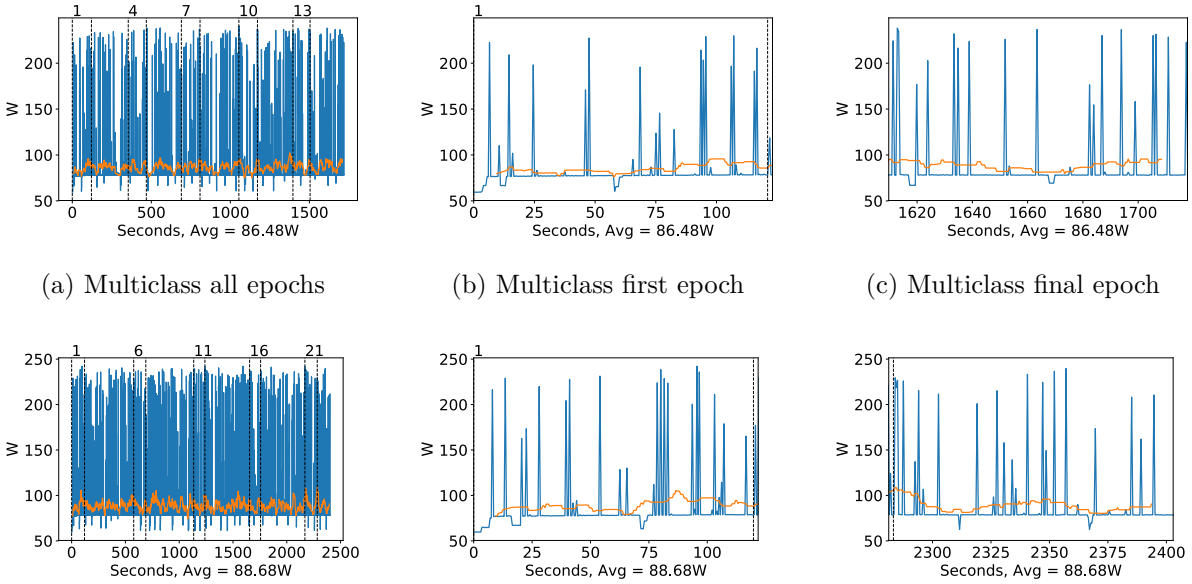
Figure A.159: Mobilenet GPU volatile usage during fine tuning, with a moving window average over 40 measurements overlaid



(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

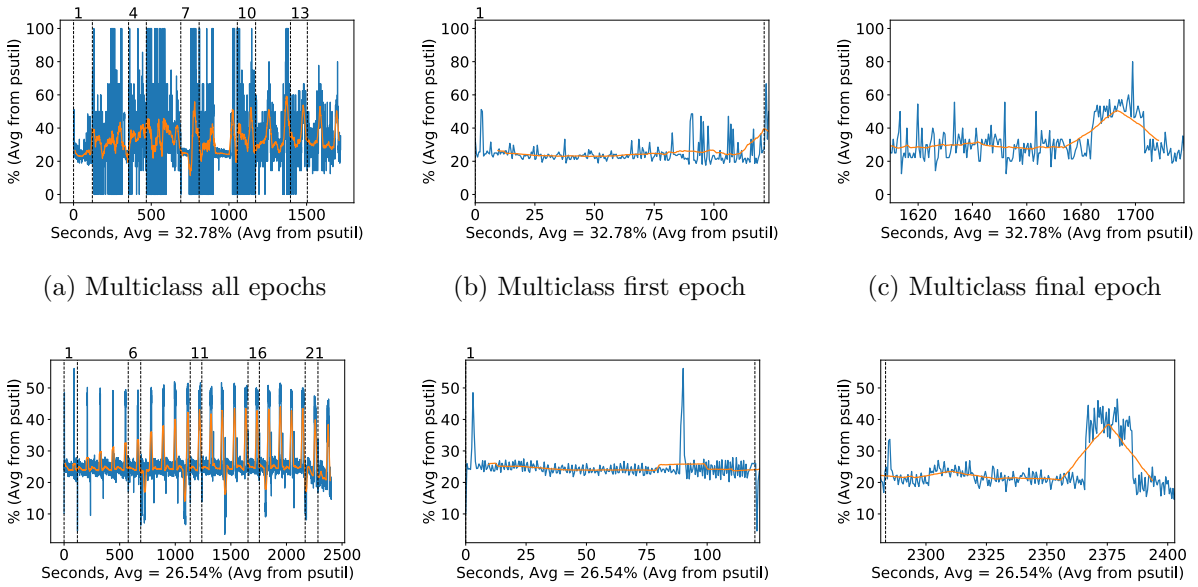
Figure A.160: Mobilenet GPU memory usage during fine tuning, with a moving window average over 40 measurements overlaid



(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

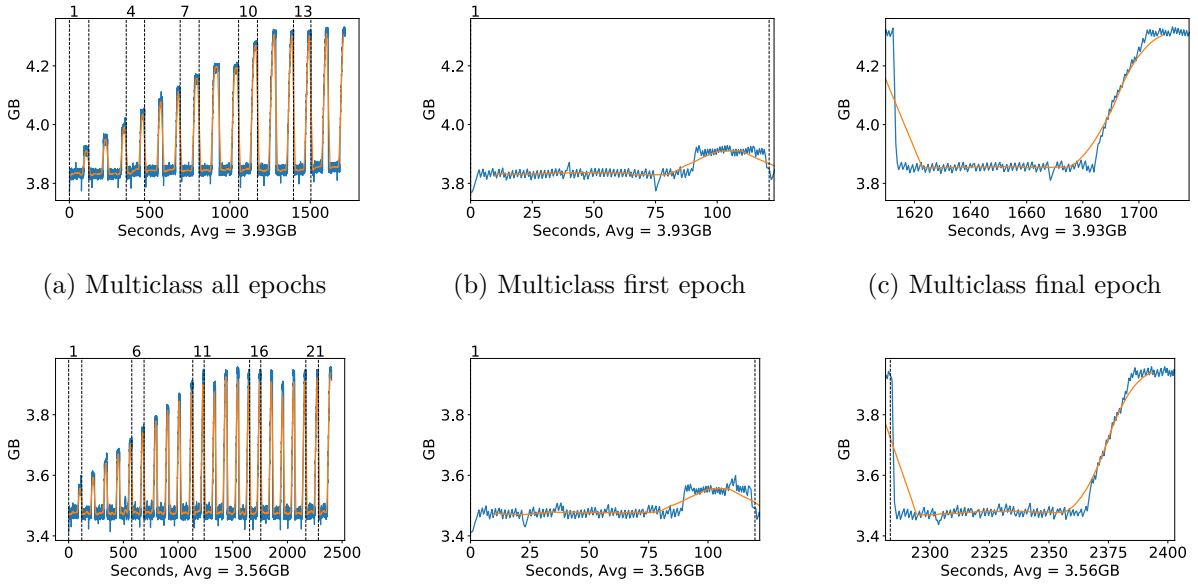
Figure A.161: Mobilenet GPU power usage during fine tuning in W, with a moving window average over 40 measurements overlaid



(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

Figure A.162: Mobilenet CPU usage (averaged over 4 cores) during fine tuning, with a moving window average over 40 measurements overlaid



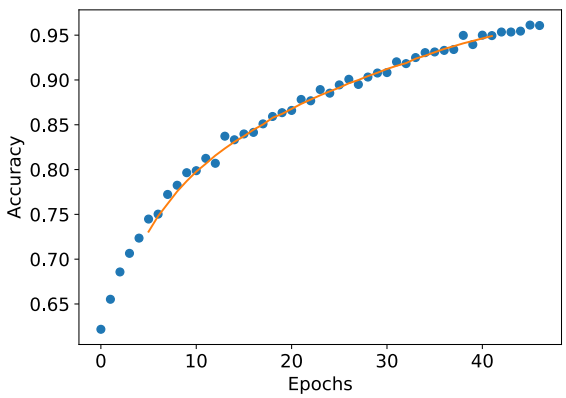
(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

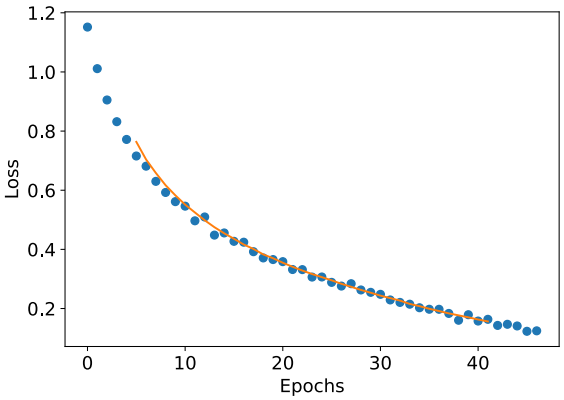
Figure A.163: Mobilenet Memory usage during fine tuning, with a moving window average over 40 measurements overlaid

A.3.12 NasNet Large

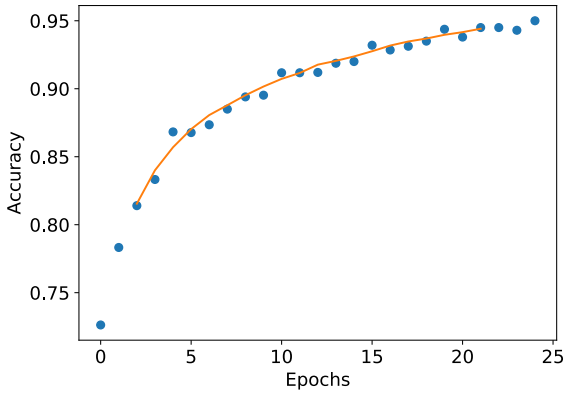
Classification Accuracy and Loss



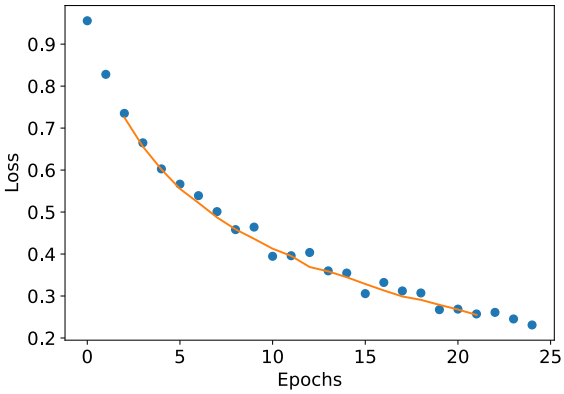
(a) Multiclass network accuracy



(b) Multiclass network loss



(c) Binary network "polyps" accuracy

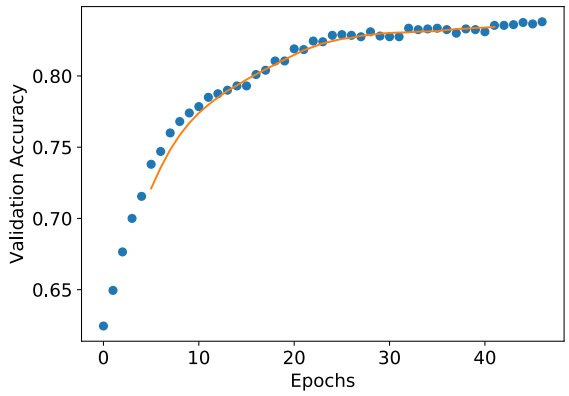


(d) Binary network "polyps" loss

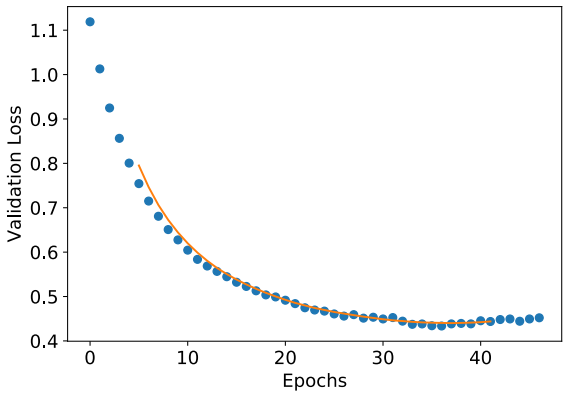
Figure A.164: NASNet Large Fine-tuning classification accuracy and loss history for training data

GPU Usage

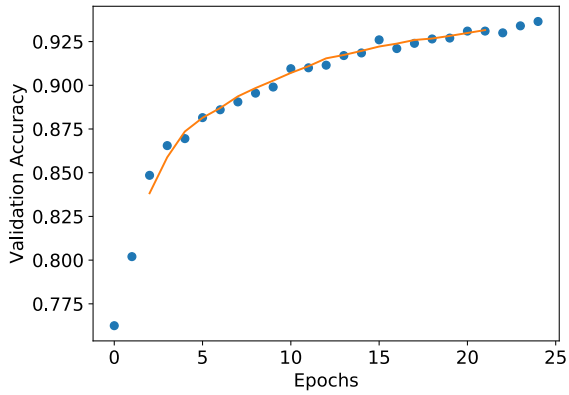
CPU and memory usage



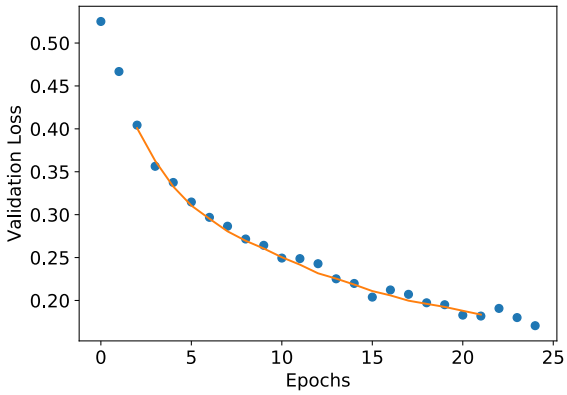
(a) Multiclass network accuracy



(b) Multiclass network loss

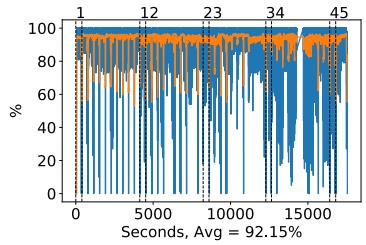


(c) Binary network "polyps" accuracy

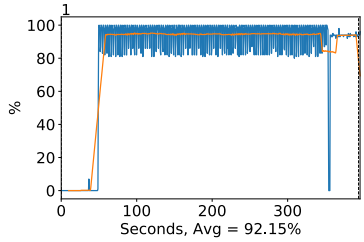


(d) Binary network "polyps" loss

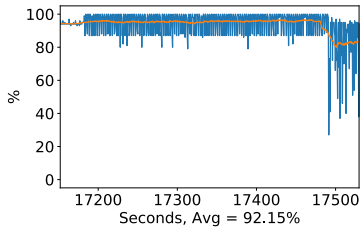
Figure A.165: NASNet Large Fine tuning classification accuracy and loss history for validation data



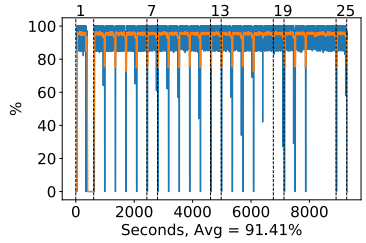
(a) Multiclass all epochs



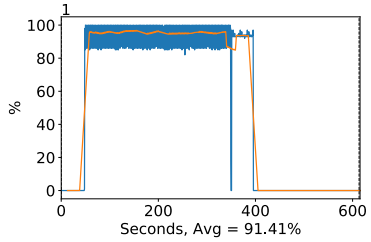
(b) Multiclass first epoch



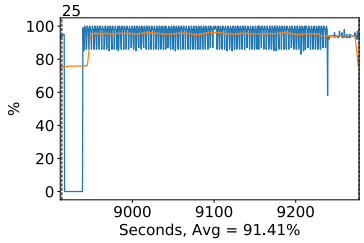
(c) Multiclass final epoch



(d) Binary all epochs

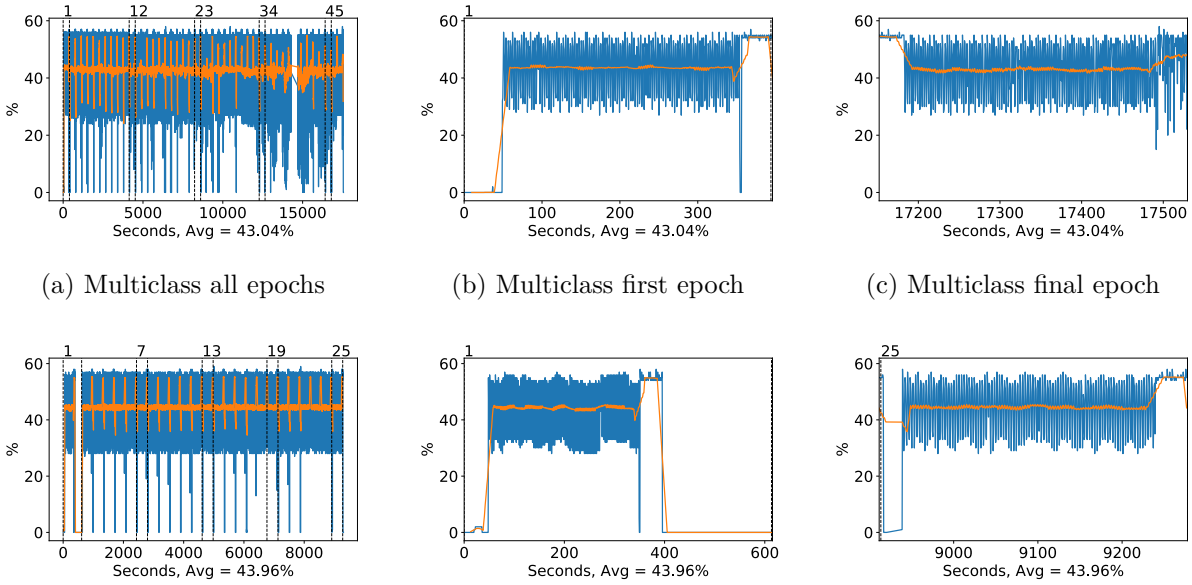


(e) Binary first epoch



(f) Binary final epoch

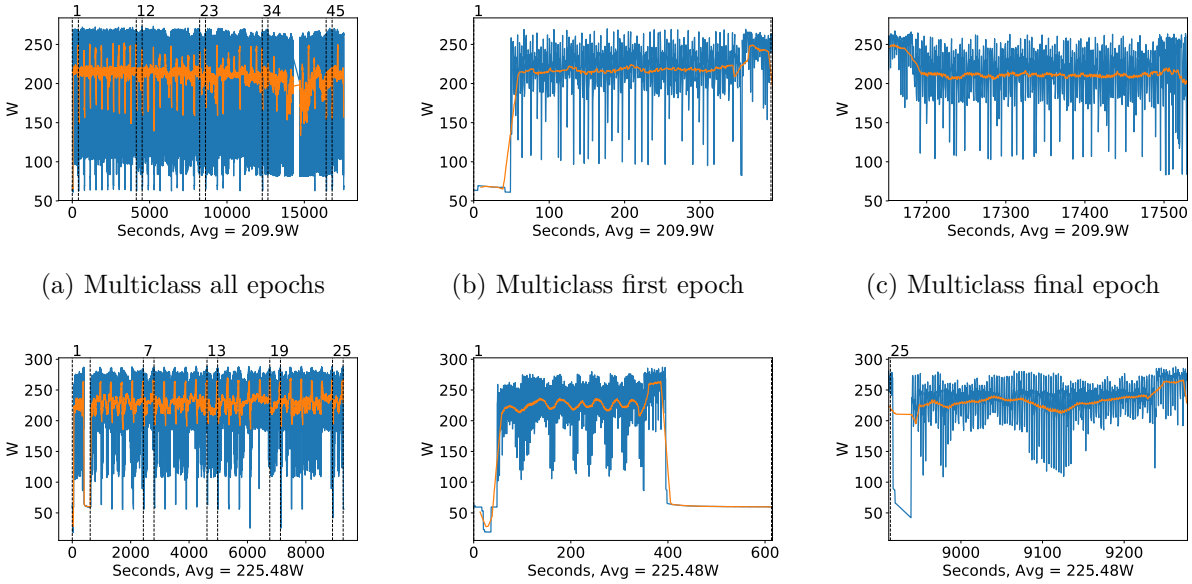
Figure A.166: NASNet Large GPU volatile usage during fine tuning, with a moving window average over 40 measurements overlaid



(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

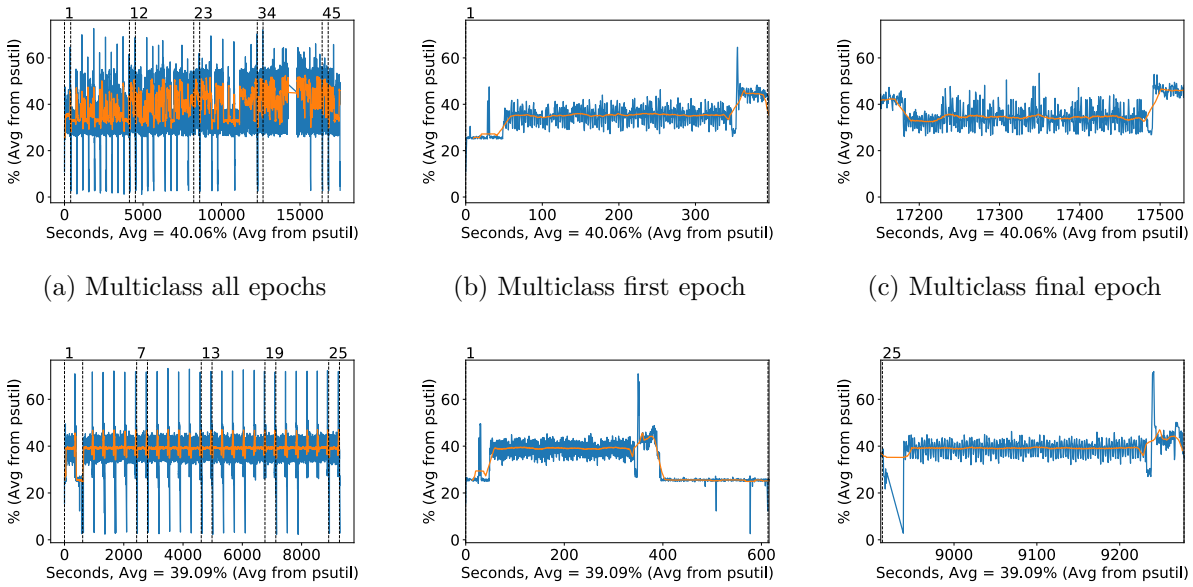
Figure A.167: NASNet Large GPU memory usage during fine tuning, with a moving window average over 40 measurements overlaid



(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

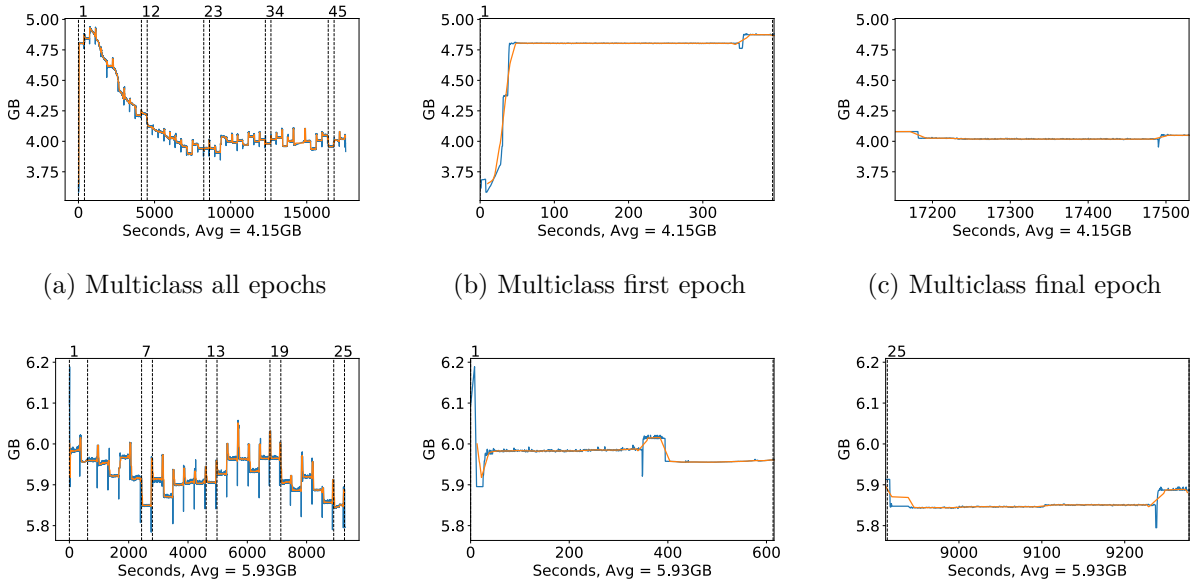
Figure A.168: NASNet Large GPU power usage during fine tuning in W, with a moving window average over 40 measurements overlaid



(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

Figure A.169: NASNet Large CPU usage (averaged over 4 cores) during fine tuning, with a moving window average over 40 measurements overlaid



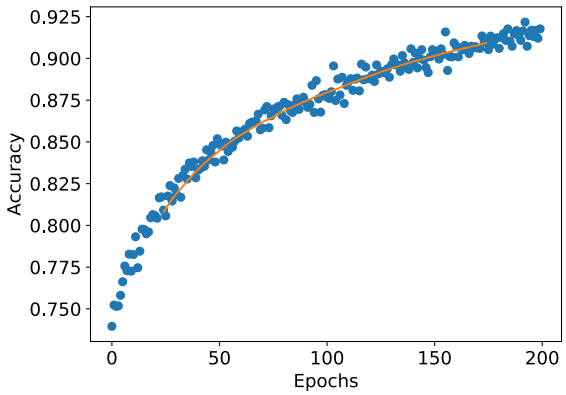
(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

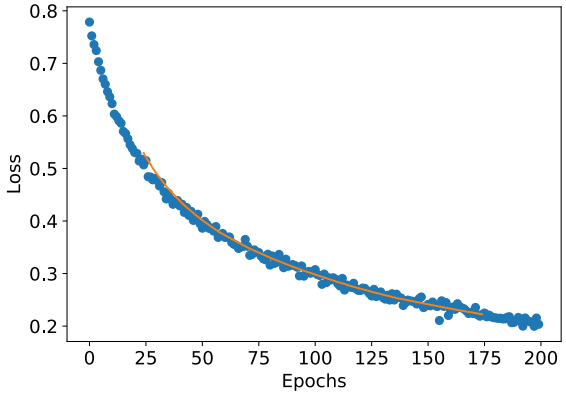
Figure A.170: NASNet Large Memory usage during fine tuning, with a moving window average over 40 measurements overlaid

A.3.13 NasNet Mobile

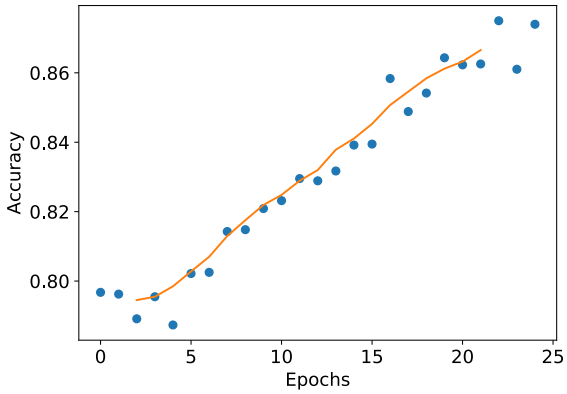
Classification Accuracy and Loss



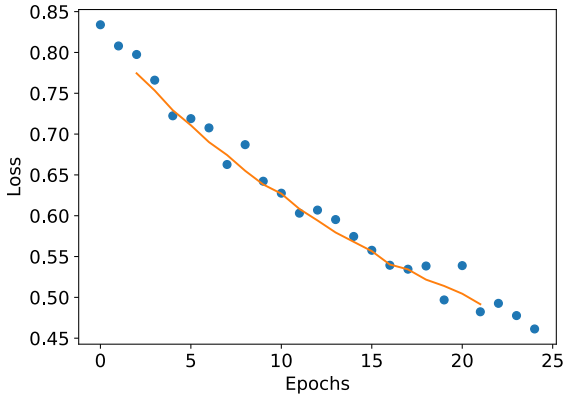
(a) Multiclass network accuracy



(b) Multiclass network loss



(c) Binary network "polyps" accuracy

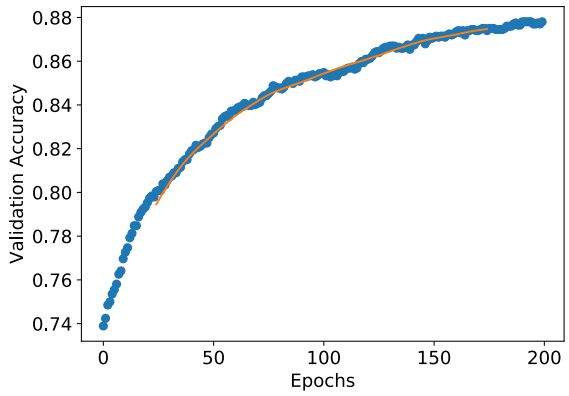


(d) Binary network "polyps" loss

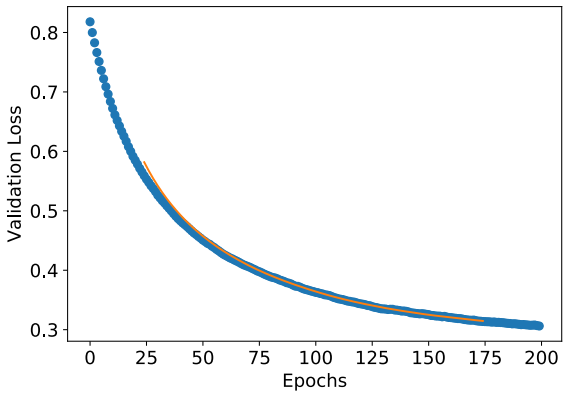
Figure A.171: NASNet Mobile Fine-tuning classification accuracy and loss history for training data

GPU Usage

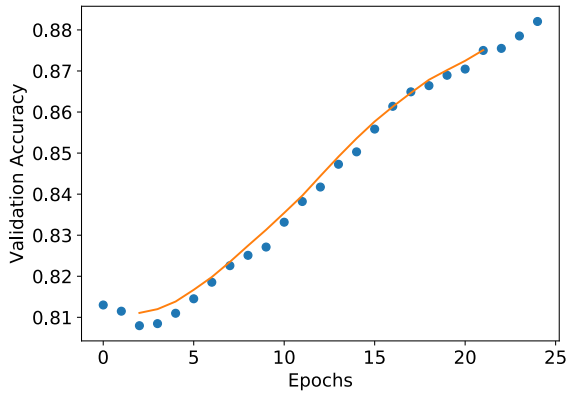
CPU and memory usage



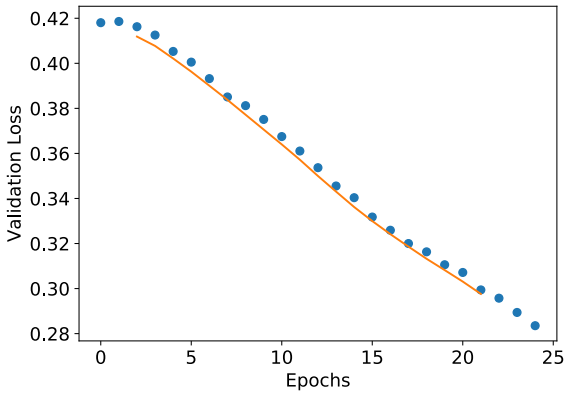
(a) Multiclass network accuracy



(b) Multiclass network loss

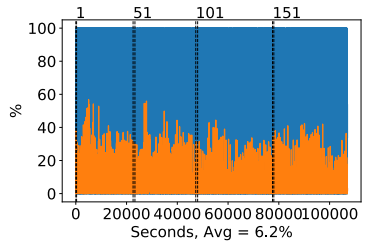


(c) Binary network "polyps" accuracy

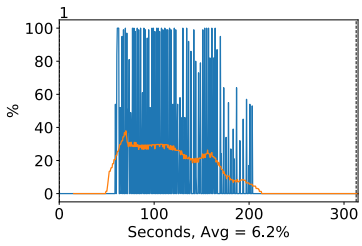


(d) Binary network "polyps" loss

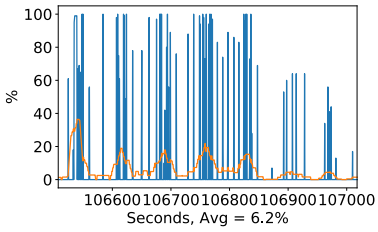
Figure A.172: NASNet Mobile Fine tuning classification accuracy and loss history for validation data



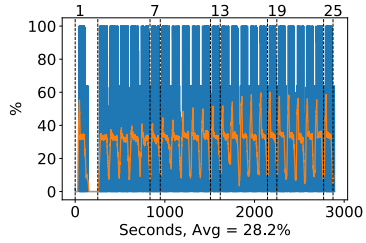
(a) Multiclass all epochs



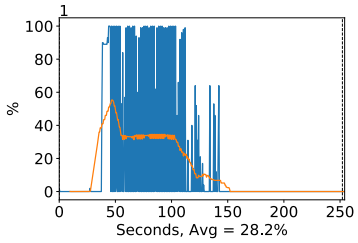
(b) Multiclass first epoch



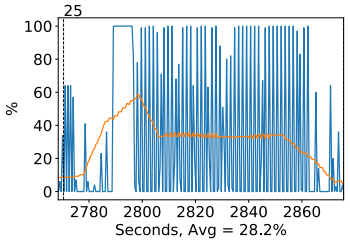
(c) Multiclass final epoch



(d) Binary all epochs

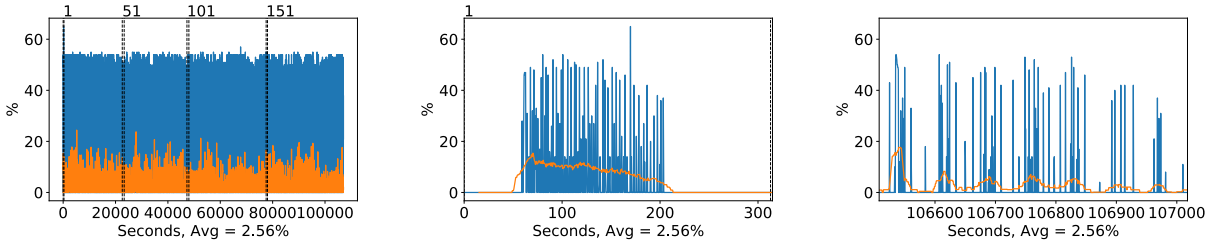


(e) Binary first epoch

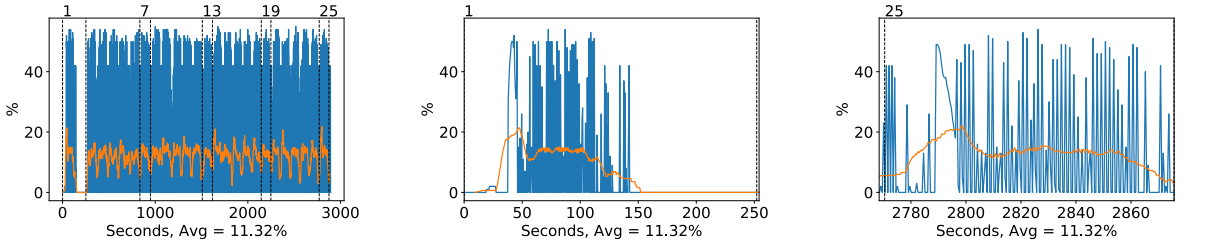


(f) Binary final epoch

Figure A.173: NASNet Mobile GPU volatile usage during fine tuning, with a moving window average over 40 measurements overlaid

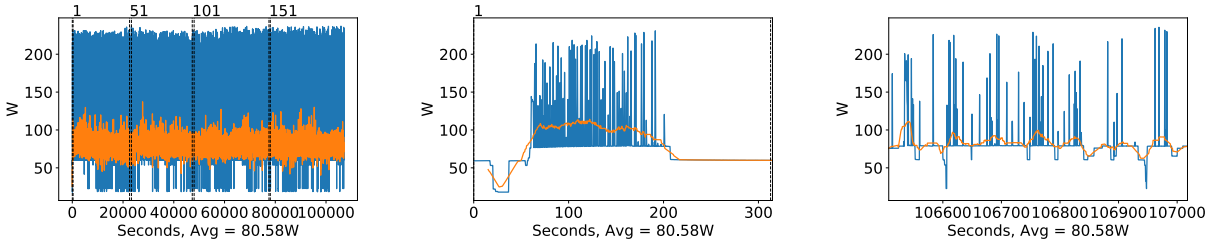


(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

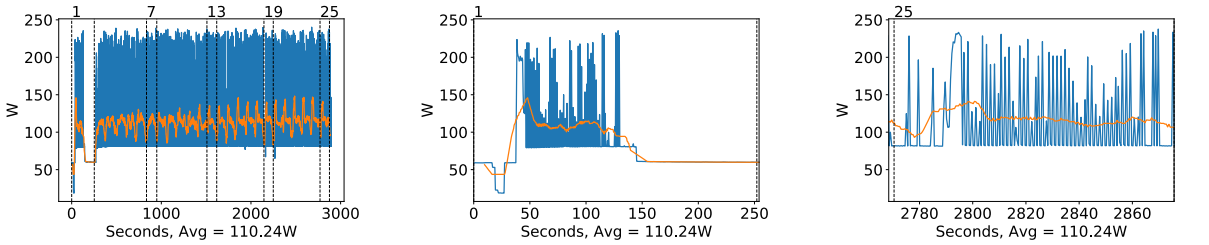


(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

Figure A.174: NASNet Mobile GPU memory usage during fine tuning, with a moving window average over 40 measurements overlaid

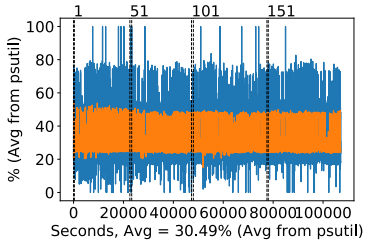


(a) Multiclass all epochs (b) Multiclass first epoch (c) Multiclass final epoch

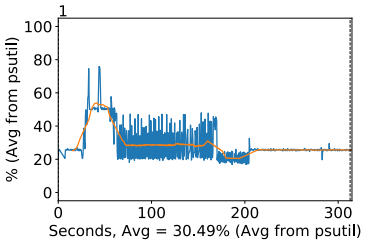


(d) Binary all epochs (e) Binary first epoch (f) Binary final epoch

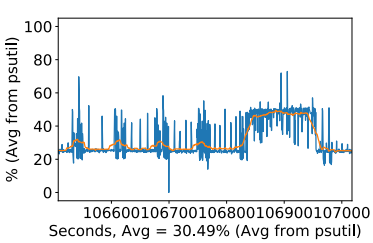
Figure A.175: NASNet Mobile GPU power usage during fine tuning in W, with a moving window average over 40 measurements overlaid



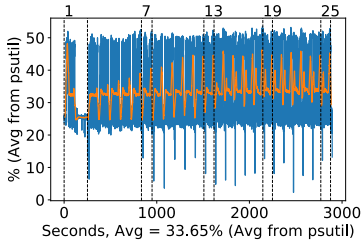
(a) Multiclass all epochs



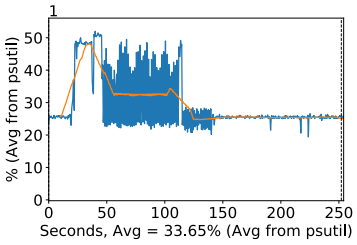
(b) Multiclass first epoch



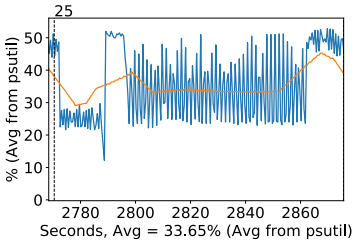
(c) Multiclass final epoch



(d) Binary all epochs

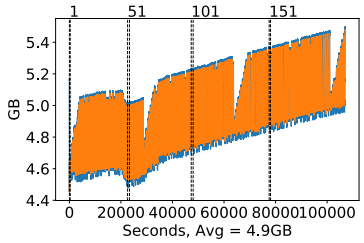


(e) Binary first epoch

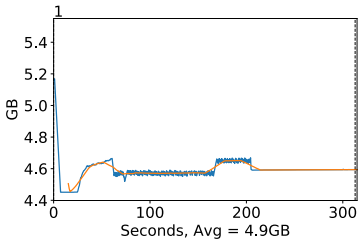


(f) Binary final epoch

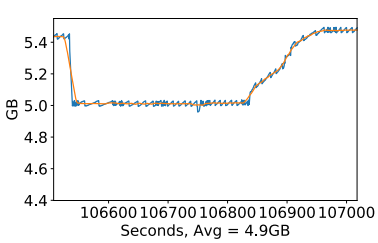
Figure A.176: NASNet Mobile CPU usage (averaged over 4 cores) during fine tuning, with a moving window average over 40 measurements overlaid



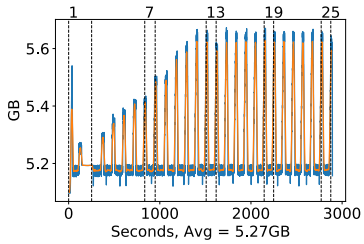
(a) Multiclass all epochs



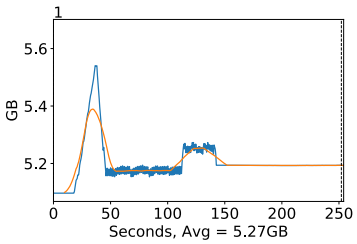
(b) Multiclass first epoch



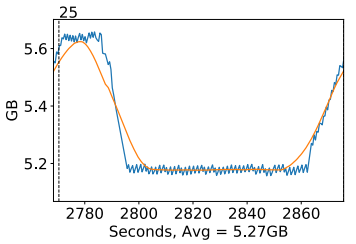
(c) Multiclass final epoch



(d) Binary all epochs



(e) Binary first epoch



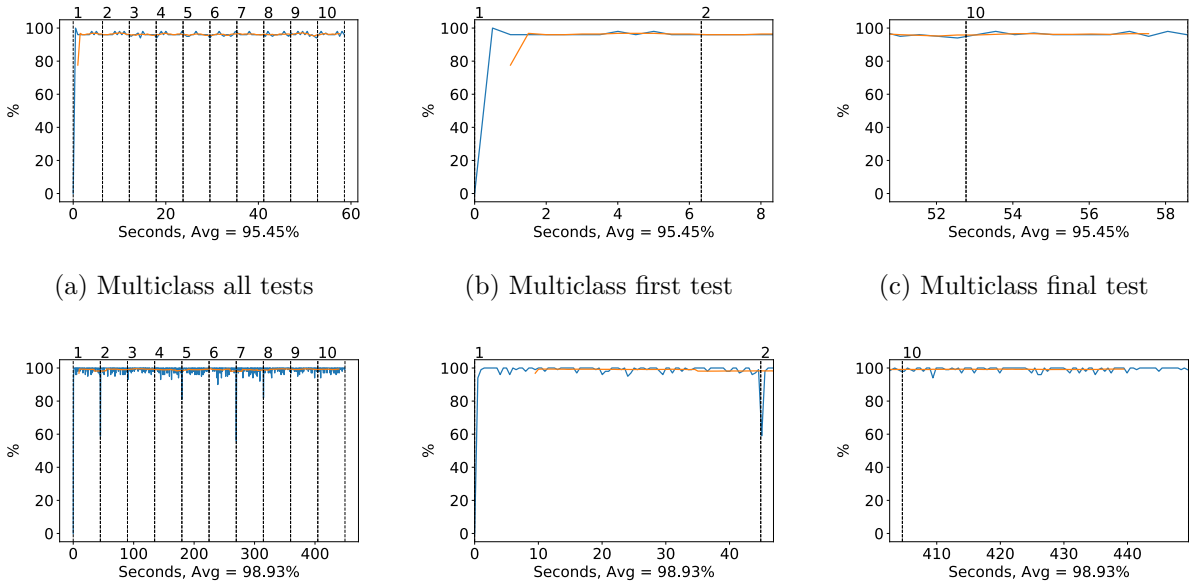
(f) Binary final epoch

Figure A.177: NASNet Mobile Memory usage during fine tuning, with a moving window average over 40 measurements overlaid

A.4 Testing

A.4.1 VGG16

GPU Usage

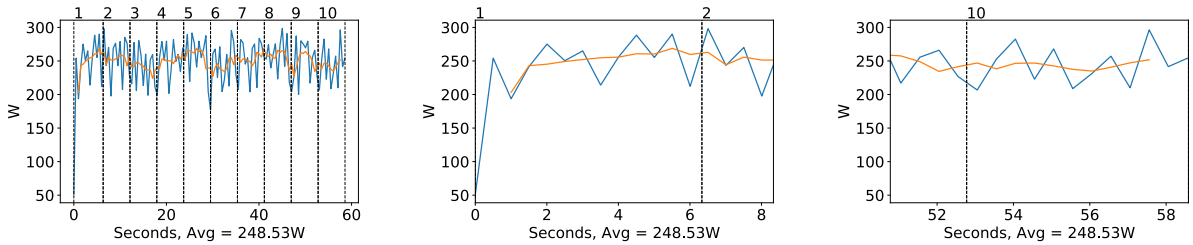


(a) Multiclass all tests (b) Multiclass first test (c) Multiclass final test

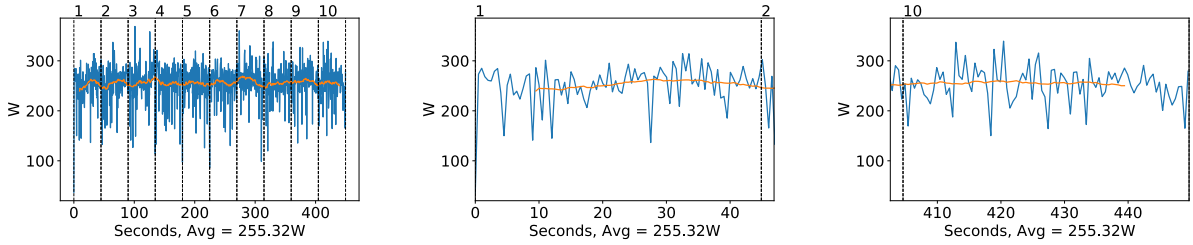
(d) Binary all tests (e) Binary first test (f) Binary final test

Figure A.178: VGG16 GPU volatile usage during testing, with a moving window average over 40 measurements overlaid

CPU and memory usage

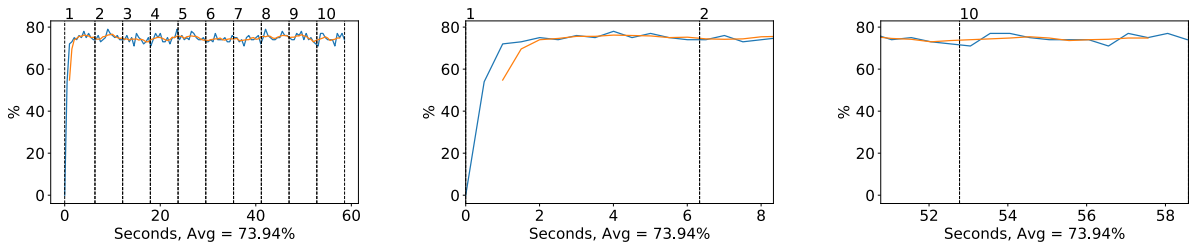


(a) Multiclass all tests (b) Multiclass first test (c) Multiclass final test

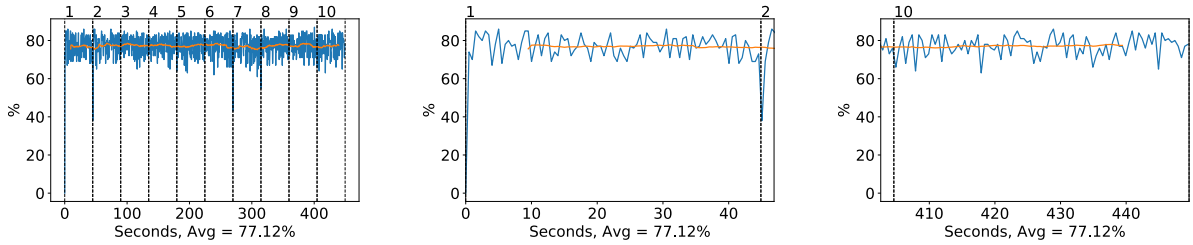


(d) Binary all tests (e) Binary first test (f) Binary final test

Figure A.179: VGG16 GPU power usage i W during testing, with a moving window average over 40 measurements overlaid

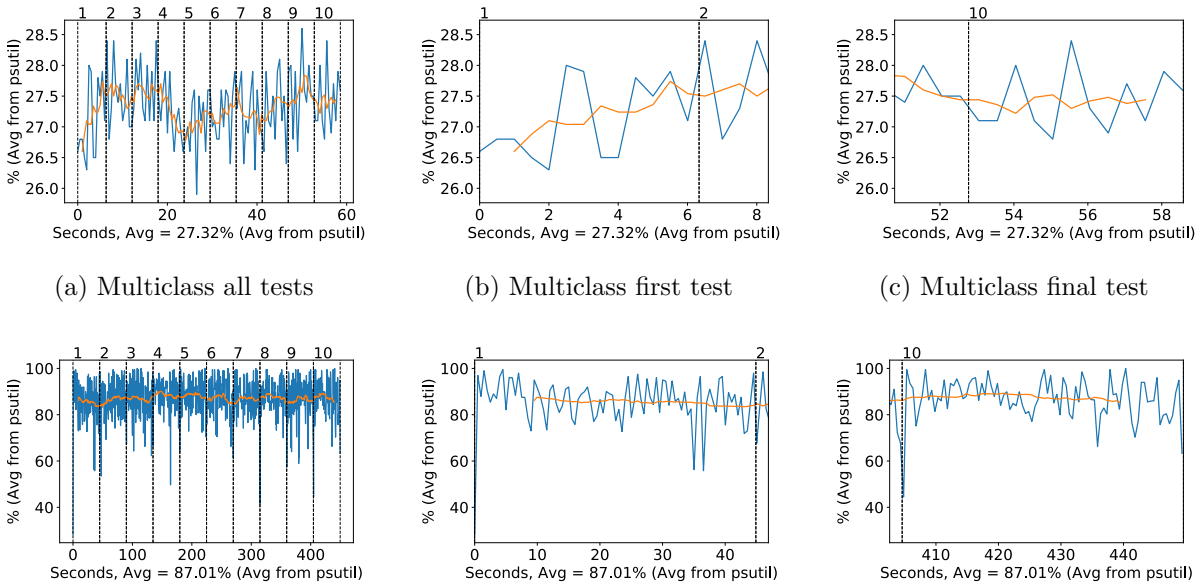


(a) Multiclass all tests (b) Multiclass first test (c) Multiclass final test



(d) Binary all tests (e) Binary first test (f) Binary final test

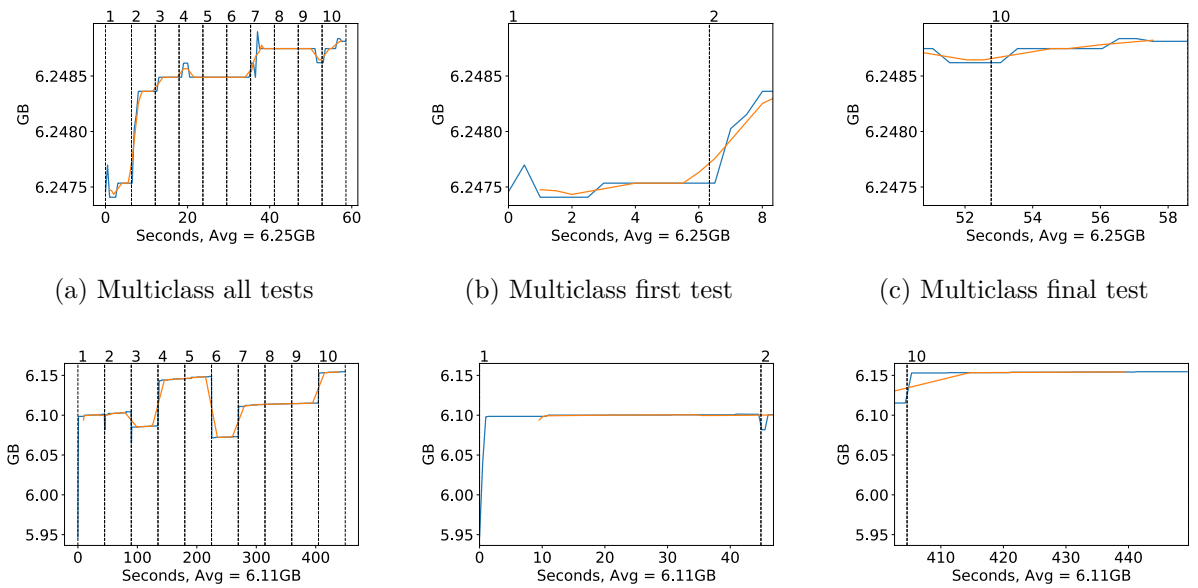
Figure A.180: VGG16 GPU memory usage during testing, with a moving window average over 40 measurements overlaid



(a) Multiclass all tests (b) Multiclass first test (c) Multiclass final test

(d) Binary all tests (e) Binary first test (f) Binary final test

Figure A.181: VGG16 CPU usage (averaged over 4 cores) during testing, with a moving window average over 40 measurements overlaid



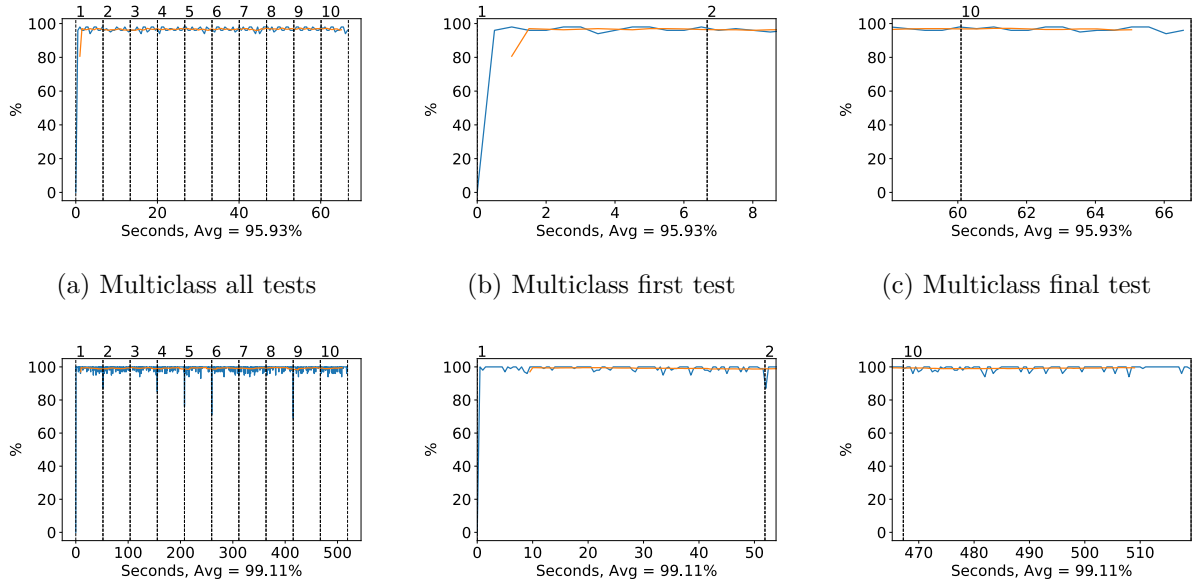
(a) Multiclass all tests (b) Multiclass first test (c) Multiclass final test

(d) Binary all tests (e) Binary first test (f) Binary final test

Figure A.182: VGG16 Memory usage during testing, with a moving window average over 40 measurements overlaid

A.4.2 VGG19

GPU Usage

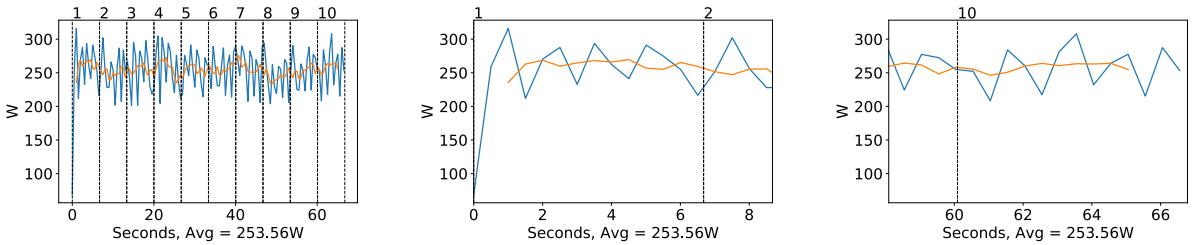


(a) Multiclass all tests (b) Multiclass first test (c) Multiclass final test

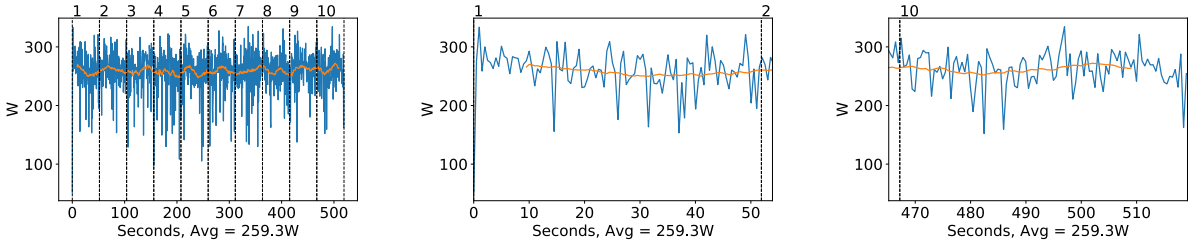
(d) Binary all tests (e) Binary first test (f) Binary final test

Figure A.183: VGG19 GPU volatile usage during testing, with a moving window average over 40 measurements overlaid

CPU and memory usage

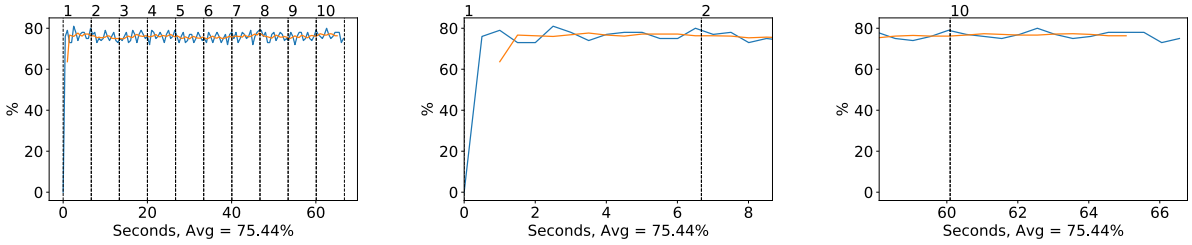


(a) Multiclass all tests (b) Multiclass first test (c) Multiclass final test

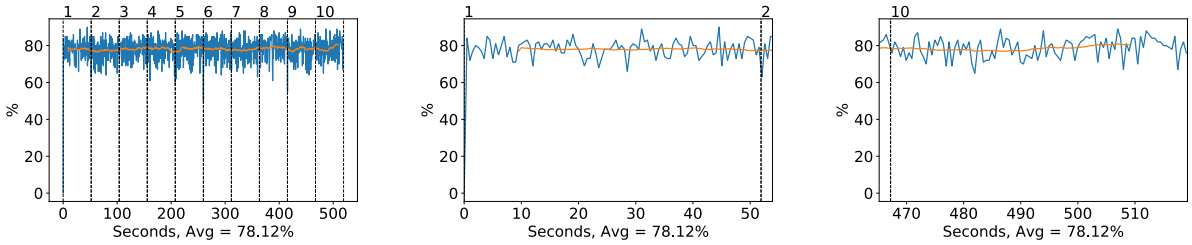


(d) Binary all tests (e) Binary first test (f) Binary final test

Figure A.184: VGG19 GPU power usage i W during testing, with a moving window average over 40 measurements overlaid

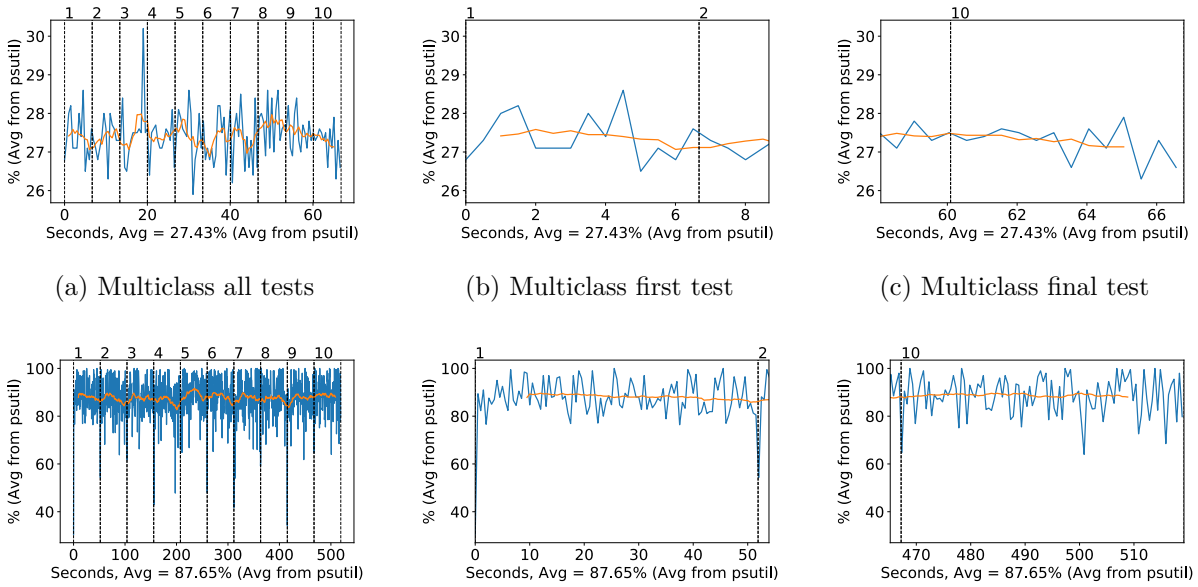


(a) Multiclass all tests (b) Multiclass first test (c) Multiclass final test



(d) Binary all tests (e) Binary first test (f) Binary final test

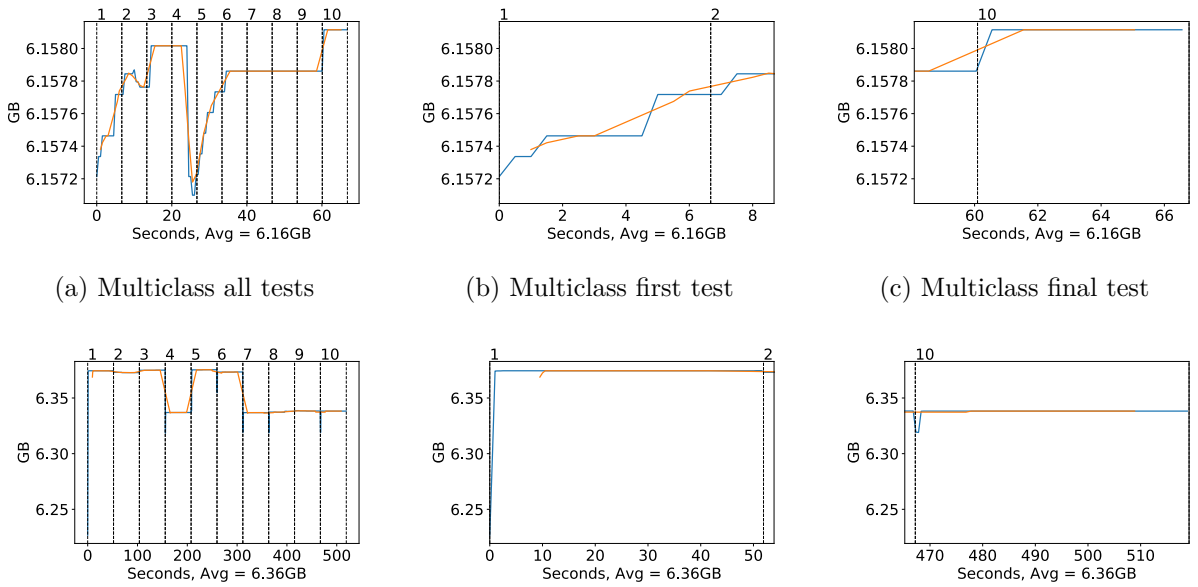
Figure A.185: VGG19 GPU memory usage during testing, with a moving window average over 40 measurements overlaid



(a) Multiclass all tests (b) Multiclass first test (c) Multiclass final test

(d) Binary all tests (e) Binary first test (f) Binary final test

Figure A.186: VGG19 CPU usage (averaged over 4 cores) during testing, with a moving window average over 40 measurements overlaid



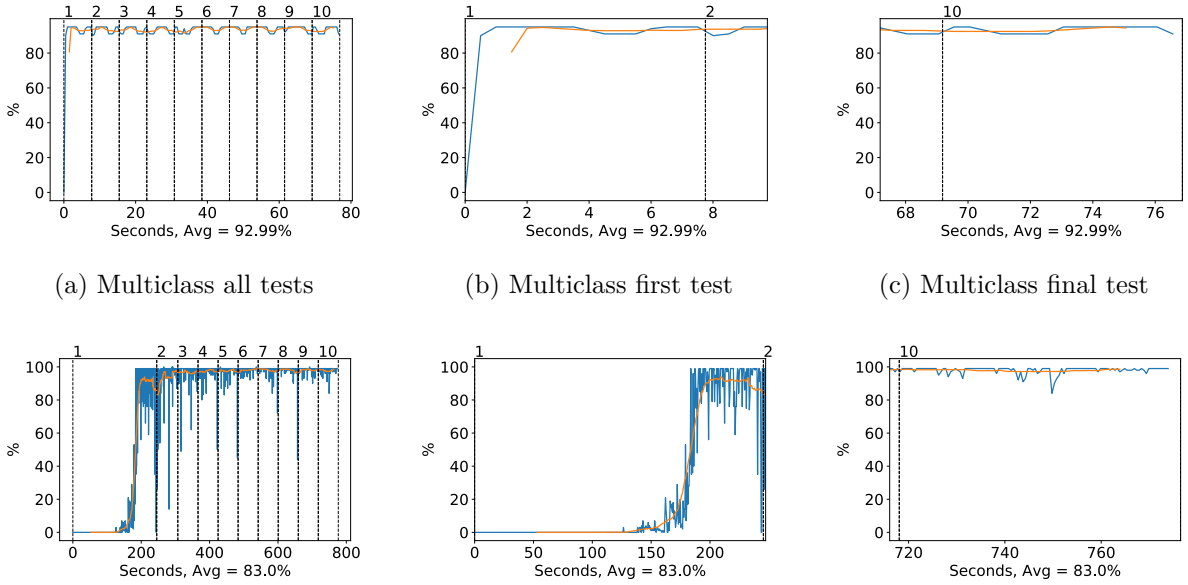
(a) Multiclass all tests (b) Multiclass first test (c) Multiclass final test

(d) Binary all tests (e) Binary first test (f) Binary final test

Figure A.187: VGG19 Memory usage during testing, with a moving window average over 40 measurements overlaid

A.4.3 Inception v3

GPU Usage



(a) Multiclass all tests

(b) Multiclass first test

(c) Multiclass final test

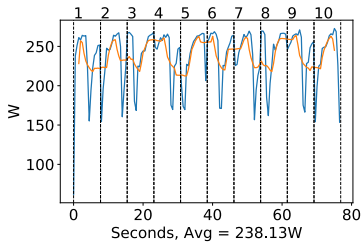
(d) Binary all tests

(e) Binary first test

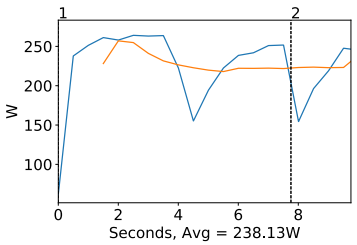
(f) Binary final test

Figure A.188: Inception v3 GPU volatile usage during testing, with a moving window average over 40 measurements overlaid

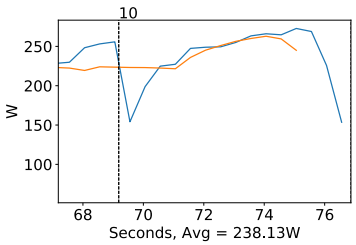
CPU and memory usage



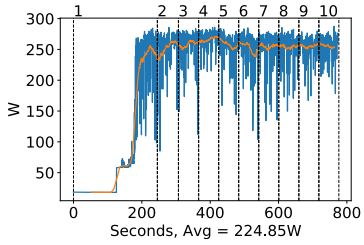
(a) Multiclass all tests



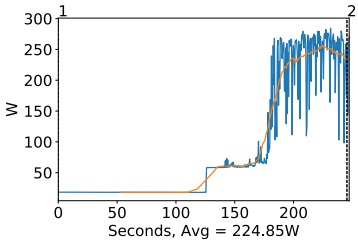
(b) Multiclass first test



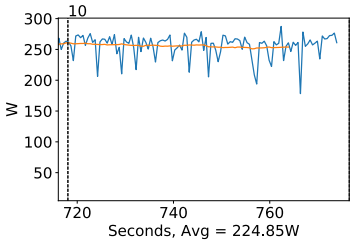
(c) Multiclass final test



(d) Binary all tests

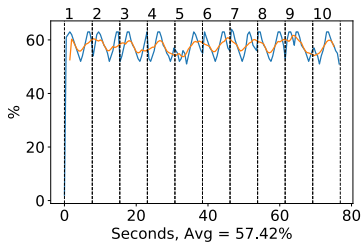


(e) Binary first test

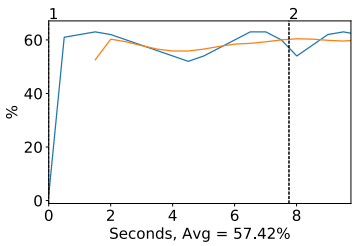


(f) Binary final test

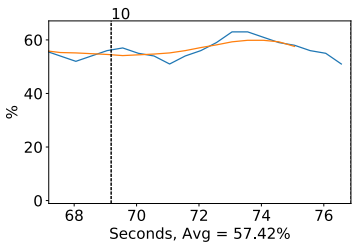
Figure A.189: Inception v3 GPU power usage i W during testing, with a moving window average over 40 measurements overlaid



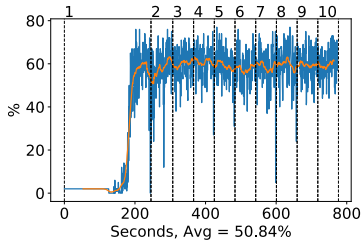
(a) Multiclass all tests



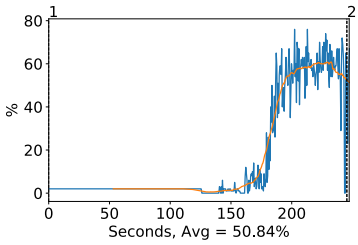
(b) Multiclass first test



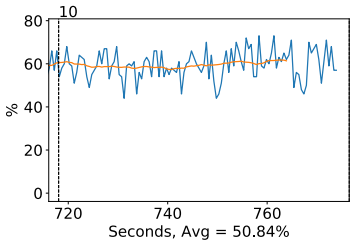
(c) Multiclass final test



(d) Binary all tests

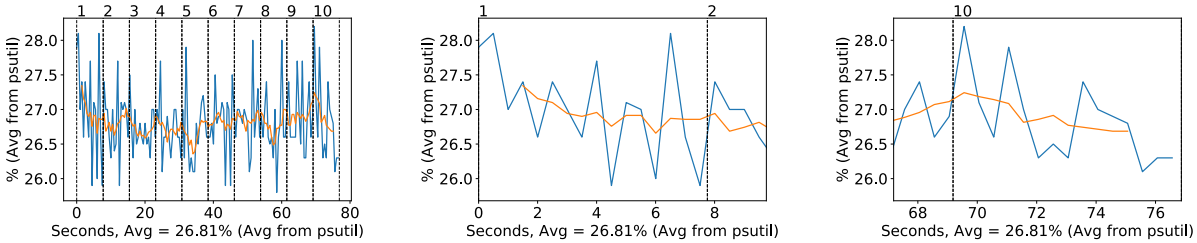


(e) Binary first test

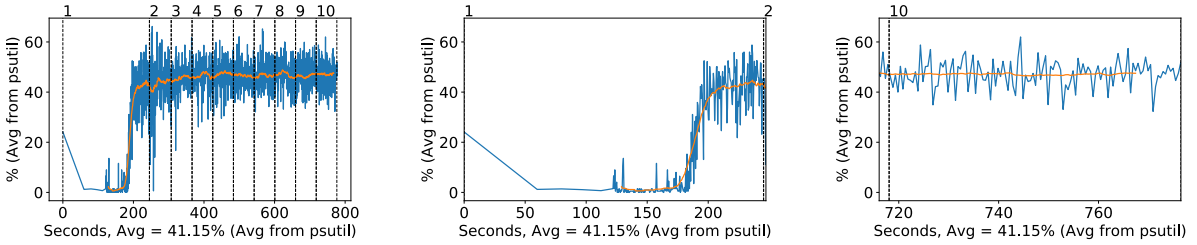


(f) Binary final test

Figure A.190: Inception v3 GPU memory usage during testing, with a moving window average over 40 measurements overlaid

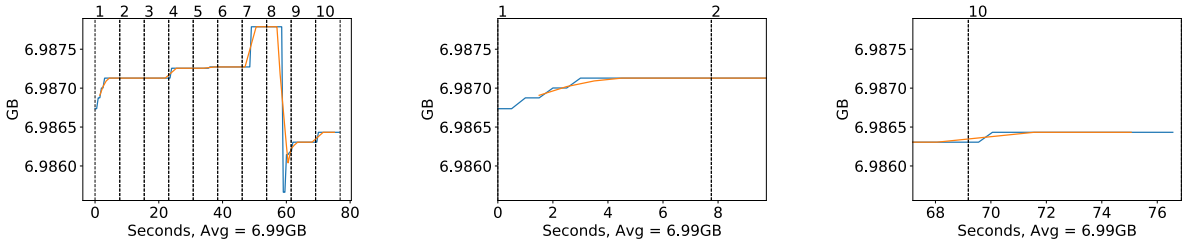


(a) Multiclass all tests (b) Multiclass first test (c) Multiclass final test

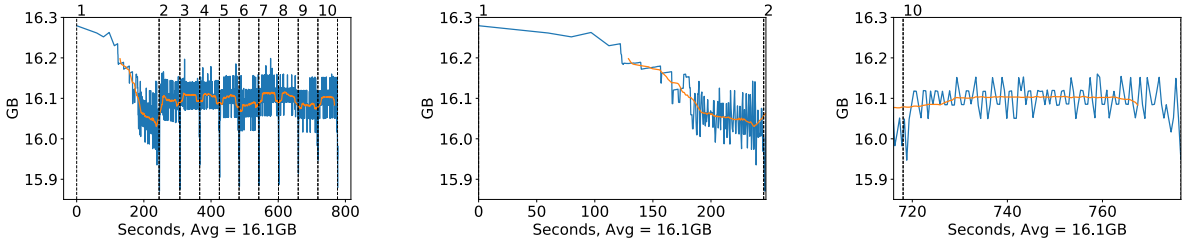


(d) Binary all tests (e) Binary first test (f) Binary final test

Figure A.191: Inception v3 CPU usage (averaged over 4 cores) during testing, with a moving window average over 40 measurements overlaid



(a) Multiclass all tests (b) Multiclass first test (c) Multiclass final test

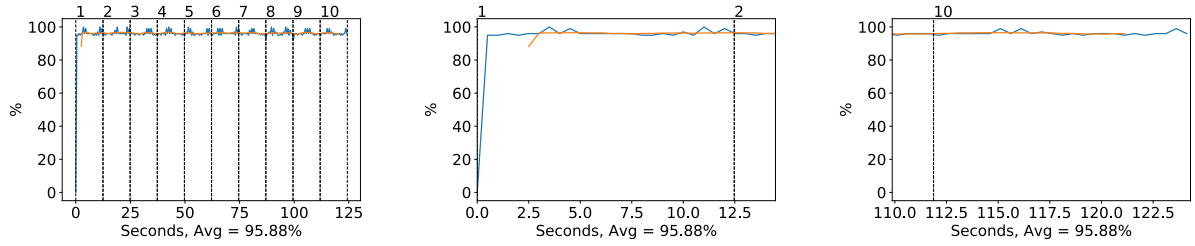


(d) Binary all tests (e) Binary first test (f) Binary final test

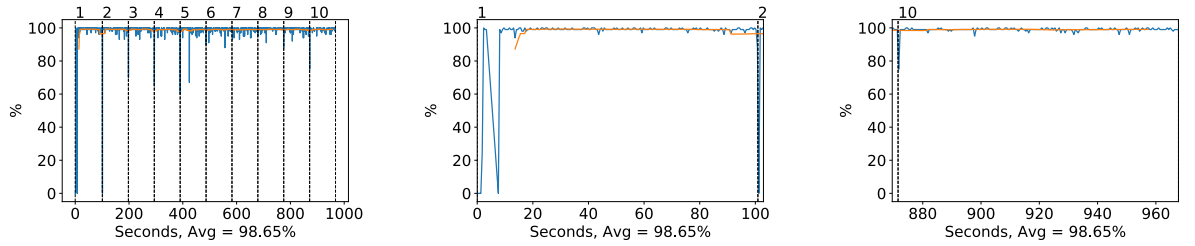
Figure A.192: Inception v3 Memory usage during testing, with a moving window average over 40 measurements overlaid

A.4.4 DenseNet 121

GPU Usage



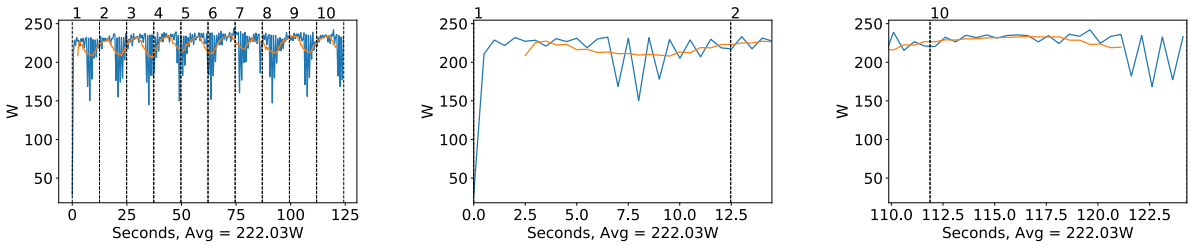
(a) Multiclass all tests (b) Multiclass first test (c) Multiclass final test



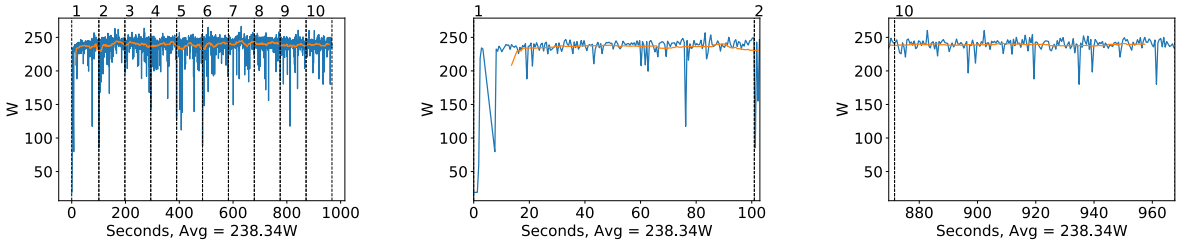
(d) Binary all tests (e) Binary first test (f) Binary final test

Figure A.193: DenseNet 121 GPU volatile usage during testing, with a moving window average over 40 measurements overlaid

CPU and memory usage

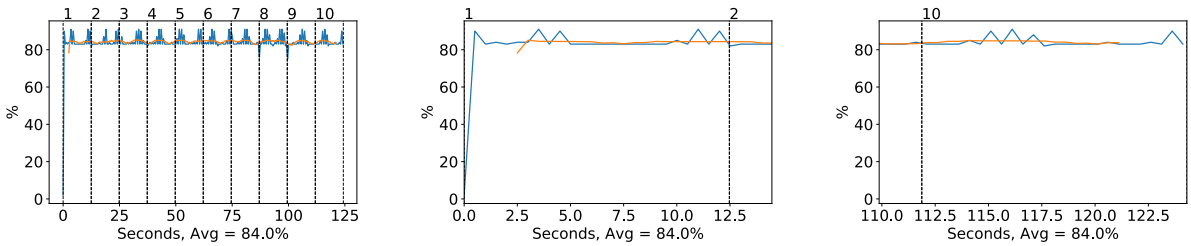


(a) Multiclass all tests (b) Multiclass first test (c) Multiclass final test

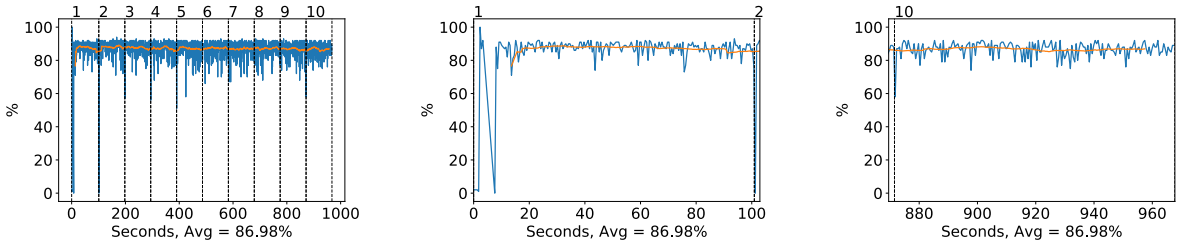


(d) Binary all tests (e) Binary first test (f) Binary final test

Figure A.194: DenseNet 121 GPU power usage i W during testing, with a moving window average over 40 measurements overlaid

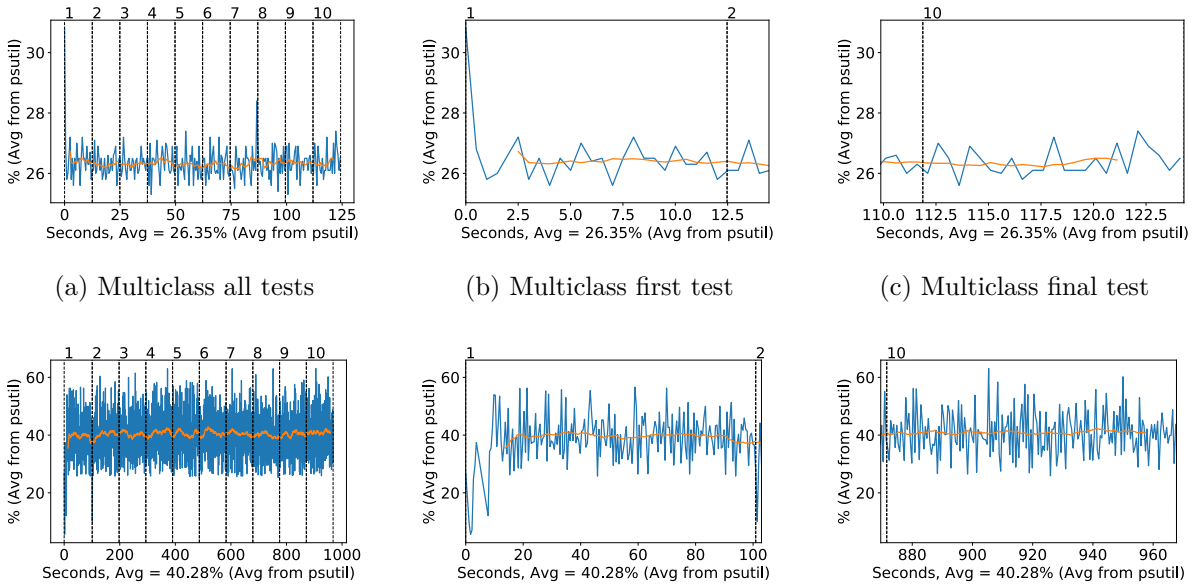


(a) Multiclass all tests (b) Multiclass first test (c) Multiclass final test



(d) Binary all tests (e) Binary first test (f) Binary final test

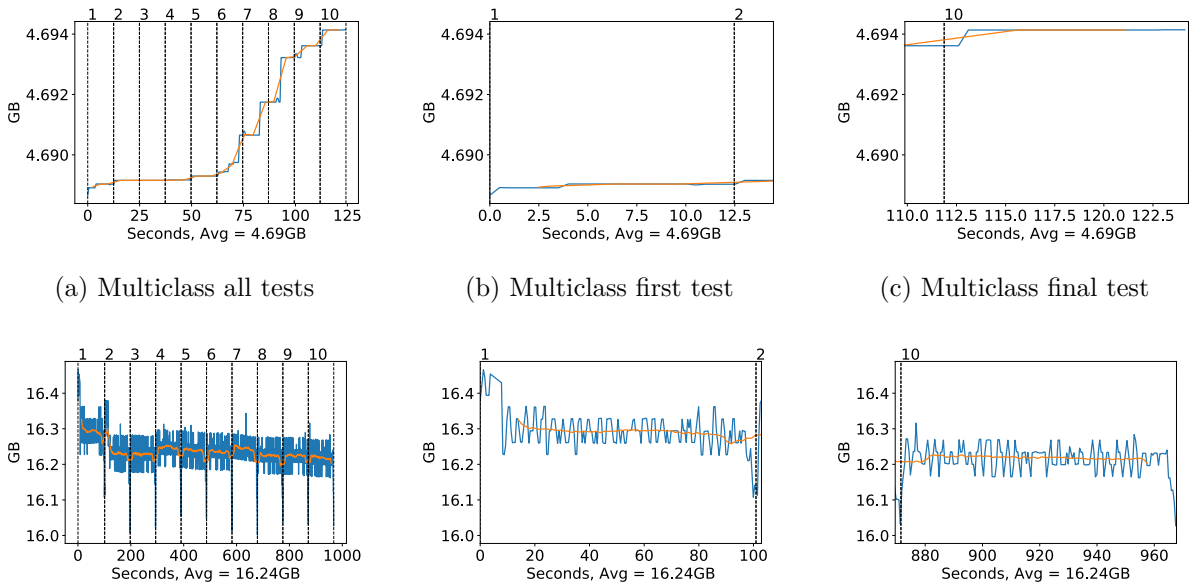
Figure A.195: DenseNet 121 GPU memory usage during testing, with a moving window average over 40 measurements overlaid



(a) Multiclass all tests (b) Multiclass first test (c) Multiclass final test

(d) Binary all tests (e) Binary first test (f) Binary final test

Figure A.196: DenseNet 121 CPU usage (averaged over 4 cores) during fine tuning, with a moving window average over 40 measurements overlaid



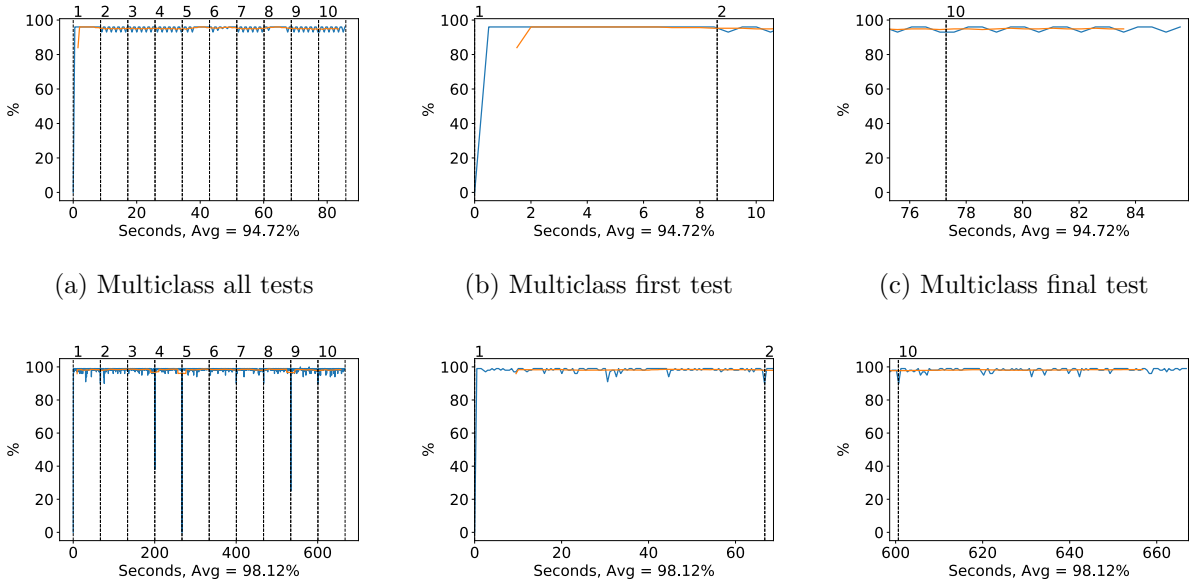
(a) Multiclass all tests (b) Multiclass first test (c) Multiclass final test

(d) Binary all tests (e) Binary first test (f) Binary final test

Figure A.197: DenseNet 121 Memory usage during testing, with a moving window average over 40 measurements overlaid

A.4.5 DenseNet 169

GPU Usage

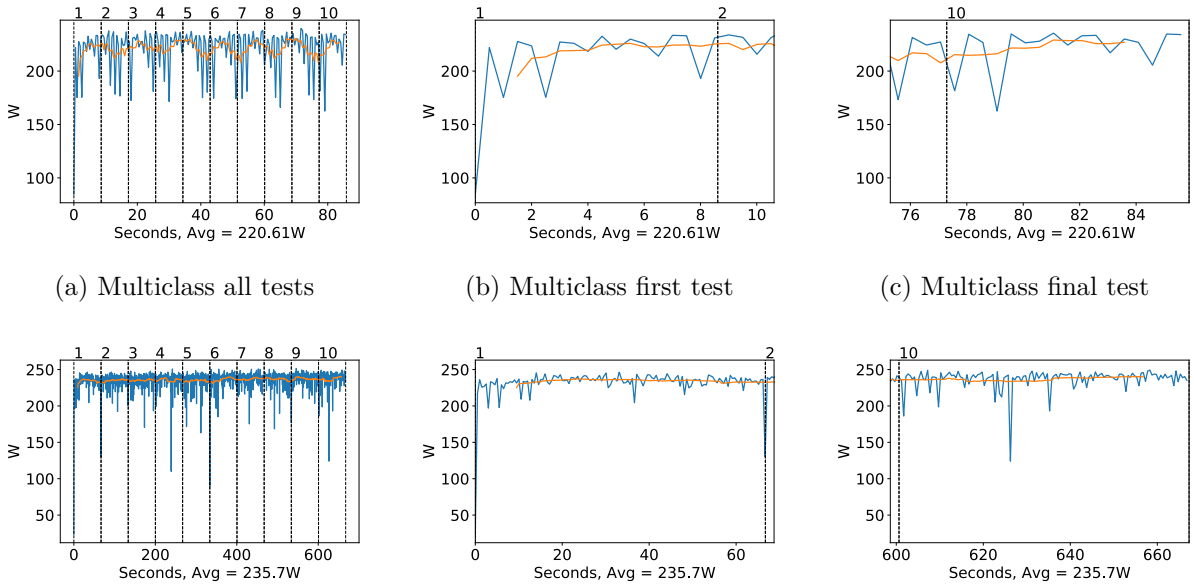


(a) Multiclass all tests (b) Multiclass first test (c) Multiclass final test

(d) Binary all tests (e) Binary first test (f) Binary final test

Figure A.198: DenseNet 169 GPU volatile usage during testing, with a moving window average over 40 measurements overlaid

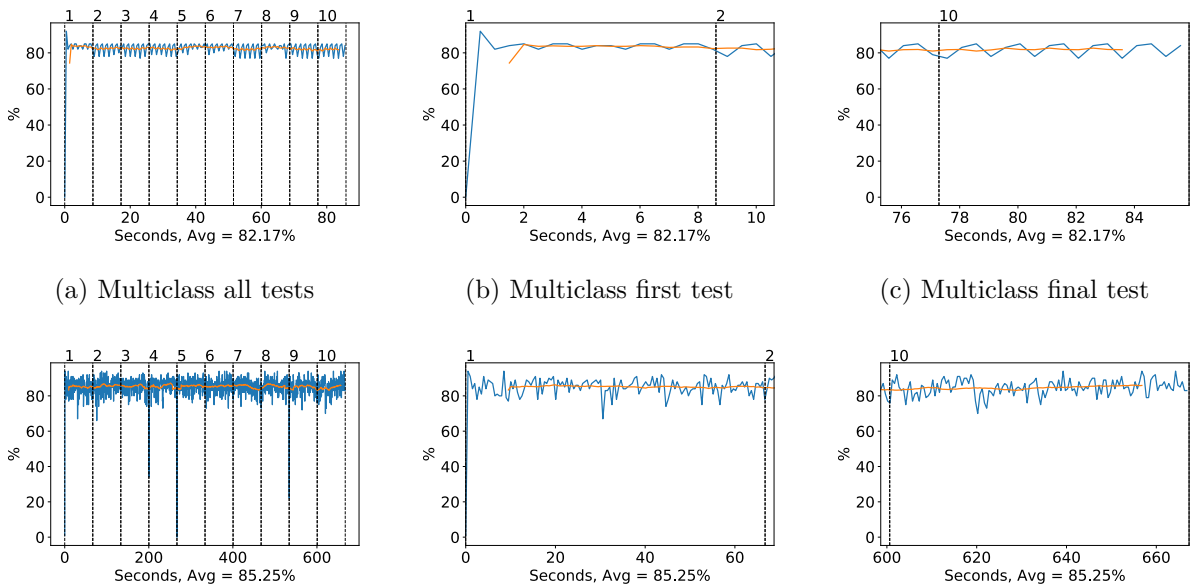
CPU and memory usage



(a) Multiclass all tests (b) Multiclass first test (c) Multiclass final test

(d) Binary all tests (e) Binary first test (f) Binary final test

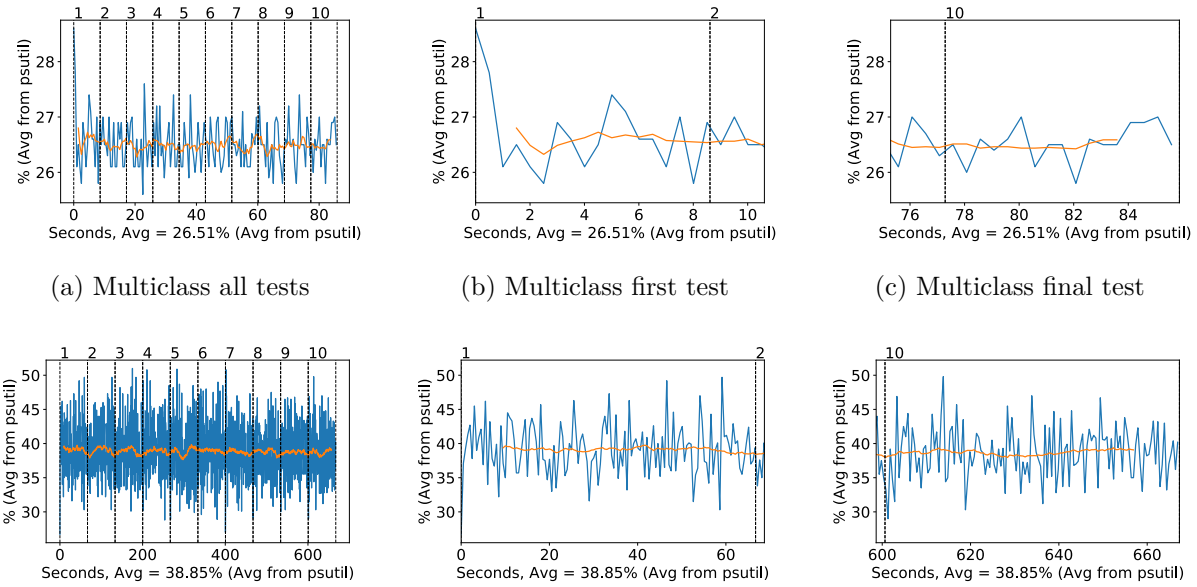
Figure A.199: DenseNet 169 GPU power usage i W during testing, with a moving window average over 40 measurements overlaid



(a) Multiclass all tests (b) Multiclass first test (c) Multiclass final test

(d) Binary all tests (e) Binary first test (f) Binary final test

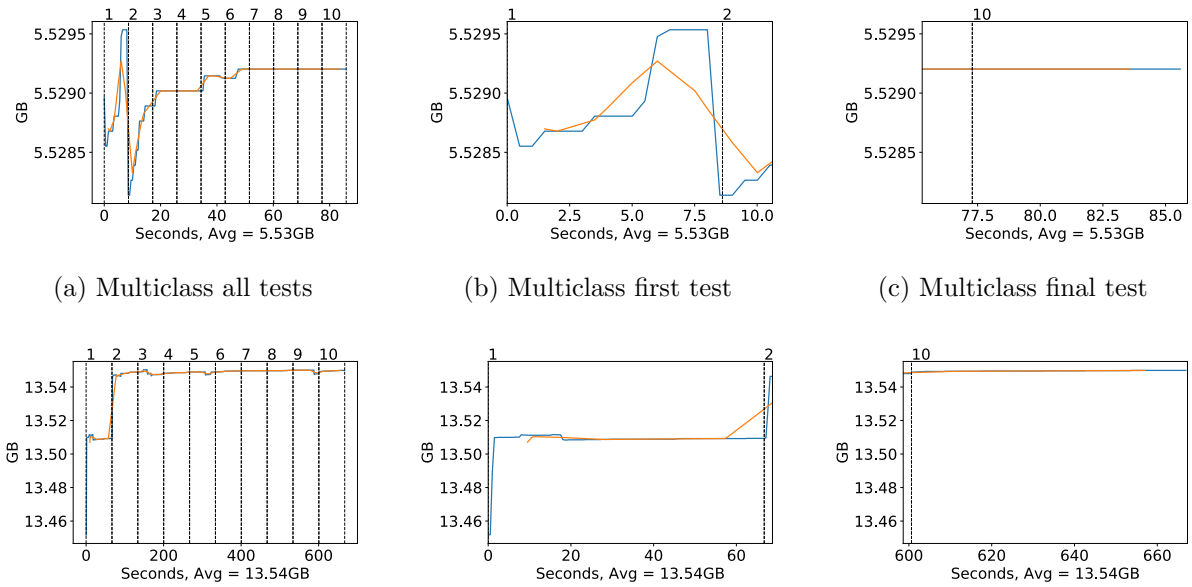
Figure A.200: DenseNet 169 GPU memory usage during testing, with a moving window average over 40 measurements overlaid



(a) Multiclass all tests (b) Multiclass first test (c) Multiclass final test

(d) Binary all tests (e) Binary first test (f) Binary final test

Figure A.201: DenseNet 169 CPU usage (averaged over 4 cores) during testing, with a moving window average over 40 measurements overlaid



(a) Multiclass all tests (b) Multiclass first test (c) Multiclass final test

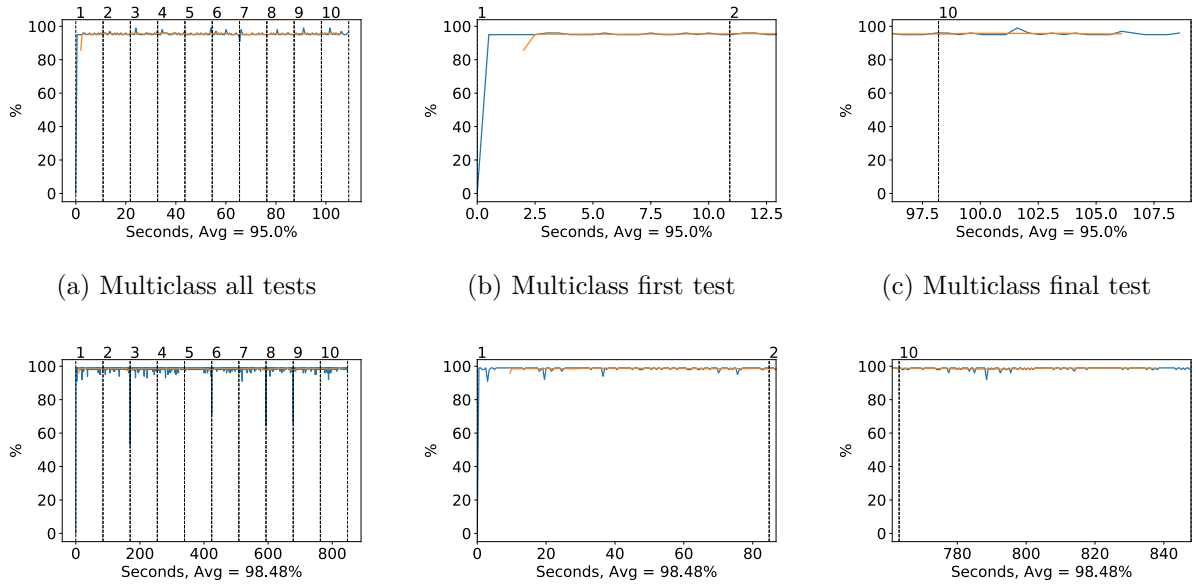
(d) Binary all tests (e) Binary first test (f) Binary final test

Figure A.202: DenseNet 169 Memory usage during testing, with a moving window average over 40 measurements overlaid

A.4.6 DenseNet 201

Testing performance

GPU Usage

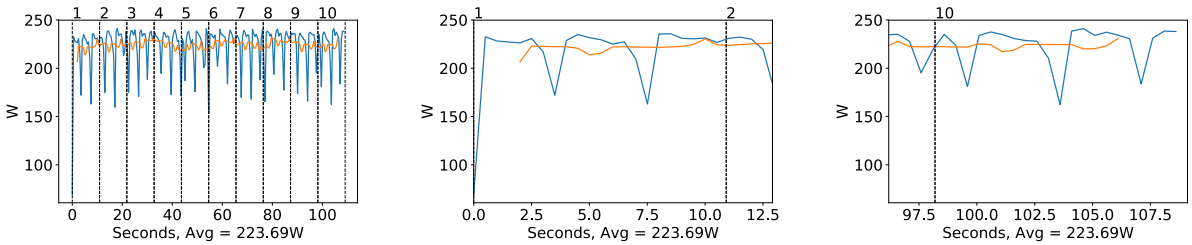


(a) Multiclass all tests (b) Multiclass first test (c) Multiclass final test

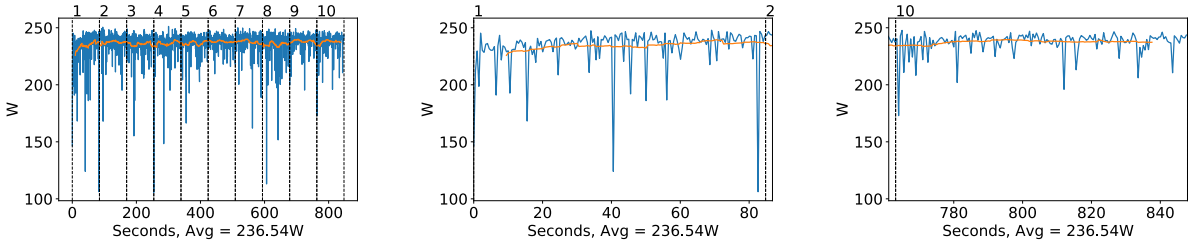
(d) Binary all tests (e) Binary first test (f) Binary final test

Figure A.203: DenseNet 201 GPU volatile usage during testing, with a moving window average over 40 measurements overlaid

CPU and memory usage

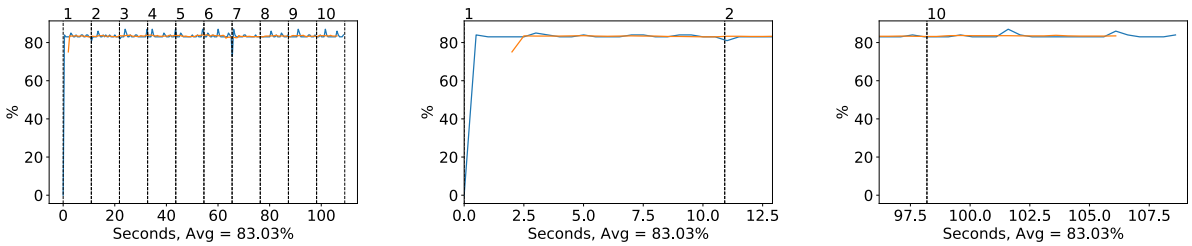


(a) Multiclass all tests (b) Multiclass first test (c) Multiclass final test

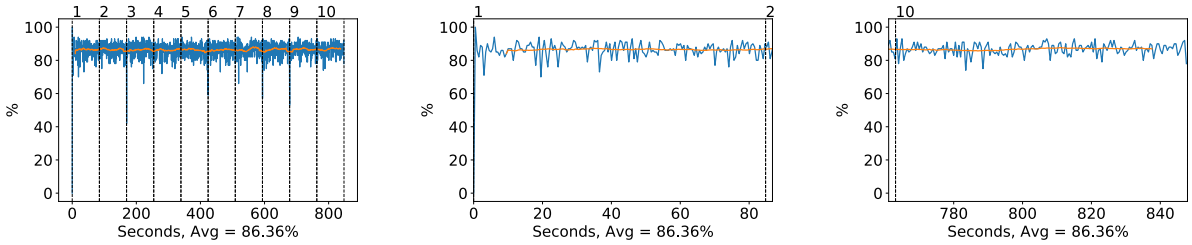


(d) Binary all tests (e) Binary first test (f) Binary final test

Figure A.204: DenseNet 201 GPU power usage i W during testing, with a moving window average over 40 measurements overlaid

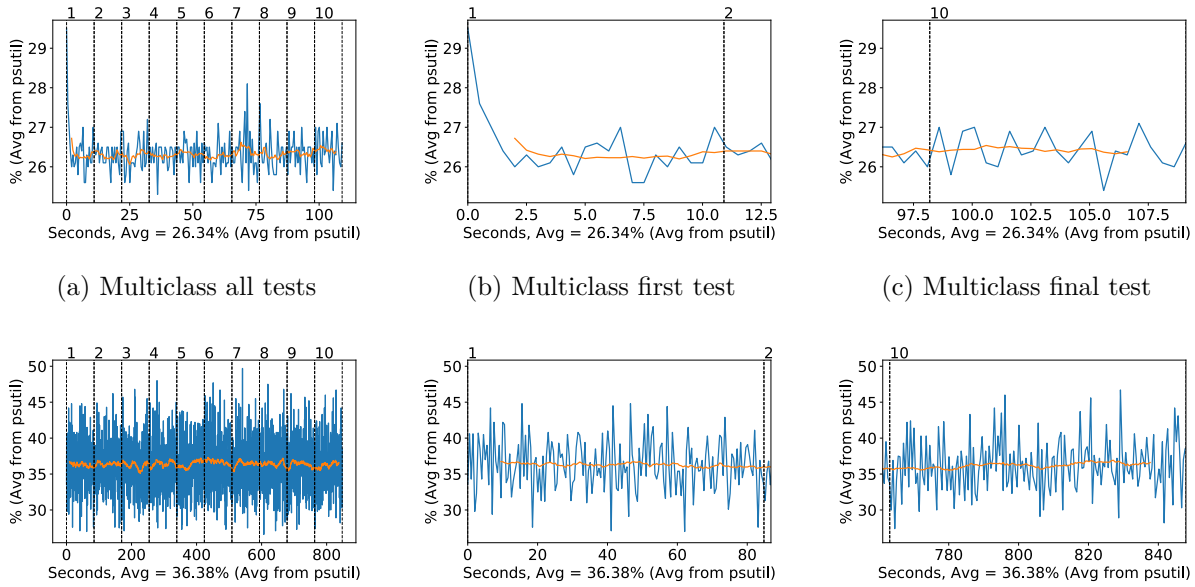


(a) Multiclass all tests (b) Multiclass first test (c) Multiclass final test



(d) Binary all tests (e) Binary first test (f) Binary final test

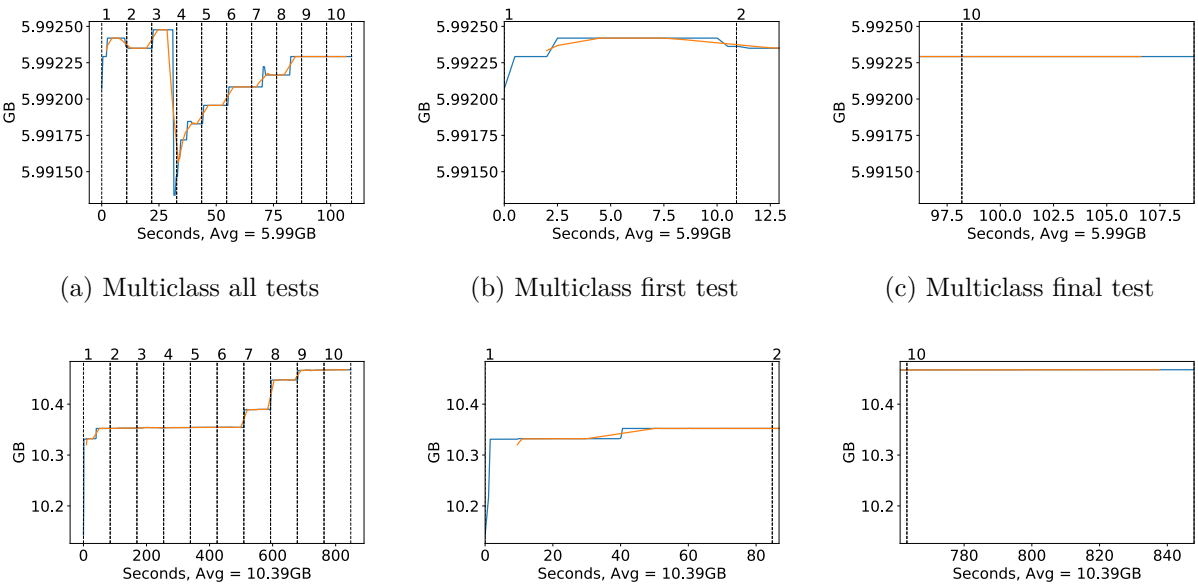
Figure A.205: DenseNet 201 GPU memory usage during testing, with a moving window average over 40 measurements overlaid



(a) Multiclass all tests (b) Multiclass first test (c) Multiclass final test

(d) Binary all tests (e) Binary first test (f) Binary final test

Figure A.206: DenseNet 201 CPU usage (averaged over 4 cores) during testing, with a moving window average over 40 measurements overlaid



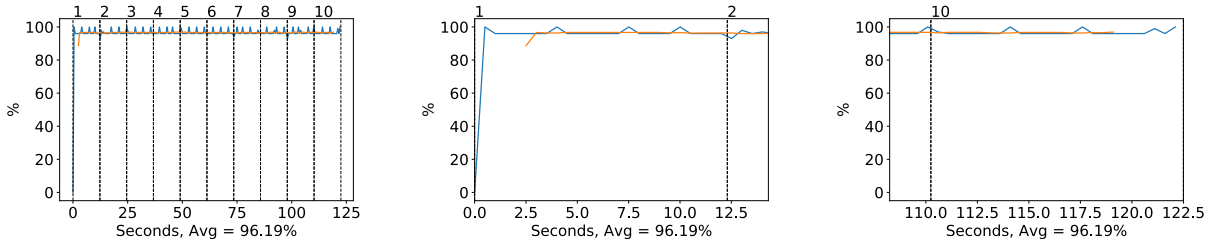
(a) Multiclass all tests (b) Multiclass first test (c) Multiclass final test

(d) Binary all tests (e) Binary first test (f) Binary final test

Figure A.207: DenseNet 201 Memory usage during testing, with a moving window average over 40 measurements overlaid

A.4.7 Xception

GPU Usage

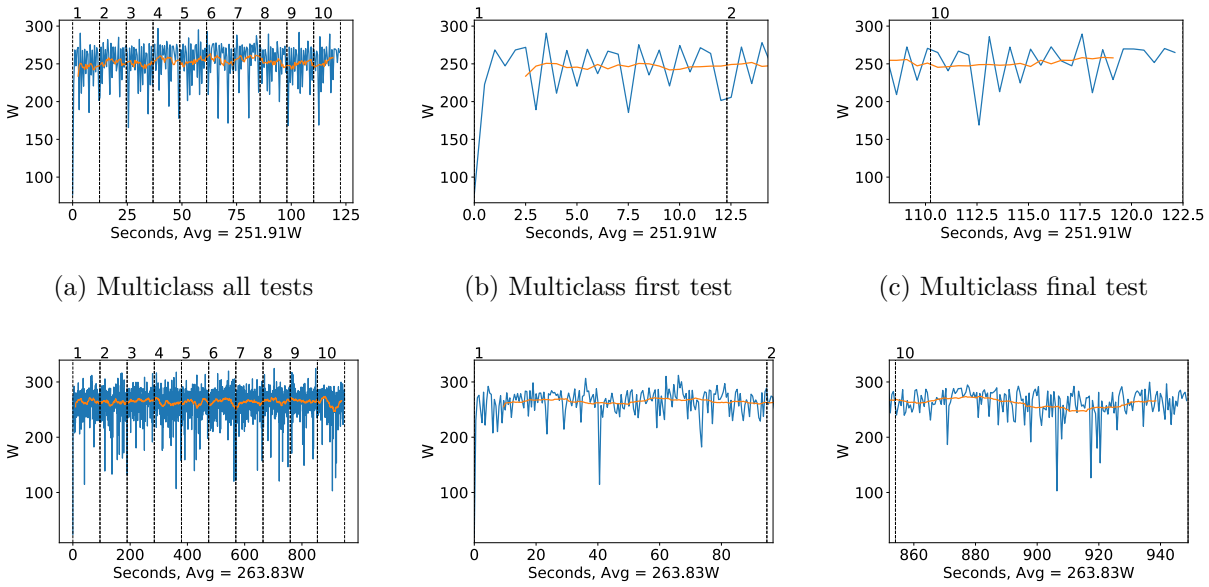


(a) Multiclass all tests (b) Multiclass first test (c) Multiclass final test

(d) Binary all tests (e) Binary first test (f) Binary final test

Figure A.208: Xception GPU volatile usage during testing, with a moving window average over 40 measurements overlaid

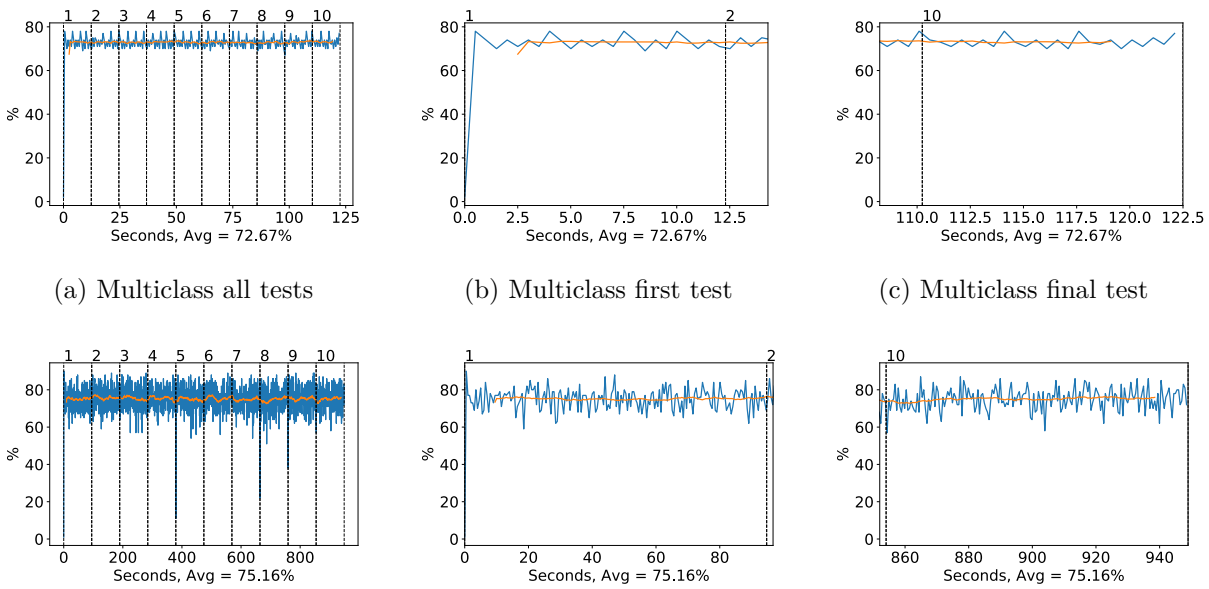
CPU and memory usage



(a) Multiclass all tests (b) Multiclass first test (c) Multiclass final test

(d) Binary all tests (e) Binary first test (f) Binary final test

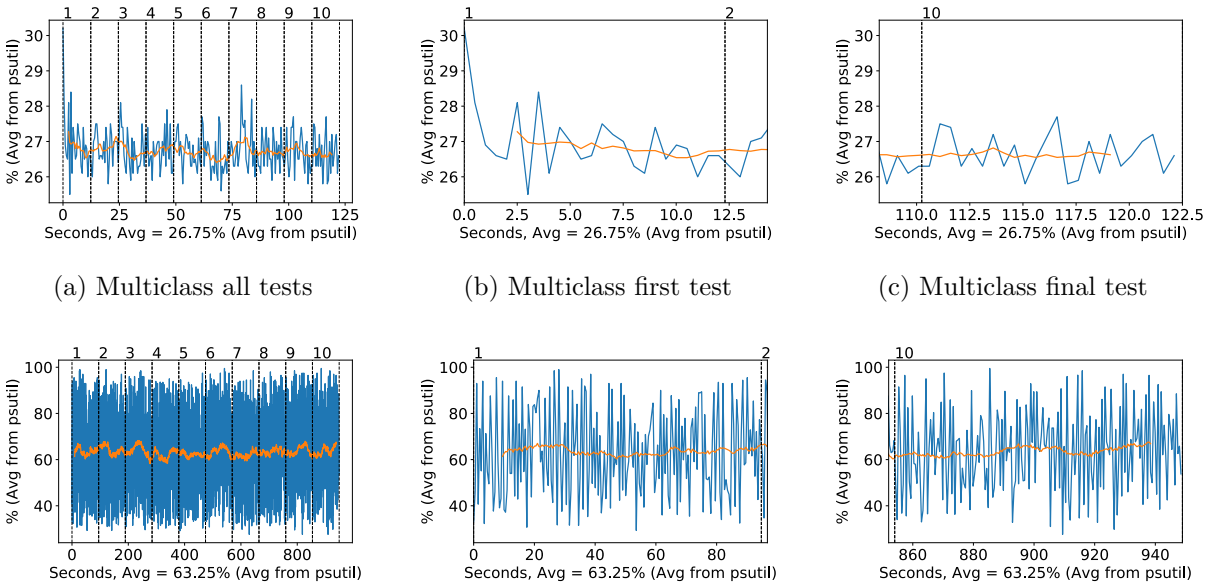
Figure A.209: Xception GPU power usage i W during testing, with a moving window average over 40 measurements overlaid



(a) Multiclass all tests (b) Multiclass first test (c) Multiclass final test

(d) Binary all tests (e) Binary first test (f) Binary final test

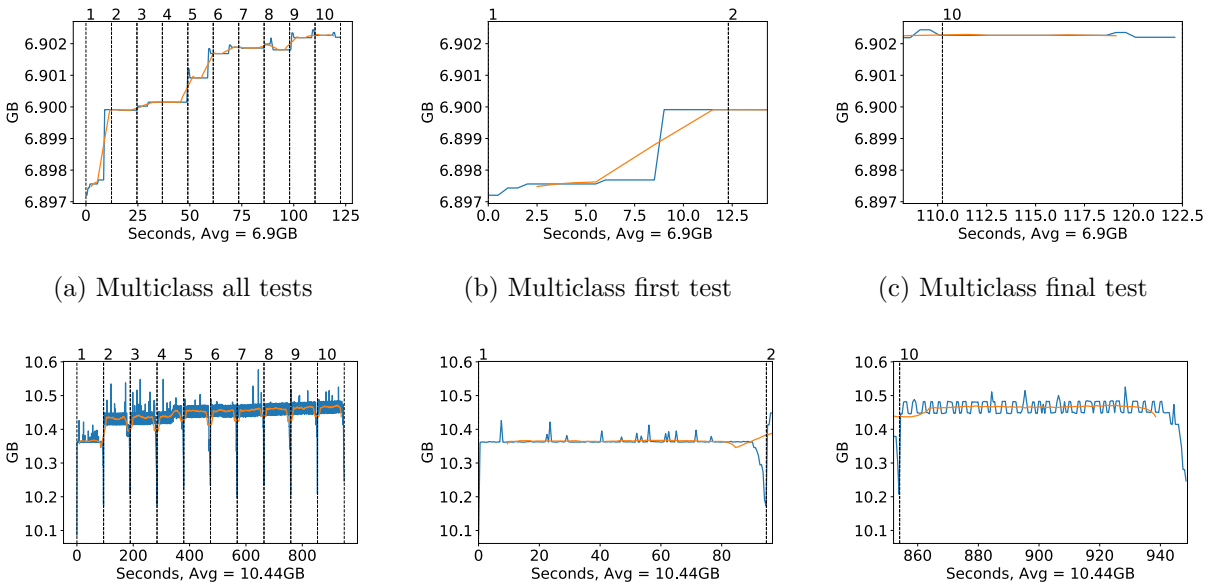
Figure A.210: Xception GPU memory usage during testing, with a moving window average over 40 measurements overlaid



(a) Multiclass all tests (b) Multiclass first test (c) Multiclass final test

(d) Binary all tests (e) Binary first test (f) Binary final test

Figure A.211: Xception CPU usage (averaged over 4 cores) during testing, with a moving window average over 40 measurements overlaid



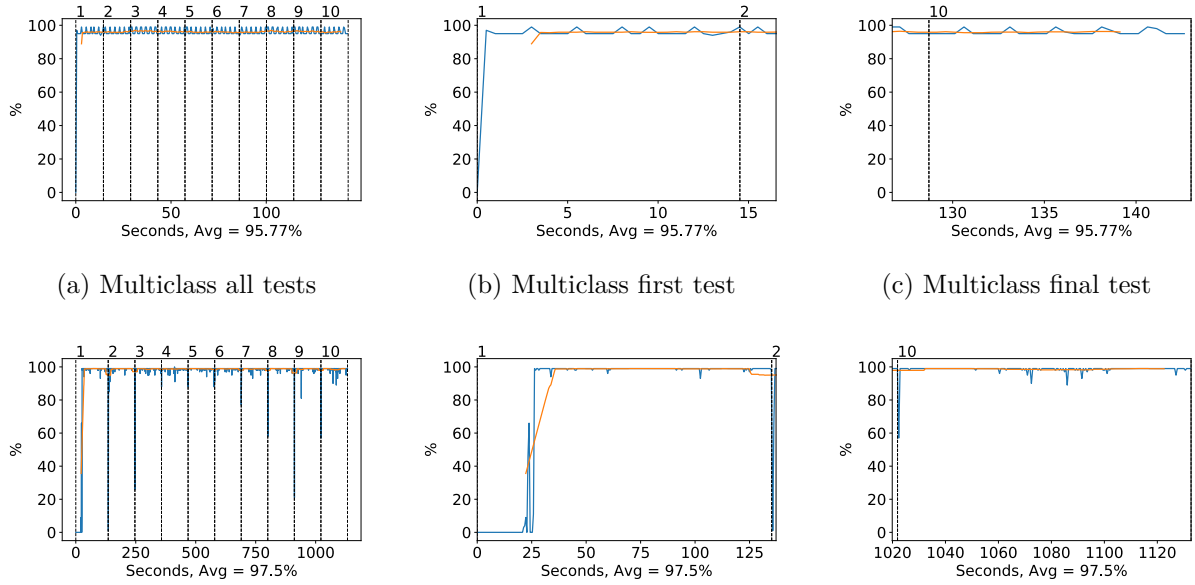
(a) Multiclass all tests (b) Multiclass first test (c) Multiclass final test

(d) Binary all tests (e) Binary first test (f) Binary final test

Figure A.212: Xception Memory usage during testing, with a moving window average over 40 measurements overlaid

A.4.8 Inception ResNet v2

GPU Usage

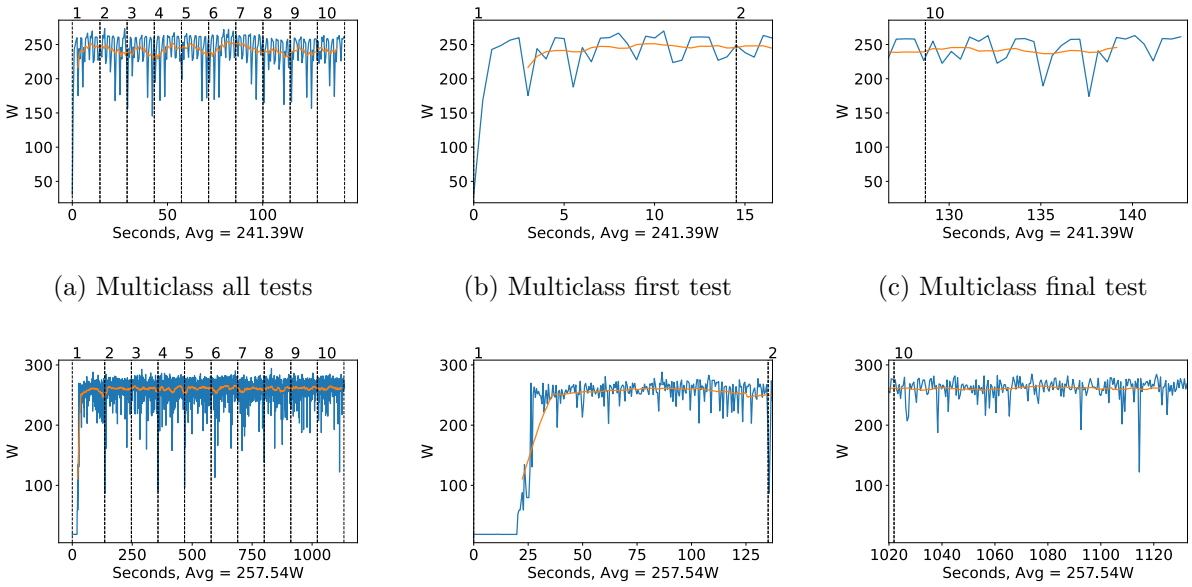


(a) Multiclass all tests (b) Multiclass first test (c) Multiclass final test

(d) Binary all tests (e) Binary first test (f) Binary final test

Figure A.213: Inception-ResNet-v2 GPU volatile usage during testing, with a moving window average over 40 measurements overlaid

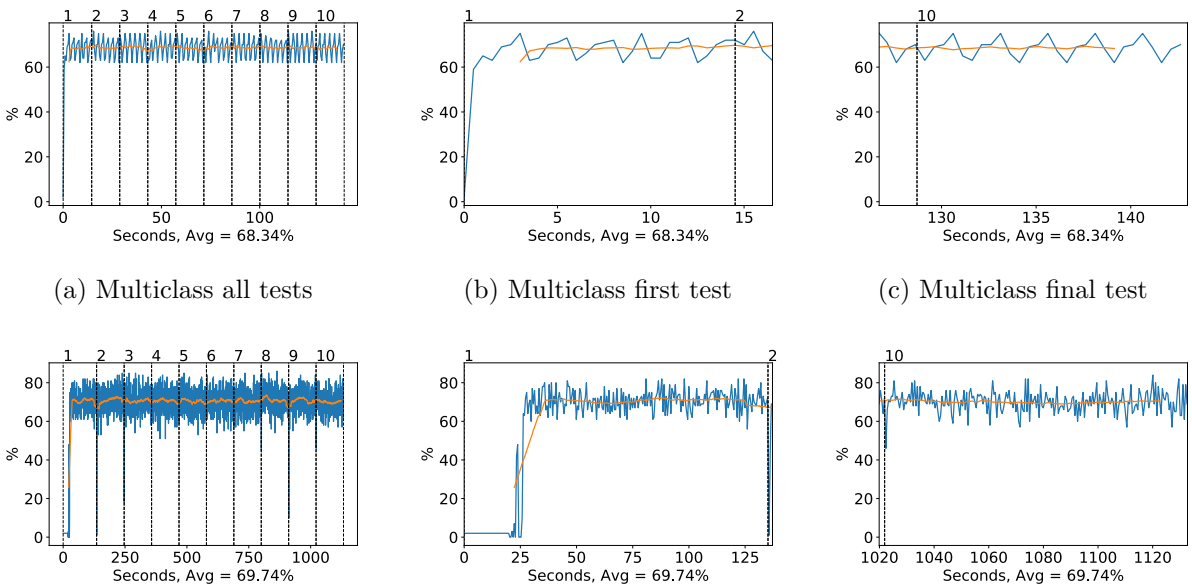
CPU and memory usage



(a) Multiclass all tests (b) Multiclass first test (c) Multiclass final test

(d) Binary all tests (e) Binary first test (f) Binary final test

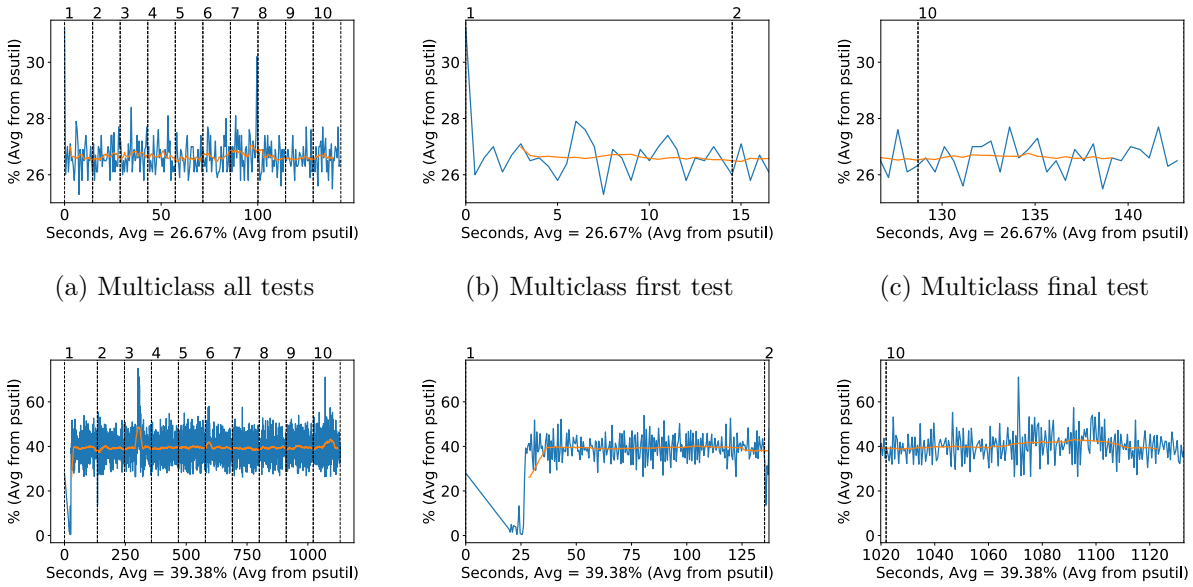
Figure A.214: Inception-ResNet-v2 GPU power usage i W during testing, with a moving window average over 40 measurements overlaid



(a) Multiclass all tests (b) Multiclass first test (c) Multiclass final test

(d) Binary all tests (e) Binary first test (f) Binary final test

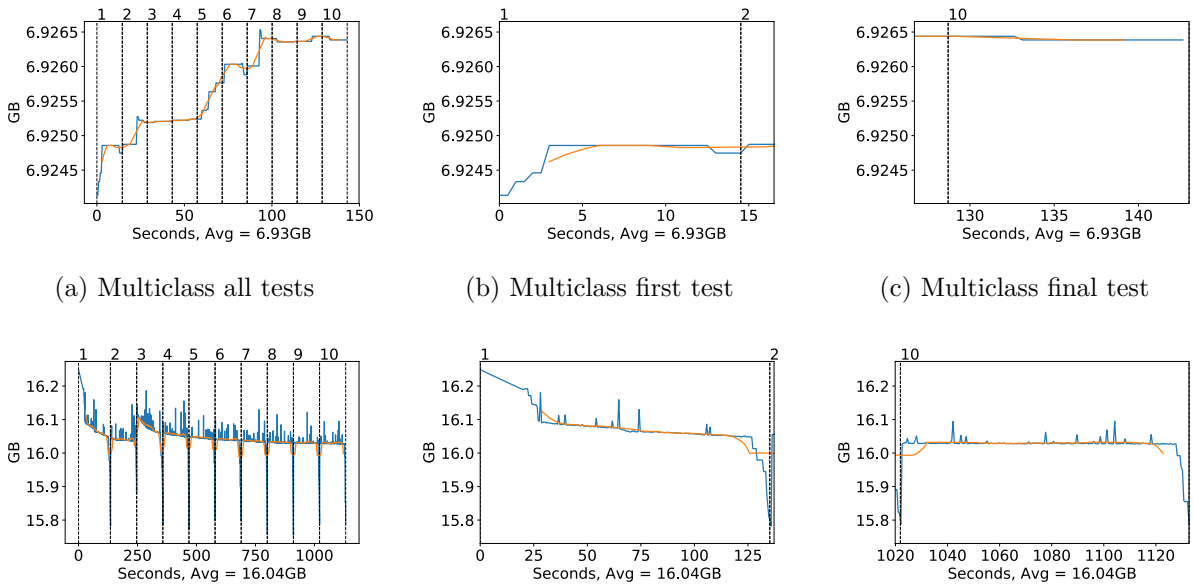
Figure A.215: Inception-ResNet-v2 GPU memory usage during testing, with a moving window average over 40 measurements overlaid



(a) Multiclass all tests (b) Multiclass first test (c) Multiclass final test

(d) Binary all tests (e) Binary first test (f) Binary final test

Figure A.216: Inception-ResNet-v2 CPU usage (averaged over 4 cores) during testing, with a moving window average over 40 measurements overlaid



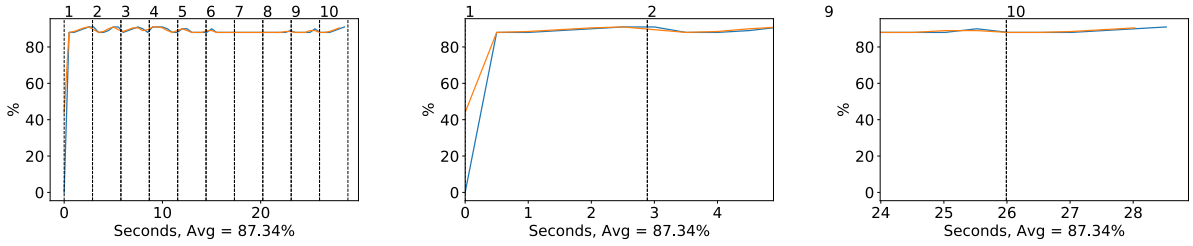
(a) Multiclass all tests (b) Multiclass first test (c) Multiclass final test

(d) Binary all tests (e) Binary first test (f) Binary final test

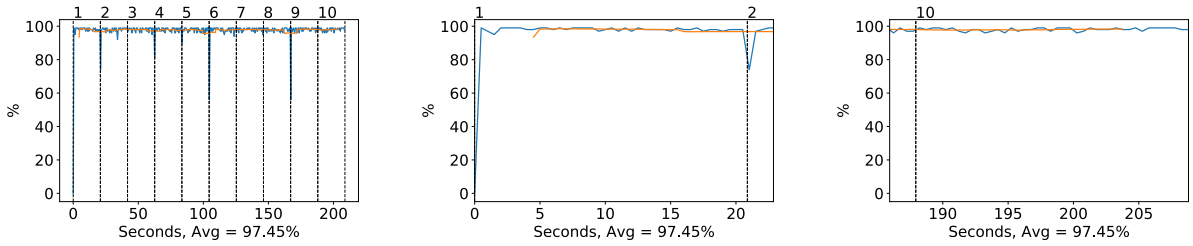
Figure A.217: Inception-ResNet-v2 Memory usage during testing, with a moving window average over 40 measurements overlaid

A.4.9 Mobilenet

GPU Usage



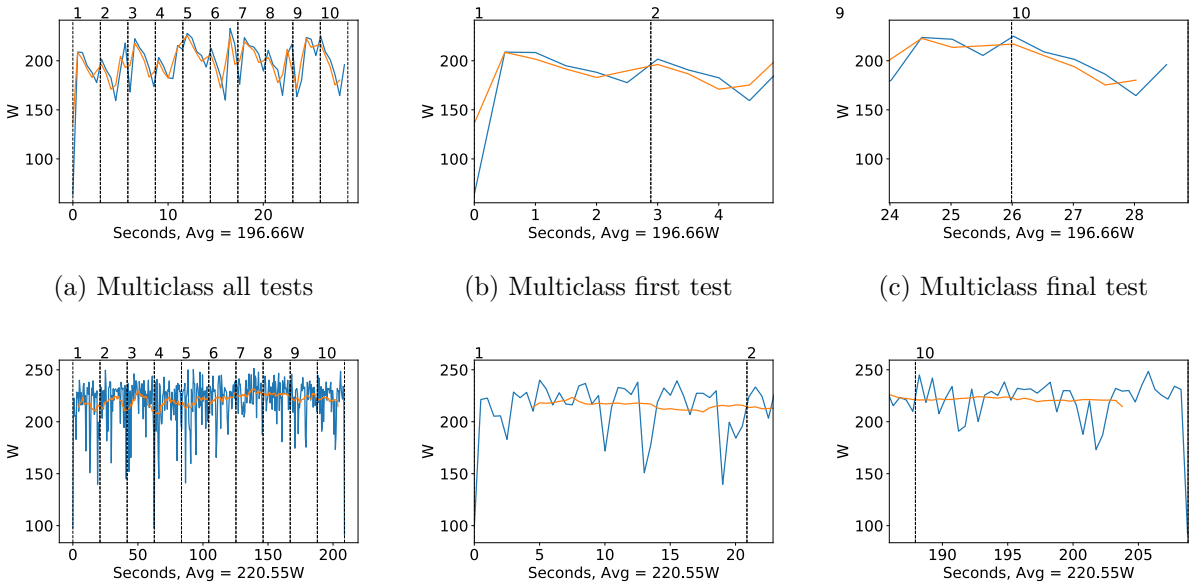
(a) Multiclass all tests (b) Multiclass first test (c) Multiclass final test



(d) Binary all tests (e) Binary first test (f) Binary final test

Figure A.218: Mobilenet GPU volatile usage during testing, with a moving window average over 40 measurements overlaid

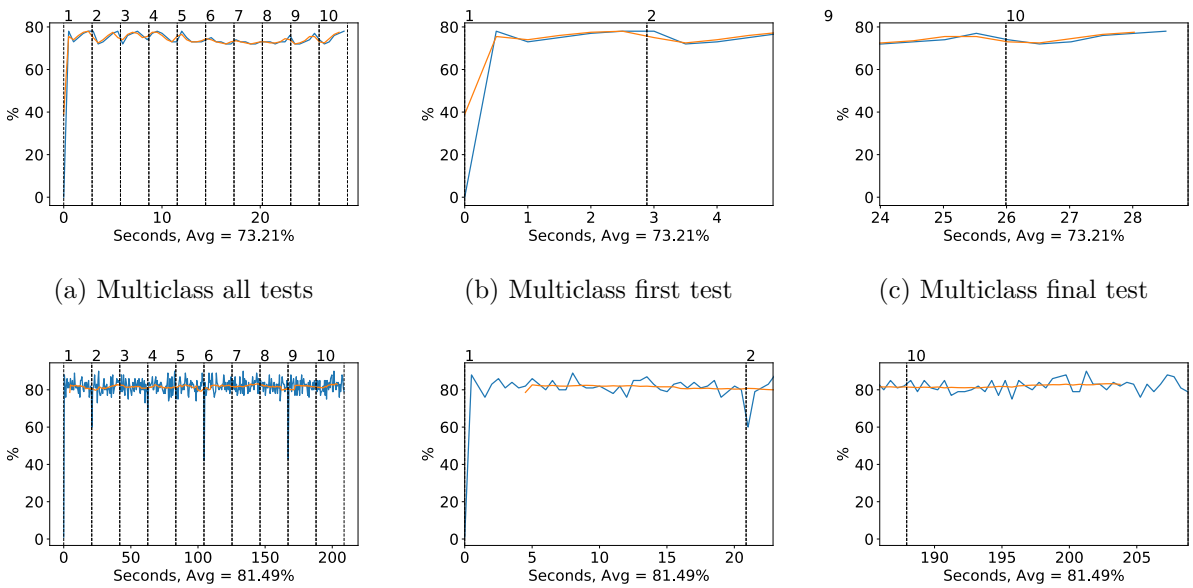
CPU and memory usage



(a) Multiclass all tests (b) Multiclass first test (c) Multiclass final test

(d) Binary all tests (e) Binary first test (f) Binary final test

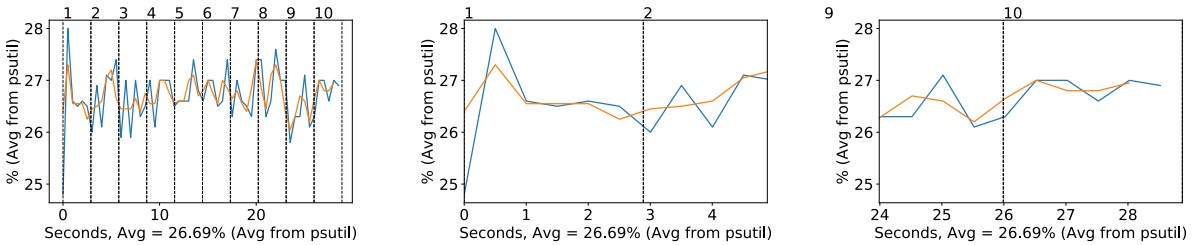
Figure A.219: Mobilenet GPU power usage i W during testing, with a moving window average over 40 measurements overlaid



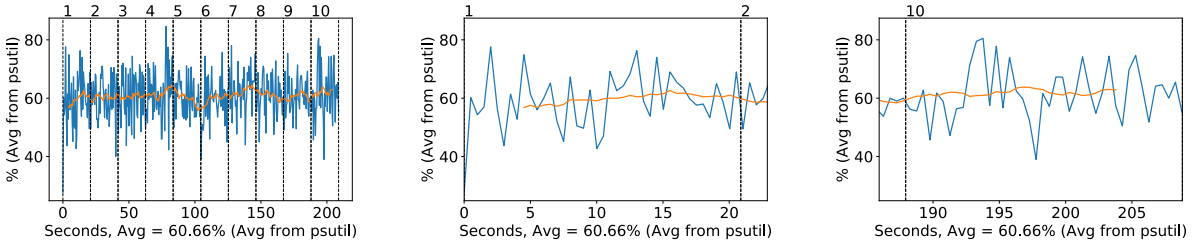
(a) Multiclass all tests (b) Multiclass first test (c) Multiclass final test

(d) Binary all tests (e) Binary first test (f) Binary final test

Figure A.220: Mobilenet GPU memory usage during testing, with a moving window average over 40 measurements overlaid

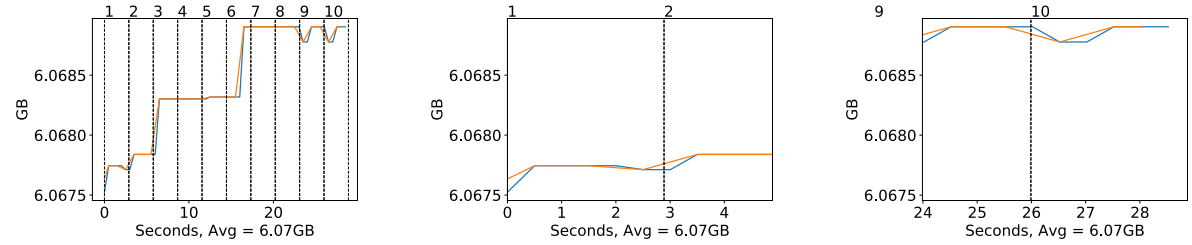


(a) Multiclass all tests (b) Multiclass first test (c) Multiclass final test

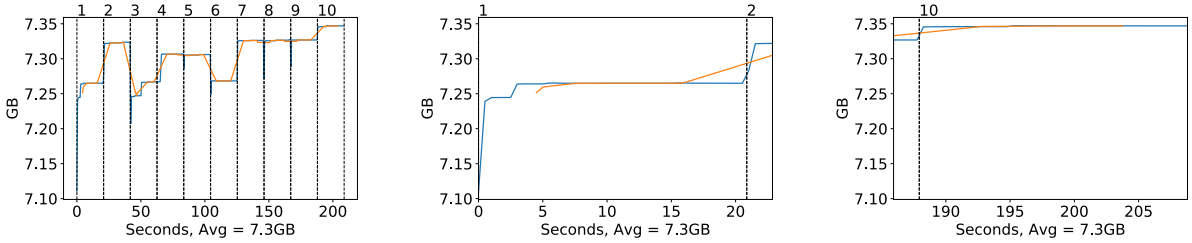


(d) Binary all tests (e) Binary first test (f) Binary final test

Figure A.221: Mobilenet CPU usage (averaged over 4 cores) during testing, with a moving window average over 40 measurements overlaid



(a) Multiclass all tests (b) Multiclass first test (c) Multiclass final test

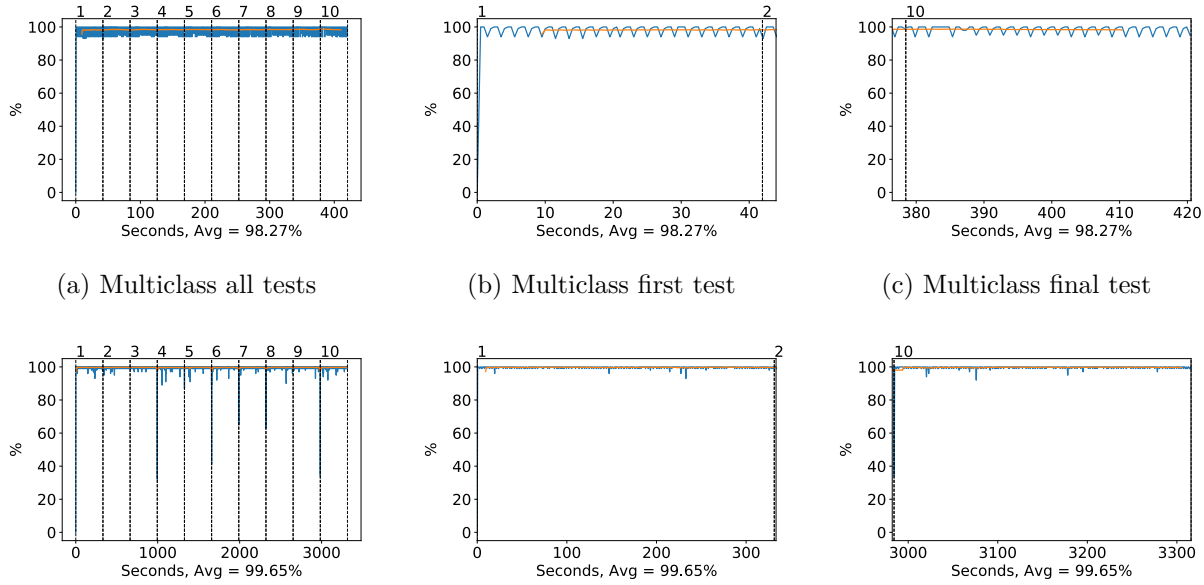


(d) Binary all tests (e) Binary first test (f) Binary final test

Figure A.222: Mobilenet Memory usage during testing, with a moving window average over 40 measurements overlaid

A.4.10 NasNet Large

GPU Usage

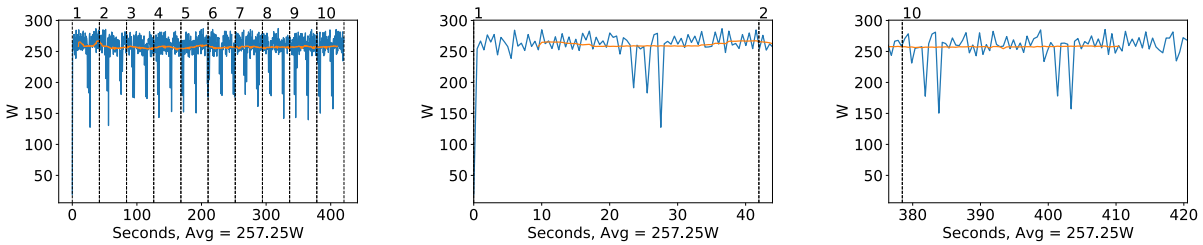


(a) Multiclass all tests (b) Multiclass first test (c) Multiclass final test

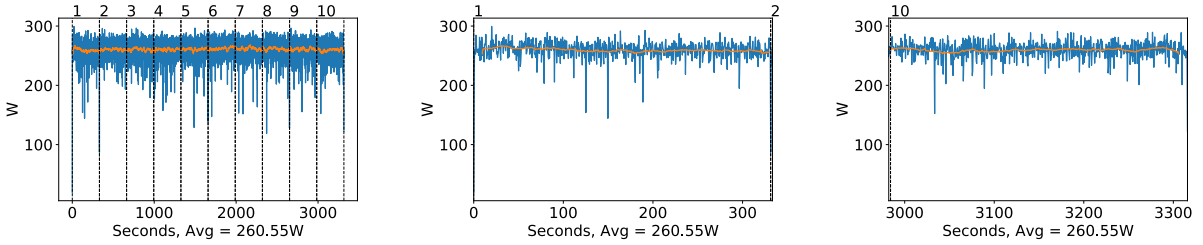
(d) Binary all tests (e) Binary first test (f) Binary final test

Figure A.223: NASNet Large GPU volatile usage during testing, with a moving window average over 40 measurements overlaid

CPU and memory usage

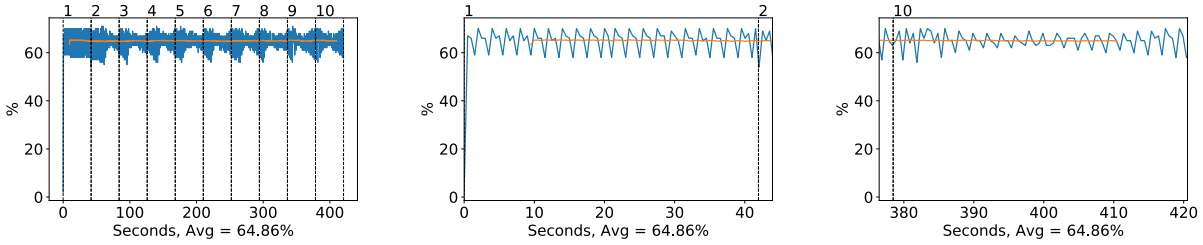


(a) Multiclass all tests (b) Multiclass first test (c) Multiclass final test

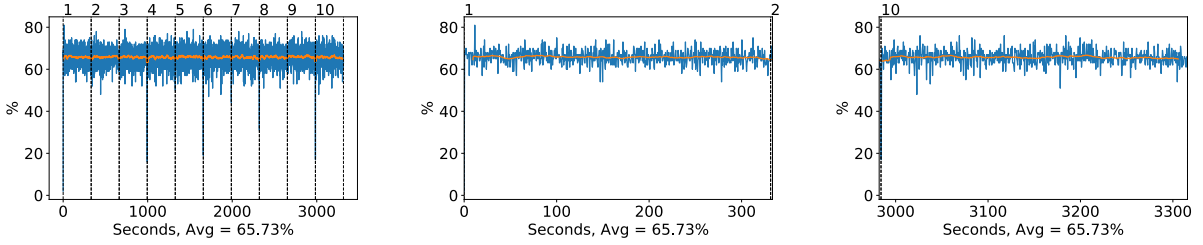


(d) Binary all tests (e) Binary first test (f) Binary final test

Figure A.224: NASNet Large GPU power usage i W during testing, with a moving window average over 40 measurements overlaid

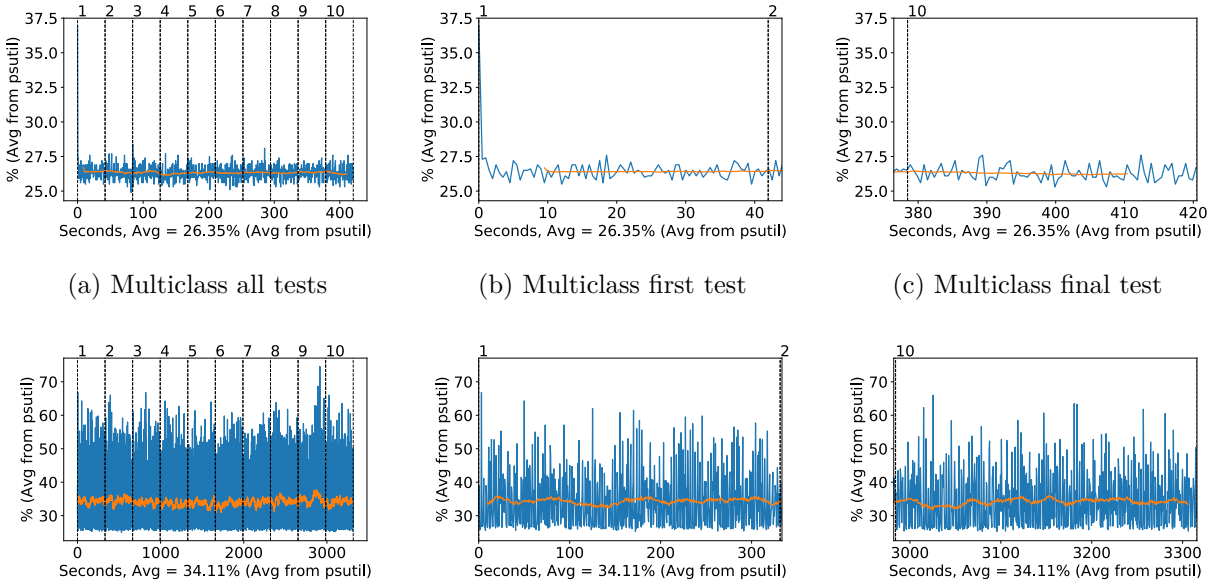


(a) Multiclass all tests (b) Multiclass first test (c) Multiclass final test



(d) Binary all tests (e) Binary first test (f) Binary final test

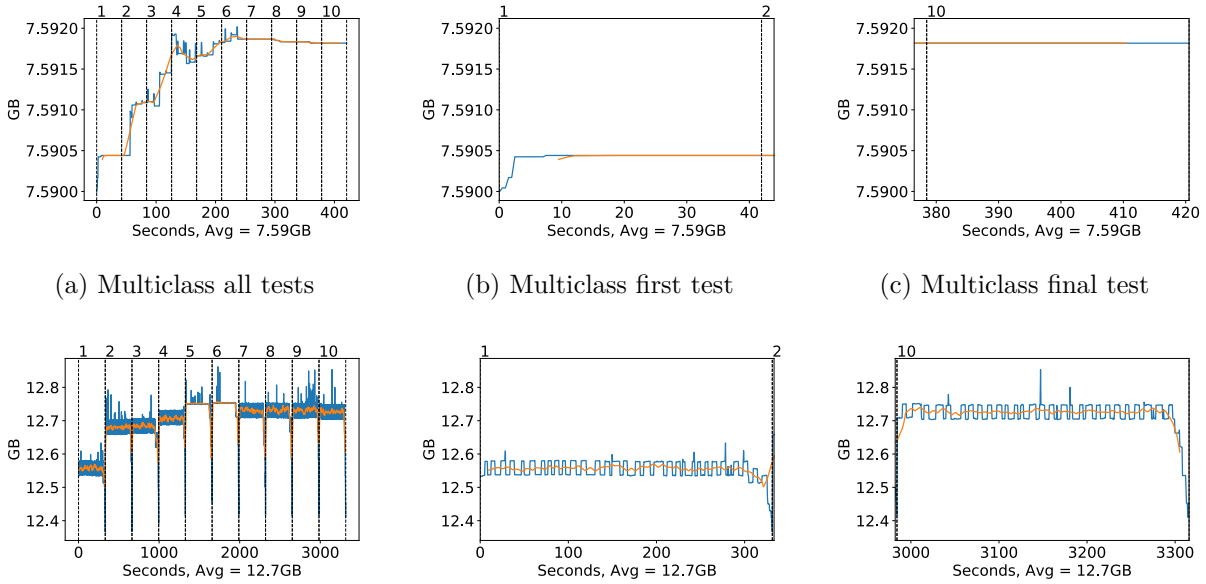
Figure A.225: NASNet Large GPU memory usage during testing, with a moving window average over 40 measurements overlaid



(a) Multiclass all tests (b) Multiclass first test (c) Multiclass final test

(d) Binary all tests (e) Binary first test (f) Binary final test

Figure A.226: NASNet Large CPU usage (averaged over 4 cores) during testing, with a moving window average over 40 measurements overlaid



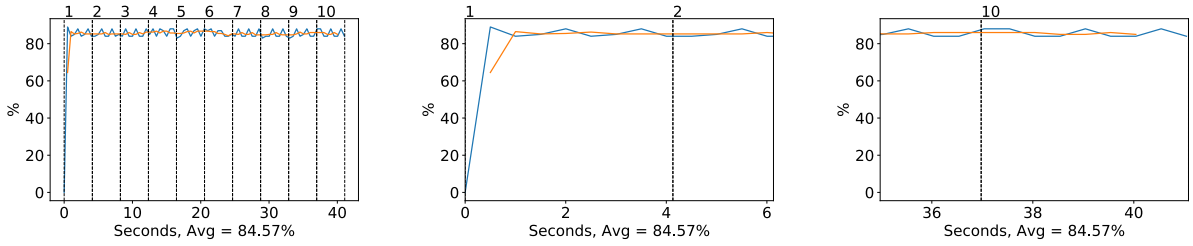
(a) Multiclass all tests (b) Multiclass first test (c) Multiclass final test

(d) Binary all tests (e) Binary first test (f) Binary final test

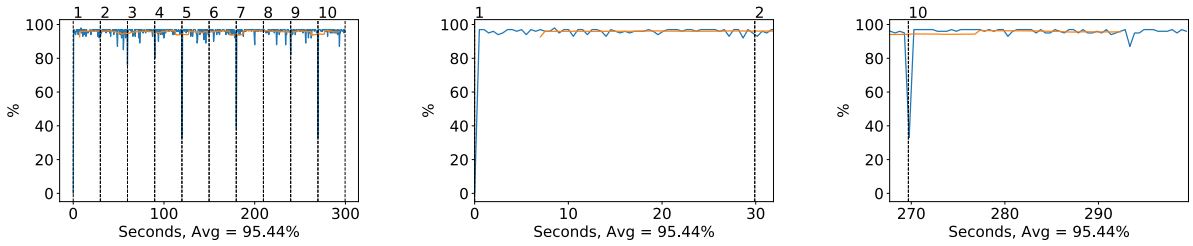
Figure A.227: NASNet Large Memory usage during testing, with a moving window average over 40 measurements overlaid

A.4.11 NasNet Mobile

GPU Usage



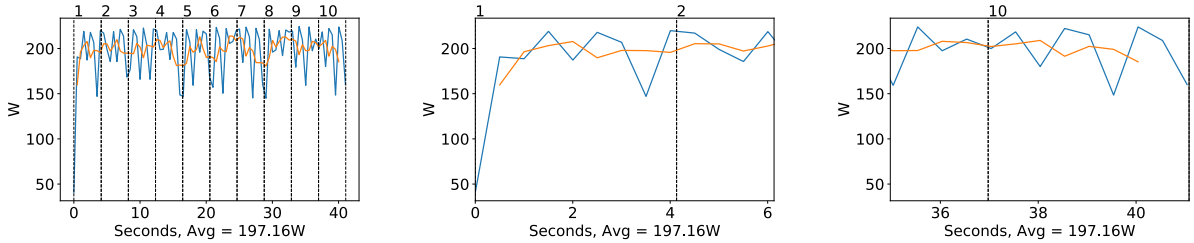
(a) Multiclass all tests (b) Multiclass first test (c) Multiclass final test



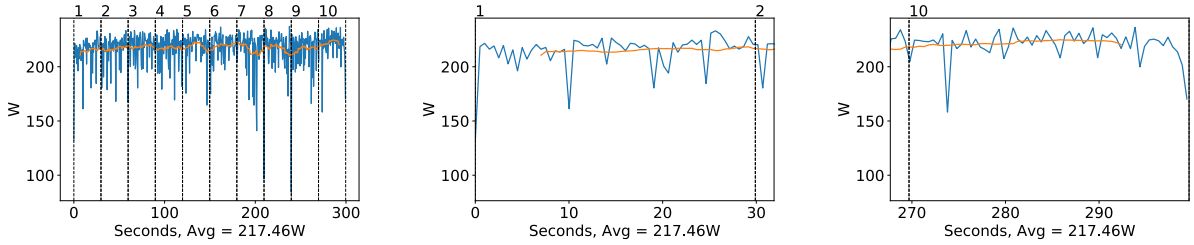
(d) Binary all tests (e) Binary first test (f) Binary final test

Figure A.228: NASNet Mobile GPU volatile usage during testing, with a moving window average over 40 measurements overlaid

CPU and memory usage

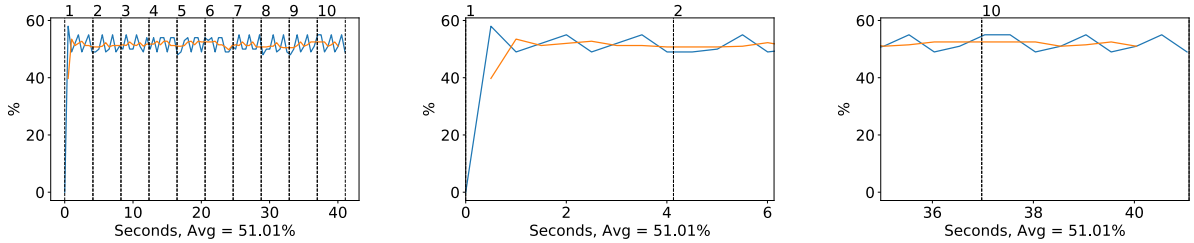


(a) Multiclass all tests (b) Multiclass first test (c) Multiclass final test

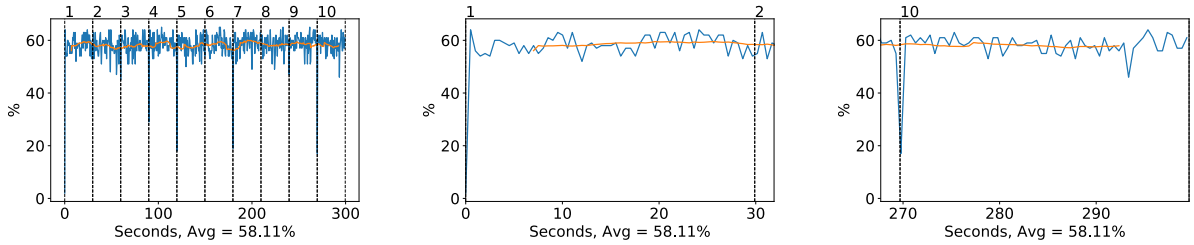


(d) Binary all tests (e) Binary first test (f) Binary final test

Figure A.229: NASNet Mobile GPU power usage i W during testing, with a moving window average over 40 measurements overlaid

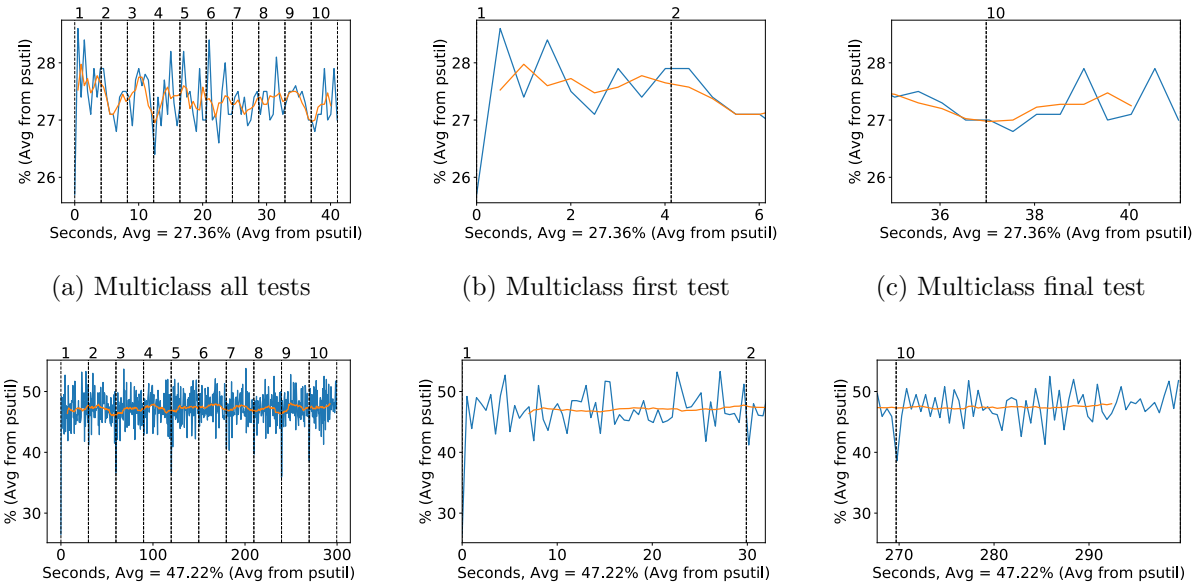


(a) Multiclass all tests (b) Multiclass first test (c) Multiclass final test



(d) Binary all tests (e) Binary first test (f) Binary final test

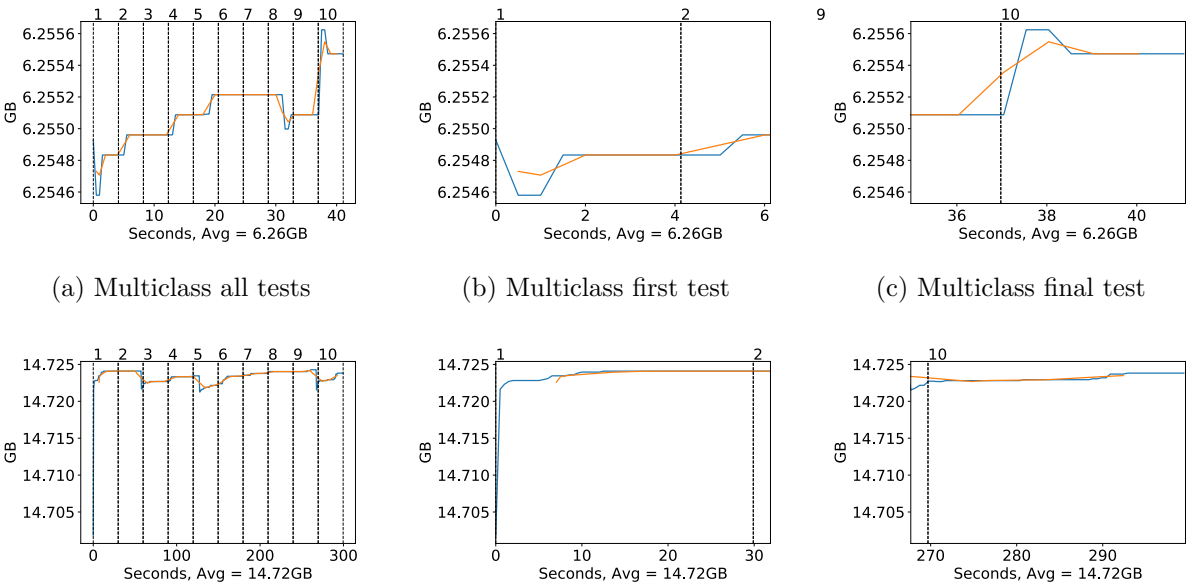
Figure A.230: NASNet Mobile GPU memory usage during testing, with a moving window average over 40 measurements overlaid



(a) Multiclass all tests (b) Multiclass first test (c) Multiclass final test

(d) Binary all tests (e) Binary first test (f) Binary final test

Figure A.231: NASNet Mobile CPU usage (averaged over 4 cores) during testing, with a moving window average over 40 measurements overlaid



(a) Multiclass all tests (b) Multiclass first test (c) Multiclass final test

(d) Binary all tests (e) Binary first test (f) Binary final test

Figure A.232: NASNet Mobile Memory usage during testing, with a moving window average over 40 measurements overlaid

A.5 Throughput and Accuracy

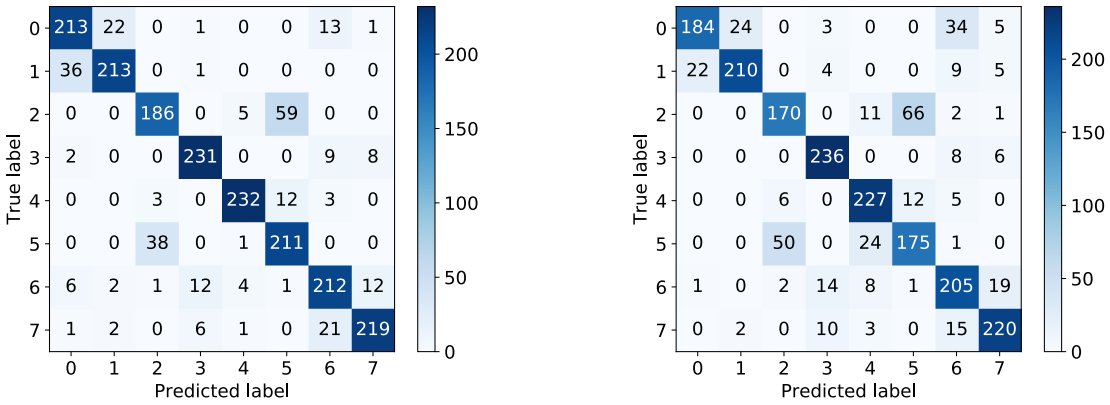
A.5.1 VGG16

Frame throughput

Network style	1	2	3	4	5	6	7	8	9	10	Average
Multiclass	315.76	345.07	344.95	344.71	344.65	344.00	344.17	344.47	344.23	344.23	341.62
Binary	44.55	44.57	44.55	44.37	44.46	44.46	44.53	44.47	44.45	44.45	44.49

Table A.1: FPS Test for Vgg16, including all 10 tests and average value.

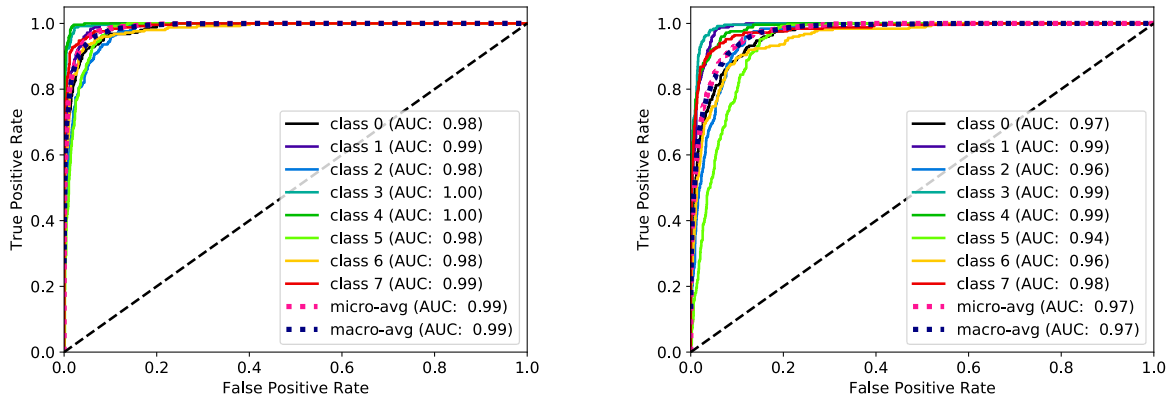
Classification Accuracy



(a) Multiclass confusion matrix

(b) Binary confusion matrix

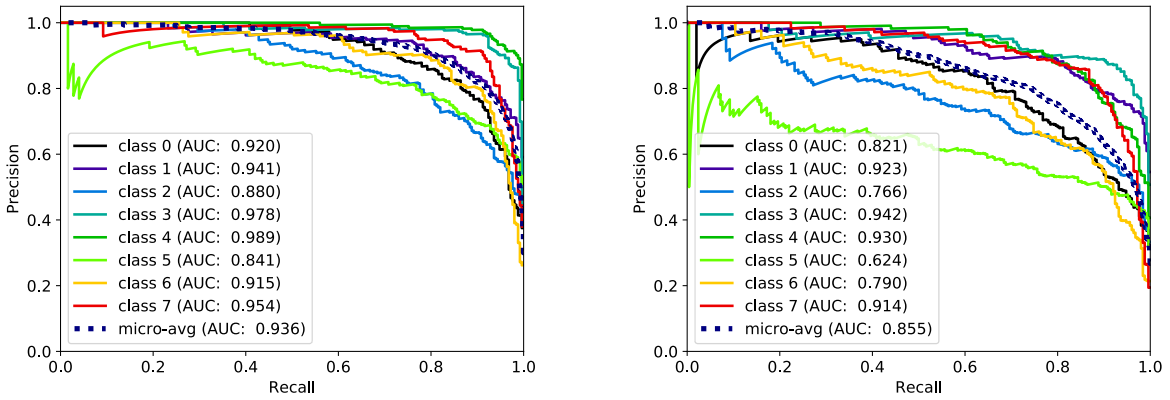
Figure A.233: Confusion matrices for VGG16, binary and multiclass, generated by Scikit-plots.



(a) Multiclass ROC curves

(b) Binary ROC curves

Figure A.234: ROC curves for VGG16, binary and multiclass, generated by Scikit-plots.



(a) Multiclass PR curves

(b) Binary PR curves

Figure A.235: PR curves for VGG16, binary and multiclass, generated by Scikit-plots.

Style, class	FN	FP	TN	TP	F_1	ACC	MCC	PREC	REC	SPEC
Multi, class 0	37	45	1705	213	0.84	0.96	0.82	0.83	0.85	0.97
Multi, class 1	37	26	1724	213	0.87	0.97	0.85	0.89	0.85	0.99
Multi, class 2	64	42	1708	186	0.78	0.95	0.75	0.82	0.74	0.98
Multi, class 3	19	20	1730	231	0.92	0.98	0.91	0.92	0.92	0.99
Multi, class 4	18	11	1739	232	0.94	0.99	0.93	0.95	0.93	0.99
Multi, class 5	39	72	1678	211	0.79	0.94	0.76	0.75	0.84	0.96
Multi, class 6	38	46	1704	212	0.83	0.96	0.81	0.82	0.85	0.97
Multi, class 7	31	21	1729	219	0.89	0.97	0.88	0.91	0.88	0.99
Average	35.38	35.38	1714.62	214.62	0.86	0.96	0.84	0.86	0.86	0.98
Binary, class 0	66	23	1727	184	0.81	0.96	0.78	0.89	0.74	0.99
Binary, class 1	40	26	1724	210	0.86	0.97	0.85	0.89	0.84	0.99
Binary, class 2	80	58	1692	170	0.71	0.93	0.67	0.75	0.68	0.97
Binary, class 3	14	31	1719	236	0.91	0.98	0.90	0.88	0.94	0.98
Binary, class 4	23	46	1704	227	0.87	0.97	0.85	0.83	0.91	0.97
Binary, class 5	75	79	1671	175	0.69	0.92	0.65	0.69	0.70	0.95
Binary, class 6	45	74	1676	205	0.78	0.94	0.74	0.73	0.82	0.96
Binary, class 7	30	36	1714	220	0.87	0.97	0.85	0.86	0.88	0.98
Average	46.62	46.62	1703.38	203.38	0.81	0.95	0.79	0.82	0.81	0.97
Average Diff.	+11.25	+11.25	-11.25	-11.25	-0.05	-0.01	-0.05	-0.05	-0.05	-0.01

Table A.2: Accuracy Test for Vgg16, including all metrics and average values.

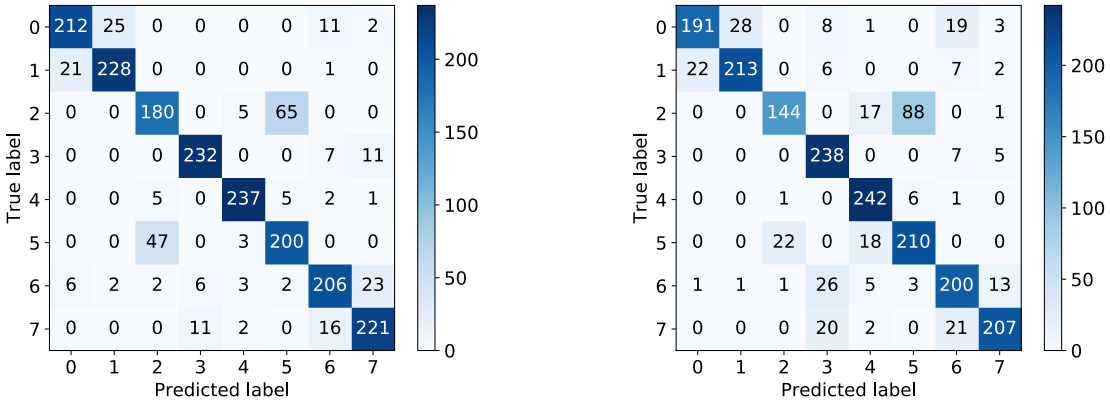
A.5.2 VGG19

Frame throughput

Network style	1	2	3	4	5	6	7	8	9	10	Average
Multiclass	299.63	300.12	299.90	299.18	299.49	299.27	299.31	299.40	299.54	299.04	299.49
Binary	38.53	38.54	38.53	38.54	38.44	38.55	38.53	38.54	38.55	38.47	38.52

Table A.3: FPS Test for Vgg19, including all 10 tests and average value.

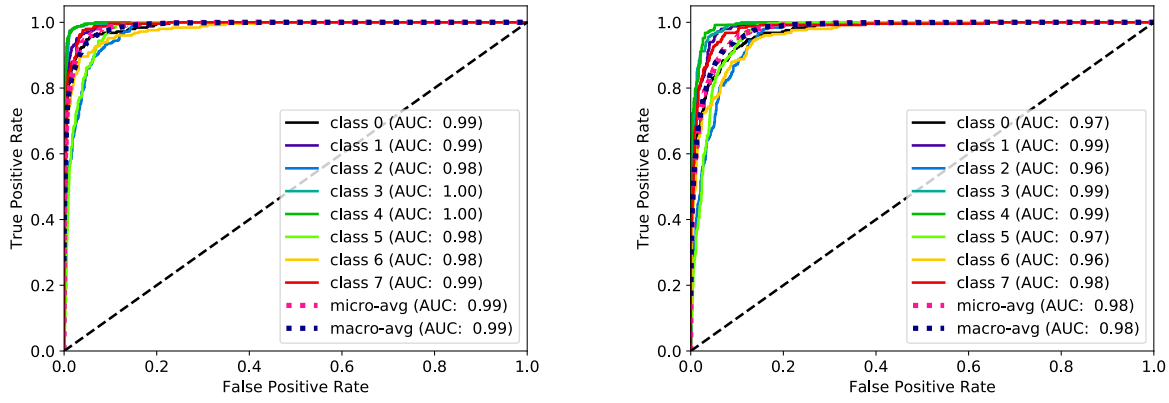
Classification Accuracy



(a) Multiclass confusion matrix

(b) Binary confusion matrix

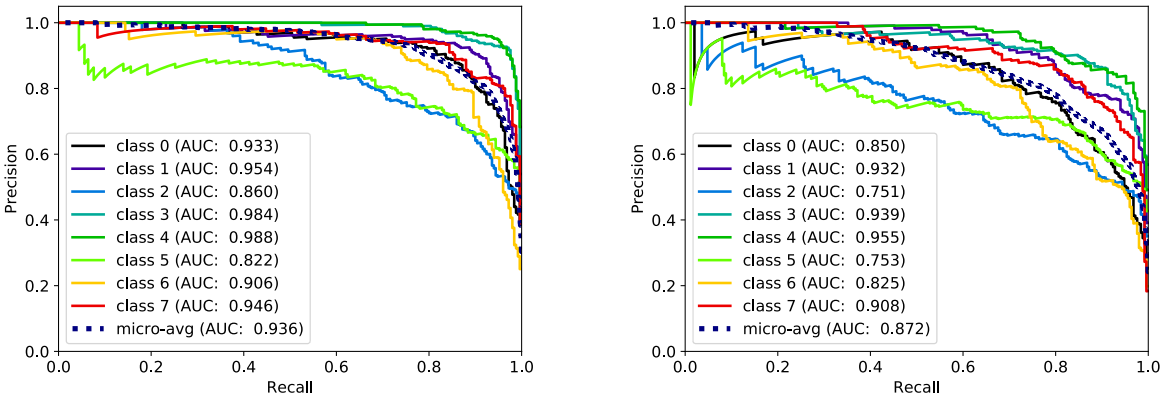
Figure A.236: Confusion matrices for VGG19, binary and multiclass, generated by Scikit-plots.



(a) Multiclass ROC curves

(b) Binary ROC curves

Figure A.237: ROC curves for VGG19, binary and multiclass, generated by Scikit-plots.



(a) Multiclass PR curves

(b) Binary PR curves

Figure A.238: PR curves for VGG19, binary and multiclass, generated by Scikit-plots.

Style, class	FN	FP	TN	TP	F_1	ACC	MCC	PREC	REC	SPEC
Multi, class 0	38	27	1723	212	0.87	0.97	0.85	0.89	0.85	0.98
Multi, class 1	22	27	1723	228	0.90	0.98	0.89	0.89	0.91	0.98
Multi, class 2	70	54	1696	180	0.74	0.94	0.71	0.77	0.72	0.97
Multi, class 3	18	17	1733	232	0.93	0.98	0.92	0.93	0.93	0.99
Multi, class 4	13	13	1737	237	0.95	0.99	0.94	0.95	0.95	0.99
Multi, class 5	50	72	1678	200	0.77	0.94	0.73	0.74	0.80	0.96
Multi, class 6	44	37	1713	206	0.84	0.96	0.81	0.85	0.82	0.98
Multi, class 7	29	37	1713	221	0.87	0.97	0.85	0.86	0.88	0.98
Average	35.50	35.50	1714.50	214.50	0.86	0.96	0.84	0.86	0.86	0.98
Binary, class 0	59	23	1727	191	0.82	0.96	0.80	0.89	0.76	0.99
Binary, class 1	37	29	1721	213	0.87	0.97	0.85	0.88	0.85	0.98
Binary, class 2	106	24	1726	144	0.69	0.94	0.67	0.86	0.58	0.99
Binary, class 3	12	60	1690	238	0.87	0.96	0.85	0.80	0.95	0.97
Binary, class 4	8	43	1707	242	0.90	0.97	0.89	0.85	0.97	0.98
Binary, class 5	40	97	1653	210	0.75	0.93	0.72	0.68	0.84	0.94
Binary, class 6	50	55	1695	200	0.79	0.95	0.76	0.78	0.80	0.97
Binary, class 7	43	24	1726	207	0.86	0.97	0.84	0.90	0.83	0.99
Average	44.38	44.38	1705.62	205.62	0.82	0.96	0.80	0.83	0.82	0.97
Average Diff.	+8.88	+8.88	-8.88	-8.88	-0.04	-0.01	-0.04	-0.03	-0.04	-0.01

Table A.4: Accuracy Test for Vgg19, including all metrics and average values.

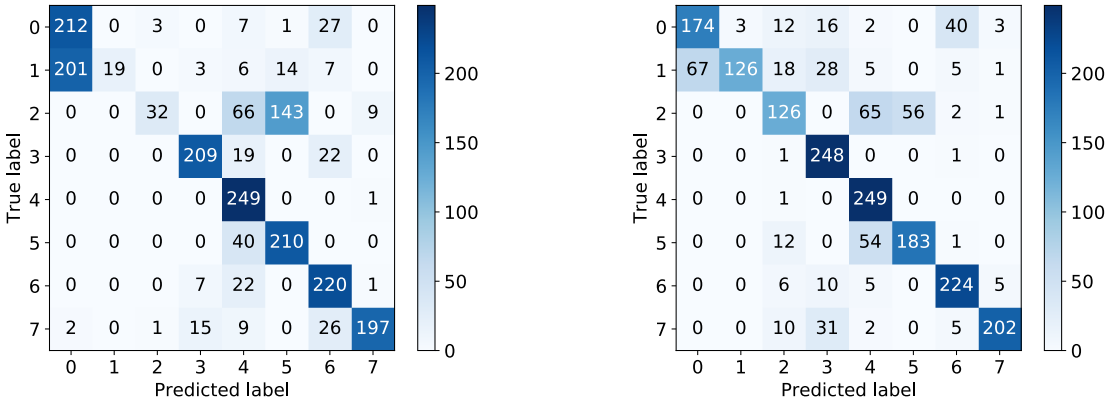
A.5.3 Inception v3

Frame throughput

Network style	1	2	3	4	5	6	7	8	9	10	Average
Multiclass	257.90	260.72	260.38	260.42	260.42	260.42	260.45	260.45	260.42	260.25	260.18
Binary	8.15	32.45	33.86	34.08	34.16	34.07	34.09	34.08	34.16	34.23	31.33

Table A.5: FPS Test for Inception-v3, including all 10 tests and average value.

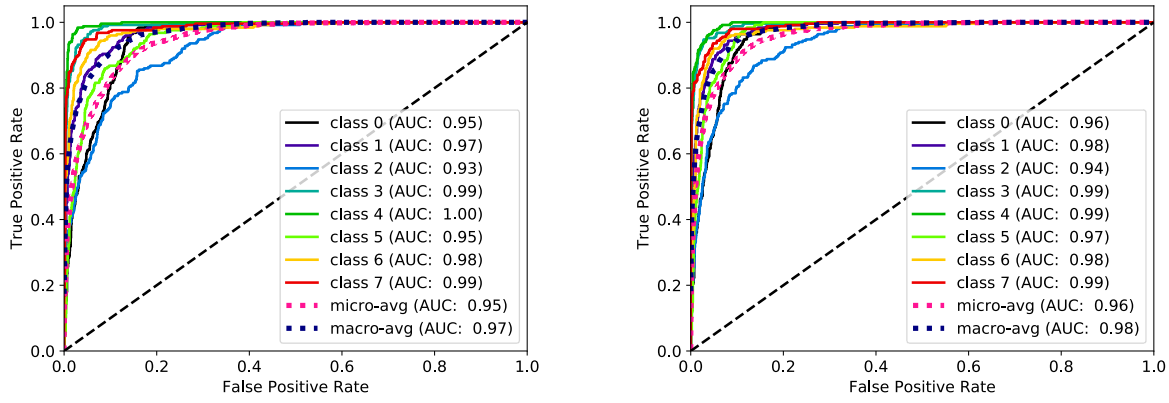
Classification Accuracy



(a) Multiclass confusion matrix

(b) Binary confusion matrix

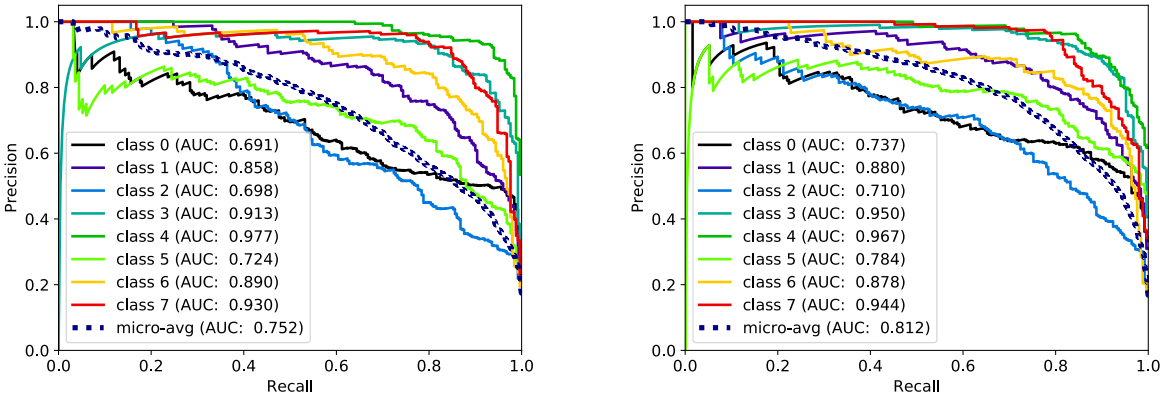
Figure A.239: Confusion matrices for Inception v3, binary and multiclass, generated by Scikit-plots.



(a) Multiclass ROC curves

(b) Binary ROC curves

Figure A.240: ROC curves for Inception v3, binary and multiclass, generated by Scikit-plots.



(a) Multiclass PR curves

(b) Binary PR curves

Figure A.241: PR curves for Inception v3, binary and multiclass, generated by Scikit-plots.

Style, class	FN	FP	TN	TP	F_1	ACC	MCC	PREC	REC	SPEC
Multi, class 0	38	203	1547	212	0.64	0.88	0.60	0.51	0.85	0.88
Multi, class 1	231	0	1750	19	0.14	0.88	0.26	1.00	0.08	1.00
Multi, class 2	218	4	1746	32	0.22	0.89	0.31	0.89	0.13	1.00
Multi, class 3	41	25	1725	209	0.86	0.97	0.85	0.89	0.84	0.99
Multi, class 4	1	169	1581	249	0.75	0.92	0.73	0.60	1.00	0.90
Multi, class 5	40	158	1592	210	0.68	0.90	0.64	0.57	0.84	0.91
Multi, class 6	30	82	1668	220	0.80	0.94	0.77	0.73	0.88	0.95
Multi, class 7	53	11	1739	197	0.86	0.97	0.85	0.95	0.79	0.99
Average	81.50	81.50	1668.50	168.50	0.62	0.92	0.63	0.77	0.67	0.95
Binary, class 0	76	67	1683	174	0.71	0.93	0.67	0.72	0.70	0.96
Binary, class 1	124	3	1747	126	0.66	0.94	0.68	0.98	0.50	1.00
Binary, class 2	124	60	1690	126	0.58	0.91	0.53	0.68	0.50	0.97
Binary, class 3	2	85	1665	248	0.85	0.96	0.84	0.74	0.99	0.95
Binary, class 4	1	133	1617	249	0.79	0.93	0.77	0.65	1.00	0.92
Binary, class 5	67	56	1694	183	0.75	0.94	0.71	0.77	0.73	0.97
Binary, class 6	26	54	1696	224	0.85	0.96	0.83	0.81	0.90	0.97
Binary, class 7	48	10	1740	202	0.87	0.97	0.86	0.95	0.81	0.99
Average	58.50	58.50	1691.50	191.50	0.76	0.94	0.74	0.79	0.77	0.97
Average Diff.	-23.00	-23.00	+23.00	+23.00	+0.14	+0.02	+0.11	+0.02	+0.09	+0.01

Table A.6: Accuracy Test for Inception-v3, including all metrics and average values.

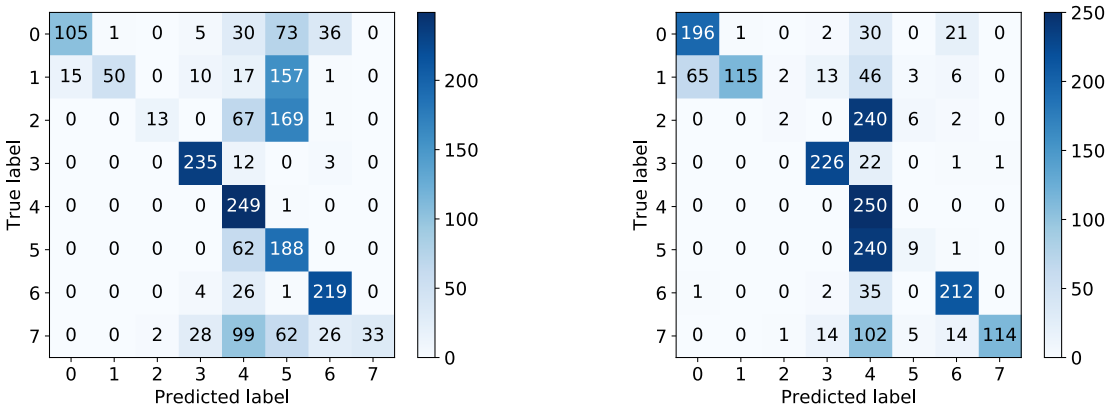
A.5.4 DenseNet 121

Frame throughput

Network style	1	2	3	4	5	6	7	8	9	10	Average
Multiclass	160.36	161.06	160.98	160.97	160.98	160.99	160.98	160.88	160.89	160.86	160.89
Binary	19.82	20.67	20.78	20.79	20.76	20.77	20.79	20.78	20.77	20.78	20.67

Table A.7: FPS Test for Densenet121, including all 10 tests and average value.

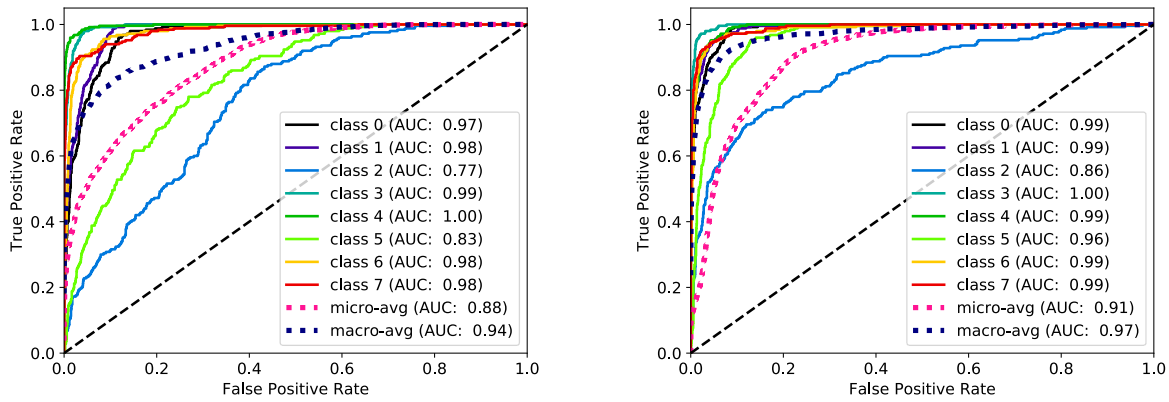
Classification Accuracy



(a) Multiclass confusion matrix

(b) Binary confusion matrix

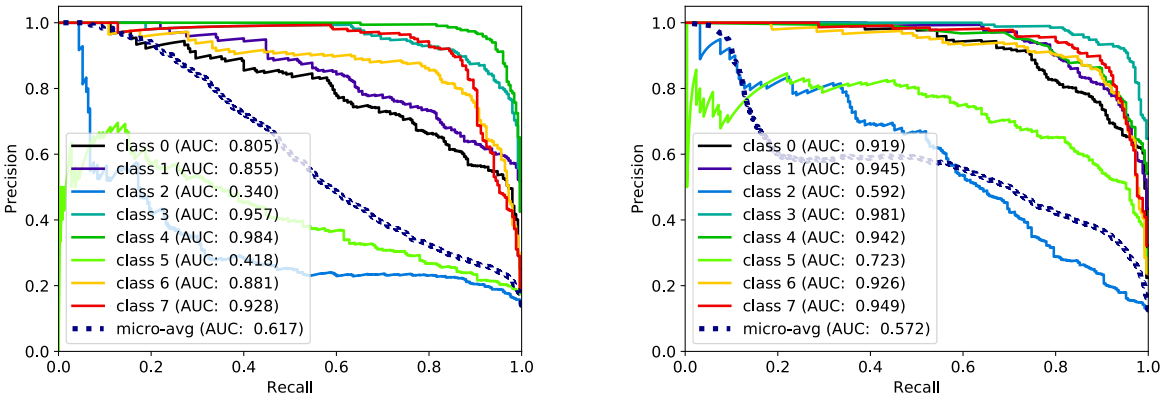
Figure A.242: Confusion matrices for DenseNet 121, binary and multiclass, generated by Scikit-plots.



(a) Multiclass ROC curves

(b) Binary ROC curves

Figure A.243: ROC curves for DenseNet 121, binary and multiclass, generated by Scikit-plots.



(a) Multiclass PR curves

(b) Binary PR curves

Figure A.244: PR curves for DenseNet 121, binary and multiclass, generated by Scikit-plots.

Style, class	FN	FP	TN	TP	F_1	ACC	MCC	PREC	REC	SPEC
Multi, class 0	145	15	1735	105	0.57	0.92	0.57	0.88	0.42	0.99
Multi, class 1	200	1	1749	50	0.33	0.90	0.42	0.98	0.20	1.00
Multi, class 2	237	2	1748	13	0.10	0.88	0.19	0.87	0.05	1.00
Multi, class 3	15	47	1703	235	0.88	0.97	0.87	0.83	0.94	0.97
Multi, class 4	1	313	1437	249	0.61	0.84	0.60	0.44	1.00	0.82
Multi, class 5	62	463	1287	188	0.42	0.74	0.34	0.29	0.75	0.74
Multi, class 6	31	67	1683	219	0.82	0.95	0.79	0.77	0.88	0.96
Multi, class 7	217	0	1750	33	0.23	0.89	0.34	1.00	0.13	1.00
Average	113.50	113.50	1636.50	136.50	0.50	0.89	0.52	0.76	0.55	0.94
Binary, class 0	54	66	1684	196	0.77	0.94	0.73	0.75	0.78	0.96
Binary, class 1	135	1	1749	115	0.63	0.93	0.65	0.99	0.46	1.00
Binary, class 2	248	3	1747	2	0.02	0.87	0.04	0.40	0.01	1.00
Binary, class 3	24	31	1719	226	0.89	0.97	0.88	0.88	0.90	0.98
Binary, class 4	0	715	1035	250	0.41	0.64	0.39	0.26	1.00	0.59
Binary, class 5	241	14	1736	9	0.07	0.87	0.09	0.39	0.04	0.99
Binary, class 6	38	45	1705	212	0.84	0.96	0.81	0.82	0.85	0.97
Binary, class 7	136	1	1749	114	0.62	0.93	0.65	0.99	0.46	1.00
Average	109.50	109.50	1640.50	140.50	0.53	0.89	0.53	0.69	0.56	0.94
Average Diff.	-4.00	-4.00	+4.00	+4.00	+0.03	0.00	+0.01	-0.07	+0.02	0.00

Table A.8: Accuracy Test for Densenet121, including all metrics and average values.

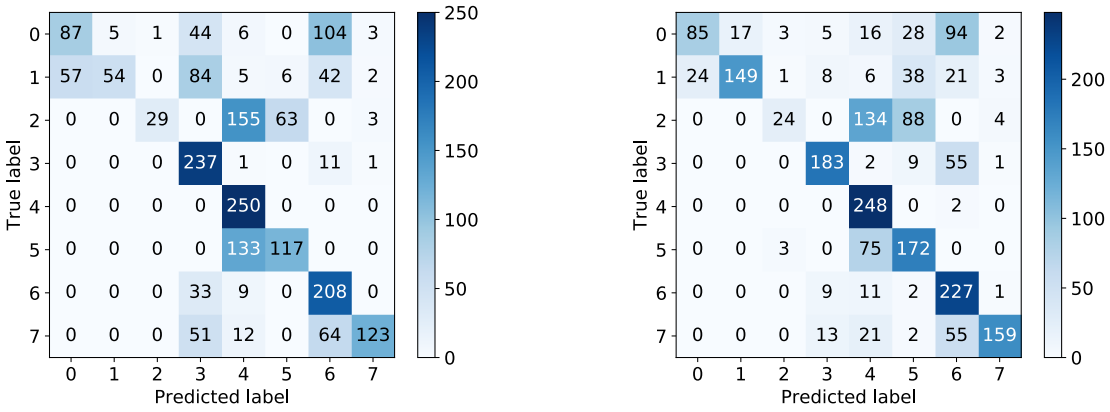
A.5.5 DenseNet 169

Frame throughput

Network style	1	2	3	4	5	6	7	8	9	10	Average
Multiclass	232.18	233.13	233.10	233.07	232.99	232.91	232.91	232.88	232.96	232.91	232.91
Binary	29.97	29.99	29.96	29.94	29.95	29.98	29.98	29.98	29.97	30.00	29.97

Table A.9: FPS Test for Densenet169, including all 10 tests and average value.

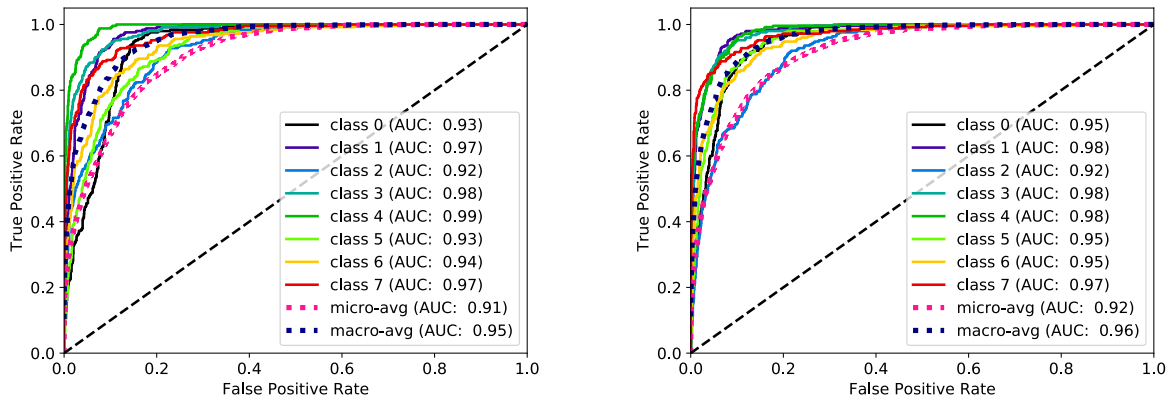
Classification Accuracy



(a) Multiclass confusion matrix

(b) Binary confusion matrix

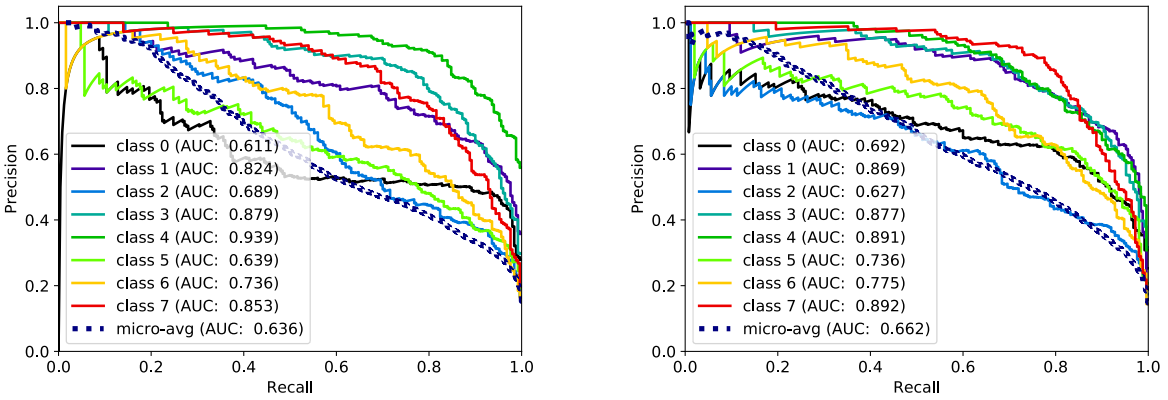
Figure A.245: Confusion matrices for DenseNet 169, binary and multiclass, generated by Scikit-plots.



(a) Multiclass ROC curves

(b) Binary ROC curves

Figure A.246: ROC curves for DenseNet 169, binary and multiclass, generated by Scikit-plots.



(a) Multiclass PR curves

(b) Binary PR curves

Figure A.247: PR curves for DenseNet 169, binary and multiclass, generated by Scikit-plots.

Style, class	FN	FP	TN	TP	F_1	ACC	MCC	PREC	REC	SPEC
Multi, class 0	163	57	1693	87	0.44	0.89	0.40	0.60	0.35	0.97
Multi, class 1	196	5	1745	54	0.35	0.90	0.42	0.92	0.22	1.00
Multi, class 2	221	1	1749	29	0.21	0.89	0.31	0.97	0.12	1.00
Multi, class 3	13	212	1538	237	0.68	0.89	0.66	0.53	0.95	0.88
Multi, class 4	0	321	1429	250	0.61	0.84	0.60	0.44	1.00	0.82
Multi, class 5	133	69	1681	117	0.54	0.90	0.49	0.63	0.47	0.96
Multi, class 6	42	221	1529	208	0.61	0.87	0.57	0.48	0.83	0.87
Multi, class 7	127	9	1741	123	0.64	0.93	0.65	0.93	0.49	0.99
Average	111.88	111.88	1638.12	138.12	0.51	0.89	0.51	0.69	0.55	0.94
Binary, class 0	165	24	1726	85	0.47	0.91	0.48	0.78	0.34	0.99
Binary, class 1	101	17	1733	149	0.72	0.94	0.70	0.90	0.60	0.99
Binary, class 2	226	7	1743	24	0.17	0.88	0.25	0.77	0.10	1.00
Binary, class 3	67	35	1715	183	0.78	0.95	0.76	0.84	0.73	0.98
Binary, class 4	2	265	1485	248	0.65	0.87	0.64	0.48	0.99	0.85
Binary, class 5	78	167	1583	172	0.58	0.88	0.52	0.51	0.69	0.90
Binary, class 6	23	227	1523	227	0.64	0.88	0.61	0.50	0.91	0.87
Binary, class 7	91	11	1739	159	0.76	0.95	0.75	0.94	0.64	0.99
Average	94.12	94.12	1655.88	155.88	0.60	0.91	0.59	0.71	0.62	0.95
Average Diff.	-17.75	-17.75	+17.75	+17.75	+0.09	+0.02	+0.08	+0.03	+0.07	+0.01

Table A.10: Accuracy Test for Densenet169, including all metrics and average values.

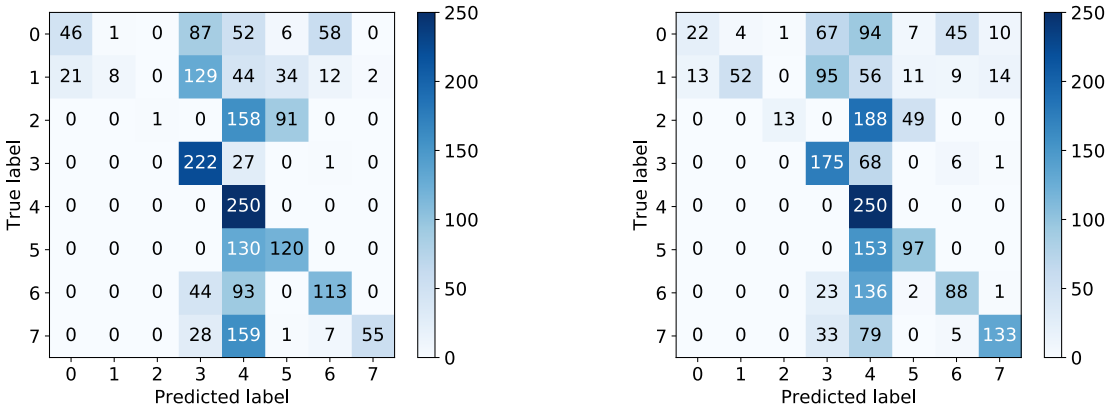
A.5.6 DenseNet 201

Frame throughput

Network style	1	2	3	4	5	6	7	8	9	10	Average
Multiclass	183.28	183.45	183.39	183.30	183.32	183.28	183.27	183.30	183.23	183.32	183.31
Binary	23.59	23.59	23.59	23.57	23.58	23.60	23.59	23.58	23.59	23.59	23.59

Table A.11: FPS Test for Densenet201, including all 10 tests and average value.

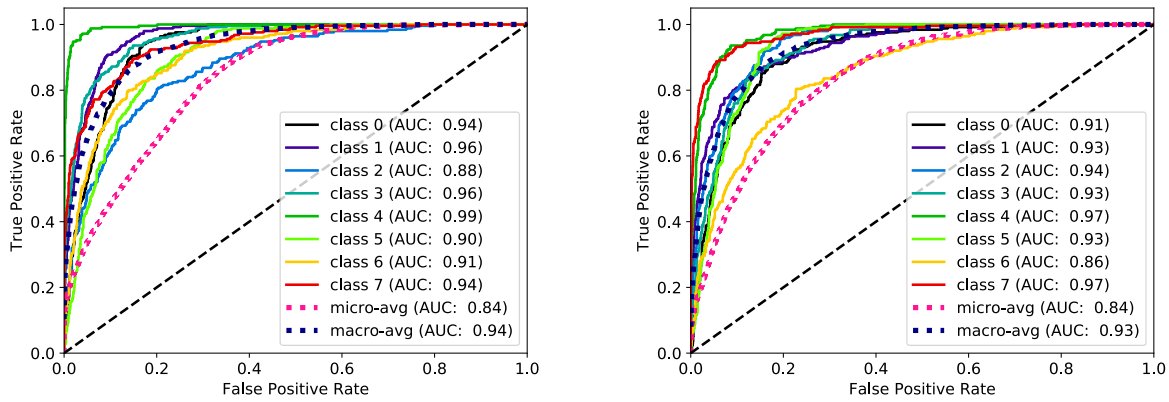
Classification Accuracy



(a) Multiclass confusion matrix

(b) Binary confusion matrix

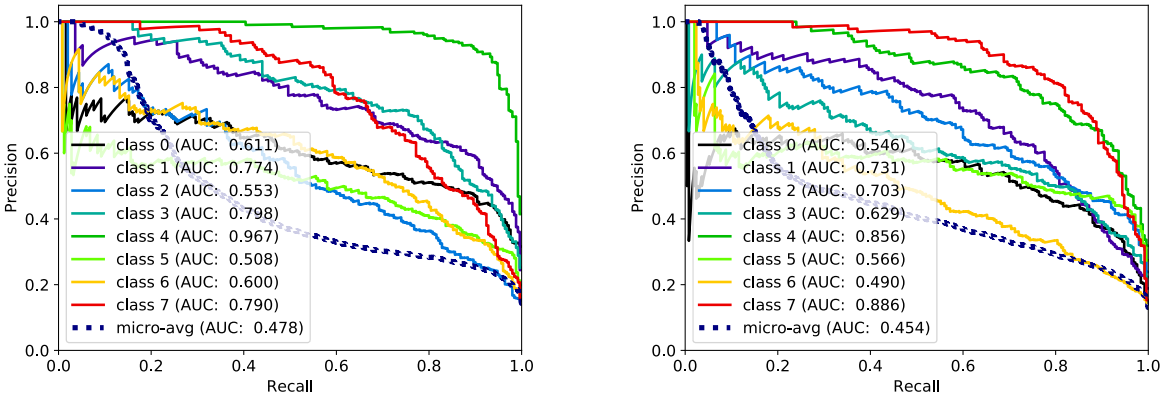
Figure A.248: Confusion matrices for DenseNet 201, binary and multiclass, generated by Scikit-plots.



(a) Multiclass ROC curves

(b) Binary ROC curves

Figure A.249: ROC curves for DenseNet 201, binary and multiclass, generated by Scikit-plots.



(a) Multiclass PR curves

(b) Binary PR curves

Figure A.250: PR curves for DenseNet 201, binary and multiclass, generated by Scikit-plots.

Style, class	FN	FP	TN	TP	F_1	ACC	MCC	PREC	REC	SPEC
Multi, class 0	204	21	1729	46	0.29	0.89	0.32	0.69	0.18	0.99
Multi, class 1	242	1	1749	8	0.06	0.88	0.16	0.89	0.03	1.00
Multi, class 2	249	0	1750	1	0.01	0.88	0.06	1.00	0.00	1.00
Multi, class 3	28	288	1462	222	0.58	0.84	0.55	0.44	0.89	0.84
Multi, class 4	0	663	1087	250	0.43	0.67	0.41	0.27	1.00	0.62
Multi, class 5	130	132	1618	120	0.48	0.87	0.40	0.48	0.48	0.92
Multi, class 6	137	78	1672	113	0.51	0.89	0.46	0.59	0.45	0.96
Multi, class 7	195	2	1748	55	0.36	0.90	0.43	0.96	0.22	1.00
Average	148.12	148.12	1601.88	101.88	0.34	0.85	0.35	0.66	0.41	0.92
Binary, class 0	228	13	1737	22	0.15	0.88	0.20	0.63	0.09	0.99
Binary, class 1	198	4	1746	52	0.34	0.90	0.41	0.93	0.21	1.00
Binary, class 2	237	1	1749	13	0.10	0.88	0.20	0.93	0.05	1.00
Binary, class 3	75	218	1532	175	0.54	0.85	0.48	0.45	0.70	0.88
Binary, class 4	0	774	976	250	0.39	0.61	0.37	0.24	1.00	0.56
Binary, class 5	153	69	1681	97	0.47	0.89	0.42	0.58	0.39	0.96
Binary, class 6	162	65	1685	88	0.44	0.89	0.39	0.58	0.35	0.96
Binary, class 7	117	26	1724	133	0.65	0.93	0.63	0.84	0.53	0.99
Average	146.25	146.25	1603.75	103.75	0.39	0.85	0.39	0.65	0.41	0.92
Average Diff.	-1.88	-1.88	+1.88	+1.88	+0.05	0.00	+0.04	-0.02	+0.01	0.00

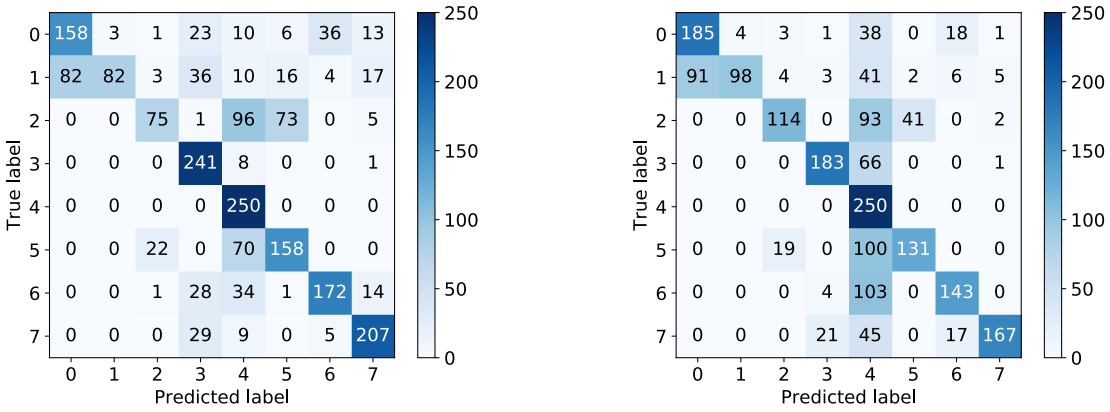
Table A.12: Accuracy Test for Densenet201, including all metrics and average values.

A.5.7 Xception

Network style	1	2	3	4	5	6	7	8	9	10	Average
Multiclass	162.56	163.44	163.37	163.40	163.43	163.28	163.39	163.28	163.35	163.20	163.27
Binary	21.12	21.11	21.06	21.07	21.07	21.08	21.06	21.04	21.07	21.08	21.08

Table A.13: FPS Test for Xception, including all 10 tests and average value.

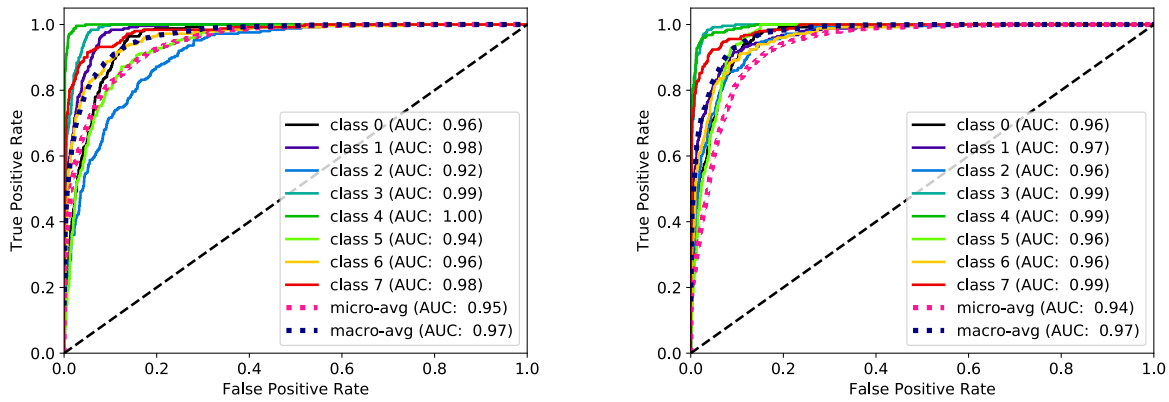
Frame throughput



(a) Multiclass confusion matrix

(b) Binary confusion matrix

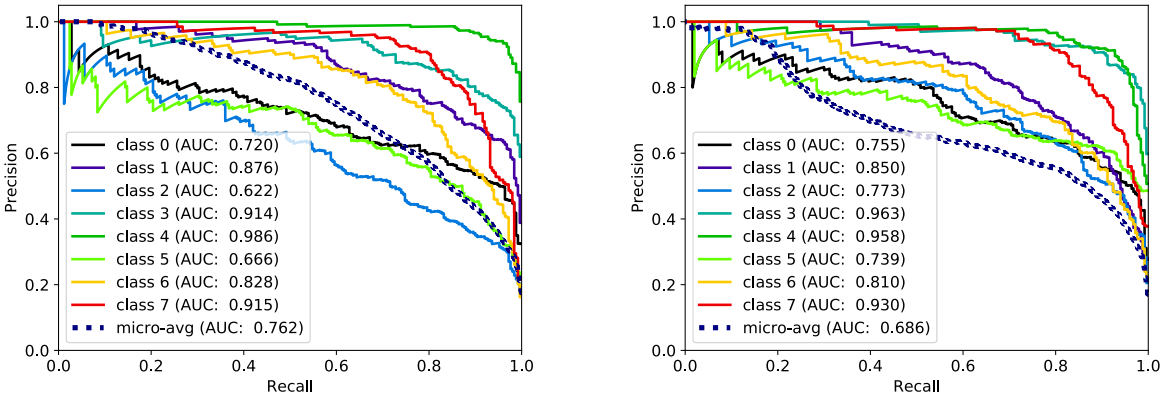
Figure A.251: Confusion matrices for Xception, binary and multiclass, generated by Scikit-plots.



(a) Multiclass ROC curves

(b) Binary ROC curves

Figure A.252: ROC curves for Xception, binary and multiclass, generated by Scikit-plots.



(a) Multiclass PR curves

(b) Binary PR curves

Figure A.253: PR curves for Xception, binary and multiclass, generated by Scikit-plots.

Style, class	FN	FP	TN	TP	F_1	ACC	MCC	PREC	REC	SPEC
Multi, class 0	92	82	1668	158	0.64	0.91	0.60	0.66	0.63	0.95
Multi, class 1	168	3	1747	82	0.49	0.91	0.53	0.96	0.33	1.00
Multi, class 2	175	27	1723	75	0.43	0.90	0.43	0.74	0.30	0.98
Multi, class 3	9	117	1633	241	0.79	0.94	0.77	0.67	0.96	0.93
Multi, class 4	0	237	1513	250	0.68	0.88	0.67	0.51	1.00	0.86
Multi, class 5	92	96	1654	158	0.63	0.91	0.57	0.62	0.63	0.95
Multi, class 6	78	45	1705	172	0.74	0.94	0.70	0.79	0.69	0.97
Multi, class 7	43	50	1700	207	0.82	0.95	0.79	0.81	0.83	0.97
Average	82.12	82.12	1667.88	167.88	0.65	0.92	0.63	0.72	0.67	0.95
Binary, class 0	65	91	1659	185	0.70	0.92	0.66	0.67	0.74	0.95
Binary, class 1	152	4	1746	98	0.56	0.92	0.59	0.96	0.39	1.00
Binary, class 2	136	26	1724	114	0.58	0.92	0.57	0.81	0.46	0.99
Binary, class 3	67	29	1721	183	0.79	0.95	0.77	0.86	0.73	0.98
Binary, class 4	0	486	1264	250	0.51	0.76	0.50	0.34	1.00	0.72
Binary, class 5	119	43	1707	131	0.62	0.92	0.59	0.75	0.52	0.98
Binary, class 6	107	41	1709	143	0.66	0.93	0.63	0.78	0.57	0.98
Binary, class 7	83	9	1741	167	0.78	0.95	0.77	0.95	0.67	0.99
Average	91.12	91.12	1658.88	158.88	0.65	0.91	0.63	0.77	0.64	0.95
Average Diff.	+9.00	+9.00	-9.00	-9.00	-0.00	-0.01	0.00	+0.05	-0.04	-0.01

Table A.14: Accuracy Test for Xception, including all metrics and average values.

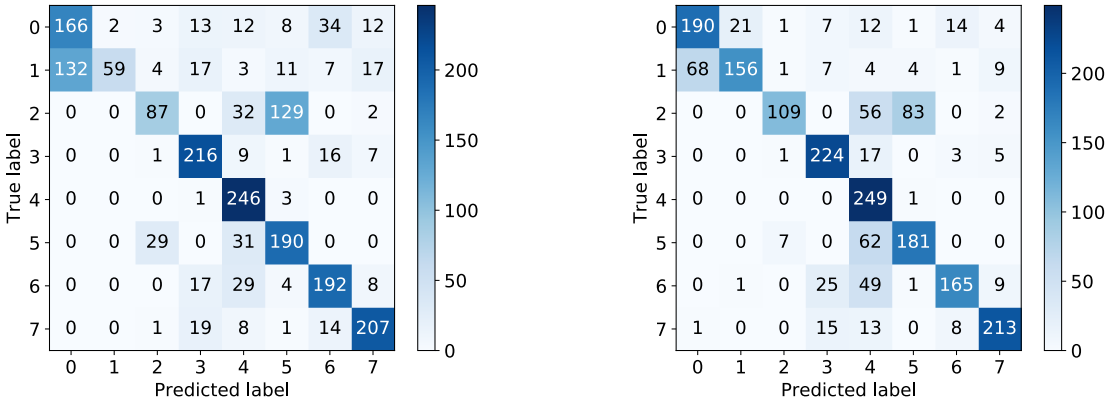
A.5.8 Inception ResNet v2

Frame throughput

Network style	1	2	3	4	5	6	7	8	9	10	Average
Multiclass	137.72	140.20	140.10	140.19	140.23	140.21	140.23	139.93	139.91	139.92	139.86
Binary	14.80	17.97	18.04	18.05	18.05	18.06	18.07	18.05	18.06	18.04	17.72

Table A.15: FPS Test for Inception-v2, including all 10 tests and average value.

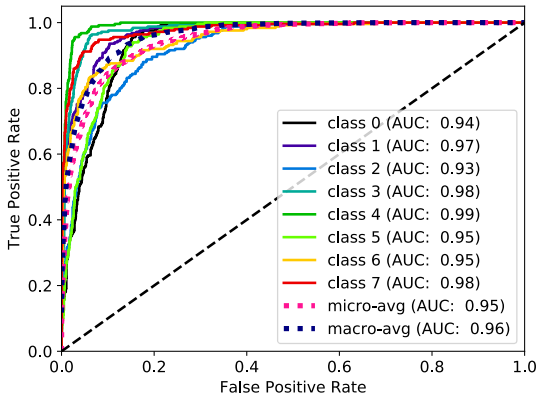
Classification Accuracy



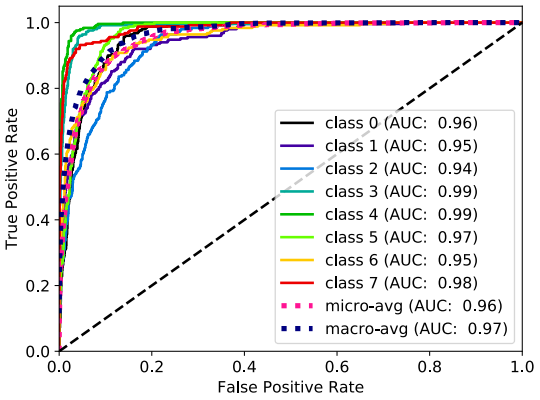
(a) Multiclass confusion matrix

(b) Binary confusion matrix

Figure A.254: Confusion matrices for Inception-ResNet-v2, binary and multiclass, generated by Scikit-plots.

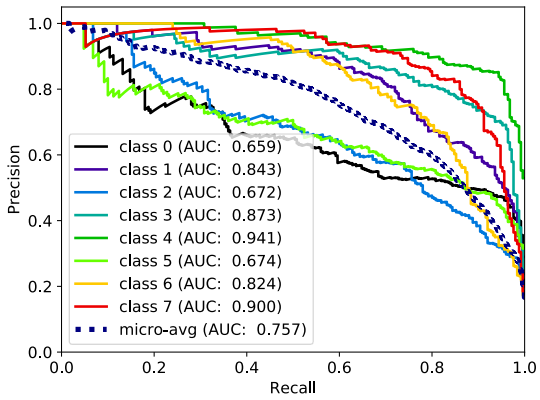


(a) Multiclass ROC curves

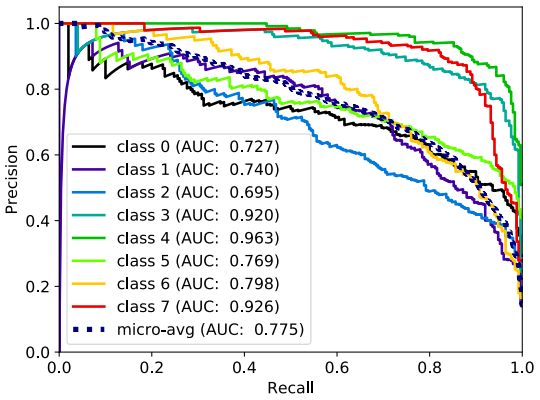


(b) Binary ROC curves

Figure A.255: ROC curves for Inception-ResNet-v2, binary and multiclass, generated by Scikit-plots.



(a) Multiclass PR curves



(b) Binary PR curves

Figure A.256: PR curves for Inception-ResNet-v2, binary and multiclass, generated by Scikit-plots.

Style, class	FN	FP	TN	TP	F_1	ACC	MCC	PREC	REC	SPEC
Multi, class 0	84	132	1618	166	0.61	0.89	0.55	0.56	0.66	0.92
Multi, class 1	191	2	1748	59	0.38	0.90	0.45	0.97	0.24	1.00
Multi, class 2	163	38	1712	87	0.46	0.90	0.45	0.70	0.35	0.98
Multi, class 3	34	67	1683	216	0.81	0.95	0.78	0.76	0.86	0.96
Multi, class 4	4	124	1626	246	0.79	0.94	0.78	0.66	0.98	0.93
Multi, class 5	60	157	1593	190	0.64	0.89	0.59	0.55	0.76	0.91
Multi, class 6	58	71	1679	192	0.75	0.94	0.71	0.73	0.77	0.96
Multi, class 7	43	46	1704	207	0.82	0.96	0.80	0.82	0.83	0.97
Average	79.62	79.62	1670.38	170.38	0.66	0.92	0.64	0.72	0.68	0.95
Binary, class 0	60	69	1681	190	0.75	0.94	0.71	0.73	0.76	0.96
Binary, class 1	94	22	1728	156	0.73	0.94	0.71	0.88	0.62	0.99
Binary, class 2	141	10	1740	109	0.59	0.92	0.60	0.92	0.44	0.99
Binary, class 3	26	54	1696	224	0.85	0.96	0.83	0.81	0.90	0.97
Binary, class 4	1	213	1537	249	0.70	0.89	0.69	0.54	1.00	0.88
Binary, class 5	69	90	1660	181	0.69	0.92	0.65	0.67	0.72	0.95
Binary, class 6	85	26	1724	165	0.75	0.94	0.73	0.86	0.66	0.99
Binary, class 7	37	29	1721	213	0.87	0.97	0.85	0.88	0.85	0.98
Average	64.12	64.12	1685.88	185.88	0.74	0.94	0.72	0.79	0.74	0.96
Average Diff.	-15.50	-15.50	+15.50	+15.50	+0.08	+0.02	+0.08	+0.07	+0.06	+0.01

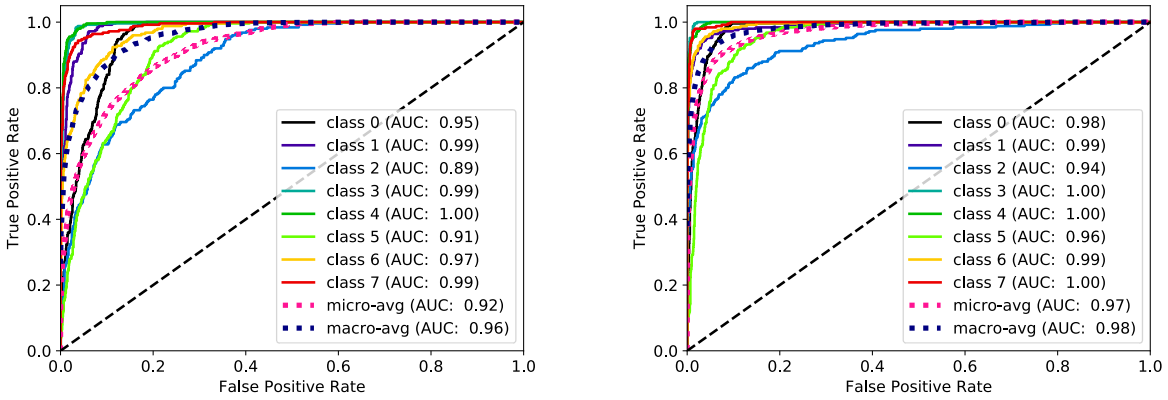
Table A.16: Accuracy Test for Inception-v2, including all metrics and average values.

A.5.9 Mobilenet

Frame throughput

Network style	1	2	3	4	5	6	7	8	9	10	Average
Multiclass	692.52	692.76	692.28	692.52	692.52	692.28	692.76	692.76	692.76	692.76	692.59
Binary	95.82	95.86	95.86	95.72	95.41	95.74	95.88	95.90	95.85	95.84	95.79

Table A.17: FPS Test for Mobilenet, including all 10 tests and average value.



(a) Multiclass ROC curves

(b) Binary ROC curves

Figure A.257: ROC curves for Mobilenet, binary and multiclass, generated by Scikit-plots.

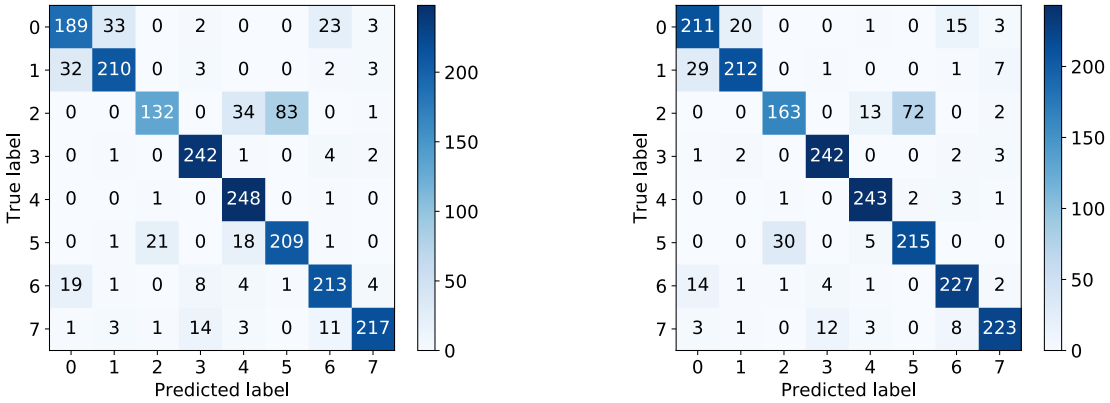
A.5.10 NasNet Large

Frame throughput

Network style	1	2	3	4	5	6	7	8	9	10	Average
Multiclass	47.68	47.60	47.54	47.54	47.54	47.54	47.53	47.54	47.54	47.54	47.56
Binary	6.04	6.03	6.03	6.03	6.03	6.03	6.03	6.03	6.03	6.03	6.03

Table A.18: FPS Test for Nasnetlarge, including all 10 tests and average value.

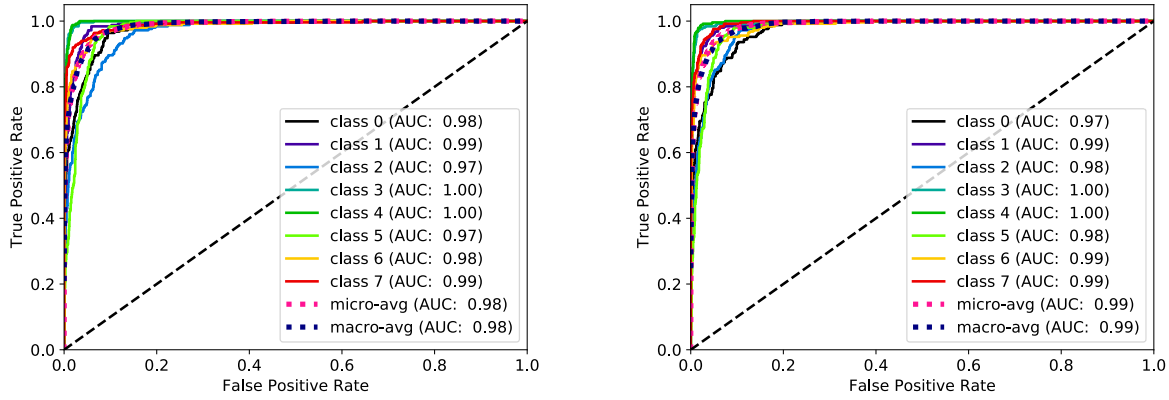
Classification Accuracy



(a) Multiclass confusion matrix

(b) Binary confusion matrix

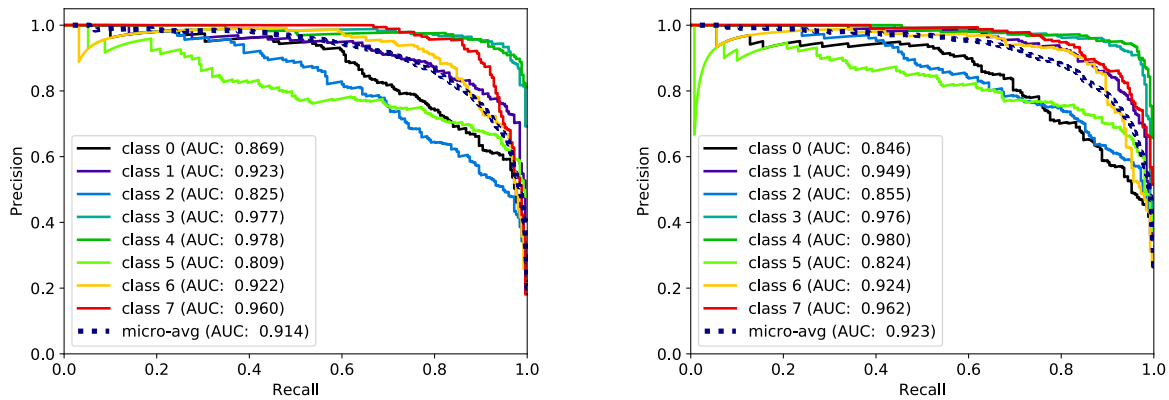
Figure A.258: Confusion matrices for NASNet Large, binary and multiclass, generated by Scikit-plots.



(a) Multiclass ROC curves

(b) Binary ROC curves

Figure A.259: ROC curves for NASNet Large, binary and multiclass, generated by Scikit-plots.



(a) Multiclass PR curves

(b) Binary PR curves

Figure A.260: PR curves for NASNet Large, binary and multiclass, generated by Scikit-plots.

Style, class	FN	FP	TN	TP	F_1	ACC	MCC	PREC	REC	SPEC
Multi, class 0	61	52	1698	189	0.77	0.94	0.74	0.78	0.76	0.97
Multi, class 1	40	39	1711	210	0.84	0.96	0.82	0.84	0.84	0.98
Multi, class 2	118	23	1727	132	0.65	0.93	0.64	0.85	0.53	0.99
Multi, class 3	8	27	1723	242	0.93	0.98	0.92	0.90	0.97	0.98
Multi, class 4	2	60	1690	248	0.89	0.97	0.88	0.81	0.99	0.97
Multi, class 5	41	84	1666	209	0.77	0.94	0.74	0.71	0.84	0.95
Multi, class 6	37	42	1708	213	0.84	0.96	0.82	0.84	0.85	0.98
Multi, class 7	33	13	1737	217	0.90	0.98	0.89	0.94	0.87	0.99
Average	42.50	42.50	1707.50	207.50	0.83	0.96	0.81	0.83	0.83	0.98
Binary, class 0	39	47	1703	211	0.83	0.96	0.81	0.82	0.84	0.97
Binary, class 1	38	24	1726	212	0.87	0.97	0.86	0.90	0.85	0.99
Binary, class 2	87	32	1718	163	0.73	0.94	0.71	0.84	0.65	0.98
Binary, class 3	8	17	1733	242	0.95	0.99	0.94	0.93	0.97	0.99
Binary, class 4	7	23	1727	243	0.94	0.98	0.93	0.91	0.97	0.99
Binary, class 5	35	74	1676	215	0.80	0.95	0.77	0.74	0.86	0.96
Binary, class 6	23	29	1721	227	0.90	0.97	0.88	0.89	0.91	0.98
Binary, class 7	27	18	1732	223	0.91	0.98	0.90	0.93	0.89	0.99
Average	33.00	33.00	1717.00	217.00	0.87	0.97	0.85	0.87	0.87	0.98
Average Diff.	-9.50	-9.50	+9.50	+9.50	+0.04	+0.01	+0.04	+0.03	+0.04	+0.01

Table A.19: Accuracy Test for Nasnetlarge, including all metrics and average values.

A.5.11 NasNet Mobile

Frame throughput

Network style	1	2	3	4	5	6	7	8	9	10	Average
Multiclass	484.14	487.33	487.09	487.21	486.97	487.21	486.97	487.09	487.21	487.09	486.83
Binary	66.91	66.87	66.69	66.66	66.80	66.86	66.75	66.56	66.61	66.82	66.75

Table A.20: FPS Test for Nasnetmobile, including all 10 tests and average value.