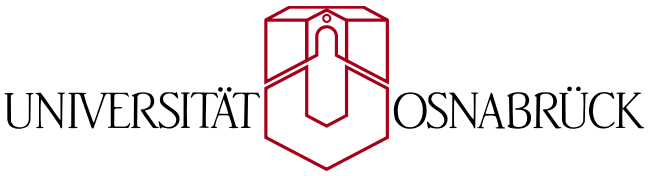


Department of Computer Science
Department of Cognitive Science

Rasmus Diederichsen

Live Introspection for Neural Network Training

Master's Thesis

First Supervisor: Prof. Dr. Oliver Vornberger
Second Supervisor: Dr. Ulf Krumnack
Project Supervisor: Anders Arpteg, PhD

ABSTRACT

Artificial neural networks have become the prevalent model class for many machine learning tasks, including image classification, segmentation, video and audio analysis, or time series prediction. With ever increasing computational resources and advances in programming infrastructure, the size of model we can train also increases. Nevertheless, it is not uncommon for training to take days or weeks, even on potent hardware. While there are many obvious causes—e.g. inherent difficulty to parallelize training with the most successful algorithms —there may still be inefficiencies due to our incomplete knowledge of training dynamics which is compensated for by expensive parameter search.

The pragmatic approach to opening the black box of deep learning is through experimentation and observation. Experiments are being performed to obtain insights into the training process, and said insights can then be used to guide the training. The speed and ease at which experiments can be performed is tied to the availability of tooling for the experimenter.

In this thesis, a software library is presented which facilitates both experimenting on metrics which can guide and ultimately accelerate the learning process and monitoring these metrics in practice. The library is then used to investigate a selection of metrics in order to find out if heretofore unknown signals can be extracted for informing parameter choices during training. Also, the library is used for validating known or claimed results, highlighting its usefulness for deep learning research.

ZUSAMMENFASSUNG

Künstliche neuronale Netze haben sich zum populärsten Modelltyp für Aufgaben des maschinellen Lernens entwickelt, beispielsweise Bildklassifikation, Segmentierung, Video- und Audioverständnis oder Zeitreihenvorhersage. Mit immer weiter wachsenden Rechenressourcen sowie Fortschritten in der verfügbaren Softwareinfrastruktur vergrößern sich die trainierbaren Modelle immer weiter. Nichtsdestoweniger ist es nicht unüblich, dass das Training eines Modells Tage oder Wochen in Anspruch nimmt, selbst auf hochleistungsfähiger Hardware. Es gibt offensichtliche Gründe—zum Beispiel die Schwierigkeit, das Training mit den erfolgreichsten Algorithmen zu parallelisieren—gibt es möglicherweise weitere Effizienzverluste durch fehlendes Verständins der Trainingsdynamiken, was durch teure Parametersuche umgangen wird.

Der pragmatische Ansatz, die Black Box des Deep Learning zu öffnen, ist das Experimentieren und Beobachten. Forscher führen Experimente durch, um Einsichten in den Trainingsprozess zu erhalten, welche dann verwendet werden können, um das Training zu leiten. Die Geschwindigkeit, mit der solche Experimente durchgeführt werden können, hängt auch von den verfügbaren Werkzeugen ab.

Diese Arbeit stellt eine Softwarebibliothek vor, die das Experimentieren mit Online-Metriken vereinfacht, welche letztendlich das Training beschleunigen könnten. Jene Bibliothek kann dann auch verwendet werden, gefundene Metriken live für beliebige Modelle zu überwachen. Die Software wird weiterhin verwendet, um solcherlei bisher unerforschte Metriken für eine Auswahl an Problemen zu erarbeiten. Zudem werden unter Zuhilfenahme der Software bekannte Resultate und Behauptungen validiert, um die Nützlichkeit für Unterschungen im Deep Learning zu demonstrieren.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

LISTINGS

1 INTRODUCTION

While the rise in performance of deep learning models has been rapid, the theoretical justification for most game-changing ideas as well as the general principles of deep learning has not kept pace (for more information, see Arora (2018)). This means that for many successful techniques, we have no clear understanding why they work so well in practice.

This explanatory gap is also a reason why developing deep learning applications is considered more of an art than a science. In contrast to traditional programming, which builds on decades of research and development in electrical engineering, logic, mathematics and theoretical computer science, there is rarely one definitive way to solve a certain problem in deep learning. Additionally, the quality of debugging tools available to a traditional programmer on every level of abstraction far exceeds what we currently have for differentiable programming. Simple questions like "Does my model learn what I want it to learn?" are not answerable at this point. We can thus identify a need to supply more useful tooling for deep learning researchers and practitioners. A standard approach to choosing a parameterisation remains trial-and-error, or only somewhat more sophisticated ways to run and test. The computational cost of training large models prohibits quick experimentation and often translates into monetary costs as well. Identifying dead ends early or points in training at which to tweak certain parameters could thus provide large savings in time and money, besides enabling a more thorough understanding of what is going on inside the pile of linear algebra that is a deep learning model.

This thesis addresses the above in two ways: A software library aiding in deep learning debugging and experimentation is designed, implemented and documented. The same library is then used to investigate experimentally whether heretofore unexplored metrics can be devised for optimising parameters of the training process while it is running.

This thesis is concerned with *deep* neural networks, meaning architectures consisting of many layers. Typically, the number of units in such a model and thus the number of tunable parameters ($n_{weights\_per\_neuron} \times n_{neurons}$) exceeds the number of training examples. Training such large networks thus involves iterating over the dataset many times which incurs a high computational cost due to the massive number of matrix operations involved. Speeding up the training is thus one of the primary endeavours in deep learning research. An overview of the workings of a neural network is given in the next (section 1.1).

1.1 ARTIFICIAL NEURAL NETWORKS

For the reader's benefit, a brief introduction to artificial neural networks is presented here, which may safely be skipped or replaced by other resources.

A Brief History of Neural Networks

The artificial neural network is a class of machine learning model which can be used for prediction tasks like regression and classification, as well as for generation and enhancement of data. At the core, neural models are simply conceptualised as an arrangement of units which receive numerical inputs, compute a weighted sum and thus produce an output activation. The ideas date back at least to Hebbian learning (a single neuron) in the 1940s, and were developed into the Perceptron model by Rosenblatt (1958). At that time, more easily trainable models like Support Vector machines eclipsed neural nets for most applications, but with the introduction of the backpropagation algorithm (Werbos, 1975) and increasing availability of computing power, this began to change. Architectural advances, such as the convolutional neural networks in the late 1980s ((LeCun et al., 1989b,a)) and the advent of GPU-accelerated neural network implementations (pioneered in Ciresan et al. (2011))—as well as subsequently, GPU-accelerated linear algebra libraries with automatic differentiation—finally made neural networks the workhorse for many AI applications today.

A Brief Primer on Neural Networks

A neural network consists of at least one input layer, $0 - n$ intermediate layers and at least one output layer. Data is processed in numerical form by multiplying it with the input layer's weights, mapping each product with the input layer's activation function and then propagating the resulting activations through subsequent layers in the same fashion. Figure 1.2 gives a graphical representation of a multi-layer feedforward network, the simplest form of neural network. Mathematically, such a model corresponds to a set of weight matrices $\mathcal{W} = \{W_i \mid W \in \mathbf{R}^{n_{i-1} \times n_i}\}$ where n_i is the number of units in layer i. Each layer can have an associated bias vector that is added to the output. The computation performed by layer i on the previous layer's output is thus

$$f_i(\mathbf{x}) = W_i\mathbf{x} + b_i \tag{1.1}$$

If only matrix multiplication and vector addition was used, a multi-layer network would be no more powerful than a single-layer one as the composition of many affine transformations can be expressed by a single one. Therefore, a non-linear *activation function* is applied component-wise to each layer's output. The nonlinearity is crucial to deeper networks' superior capabilities. Many activation functions have been proposed, for instance the hyperbolic tangent, the sigmoid or the rectified linear unit, the latter in many variations. Figure 1.1 shows plots for these functions. The choice activation function has been known impact performance significantly, and rectified linear units are most commonly used nowadays as they are computationally easy to evaluate and do not suffer from vanishing gradient problems.

Training a neural network simply involves a loss function measuring the distance of the network's output to the desired output and differentiating it with respect to the model parameters. The backpropagation algorithm provides an efficient way of computing all the partial derivatives of the loss with respect to each parameter (network weight). Parameters are usually updated with some form of the gradient

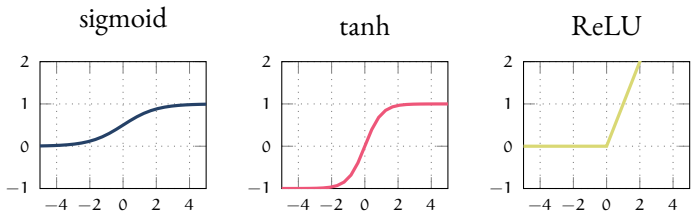


Figure 1.1: Shapes of different activation functions

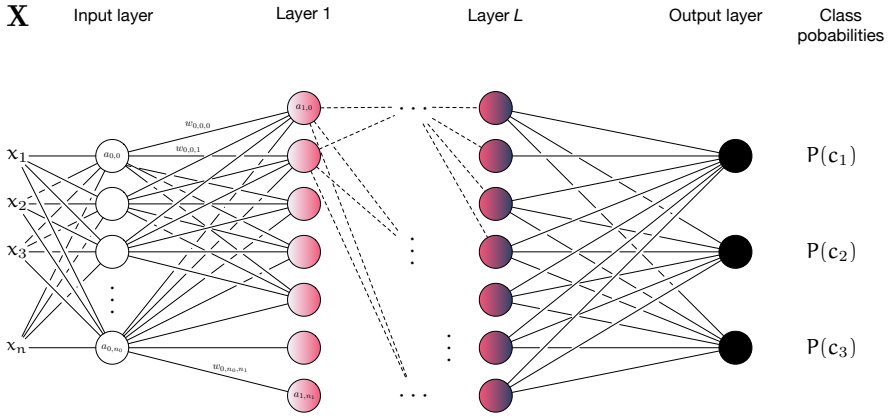


Figure 1.2: Schema of a multi-layer neural network. $\mathbf{x}_i$ are the input values, $a_{l_n,i}$ the i-th activation in layer l_n and $w_{l_n,i,j}$ the weight between the i-th unit in layer l_n and the j-th unit in layer l_{n+1}

descent optimisation algorithm. An introduction to the mathematics of gradient descent optimisation is given in the following section 1.2.

The schema shown in figure 1.2 is an example of a fully *linear* architecture as it only consists of matrix multiplication. The terms *linear*, *dense* and *fully-connected* are used interchangeably for matrix-multiplication layers. The introduction of convolutional layers and accompanying pooling operations made neural networks significantly more powerful while also reducing the computational power and making training larger models possible. The convolutional layer (figure 1.3) is inspired by the human visual system acting on natural images and thus has a two- or three-dimensional domain as opposed to the one-dimensional one for linear layers. It consists of one or more rectangular (usually square) filter kernels which slide over the input array and compute the dot product between the kernel and the data at each position. Each pixel in the output feature map is thus computed from a rectangular area of the input. The intuition is that learned filter matrices will correspond to visual features in the input, e.g. lines or corners and higher layers combine these feature detectors into more and more abstract features. The filter matrix can be a three-dimensional tensor, in which case it also sums over all input channels (for instance colour channels in natural images). The number of different filter kernels used determines the number channels in the layer's output.

Convolutional layers can be conceptualised as fully-connected ones where many weights are zero (all weights between an output pixel and all pixels outside of its rectangular input region) and all output units share their weights (since they all use the same filter kernel). This makes convolutional layers with large output sizes possible since the number of weights is merely the product of the kernel dimensions, not the product of input pixels and output pixels.

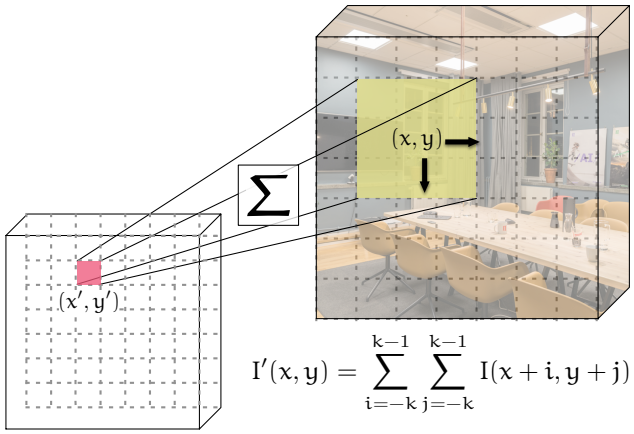


Figure 1.3: Illustration of a 3x3/1 convolution ($k = 3$). The layer depicted here operates on single-channel inputs and outputs single channels. Since only values at those positions (x, y) can be calculated for which the kernel overlaps the image completely, border pixels need special treatment like omission or padding.

In addition to linear and convolutional layers, modern architectures use the following basic building blocks:

- Spatial pooling operations for subsampling (taking maximum, minimum, or average over a region)
- Dropout layers which randomly zero out a subset of activations in the previous layer
- Batch normalisation which normalises layer outputs by dividing by the batch variance after subtracting the batch mean

Many variations and augmentations of these building blocks are subject of the literature, but have no bearing on the experiments performed for this work.

1.2 STOCHASTIC GRADIENT DESCENT

In this section, we briefly survey the stochastic gradient descent algorithm. This algorithm is used (in some variant) for virtually all practical deep learning models. The name derives from the fact that the model parameters are updated in the direction of the negative gradient, which is the high-dimensional partial derivative of the scalar loss function with respect to the model parameters. It is stochastic as it only uses a subset of the training data (called *batch*) at every time step and thus only approximates the true gradient which would have to be computed over the entire dataset to be learned.

In standard stochastic gradient descent, a loss J of some the model parameters θ (here, the layer weights) is computed over the training set of m examples by forming the expectation over the sample loss L:

$$J(\theta) = \mathbb{E}_{x,y\sim\hat{p}_{\text{data}}} L(\mathbf{x}, y, \theta) \tag{1.2}$$

$$= \frac{1}{m}\sum_{i=1}^{m} L(x^{(i)}, y^{(i)}, \theta) \tag{1.3}$$

The cumulative loss can then be derived for θ

$$\nabla_\theta J = \frac{1}{m}\sum_{i=1}^{m} \nabla_\theta L(x^{(i)}, y^{(i)}, \theta) \tag{1.4}$$

per the sum rule of differentiation. The simplest form of parameter update rule is then

$$\theta \leftarrow \theta - \eta\nabla_\theta J \tag{1.5}$$

with the learning rate η. There is no hard and fast rule on what this parameter should be, and it is subject of large swathes of literature. Popular modifications to the vanilla update rule are the use of momentum (Jacobs, 1988), per-layer learning rates (ibd.), reducing the rate throughout training, or adapting the learning rate based on mean and variance of gradients across past time steps (Kingma and Ba, 2014). Nevertheless, most of the time, training begins with an arbitrarily chosen small learning rate around 0.001 which is then adapted either by the aforementioned mechanisms or by search over the parameter space on a subset of the training data when computationally feasible.

1.3 GOALS OF THIS THESIS

The objective of this work is twofold:

1. Create a software library that enables easy and reusable implementation of training metrics and abstracts away the concrete model architecture
2. Perform experiments on common datasets to investigate whether common problems in neural network training can be detected on-line by the use of appropriate metrics. In other words, the library shall be used to verify or falsify that training dynamics can be analysed during training. Issues which could be investigated include, but are not limited to
    - inappropriate learning rate
    - layer/model saturation
    - bad initialisations
    - inappropriate network architecture
    - bad generalisation/overfitting
    - susceptibility to adversarial attacks

1.4 MOTIVATION

In contrast to classical machine learning models, training deep neural networks requires navigating a huge parameter space. While most non-neural regression or classification algorithms only require specification of a parameter set up-front and often no more than a few, some parameters can (and should) be varied over training time for neural networks. Looking at the popular scikit-learn library (Pedregosa et al., 2011), it can be seen that traditional methods such as SVMs, Gaussian Processes, Decision Trees or Gradient Boosting typically require less than 10 hyperparameters[1].

In neural networks the parameter space can have arbitrarily many dimensions when factoring in the fact that some parameters can change over time, such as

- learning rate (can vary according to some schedule)
- batch size[2]
- trainability of layers (not all layers need to be trained throughout the entire training)

Other parameters that need to be set initially are

- network architecture (how many layers, how many units per layer, what kind of layers)
- nonlinearity function for each layer
- loss function
- optimisation algorithm
- initial learning rate
- momentum of the weight updates
- weight decay
- regularisation methods for the weights

This makes finding an optimal training regimen very hard, particularly since training deep neural networks for realistic problems can take much longer than traditional methods, meaning cross-validating different models or different parameterisations can be prohibitively expensive. It is therefore desirable to detect dead ends early during training, or be able to tweak parameters in such a way as to maximise convergence speed.

This thesis work is motivated by the scarcity of useful tools to debug and monitor deep learning training. Arpteg et al. (2018) discuss several challenges arising in the context of engineering larger-scale machine learning software and note that a lack of useful debugging tools can lead to a lot of wasted time and money in diagnosing problematic behaviour of a neural network.

Without a lot of training and mathematical intuition and expertise, it is often very hard to figure out why a network is not learning or how to ensure timely

1 A look through scikit-learns selection of regressors and classifiers shows most classes require between 5 and 10 parameters.

2 It is not usual to change the batch size during training, but it can have an effect similar annealing the learning rate (see Smith et al. (2017))

convergence. And even with this expertise, visualisations or metrics need to be implemented over and over again because common tools do not abstract from the concrete model architecture. Providing easy-to-use tooling and live insights also has the side-effect of democratising access to machine learning software. Ideally, the metrics created with the help of this work would enable non-experts to better understand their models and reduce the dependence on rare machine learning expertise. While this goal will likely not be achieved by this work alone, steps in the general direction are needed for disseminating AI advances throughout the industry, and counteract the tendency of large companies to reap the majority of the benefits brought about by deep learning.

There exist some monitoring tools (see section 1.5), but they are mostly low-level tools which provide visualisation primitives (drawing and interacting with graphs). They may enable visualisation of certain network metrics on top of the primitives, but there is no native support for a concept such as *Maximum singular value of the weight matrix* which can be simply applied automatically to all layers.

In contrast, the library developed in this work is geared towards modularising introspection metrics in such a way that they are usable for any kind of model, without modifications to the model code. The secondary purpose of the library is the enabling quick iterations on hypothesised metrics extracted from the training in order to diagnose problems such as those outlined in section 1.3. Most deep learning research and its application to real-world problems involves experiments on a variety of architectures, datasets, and hyperparameter sets in order to validate an idea. All of these experiments must be implemented, which can easily lead to haphazard duplication of code for all the different settings, or creation of ad-hoc libraries that are only used in this specific work. Thus, a lot of effort is wasted due to the lack of more general tools and the fact that code produced for research often is not made public, or would require significant effort to generalise to other problems.

As such, the library shall not only be useful to end users who will make use of established metrics and thus save time in their model training, but also to researchers and the author of this thesis in evaluating hypotheses about training metrics.

In summary, this library is supposed to be both a developer tool, reducing implementation effort and decreasing opacity of the training process, and a research tool for abstracting away some of the nuisances of machine learning experimentation and simplifying experiments on new live metrics in neural networks.

1.5 EXISTING APPLICATIONS

There exist a variety of libraries for machine learning visualisation and online metric tracking which we will briefly survey in this section.

PyTorch Ignite

The project perhaps closest to `ikkuna`[3] in spirit is PyTorch's `ignite` subproject[4]. It defines abstractions for simplifying the setup of the training loop and for automatically computing metrics when events happen. It also encapsulates metrics into separate classes so they can be used on any model. `ignite` per default only includes

3 The name of the library developed for this thesis is the Finnish word for *window* as it permits insights into the training process from the outside.

4 https://pytorch.org/ignite/

events for beginning and end of iterations or epochs, but custom events can be registered by the user. `ikkuna`, on the other hand, will provide more fine-grained information on the level of individual layers, which is much more than `ignite` does. With respect to ergonomics, `ignite` is simpler to use due to the fact that metrics are registered by the help of function decorators on plain functions, while `ikkuna` requires implementation of a dedicated subscriber subclass. `ignite` also does not provide any visualisation capabilities out of the box.

We can thus summarise that `ikkuna` is focused more on collaboration across projects through it's plugin-based structure while `ignite` is a less powerful but simpler to use tool for removing boilerplate in one specific application, as it has no notion of reusing metrics outside of where they were defined.

TensorBoard

TensorBoard is a visualisation toolkit originally developed for the TensorFlow (Abadi et al., 2015) deep learning framework. It is composed of a Python library for exporting data from the training process and a web server which reads the serialised data and displays it in the browser. The server can be used independently from TensorFlow, provided the data is serialised in the appropriate format. This enables, e.g., a PyTorch port—termed TensorBoardX—to use TensorBoard from frameworks other than TensorFlow.

For exporting data during training, the developer adds operations to the computation graph which write scalars, histograms, audio, or other data asynchronously to disk. This data can then be displayed in approximately real-time in the web browser. Besides scalar-valued functions, which could be e.g. the loss curve or accuracy measure, TensorBoard supports histograms, audio, and embedding data natively. However, concrete instances of these classes of training artefact must be defined by the user and can only be reused if the developer creates a separate library for the computations involved. TensorBoard or TensorFlow also have no built-in way of intelligently discovering the model structure to automatically add visualisations of every layer with a higher-level API.

New kinds of visualisations can be added with plugins, which require not only writing the Python code exporting the data and for serving it from the web server, but also JavaScript for actually displaying it (the Polymer library is used for this[5]).

An attempt to abstract over the programming language for talking to the server is Crayon which so far supports Python and Lua.

In summary, TensorBoard is a possible backend for the library developed here, but operates at a lower level of abstraction.

Visdom

Visdom by Facebook Research fulfil is more or less the same purpose as TensorBoard, but supports NumPy and Lua Torch. In contrast to TensorBoard, Visdom includes more features for organising the display of many visualisations at once. Still, the framework is mostly geared towards improving workflows for data scientists, and is not concerned with providing useful metrics out-of-the-box.

5 https://www.polymer-project.org/

Others

There are other tools such as DeepVis for offline introspection by e.g. visualising learned features, which offer insights into the training after the fact, but do not help guiding the training process while it is running.

General-Purpose plotting libraries such as MatplotLib fall into the same category as TensorBoard—they offer primitives but there are no dedicated extensions to work with neural networks.

2 IKKUNA

Ikkuna is the Python library developed for this thesis. It targets Python 3.6 and was designed with the following goals in mind:

1. Ease of use through minimal configuration overhead
2. Flexible and all-encompassing API enabling creating arbitrary metrics which act on training artifacts
3. Metrics shall be agnostic of model code.
4. Plugin architecture so metrics written once can be used for any kind of model
5. Framework agnosticism. Ideally, the library would support every deep learning framework through an extensible abstraction layer.

What it provides over the aforementioned tools (section 1.5) is that it enables working at a higher level of abstraction, liberating the developer from having to repeat themselves, exchanging visualisations and metrics with others and reduces the friction between development and debugging.

This chapters gives a high-level overview of the library components and elaborates on the design decisions made during the creation. Throughout the chapters, UML class and package diagrams will serve as a mental map for the reader. For brevity, not all parts of the library are diagrammed down to the same level of detail.

2.1 DESIGN PRINCIPLES

Of the aforementioned goals, all except one have been accomplished. The objective of making the library agnostic to the deep learning framework being used (TensorFlow, PyTorch, PyCaffe, Chainer, etc.) has been neglected for practical reasons. Enabling this kind of support is beyond the scope of this thesis and only requires the implementation of a software layer which offers framework-agnostic access to network modules, activations, gradients and all the other necessary information. While this is certainly possible and useful, the PyTorch framework has been chosen for this work to create a proof of the concept. The choice is motivated in section 2.2.

The overarching architecture of this software must lend itself to this agnosticity goal, however. As such, a very loose coupling between model code, metric computation and visualisations is desired. Not only will this aid in extending the library to different deep learning frameworks, but it is also a prerequisite for allowing for modular, self-contained visualisations or metrics which can be installed and used separately and independently of specific model code. The Publisher-Subscriber design pattern has been chosen for these reasons (section 2.3) and the fact that it is conceptually simple and thus easy to implement and understand by other developers interested in the library.

2.2 DEEP LEARNING FRAMEWORKS

As neural networks are essentially sequences of matrix operations and elementwise function application followed by reduction operations, deep learning frameworks are not much different from previously available matrix libraries such as Eigen or NumPy. The value they provide lies in the fact that they embrace the concept of automatic differentiation and GPU acceleration. Unless specifically noted, all operations provided by these libraries are accompanied by their respective derivatives and a mechanism is provided to apply the chain rule to arbitrary sequences of operations in order to compute the derivative of their output with respect to the inputs.

All frameworks have in common that they build a graph representation of the model, whether implicitly or explicitly, and use it to parallelise propagations and factor dependencies into paths for computing derivatives in parallel. Nodes in the graph are operations while edges are data flowing between operations. This allows naturally parallelising independent computations. To compute gradients, the graph can be traversed backwards from the output node by applying the reverse-mode autodifferentiation algorithm (a generalisation of the plain backpropagation used in the multilayer perceptron). Define-and-run frameworks like TensorFlow create the graph explicitly; the user uses the API to do exactly this. The graph—once compiled—is fixed for the entire training process. PyTorch on the other hand implicitly records all operations as they execute and also overloads arithmetic operators for automatically constructing the graph. The graph is recreated for each propagation through the network and the user never directly interacts with it. This precludes some optimizations, but makes dynamically changing networks easily achievable and the API more natural to users.

The currently available deep learning libraries can be located on a spectrum between define-by-run and define-and-run. The first extreme would be frameworks such as PyTorch (Paszke et al., 2017) or Chainer (Tokui et al., 2015) , where there exist no two distinct execution phases – just as in an ordinary matrix library like NumPy, each statement immediately returns or operates on an actual value. By contrast, frameworks like TensorFlow[1] require specifying the model graph in a domain-specific language (TensorFlow has Python, Java and C++ APIs, Caffe uses Prototxt files), compile it to a different representation and the model is run and trained in a second phase. While this enables graph-based optimizations, the main downsides are that

- control flow cannot use the host language features, but must be done with the API used for defining models. Instead of

```
counter = torch.tensor(0)
# repeated matrix multiplication
while counter < N:
    counter += 1
    h = torch.matmul(W, h) + b
```

  one must use a construction like this

```
counter = tf.constant(0)
while_condition = lambda counter: tf.less(counter, N)
```

1 Since version 1.4, TensorFlow gravitates toward define-by-run through the introduction of *eager execution*, which becomes the default mode in version 2.0. Graph-based execution is still available, but not the default any longer.

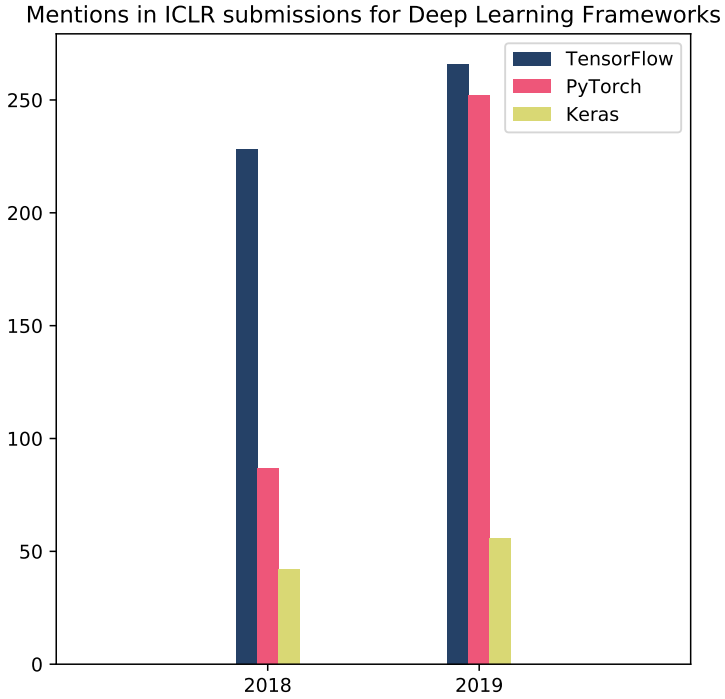


Figure 2.1: Changes in popularity of different deep learning libraries in research. Data was collected by keyword search over ICLR submissions (http://search.iclr2019.smerity.com/search; analogously for 2018)

```
# loop body
def body(counter):
    h = tf.add(tf.matmul(W, h), b)
    # increment counter
    return [tf.add(counter, 1)]

# do the actual loop
r = tf.while_loop(while_condition, body, [counter])
```

- As a corollary, the barrier of entry is higher, since a beginner cannot rely on the language feature they know but must learn how to express many concepts without the host language.
- halting execution at arbitrary points in the training is not possible, since the actual training is not happening in the host language, but is more often handed off to lower-level implementations in its entirety.

This makes conditional processing and debugging much less ergonomic.

For this work, the PyTorch framework has been chosen, due to the fact that it is growing quickly in popularity (see figure 2.1) and relatively new, so the ecosystem is not fully developed and some utilities available for e.g. TensorFlow are not available for PyTorch. Because of this, an introspection framework for training monitoring is judged to present the best value proposition for PyTorch users.

2.3 PUBLISHER-SUBSCRIBER

The Publisher-Subscriber pattern (for a detailed overview see Eugster et al. (2003)) is a pattern for distributed computation in which publishers publish messages either directly to any subscribers which have registered interest in them, or to a central

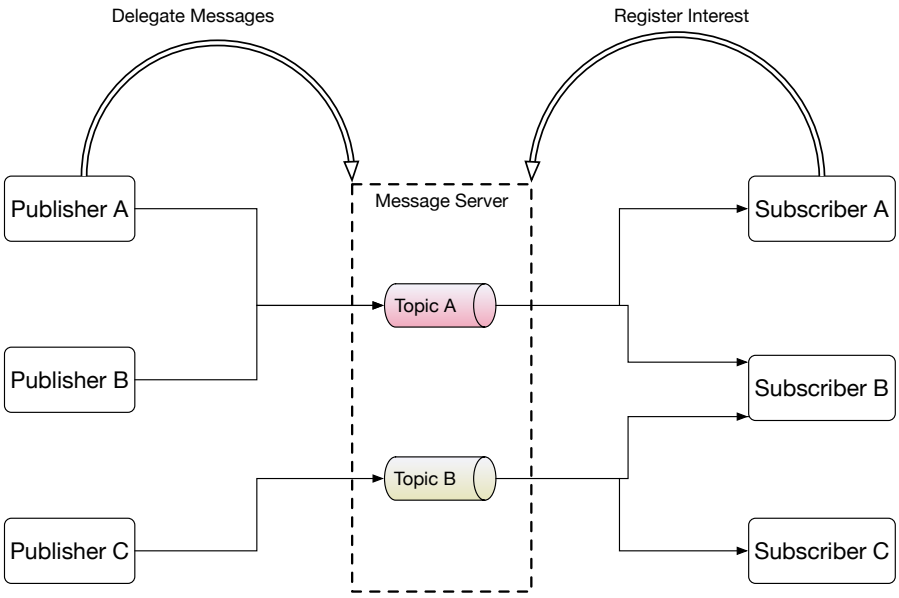


Figure 2.2: One possible implementation of the Publisher-Subscriber pattern.

authority orchestrating the exchange. Messages are generally associated with one or more topics and subscribers register interest in receiving messages on one or more topics.

The components are very loosely coupled; the subscribers need not even be aware of the publishers at all, and the publishers' only interaction with their subscribers is relaying messages through a uniform interface or through an optional server. A graphical schema of one possible incarnation of this pattern is shown in figure 2.2.

This project is not distributed, but can benefit from the loose coupling in another way: Subscribers can be defined in terms of the kind of messages they need to compute their metric, without knowing anything about where the messages are coming from. Concretely, as long as the appropriate data is emitted from the training process, subscribers can work without modifications with any possible model.

Since real-world neural networks are trained on the GPU, and communication between host and GPU memory is already expensive, making this library truly distributed across processes is not an objective. However, the design will simplify asynchronous computation of metrics in the future. The Python language does not support true multithreading[2], but since the expensive part of the work is running on the GPU while the host code is waiting (or prefetching data), metric computation could happen asynchronously on the GPU as well while the expensive forward or backward passes through the network are running. This is not currently implemented but can be added later, in case more computationally demanding metrics are to be explored.

In the context of neural network training, there is only one source of information—the model—and hence only one publisher. Nevertheless, a message server is introduced to segregate responsibilities. The singular publisher extracts data from the training model, passes it on to the server which also accepts subscriber registrations and relays messages appropriately.

2 The `multiprocessing` module allows for truly asynchronous computation and communication, but the inter-process-communication is more expensive than memory shared between threads.

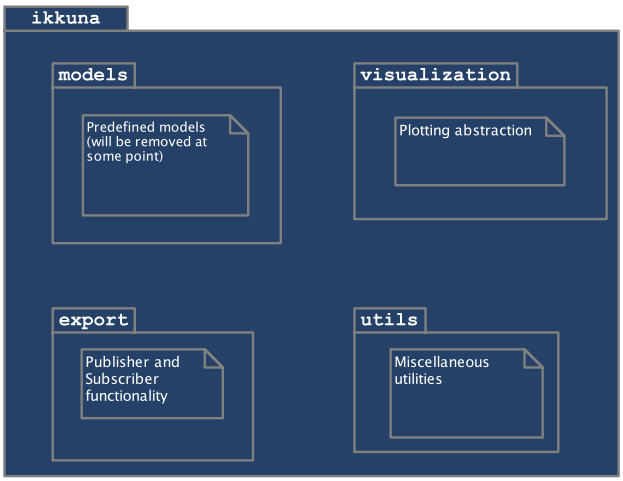


Figure 2.3: `ikkuna` package diagram

2.4 OVERVIEW OF THE LIBRARY

The software is structured into several subpackages of the `ikkuna` package. Other packages and modules contain utilities implemented for this work specifically, but will generally not be relevant to other users. A survey of these tools will be given in section 2.4.5. This section surveys the structure of the codebase and elaborates on implementation strategies, but does not constitute a full documentation. For a detailed API reference, the reader is referred to the HTML documentation[3].

The root package diagram is shown in figure 2.3

The most important bits of the software live in the `export` subpackage (section 2.4.1). It implements the Publisher-Subscriber pattern. Extracting data from the training process, defining subscriber functionality and messages used for communication is done here.

The `models` subpackage (see section 2.4.2) contains a few exemplary neural network definitions which are wired up with the library and can thus be used to showcase the library's functionality. The `utils` subpackage (see section 2.4.3) contains miscellaneous utility classes and functions used throughout the core library. Lastly, the `visulization` subpackage (section 2.4.4) contains the plotting functionality to actually show the metrics computed during the training process.

2.4.1 *The `ikkuna.export` subpackage*

The `ikkuna.export` subpackage contains the core part of the library, i.e. it provides the classes that handle discovering the structure of the neural network model, attaching the appropriate callbacks and intercepting method calls on the model so the library is informed about everything entering and exiting the model and its individual layers. It also contains the definition for the subscriber API, i.e. the messages that subscribers can receive, synchronisation facilities when multiple topics are needed by a subscriber, as well as the subscriber class interface. The package diagram is displayed in figure 2.4.

The package comprises three subpackages or modules listed in table 2.1

3 At the time of writing located at https://peltarion.github.io/ai_ikkuna/

Table 2.1: `ikkuna.export` functionalities

Submodule	Function
`export`	Define the publisher discovering an arbitrary model and relaying messages to the message server
`messages`	Define message interface, i.e. what topics exist and which information a message must contain
`subscriber`	Define the base class for subscribers and classes for message synchronisation and filtering

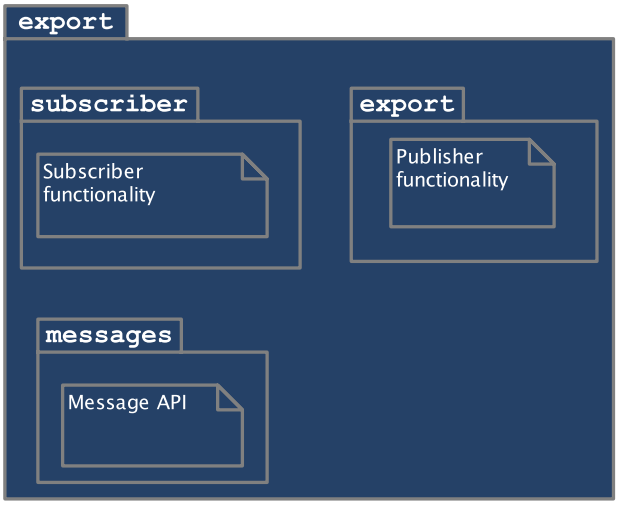


Figure 2.4: `ikkuna.export` package diagram

The `ikkuna.export.export` submodule

In in slight deviation from the Publisher-Subscriber framework as displayed in figure 2.2, the `ikkuna.export.Exporter` class (figure 2.6) is the sole publisher of data. There is only one source of data during training, so it is unnecessary to accommodate multiple publishers. The `Exporter` is informed of the model with its methods `set_model()` and `set_loss()`, the latter of which is only necessary if metrics which rely on training labels should be displayed. It can accept a filter list of classes which are to be included when discovering the modules in the model. For instance, it could be desirable to only observe layers which have weights and biases associated with them, not e.g. normalisation or reshaping layers. The `Exporter` then traverses the model (which is really just a tree structure of modules) and adds to each a callback invoked when input enters the layer—in order to retrieve activations—and when gradients are computed for the layer outputs. PyTorch provides the `add_forward_hook()` and `add_backward_hook()` functions on modules (layers) and `add_hook()` function on Tensors to register callbacks. All added modules are also given a name—either set by the user during registration of the module or generated automatically. The name is necessary for display purposes.

The callbacks also use cached weights—if present—in order to publish updates to the weights.

Furthermore, the `Exporter` employs monkey patches to the model; it replaces a few of the model's methods with closure wrappers—functions defined locally which have access to the `Exporter` instance—so it can

- be notified when the model is set to training or testing mode (this switch disables or enables layers which only make sense during one of the phases[4])
- increase its own step counter automatically when a new batch is seen. This can be deduced from the fact that `forward()` is called on the model.
- add a parameter to the model's `forward()` method—called by the runtime when data is propagated through the model—which can be used be subscribers to temporarily turn off training mode and have it revert automatically. This is useful for subscribers which need to evaluate the model and are thus given limited access through the `forward()` method, but want the following messages emitted by the `Exporter` to carry a specific tag, so that subscribers listening to the default tag are not notified.
- intercept inputs and labels passed to the loss function during training and publish them as messages so the user need not concern himself with this task. This is useful for e.g. computing the accuracy on the current batch with the subscriber framework.
- intercept the final output of the network. This could be realised alternatively by identifying the last module in the network and attaching specific callbacks only to it, but is easier to do by patching the loss function set with `set_loss()`, since the loss function will always be evaulated with the network output.

At every time step (training batch), the `Exporter` publishes the following information on to the message bus (see figure 2.5):

- gradients for each module
- gradients for each parameter (loss derivative w.r.t the weights and biases)
- activations for each module
- current values of weights and biases for each module that has these properties (e.g. convolutional or fully-connected layers)
- updates to the weights and biases applied at the end of the current step
- Training labels used for the parameter updates. This requires that the `Exporter` be informed of the loss function object with `set_loss()`.
- The batch of input data passed to the network at the current training step
- The final output of the network for the current batch of training data. This is simply the tensor of activations from the last layer and is thus technically duplicated since activations are published anyway. The reason is that some

4 There are two built-in layers this applies to. One is the batch normalisation layer. It normalises the output of the previous layer with the mean and variance over the entire batch of data—optionally with running means and variances over the previous training steps. The variance is not defined for single data point entering the layer, as could be the case during inference/testing time. The layer therefore stores running estimates of mean an variance during training and uses them during test time. The second case is the dropout layer, which randomly zeroes out a percentage of the previous layer's activations. This is used during training to prevent subsequent units from becoming correlated with a fixed set of units in the previous layer, instead of picking up patterns invariant of where in the input they occur. During inference time, this is turned off to make full use of the trained layers.

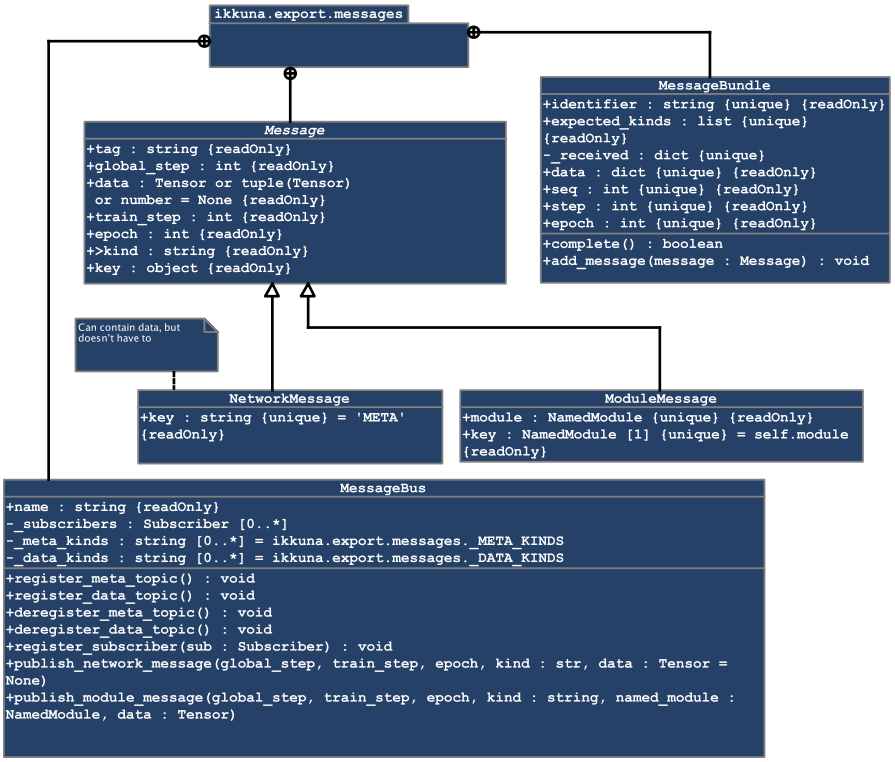


Figure 2.5: Classes in the `ikkuna.export.messages` submodule

subscribers may only be interested in the network predictions and it is unnecessary to determine automatically the last layer in the network as the loss function has automatic access to the activations and must be patched anyway for the training labels

Further messages are published only at certain points in the training process

- When a batch starts or ends, a message with the current batch index is published
- When an epoch starts or ends, a message with the current epoch index is published. This requires the `Exporter` be notified with `epoch_finished()` by the user, since it is impossible to determine when an epoch is over from inside the model.

The `ikkuna.export.messages` submodule

This submodule contains definitions of all permissible messages kinds, message classes and a collection class for message objects. An overview of the classes defined in this module is shown in figure 2.5.

Messages are of one of two types: They are either directly tied to a layer in the network and are thus published for each layer, or they contain information for the current epoch or training step applying to the entire network. In that case, they appear only once per epoch or training step, not once per layer. The meta-messages can carry tensor data (e.g. input data or labels), but need not to (e.g. notifications about a starting or ending epoch). All message kinds are summarised in table 2.2.

Messages can be assembled into bundles if a subscribers wants to subscribe several topics at once. The `MessageBundle` class performs all necessary error checking

Table 2.2: Subscribable message kinds

Meta topics		
Identifier	Frequency	Description
`batch_started`	Once every batch	
`batch_finished`	——— " ———	
`epoch_started`	Once every epoch	
`epoch_finished`	——— " ———	
`input_data`	Once every batch	
`input_labels`	——— " ———	
`network_output`	——— " ———	Activations of the last layer
`loss`	——— " ———	Loss over the current batch
Data topics		
Identifier	Frequency	Description
`weights`	Once per layer per batch	Gradients of loss function w.r.t. layer weight matrix
`weight_gradients`	——— " ———	
`weight_updates`	——— " ———	
`biases`	——— " ———	Gradients of loss function w.r.t. layer bias vector
`bias_gradients`	——— " ———	
`bias_updates`	——— " ———	
`activations`	——— " ———	
`layer_gradients`	——— " ———	Gradients of loss function w.r.t. layer output

to ensure consistency of the contained messages (e.g. only messages from the same time step are allowed).

The message server from figure 2.2 is implemented by the `MessageBus` class which receives publishers' messages, accepts subscriber registrations, and maintains the lists of known topics. Each subscriber may in turn announce one or more new topics which can then be subscribed to by others. This is useful since it allows chaining of subscribers in order to realise arbitrary post-processing of computed metrics.

The `ikkuna.export.subscriber` subpackage

The third subpackage contained in the `ikkuna.export` package defines the subscriber part of the Publisher-Subscriber pattern. The diagram of the defined classes is shown in figure 2.7. The `Subscriber` base class is rudimentary and mandates only the implementation of the metric computation by subclasses. In the simplest case, a subscriber is interested in only one topic and therefore is coupled to a simple `Subscription` object, which handles bookkeeping tasks such as subsampling the message stream, routing only relevant messages to the subscriber and counting the received messages.

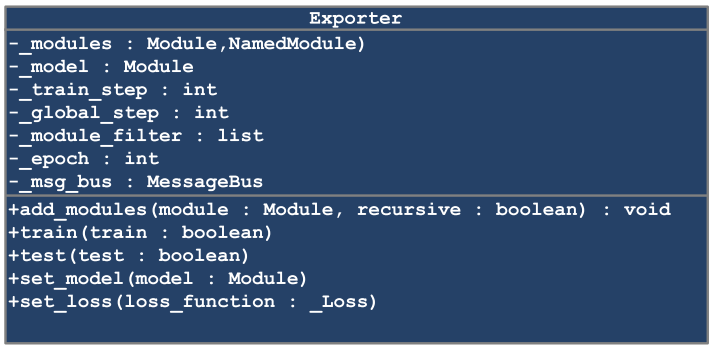


Figure 2.6: `ikkuna.export.Exporter` class diagram

More generally however, a subscriber may want to receive several pieces of information for each layer in each time step (i.e. for computing the ratio between weight updates and weights). Since the order of messages is not guaranteed, the desired messages are unlikely to occur one after the other; instead the topics must be synchronised. A `SynchronizedSubscription` buffers messages of the relevant topics until all requested kinds have been received for the current training step, before releasing them to the subscriber.

A subscriber can thus receive a single message or a bundle of messages for each subscription. It can also have multiple subscriptions, but each topic can only be associated with one subscription.

The library comes with a few subscribers already installed (they are themselves plugins, see section 2.4.6). Details are given in table 2.3

2.4.2 *The `models` subpackage*

This package shown in figure 2.8 contains model definitions for demonstration purposes and for experimentation. Four architectures are currently implemented:

1. A minified version of AlexNet, since the original architecture requires larger images (Krizhevsky et al., 2012). The code is adapted from Suki Lau[5].

2. DenseNet (Huang et al., 2017). The implementation is basically the one from (Pleiss et al., 2017)[6] with minor modifications

3. ResNet (He et al., 2016). This implementation comes from GitHub user liukang[7] and can handle CIFAR10-sized images of 32 pixels per side, as opposed to most implementaions that are geared towards ImageNet examples which are much larger.

5 https://github.com/sukilau/Ziff-deep-learning/blob/master/3-CIFAR10-lrate/CIFAR10-lrate.ipynb

6 At the time of writing, the implementation is available here: https://github.com/gpleiss/efficient_densenet_pytorch/blob/master/models/densenet.py. The licensing is unclear as the author references the original BSD-licensed implementation at https://github.com/pytorch/vision/blob/master/torchvision/models/densenet.py which was licensed by PyTorch core contributor Soumith Chintala. However, the code does not reproduce the BSD license text and can thus only be inspired by the original but cannot contain any of the code verbatim. It would require careful examination in order to determine whether this is the case.

7 The implementation is MIT-licensed. https://github.com/kuangliu/pytorch-cifar/blob/master/models/resnet.py

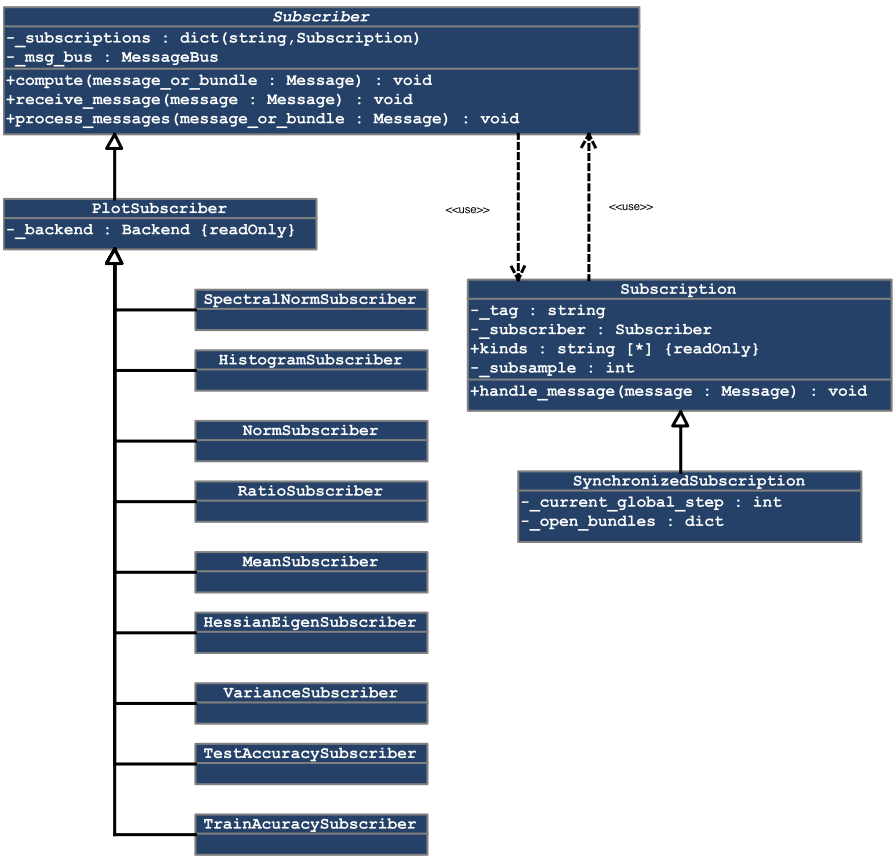


Figure 2.7: Classes defined in `ikkuna.export.subscriber`

Table 2.3: Pre-packaged subscriber subclasses

Name	Functionality
`MeanSubscriber`	Computes the mean $\mu = \frac{1}{n}\sum_{i=1}^{n} w_i$ of a tensor
`VarianceSubscriber`	Computes the variance $\sum_{i=1}^{n}(w_i - \mu)^2$ for a tensor
`HessianEigenSubscriber`	The top-k eigenpairs of the Hessian matrix
`NormSubscriber`	Computes the p-Norm $\sqrt[p]{\sum_{i=1}^{n} w_i^p}$
`RatioSubscriber`	Computes the ratio of norms $\frac{\|\|T_1\|\|_2}{\|\|T_2\|\|_2}$ of two tensors
`HistogramSubscriber`	Computes the histogram of a given tensor. This is computationally heavy.
`SpectralNormSubscriber`	Computes the spectral norm (largest singular value) $\max_{\mathbf{h}:\mathbf{h}\neq 0} \frac{\|\|A\mathbf{h}\|\|_2}{\|\|\mathbf{h}\|\|_2}$ of a tensor reshaped to 2-d
`TestAccuracySubscriber`	Computes the ratio of correctly classified examples to total examples over the test set
`TrainAccuracySubscriber`	Computes the ratio of correctly classified examples to total examples over current batch of training data

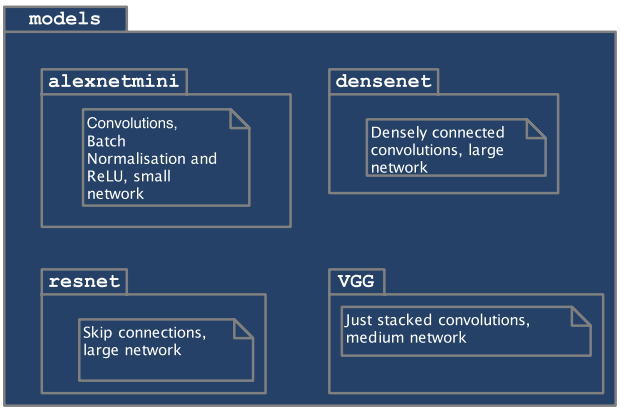


Figure 2.8: `ikkuna.models` package diagram

4. VGG18, a variant of the deep convolutional network introduced in (Simonyan and Zisserman, 2014). The implementation is adapted from GitHub user liukang as well. https://github.com/kuangliu/pytorch-cifar/blob/master/models/vgg.py

All models are modified such that their training can be supervised by the library.

2.4.3 *The `utils` subpackage*

As shown in figure 2.9, this package defines classes for traversing a model into a hierarchical tree of layers (called *modules* in PyTorch lingo) with some added metadata, and a set of miscellaneous functions for

1. Seeding random number generators to make experiments reproducible (see appendix B)
2. Creating instances of weight optimizers by name
3. Loading datasets and inferring all metadata about it
4. Creating models by name
5. Initialize the weights of any model

Additionally, it contains the `numba` module which is intended to allow interoperability with the Numba library[8]. While currently not used due to the incomplete nature of the Numba GPU array interface, it could enable leveraging Numba in the future without transferring data to the CPU. The core function was later obsoleted by an addition to the PyTorch library[9].

8 https://numba.pydata.org/. Numba is a library for transforming high-level Python code into performant compiled code and for allowing to use the CUDA library from Python with Python arrays. This enables performance improvements for numeric calculations, but there is only a limited set of higher-level functions implemented on GPU arrays.

9 The main contribution of the submodule was to make PyTorch tensors accessible to Numba by monkey-patching the `__cuda_array_interface__` property. This has since been added via pull request #11984 to the PyTorch repository.

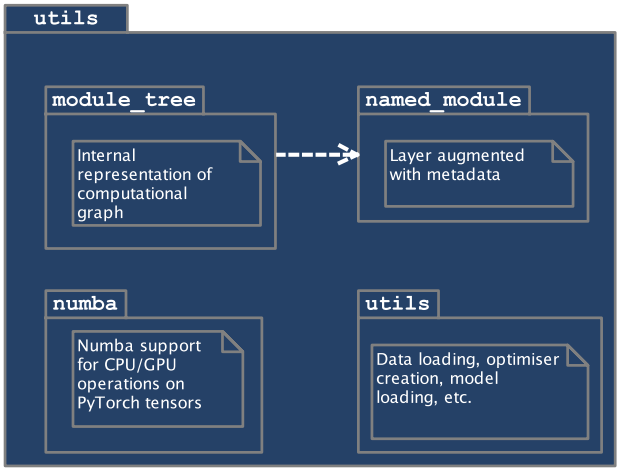


Figure 2.9: `ikkuna.utils` package diagram

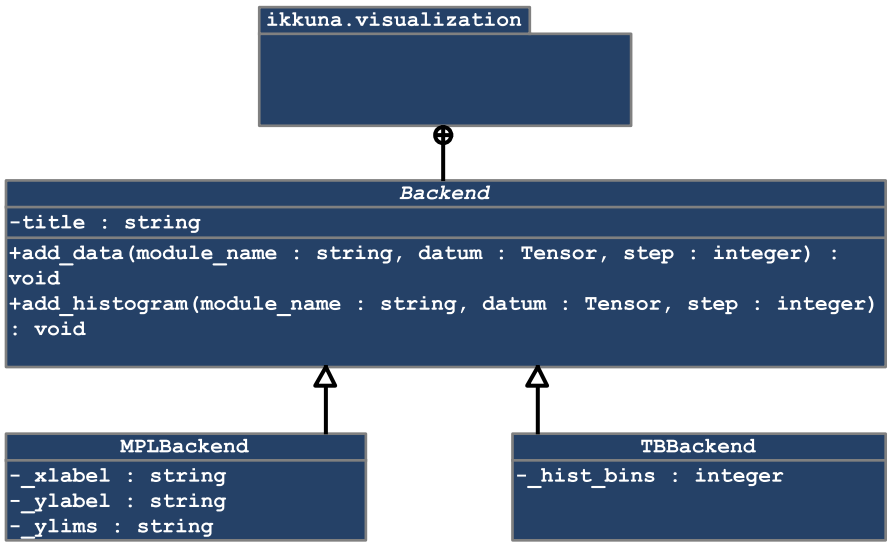
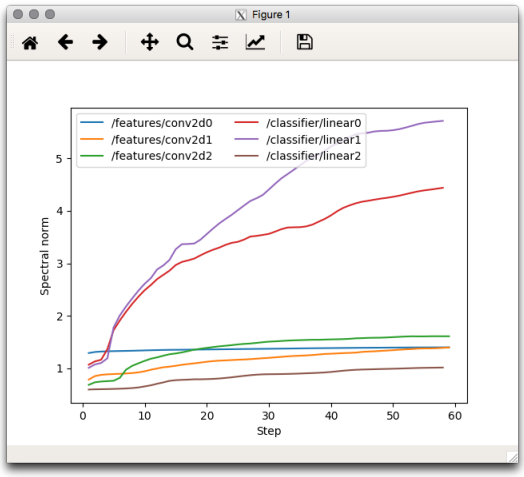


Figure 2.10: Class diagram for classes in `ikkuna.visualization`

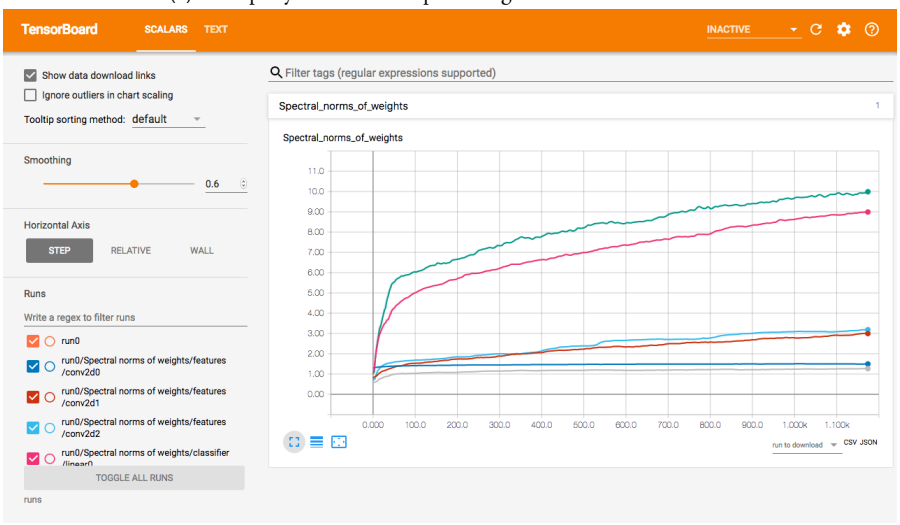
2.4.4 *The `visualization` subpackage*

This package contains only a single module: `backend`. It defines the classes shown in figure 2.10. The module serves as an abstraction over plotting libraries (or more generally, information sinks) so that metrics need not concern themselves with how the data is actually presented to the user. A given metric will compute its value and dispatch it to its backend, which can currently accept scalar and histogram data. The metric class itself need not care about how it is going to be displayed. While not currently implemented, monitoring or database backends (e.g. based on MongoDB or Grafana) could complement the already present plotting backends.

For running the library locally, a `matplotlib`-based backend has been implemented. Plotting routines from this library open a window directly on the system executing the software. In practice however, deep learning code will be executed remotely on a server with adequate compute capability and the programmer connected via SSH. While it is possible to have remote windows show up locally on Linux-based systems by use of X11-Forwarding, this is generally slow and not useful for responsive plotting. An example is shown in figure 2.11a. To remedy this issue, a



(a) Exemplary view of a Matplotlib figure forwarded over SSH



(b) Exemplary view of a TensorBoard session

plotting backend for TensorBoard (see section 1.5) is also provided. The plotting data is generated and processed on the remote system, but served over the web so it can be viewed and interacted with locally (provided the network is configured so that the server responds to HTTP requests). An example is shown in figure 2.11b.

2.4.5 *Miscellaneous tools*

There are a few modules which simplify development with the library but are not part of the distribution obtained from PyPi or by running the setup script. Nevertheless, they can be used by the reader to start using the library.

The `train` package defines a `Trainer` class which encapsulates all the logic and parameters needed to train a neural network on one of the datasets provided with PyTorch. The class's capabilities include the following

- Look up model and dataset by name
- Bundle all hyperparameters
- hook the `Exporter` into the model for publishing data
- configure the optimisation algorithm to use for training

Table 2.4: Named arguments to `main.py`

Parameter	Explanation
`-m`,`--model`	Model class to train
`-d`,`--dataset`	Dataset to train on. Possible choices: `MNIST`, `FashionMNIST`, `CIFAR10`, `CIFAR100`
`-b`,`--batch-size`	Default: 128
`-e`,`--epochs`	Default: 10
`-o`,`--optimizer`	Optimizer to use. Default: `Adam`
`-a`,`--ratio-average`	Number of ratios to average for stability (currently unused). Default: 10
`-s`,`--subsample`	Number of batches to ignore between updates. Default: `1`
`-v`,`--visualisation`	Visualisation backend to use. Possible choices: `tb`, `mpl`. Default: `tb`
`-V`,`--verbose`	Print training progress. Default: `False`
`--spectral-norm`	Use spectral norm subscriber on weights. Default: `False`
`--histogram`	Use histogram subscriber(s)
`--ratio`	Use ratio subscriber(s)
`--test-accuracy`	Use test set accuracy subscriber. Default: `False`
`--train-accuracy`	Use train accuracy subscriber. Default: `False`
`--depth`	Depth to which to add modules. Default: `-1`

- train the model for one batch

The `Trainer` class is used in the main script (`main.py`), which serves as a command line interface to the library while developing. When trying out the library, it can also be used as an initial starting point.

All experimental code used for chapter 3 is located in the `experiments` subfolder accompanying this thesis. It can be used for reproducing all experiments conducted for this work.

The library can be installed to the local Python environment by use of the provided setuptools script (`setup.py`). It can also be downloaded from the Python package index[10] by use of the package manager `pip`:

```
pip install ikkuna
```

2.4.6 *Plugin Infrastructure*

Among the main selling points of this library is the provision to add new metrics as plugins and reuse them system-wide for all architectures. Plugins in Python projects can be enabled through appropriate use of the `setuptools` library. During the setup process for installing the library, entry points (which are names) are defined by the library which can be used by plugins to announce themselves. `Ikkuna` provides

10 https://pypi.org/

the `'ikkuna.export.subscriber'` entry point. For registering a plugin, the author must simply use that entry point to make a plugin available. For illustration, listing 2.1 shows how to setup a `setup.py` setuptools file. The plugin can be installed like any other Python package with

```
python setup.py install
```

which will install all required dependencies inside the current environment. The PyTorch library must be installed manually since the binary distribution is too old at the time of writing. Detailed instructions can be found in the user guide which is part of the documentation.

```
#!/usr/bin/env python

from distutils.core import setup
import setuptools

setup(name='<your package name>',
    version='<version>',
    description='<description>',
    author='<your name',
    author_email='<your email>',
    packages=['<package name>'],
    # ... any other args
    entry_points={
        'ikkuna.export.subscriber': [
            'YourSubscriber = module.file:YourSubscriber',
        ]
    })
```

Listing 2.1: Sample setup script for subscriber plugins

2.4.7 *Documentation*

The entire codebase is liberally documented using the Sphinx documentation processor[11]. The documentation contains further documents with a detailed user guide and installation instructions. Sphinx allows generating documentation in many formats from the same source, most usefully HTML and PDF. At the time of writing, the HTML documentation and API reference is hosted at https://peltarion.github.io/ai_ikkuna/.

2.5 BUSINESS CASE FOR THE LIBRARY

This work is done in cooperation with Peltarion AB[12], a software company based in Stockholm, Sweden. Peltarion's stated mission is to

> [provide] an operational AI platform for producing real-world AI applications at scale and at speed.[13]

The Peltarion platform is a web-based deep learning platform with which users can upload, preprocess and modify datasets, create deep neural architectures without having to write code and track performance of each model and dataset version

11 http://www.sphinx-doc.org/
12 http://www.peltarion.com/
13 https://www.peltarion.com/about, accessed 17/01/19

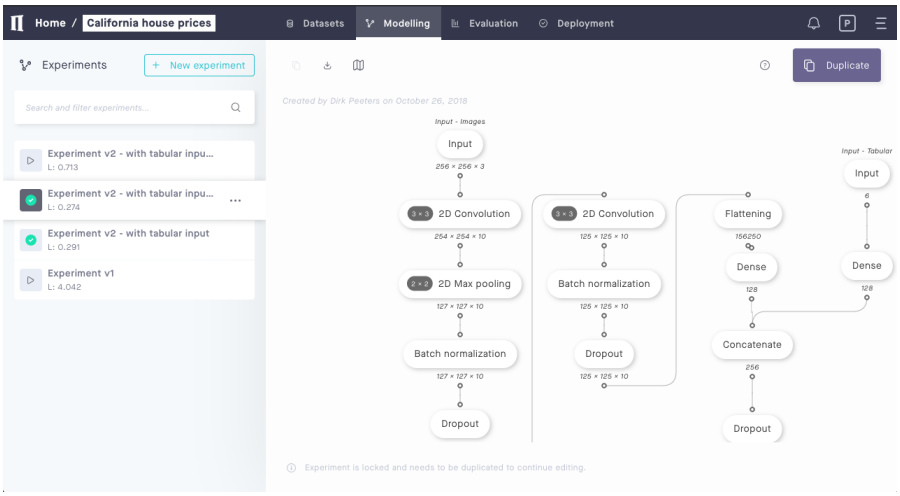


Figure 2.12: The Peltarion platform modeling screen

through experiment versioning. Trained models can be directly deployed as a web service.

The modeling interface presented to the user is shown in figure 2.12

While the product in question is code-free for the user, it is powered by a deep learning framework on the server side. The business proposition made by Peltarion is to make training deep neural networks more affordable, which the company wants to realise through savings in development time from problem statement to model deployment. As outlined in section 1.4, development of state-of-the art deep learning applications requires both expertise and education as well as experience. Since experts in any discipline are rare and expensive, lowering the cost in this area requires simplifying the process of creating deep learning solutions. A code-free platform is one way to accomplish this and make deep learning more accessible to users and companies without the previously required expertise in research and engineering.

This thesis ties into this objective in two ways. Firstly, by providing an API against which training metrics can easily be implemented and served to arbitrary backends, engineering effort for metric features of the platform could be reduced. For this work, plotting backends have been implemented, but the decoupled compontent architecture of the `ikkuna` library was chosen precisely to enable arbitrary data sinks for the computed metrics, for instance a web service or a database which is accessed by the Peltarion platform to display metrics to the user. The engineering team at Peltarion could make use of this library to handle metric logging on arbitrary models created by the platform's users without having to resort to code generation during translation of the abstract model definition created in the browser to the model implementation in the backend.

Secondly, the library—or the ideas prototyped therein—will be helpful in providing feedback to the user about the state of their experiment. Since the platform is at least partially aimed at non-experts, even well-known and simple metrics can be of great help in avoiding common pitfalls in model training. This directly reduces the opaqueness of the training process to the non-expert user, reducing time wasted on fruitless experiments and increasing confidences in the product.

It should be stressed, however, that the software implemented for this thesis is free and open-source and not owned or licensed by Peltarion AB.

3

EXPERIMENTS IN LIVE INTROSPECTION

This chapter serves the purpose of showcasing the library and validating its usefulness for actual deep learning research. While the ultimate goal is to have a set of well-researched metrics in place which can be used live during training, the way to acquire these metrics requires extensive analysis from many experiments. Therefore, the goal of this work is to research candidate metrics, not actually use them to improve any particular model's performance on any specific problem. This entails that the library is used primarily for easily gathering all the data from the necessary experiments and computing candidate metrics, and not for supervising the actual training.

The chapter is divided between a reproduction part (see section 3.2) and an original research part (see section 3.3 and section 3.4). In the former, experiments conducted for a popular variant of stochastic gradient descent (hence referred to as SGD) are reproduced and extended with the help of the library. The goal is to fact-check claims made by the authors and validate said claims on more scenarios than are shown in the publication. This will serve to show how `ikkuna` could have been employed for this type of work. The second part will be concerned with employing the library to investigate hypotheses for diagnosing roadblocks in the training process. Recall from section 1.4 the multitude of hyperparameters for a deep learning model and training regimen. In this work, we will concern ourselves with two questions:

1. How can we figure out a good learning rate?
2. (When) should we stop training layers to reduce computation time?

This thesis is not concerned with advancing the state of the art in classification. Instead, classical datasets are used on toy models and standard models in order to prove or disprove that the proposed metrics have the potential to be useful in guiding the training. A more thorough evaluation of the results on realistic architectures and problem sizes would require significantly more time and computational resources and is left for future work.

3.1 EXPERIMENTAL METHODOLOGY

For all experiments, `ikkuna` is used for recording various metrics during training without having to adapt to any specific model. Experiments are run on a Google Cloud virtual instance with between 4 and 8 Intel Xeon CPUs and 1 to 2 Nvidia Tesla K80 graphics cards with 11 GB of video memory. The information captured by `ikkuna` during training is logged to a MongoDB schemaless database with the help of the `sacred` library. The data can then be analysed off-line with MongoDB's Python interface. Plots are created with `matplotlib`.

A subscriber is used to log all the computed metrics to a MongoDB database via the `sacred` library (listing 3.1). While the cleaner approach here would be to implement a `SacredBackend` to log the metrics automatically, this would require some thought about the differing capabilities of different backends. For instance,

there is no native form of histogram storage in `sacred`, while it is easy to provide through TensorBoard or Matplotlib. Therefore, this backend is left for future work and a Subscriber is used instead for this limited set of experiments.

```
class SacredLoggingSubscriber(Subscriber):
  '''Subscriber which logs its subscribed values with sacred's
      metrics API'''

  def __init__(self, experiment, kinds):
    self._experiment = experiment
    subscription     = Subscription(self, kinds)
    super().__init__([subscription], get_default_bus())

  def compute(self, message):
    if message.key != 'META':
      name = f'{message.kind}.{message.key.name}'
    else:
      name = message.kind
  self._experiment.log_scalar(name, float(message.data), message.
      global_step)
```

Listing 3.1: Subscriber for logging metrics

3.2 VALIDATING OPTIMISER RESEARCH

In this section, we want to highlight the `ikkuna`'s usefulness in scientific research by conducting experiments in the context of (Kingma and Ba, 2014) and investigating in how far the claims made therein hold true on a range of problems. The work investigates the Adam optimisation algorithm on several machine learning problems, notably learning a convolutional network on the CIFAR10 dataset.

3.2.1 *The Adam update rule*

While standard SGD only makes use of the gradients in the current time step to compute a parameter update, Adam keeps an exponential moving average of the gradient (m_t) and its square (v_t), supposedly as an estimate for the mean and variance of the gradient around the current point in parameter space. This estimate is computed for each parameter, meaning Adam incurs a memory overhead of $\mathcal{O}(d)$ where d is the number of model parameters. Since the estimates are initialised with zero, a bias correction (denoted by $\widehat{\cdot}$) is applied.

The parameter updates are computed as

$$\theta_t \leftarrow \theta_{t-1} - \eta \frac{\widehat{m}_t}{\sqrt{\widehat{v}_t} + \epsilon} \tag{3.1}$$

Kingma and Ba justify the rule like this:

> With a smaller SNR [quotient of mean and square root of variance] the effective stepsize Δ_t will be closer to zero. This is a desirable property, since a smaller SNR means that there is greater uncertainty about whether the direction of m_t corresponds to the direction of the true gradient. For example, the SNR value typically becomes closer to 0 towards an optimum, leading to smaller effective steps in parameter space: a form of automatic annealing.

Intuitively, the magnitudes of the update to a parameter will scale linearly with the magnitude of the running gradient mean (which is effectively a form of momentum), but normalised with the magnitude of the running gradient variance so that the update is smaller when the gradient is very noisy and larger when there is more certainty of the direction of the true gradient.

The Role of `ikkuna`

We use the `ikkuna` library to compute and record various metrics during reproduction and extension of Kingma and Ba's experiments. This section showcases how little code is required to implement a recording setup portable to any neural network model. A subscriber is employed to record

1. The biased first moment estimate (exponential running gradient)
2. The biased second moment estimate (exponential running squared gradient)
3. The bias-corrected first moment estimate
4. The bias-corrected second moment estimate
5. The effective learning rate

For all metrics, mean, variance, median, and norm are logged in some interval, as computing the median incurs $\mathcal{O}(n \log n)$ runtime cost for each layer. Since Adam doesn't directly use the current gradients for its parameter updates, we need to extract the effective learning rate used for a layer somehow. Since we have access to the gradient $\nabla_t(L)$ and the update Δ_t in each train step, we can divide the two to obtain $\frac{\Delta_t}{\nabla_t(L)} = \eta$ which is the learning rate we would need to use in a vanilla SGD step with the current gradients. However, gradients can be zero at any time, and the corresponding update may not be, since Adam uses gradients from the previous steps as well. We thus simply ignore invalid values through division by zero in computing the metrics. In the pathological case where this leads to an empty tensor, we plot NaN (leading to an omission of the value), since we would otherwise have to work around metric logs with different step vectors. The subscriber used for recording all the information is displayed in listing 3.2.

```
class BiasCorrectedMomentsSubscriber(PlotSubscriber):

  def __init__(self, lr, beta1, beta2, eps, message_bus=
      get_default_bus(), tag=None, subsample=40, ylims=None, backend=
      'tb'):

    title        = 'gradient_moments'
    ylabel       = 'Gradient Moments'
    xlabel       = 'Train step'
    subscription = Subscription(self, ['weight_gradients'], tag,
        subsample)
    super().__init__([subscription], message_bus,
                     {'title': title,
                      'ylabel': ylabel,
                      'ylims': ylims,
                      'xlabel': xlabel},
                     backend=backend)

    # all parameters to Adam
```

```
    self._lr    = lr
    self._beta1 = beta1
    self._beta2 = beta2
    self._eps   = eps

    # records of the running moment estimates
    self._means = dict()
    self._vars  = dict()

    # here we set up all the metrics to be published
    for pub_name in {
      'biased_grad_mean_estimate_mean',
      'biased_grad_mean_estimate_median',
      'biased_grad_mean_estimate_var',
      'biased_grad_var_estimate_mean',
      'biased_grad_var_estimate_median',
      'biased_grad_var_estimate_var',
      'biased_grad_mean_estimate_norm',
      'biased_grad_var_estimate_norm',
      'grad_mean_estimate_mean',
      'grad_mean_estimate_median',
      'grad_mean_estimate_var',
      'grad_var_estimate_mean',
      'grad_var_estimate_median',
      'grad_var_estimate_var',
      'grad_mean_estimate_norm',
      'grad_var_estimate_norm',
      'effective_lr_mean',
      'effective_lr_median',
      'effective_lr_var',
      'effective_lr_norm',
    }:
      self._add_publication(pub_name, type='DATA')

  def compute(self, message):

    named_module = message.key

    grad         = message.data
    t            = message.global_step + 1

    # init moving moments if not present
    if named_module not in self._means:
      self._means[named_module] = torch.zeros_like(grad)
      if named_module not in self._vars:
        self._vars[named_module] = torch.zeros_like(grad)

    exp_avg, exp_avg_sq = self._means[named_module], self._vars[
        named_module]
    beta1, beta2        = self._beta1, self._beta2

    exp_avg.mul_(beta1).add_(1 - beta1, grad)
    exp_avg_sq.mul_(beta2).addcmul_(1 - beta2, grad, grad)

    # we use the more efficient ordering of computation from Kingma
        et al. (p.
    # 2) used in PyTorch's implementation
    bias_correction1    = 1 - beta1 ** t
    bias_correction2    = 1 - beta2 ** t
    unbiased_exp_avg    = exp_avg / bias_correction1
    unbiased_exp_avg_sq = exp_avg_sq / bias_correction2
```

```
    step_size           = self._lr * math.sqrt(bias_correction2) /
        bias_correction1
    denom               = exp_avg_sq.sqrt().add_(self._eps)

    # here we basically revert the entire thing to get the effective
        learning
    # rate
    update              = step_size * exp_avg / denom
    update.div_(grad)
    nan_tensor          = torch.isnan(update)
    inf_tensor          = torch.isinf(update)
    effective_lr        = update[(1 - nan_tensor) & (1 - inf_tensor)]

    # it's possible to end up with no valid values -> log 0 so
        plotting doesn't
    # crash (nan may also have worked)
    if grad.sum() == torch.tensor(0.0).cuda():
      # this would mean all entries are nan or inf because the
          current gradient was
      # zero
      effective_lr = torch.tensor(0.0).cuda()

    # instead of repeating the call to publish_module_message for
        each topic, look at
    # all topic names and infer the local variable from the topic
        name
    for topic in self.publications['DATA']:

      if topic.startswith('biased_grad_mean'):
        data = exp_avg
      elif topic.startswith('biased_grad_var'):
        data = exp_avg_sq
      elif topic.startswith('grad_mean'):
        data = unbiased_exp_avg
      elif topic.startswith('grad_var'):
        data = unbiased_exp_avg_sq
      elif topic.startswith('effective_lr'):
        data = effective_lr
      else:
        raise ValueError(f'Unexpected topic "{topic}"')

      if topic.endswith('norm'):
        data = data.norm()
      elif topic.endswith('mean'):
        data = data.mean()
      elif topic.endswith('median'):
        data = data.median()
      elif topic.endswith('var'):
        data = data.var()
      else:
        raise ValueError(f'Unexpected topic "{topic}"')

      self.message_bus.publish_module_message(message.global_step,
                                              message.train_step,
                                              message.epoch, topic,
                                              message.key, data)
```

Listing 3.2: Subscriber to record Adam terms

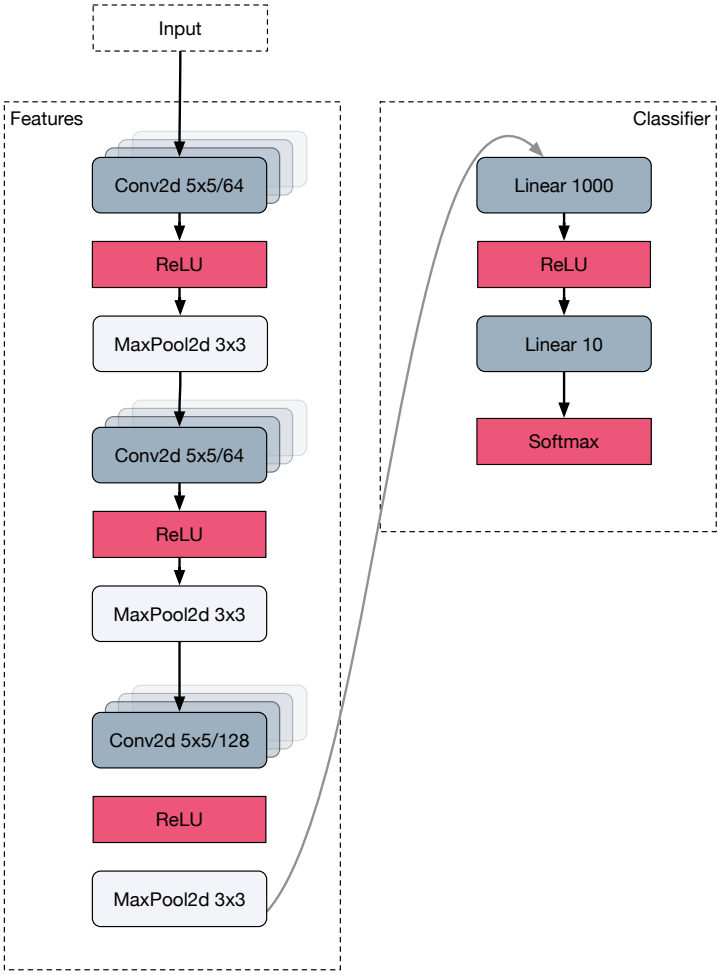


Figure 3.1: Probable Adam model architecture

3.2.2 *Experiments*

The experiments conducted here all use the CIFAR10 dataset learned by a range of models (among them the one presumably used in Kingma and Ba (2014)). The dataset is whitened[1].

The convolutional network (figure 3.1) used by the authors is not unambigously described in the publication, so it cannot be guaranteed that the reproduction is accurate. The experiments here also omit the dropout applied to the input layer (it is unclear whether Kingma and Ba refer to the float-converted input data itself, which would generally be called the input layer to or the initial layer of convolutions) because the network did not learn anything in that case and the dropout probability is not specified. We further perform the experiments on a fully-connected network without convolutional layers (see figure 3.2) as well as the previously introduced VGG architecture (figure 3.14).

We want to employ `ikkuna` to monitor the quantities in Adam's update rule on varous architectures. This will allow us to investigate whether there is a qualitative difference in how Adam operates on different architectures, as well as shed light on its behaviour in cases where it fails to optimise the network. We also track the same

1 Since I did not manage to implement the preprocessing correctly, the dataset used here is from https://github.com/szagoruyko/wide-residual-networks

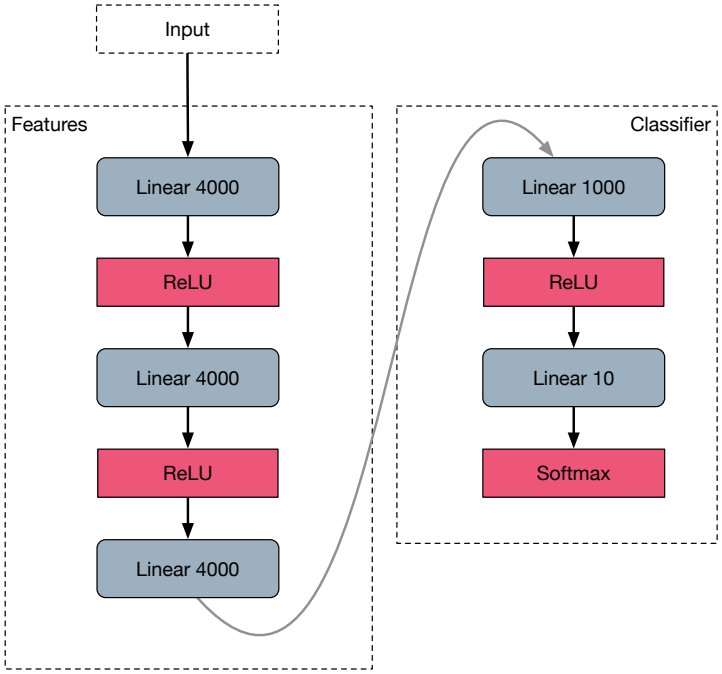


Figure 3.2: Purely dense model

quantites with vanilla SGD for comparison. Additionally, the authors make claims in the publication without accompanying evidence, notably

- Convolutional layers should be assigned a smaller learning rate than linear ones and that—presumably—Adam does this automatically.
- That the update for the experiment with the convolutional network is dominated by the mean estimate and ϵ while the gradient variance estimate vanishes to zero after a few epochs.

Whether these assertions hold true will also be investigated.

It should be noted that the quantities used in equation (3.1) are tensor-valued, i.e. the estimates of the gradient moments are computed for each parameter. Since visualising all parameters individually is not only computationally infeasible, but also useless, some level of granularity with an appropriate summary measure needs to be defined. The next level of organisation above individual layer weights are the weight tensors themselves. Since layers are the basic building blocks, it makes sense to investigate Adam's behaviour on the level of individual layers. The question of how to summarise e.g. the gradient mean estimate for an entire layer remains. Candidates are

1. the average over all units
2. the norm over all units
3. the median over all units

The first option suffers from the fact that the gradient distribution is centered around zero, so the values for most layers empirically average out. The norm of the estimate on the other hand gives a sense of the total magnitude and thus the total change to a layer. The last option is computationally more demanding, but prevents the problem of outliers, while also suffering from the fact that the gradient

distribution is mostly centered around zero. We will therefore plot the norm of the first moment estimate since we are mostly interested in how much total contribution the term has for in the update. For the second moment, we plot the median, since all values are larger than 1 and thus cannot average out. Norm, mean and median will exhibit similar qualitative behaviour for this term anyway, so the choice is less critical. For plotting effective learning rates, we once again report the median, as they vary a lot within a layer and we would like to have some form of average so it fits with our concept of a actual learning rate.

Reproducing the Convnet experiment

As a start, we check whether we can successfully recreate the experiment from (Kingma and Ba, 2014) with the convolutional architecture. On p. 7 they show the performance of the network over the first three epochs and the entire training time. Our reproduction with the exact same parameters is shown in section 3.2.2.

Kingma and Ba report a decrease of the training cost from ~ 2.2 down to around 0.4 over the course of the first three epochs. In our reproduction (figure 3.3a), the loss does not decrease that rapidly, but the trajectory is broadly similar, also when considering the entire training (figure 3.3b). Nevertheless, the original setup achieves training loss of less than 0.001 while our experiment does not improve beyond 0.05. In Kingma and Ba's experiment, the training loss almost saturates after about 20 epochs, the same holds for our reproduction. However, the original result shows the network gradually learning the entire time whereas performance basically stalls in our reproduction after 20 epochs. We can thus not completely reproduces the original experiment. Possible reasons are an incorrectly specified architecture, a different loss function[2], or details of the training procedure which Kingma and Ba do not mention.

We can also see that in our experiment, the claim that

> the second moment estimate v_t vanishes to zeros after a few epochs and is dominated by the ϵ in [the update rule]

does not hold true. Just as in the paper, we set $\epsilon = 10^{-8}$, but the median second moment estimate only vanishes for some layers and increases even for the first convolutional layer. Saying that the update is "dominated" by ϵ may therefore be inaccurate, at least we cannot show this to be the case. It is unclear how Kingma and Ba evaluate the tensor-valued second moment estimate to arrive at this conclusion.

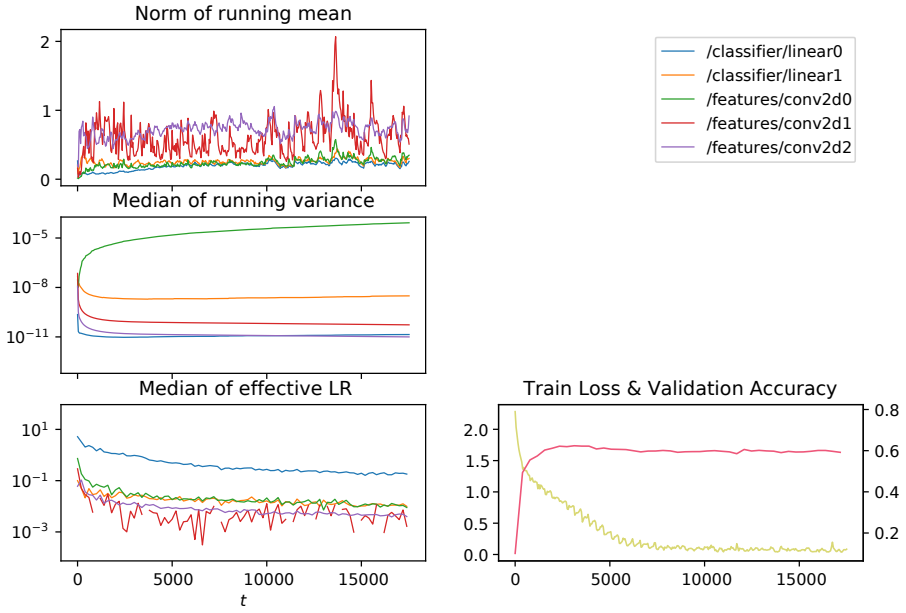
Behaviour of the Adam Update Rule on different Neural Architectures

We will now turn to a qualitative and quantitative evaluation of how Adam performs on fully-connected and convolutional architectures and whether it actually tunes the learning rate like Kingma and Ba (2014) suggest.

Figure 3.4 shows the evolution of Adam's terms and the effective learning rate for different base learning rates. We can see that, surprisingly, Adam appears very sensitive to the base learning rate, despite the fact that it is supposed to automatically tune it. It is curious that Adam performs best with the smallest learning rate and gradually worse with larger ones. We find that in the best-performing case, the second moment estimate does not actually vanish toward zero. To the contrary,

2 This is very unlikely, as the categorical crossentropy is almost always used for classification tasks, and also the absolute values are rather similar.

(a) Adam convnet with learning rate 0.001, first three epochs



(b) Adam convnet with learning rate 0.001, entire training

gradient variance gradually increases for all layers througout training. Moreover, the per-layer learning rates used by Adam for the smaller two learning rates are quite similar (beware of the log-scale), but still a large performance gap exists. Their observation that the second moment estimate vanishes leads Kingma and Ba to the hypothesis that

> The second moment estimate is therefore a poor approximation to the geometry of the cost function in CNNs comparing to fully connected network [...].

and we see here that there may be a point to this and a vanishing second moment is a bad sign for the learning process. The best-performing learning rate is the one with second moments most strongly diverging from zero. This could mean it is a useful metric to monitor during training and anneal the learning rate accordingly. The Adam update rule uses the second moment to do precisely this, so we can draw the conclusion that the learning rate schedule automatically instituted by a significant gradient variance is superior to one where the variance stagnates. It is not clear how to derive a "learning-rate-appropriateness metric" from this finding, but it is a hint that the running gradient variance could play a role in such a metric.

If we consider the SGD algorithm instead (figure 3.6), we can see however, that the variance estimate does not vanish in any of the cases, but instead approaches its asymptote more quickly with increasing learning rate. This paired with the insights from figure 3.4 hints at the importance of increasing gradient variances in some fashion througout training. Possibly, an increase in variance is correlated with layers becoming more differentiated and sensitive to more patterns and is thus indicative of learning progress. It would be interesting to perform these same experiments with batch normalisation applied between layers to see whether their influence—which should counteract any variance increase—invalidates this hypothesis.

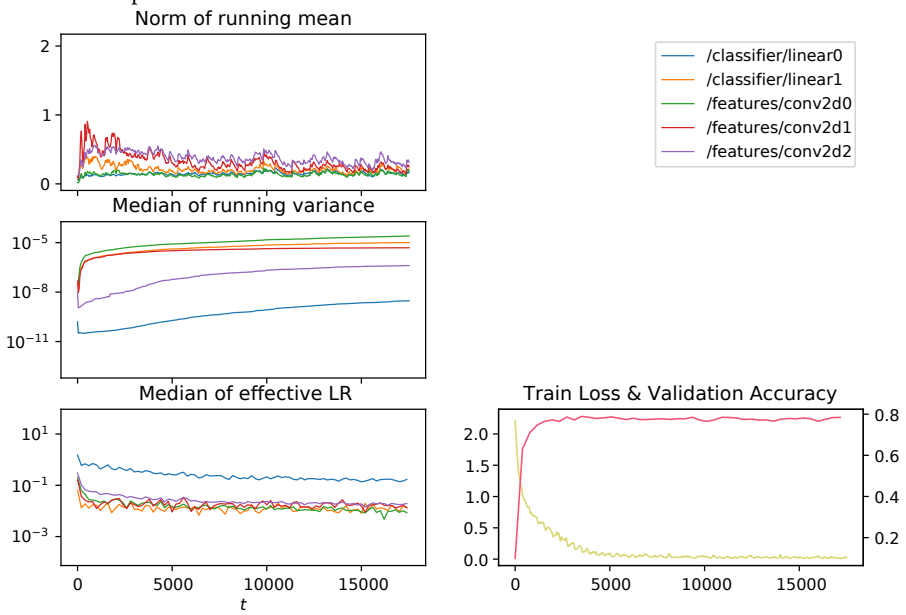
The experiments above were run on the fully connected model as well (see figure 3.2). The results are shown in figure 3.8. The first thing we notice is that Adam behaves very differently in the absence of convolutional layers. Whereas we saw a difference in performance between learning rates of 0.0005 and 0.001 on the Convnet, the fully connected model behaves identically in both cases. Kingma and Ba state that there is a qualitative difference between the gradients of dense and convolutional layers, and we see this fact demonstrated here, as the learning rate does not seem to affect the gradient distributions much, in contrast with the previous experiment.

Similarly to earlier findings, we see again that Adam is in fact quite sensitive to the base learning rate set. Once again, a value of one order of magnitude beyond the best one makes performance deteriorate. In this case, the first moment estimate vanishes completely, leading to zero-valued gradients and no learning at all.

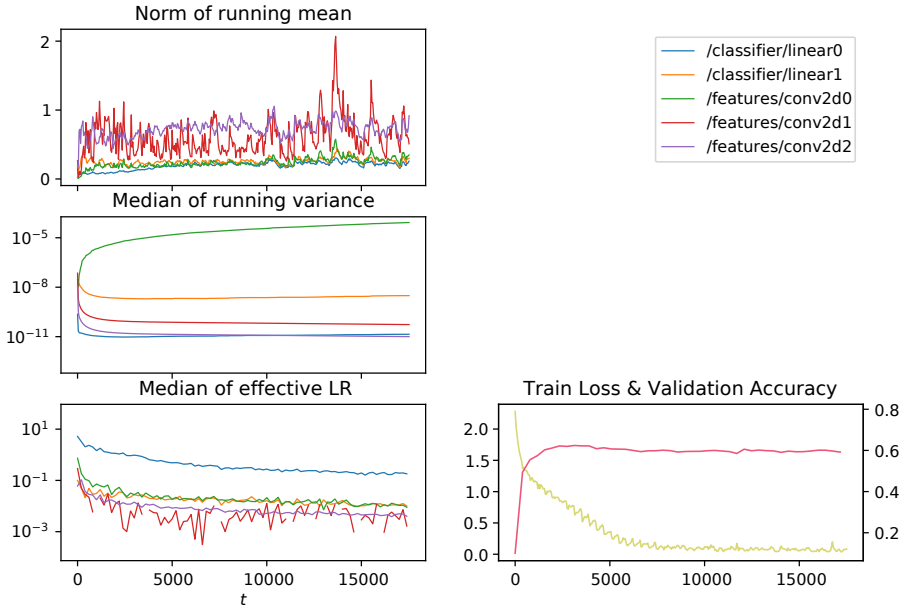
One question not answered yet is section 3.2.2. From figure 3.1 and figure 3.8 we clearly see that Adam does in fact tend to assigning higher learning rates to dense layers and lower ones to convolutions. If this heuristic were to be used for manually tuning SGD, it would indeed be automatically handled by Adam. Ignoring the final classification layer (`/classifier/linear1`) in the Convnet, this seems to come from the fact that dense layers exhibit a gradient variance several orders of magnitude smaller, while the gradient mean—while still being smaller—differs by less than one order of magnitude.

We omit here the results for the fully connected model trained with SGD (because it learns only for the largest learning rate), as well as the larger VGG16 convolutional

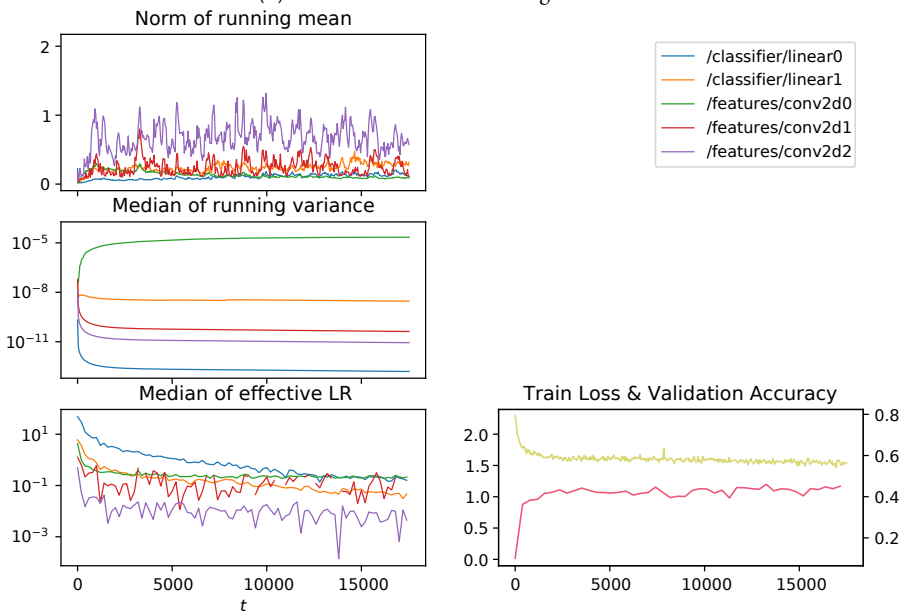
Figure 3.4: Adam metrics on the Adam-Convnet with different learning rates and Adam optimiser



(a) Adam convnet with learning rate 0.0005

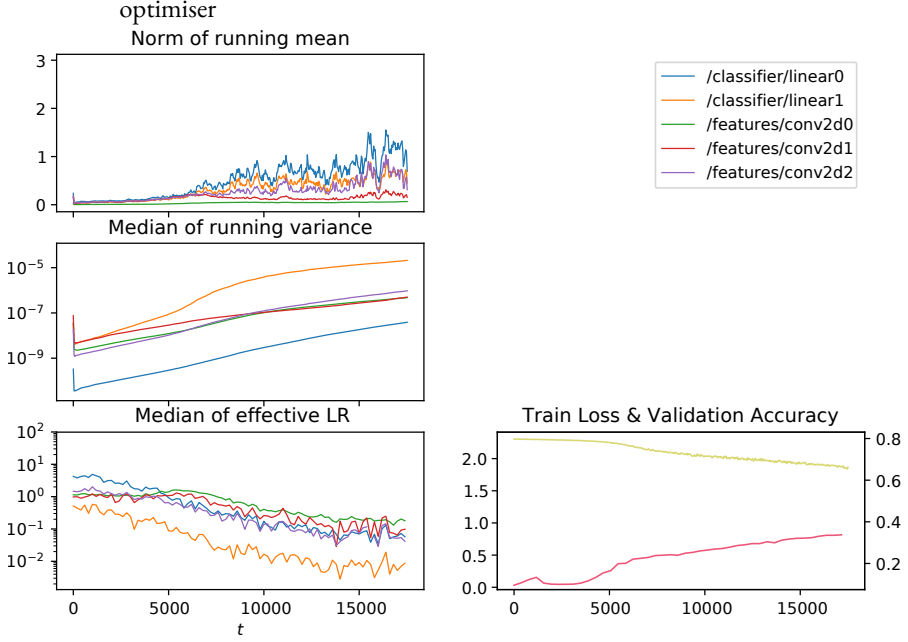


(b) Adam convnet with learning rate 0.001

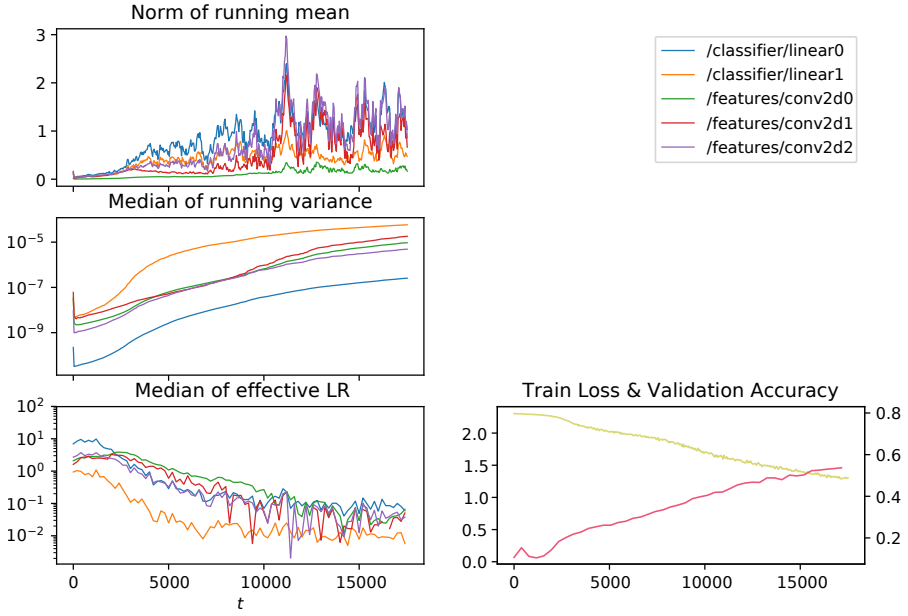


(c) Adam convnet with learning rate 0.01

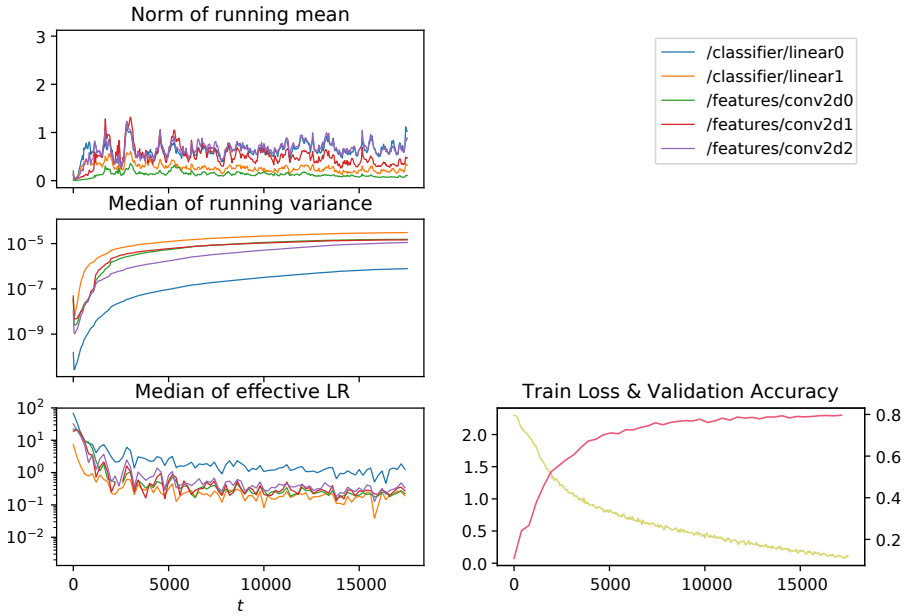
Figure 3.6: Adam metrics on the Adam-Convnet with different learning rates and SGD optimiser



(a) Adam convnet with learning rate 0.0005, optimised with SGD

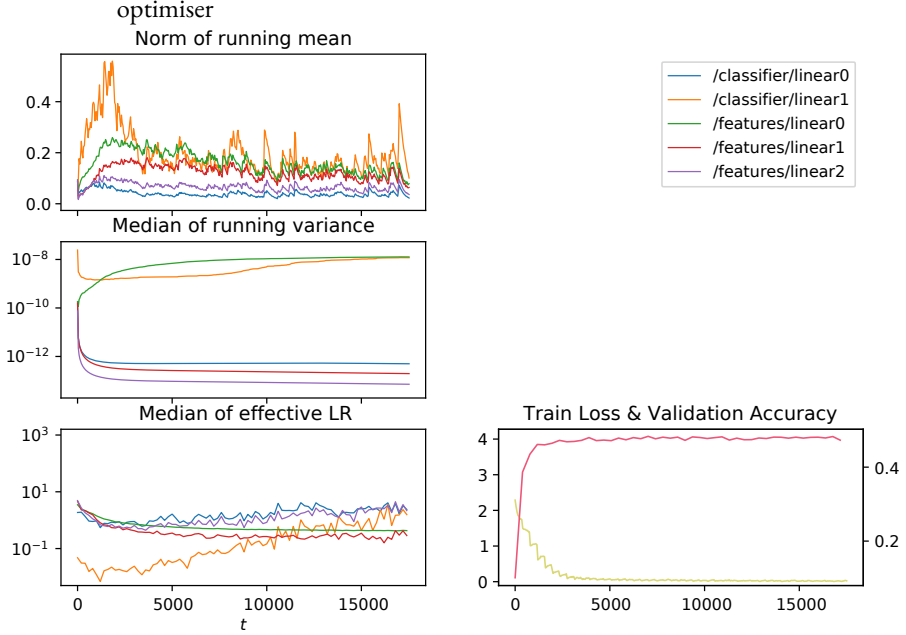


(b) Adam convnet with learning rate 0.001, optimised with SGD

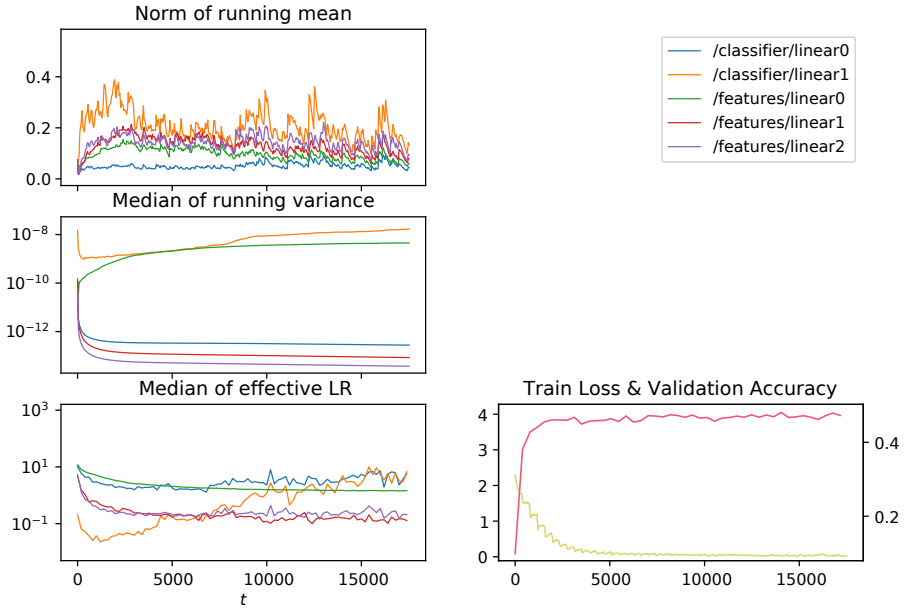


(c) Adam convnet with learning rate 0.01, optimised with SGD

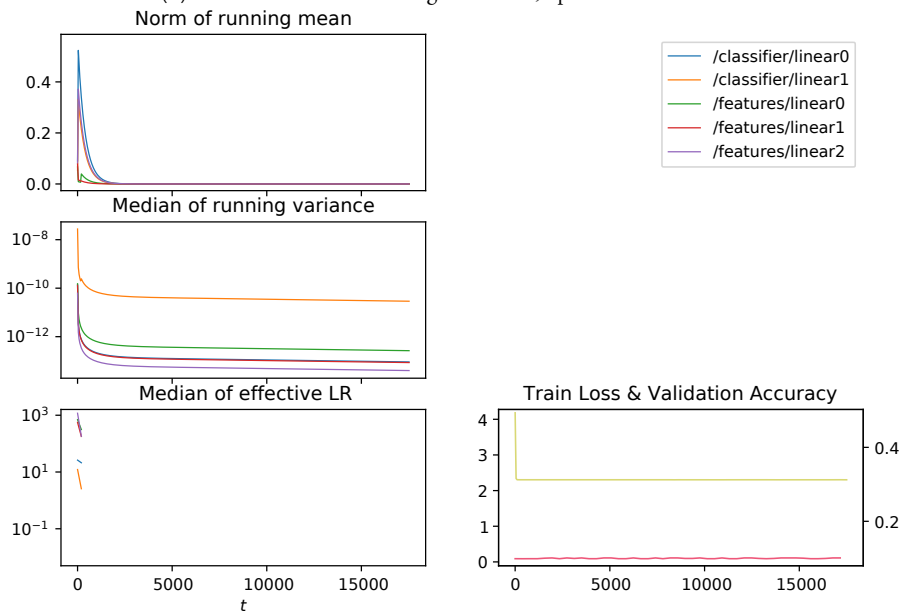
Figure 3.8: Adam metrics on fully linear model with different learning rates and Adam optimiser



(a) Dense model with learning rate 0.0005, optimised with Adam



(b) Dense model with learning rate 0.001, optimised with Adam



(c) Dense model with learning rate 0.01, optimised with Adam

architecture trained with both optimisers, since the latter appears to be invariant to any change and may simply be too powerful to exhibit large differences.

3.2.3 *Summary*

With the experiments performed here, we have evaluated the qualitative behaviour of the Adam optimisation algorithm compared to vanilla stochastic gradient descent on three architectures, with several learning rates. We find that Adam is—in spite of its promise—very sensitive to the learning rate and the correct order of magnitude must be determined beforehand. We have validated that it automatically assigns smaller learning rates to convolutional layers than to dense once, a finding in line with a popular heuristic employed manually for SGD. We also examined the behaviour of the second moment estimate and find that it appears to be related to the speed of learning, albeit not in a trivial fashion. A further interesting phenomenon is that median learning rates for individual layers can range from 10^{-3} all the way up to 10, which would never work when applied to an entire network. More work would be necessary to check whether using these learning rates for SGD would achieve the same performance. If not, then this would prove that Adam finds qualitatively different paths through the weight space by incorporating past gradient information. Reason to believe that this is indeed the case is supported by Morcos et al. (2018) who find that the representations learned by the same network with different learning rates are qualitatively different.

The `ikkuna` library was used to extract all metrics during training and log them to a database. While code for the experimental setup had to be written, the code for computing and logging the metrics is simple and short, and only needs to be written once. This software could have been used to easily perform more experiments than Kingma and Ba (2014) provided.

3.3 DETECTING LEARNING RATE PROBLEMS

In this section, we will investigate a new way to automatically detect learning rate problems based on a statement made by Andrej Karpathy. We will discuss the validity of his recommendation by tracking the proposed Update-to-Weight-Ratio in a variety of scenarios and implementing a learning rate schedule based on it. The question to be answered here is whether a metric based on the ratio can be used for automatically adjusting the learning rate, or—if unsuccessful—for detecting pathological states during training.

3.3.1 *Ratio-Adaptive Learning Rate Scheduling*

We investigate a claim made by Karpathy (2015) who states that the ratio between updates and weights $\frac{||\Delta W_i||}{||W_i||}$, is a quantity which should be monitored and constrained. He suggests a target of $\frac{1}{t} = 10^{-3}$ as a reasonable value, but to the author's knowledge there has never been a thorough investigation of this hypothesis. Because of this lack of exploration and the celebrity of the proponent, this merits further investigation.

It seems intuitive that this target cannot be static throughout training, since we usually decay the learning rate towards the end, leading to smaller magnitude of updates (this is a prerequisite for theoretical convergence guarantees for SGD, see

Saad (1998, p. 20)). As a matter of fact, this can be easily verified by running training with an update rule that scales each gradient so that the update hits the target exactly (the network does learn, but to a significantly smaller final accuracy; see figure 3.10). It is also not clear whether the target should apply to all layers equally, since this would constitute a strong regulariser on the network parameters, limiting expressiveness of the model. We also verified in section 3.2 that convolutional layers should use a smaller learning rate then linear ones.

For another instance, in networks with the (now outdated) tanh activation function, the later layers' gradients are larger as fewer backpropagation steps have been done to them, and for this activation function at least, each application of the chain rule entails multiplication with a factor < 1 (as the tanh derivative never exceeds 1 and decreases to almost 0 in both limits), so gradients become exponentially small. In fact, the vanishing gradient problem was the main driver to introduce non-saturating activation functions. Other nonlinearites such as the rectified linear unit do not exhibit the same behaviour, but that does not mean all layers must change at the same rate.

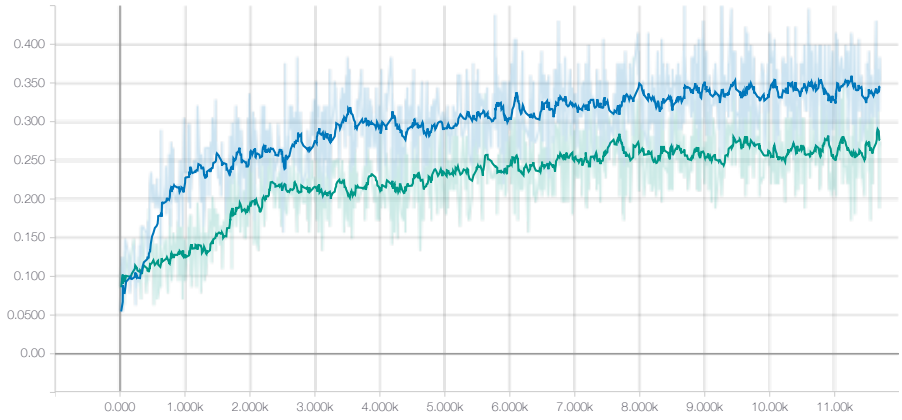


Figure 3.10: AlexNetMini on CIFAR10 at 0.1 learning rate. The blue line is vanilla SGD while the green one is obtained by scaling the gradients to have the updates meet a target ratio of 10^{-3}.

As a first approach to an evaluation, we want to try and select the learning rate in such a way as to hit the target postulated by Karpathy, but not precisely for each layer, but on average over all layers. This would allow for more flexibility in the weight updates compared to fixing each update to the same value.

Table 3.1: Hyperparameters for the fixed-ratio experiment

Parameter	Value
Architecture	Minified AlexNet (figure 3.11)
Optimiser	Vanilla SGD
Epochs	100
Dataset	CIFAR10[3]
Batch size	128
Learning rate	0.2

The settings for this experiment are listed in table 3.1. The dataset does not constitute a hard problem to solve; state of the art accuracies lie around 95%. For this reason, a decision must be made about how to make the problem hard enough

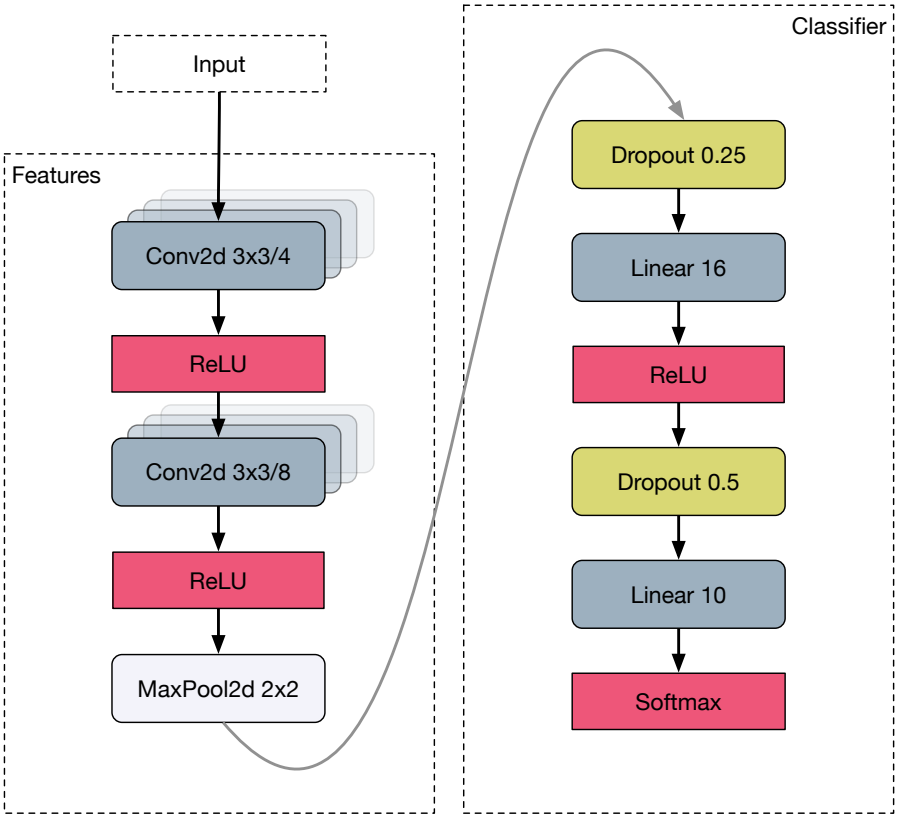


Figure 3.11: The network used in this experiment. Image features are extracted by 4 and 8 convolutional filters, respectively, with ReLU nonlinearities. Maximum pooling is applied with a filter and stride size of 2 leading to a resolution a fourth of the original size. The classifier portion employs dropout layers to reduce co-adaptation of units and a final softmax activation to map outputs to class probabilities in $(0, 1)$.

so that improvements to the training schedule can actually be made. The learning rate has thus been fixed to a high value of 0.2 which is not the optimal value (a learning rate of 0.1 solves the problem to a better 45% accuracy).

In order to validate that there is room for improvement (i.e. the task is not too easy), the training has been run about twenty times for both a constant learning rate and an exponentially decaying rate according to

$$\eta_{e+1} = 0.98^{e+1}\eta_e, \tag{3.2}$$

e being the epoch index. The final accuracies after 100 epochs of training for constant learning rate, ratio-adaptive learning rate and a control condition of exponential decay are shown in figure 3.12. As can be seen, there is a significant improvement when decaying the learning rate at a commonly used exponential schedule, meaning the problem constructed here has lots of room for improvement. We also show results for adapting the learning according to the update-to-weight-ratio which will be introduced in the next section 3.3.1.

The Adaptive Update-to-Weight-Ratio Schedule

As a first showcase of the library and a test of the update-weight ratio hypothesis, an adaptive learning rate based on the aforementioned ratio is implemented with

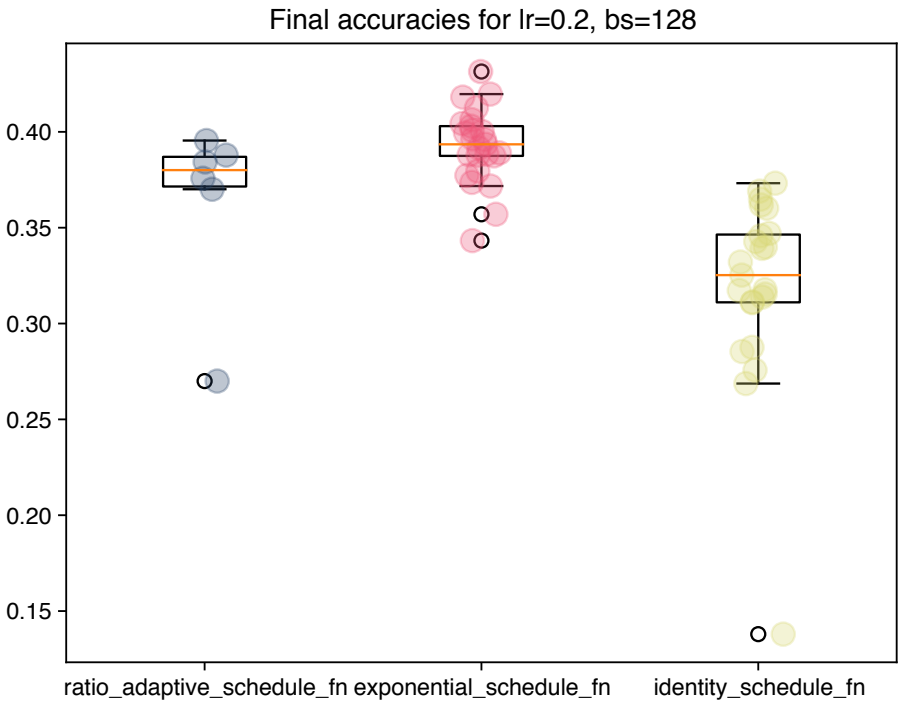


Figure 3.12: Final accuracies after 100 epochs with a learning rate of 0.2 and batch size of 128. The adaptive ratio schedule is discussed in section 3.3.1. The exponential schedule performs slightly better than the adaptive one, but also exhibits greater variance in its results. This may be caused by it not being adaptive to the problem at hand. We thus surmise that using the ratio-based schedule is worth investigating.

the help of the library. We will start by formally describing the update rule and then show how it is implemented with `ikkuna`.

Let l be the number of layers with weight matrices associated with them (for instance linear or convolutional layers, but not activation functions, dropout, or the like). Let $\{W_{i,k} \mid i = 0 \dots l-1\}$ be the set of weight matrices at training step k. Let η be the base learning rate and $\frac{1}{t}$ be a target value to which we want the update-to-weigh ratio to move. Furthermore, let $\gamma \in (0, 1)$ be a decay factor for exponential smoothing. Now, let

$$R_{i,k} = \frac{\|W_{i,k} - W_{i,k-1}\|_2}{\|W_{i,k}\|_2} \tag{3.3}$$

be the ratio between the L2-Norms of layer i's weight updates before step k step and the weights at step k themselves. We then select the new learning rate for batch step $k+1$ as

$$\eta_{k+1} = \eta_k \left(t \cdot \frac{1}{l} \sum_{i=0}^{l-1} \gamma R_k + (1-\gamma) R_{k-1} \right)^{-1} \tag{3.4}$$

for $k \geqslant 2$. This is the average exponentially smoothed update-weight-ratio, divided by the target range. This learning rate is used for vanilla gradient descent without any other modifications beyond capping it to some value in case of very small ratios. The effect of adapting the learning rate according to this schedule is that the average ratio between the weight updates and the weights moves towards the target

range. It should be noted that this update rule biases the learning rate in favour of the smaller layers since all ratios are weighted equally, regardless of the number of weights. Figure 3.13 displays a set of accuracy traces for each of the schedules (constant, exponential decay, ratio-adaptive) with different base learning rates. The network was trained from scratch 5 times for each combination.

The results are not overwhelming, which is unsurprising for such a simple schedule. For the smaller learning rates, it is not better or worse than a constant or exponentially decaying schedule. However, for unnecessarily high learning rates, the adaptive schedule outperforms the constant one, hinting at a possible signal for identifying too high learning rates. This holds for the smaller of the three batch sizes, which makes sense as a high batch size is generally more amenable to a high learning rate, as the larger sample size reduces the noise in the gradient and makes for a smoother loss landscape as the gradients for more samples are averaged. On the other hand, the adaptive schedule is also better than a constant one for very small learning rates on large batch sizes. So it not only works in preventing too-high learning rates, but also too low ones. This signal could hence be useful for identifying inappropriate learning rates in small or large batch sizes.

As an impression of how the library presented in chapter 2 simplifies a general implementation of such a learning rate schedule, code is provided here.

When an `Exporter` is configured for a given model, a `RatioSubscriber` (see table 2.3) must be added to the message bus in order for the update-weight-ratio ($R_{i,k}$ in the above equations) to be published. One can then subscribe them and process the information with this subscriber:

```
class RatioLRSubscriber(PlotSubscriber):
    def __init__(self, base_lr, smoothing=0.9, target=1e-3,
        max_factor=500):
        subscription = Subscription(self, ['
            weight_updates_weights_ratio', 'batch_started'],
                                    tag=None, subsample=1)
        super().__init__([subscription], get_default_bus(),
                         {'title': 'learning_rate',
                          'ylabel': 'Adjusted learning rate',
                          'ylims': None,
                          'xlabel': 'Train step'})
        # exponential moving avg of R_{i,k}
        self._ratios     = defaultdict(float)
        # maximum multiplier for base learning rate (in pathological
            cases)
        self._max_factor = max_factor
        # exp smoothing factor
        self._smoothing  = smoothing
        # target ratio
        self._target     = target
        # this factor is always returned to the learning rate
            scheduler
        self._factor     = 1
        self._base_lr    = base_lr

    def _compute_lr_multiplier(self):
        '''Compute learning rate multiplicative. Will output 1 for
            the first batch since no layer
        ratios have been recorded yet. Will also output 1 if the
            average ratio is close to 0.
        Will clip the factor to some max limit'''

        n_layers = len(self._ratios)
```

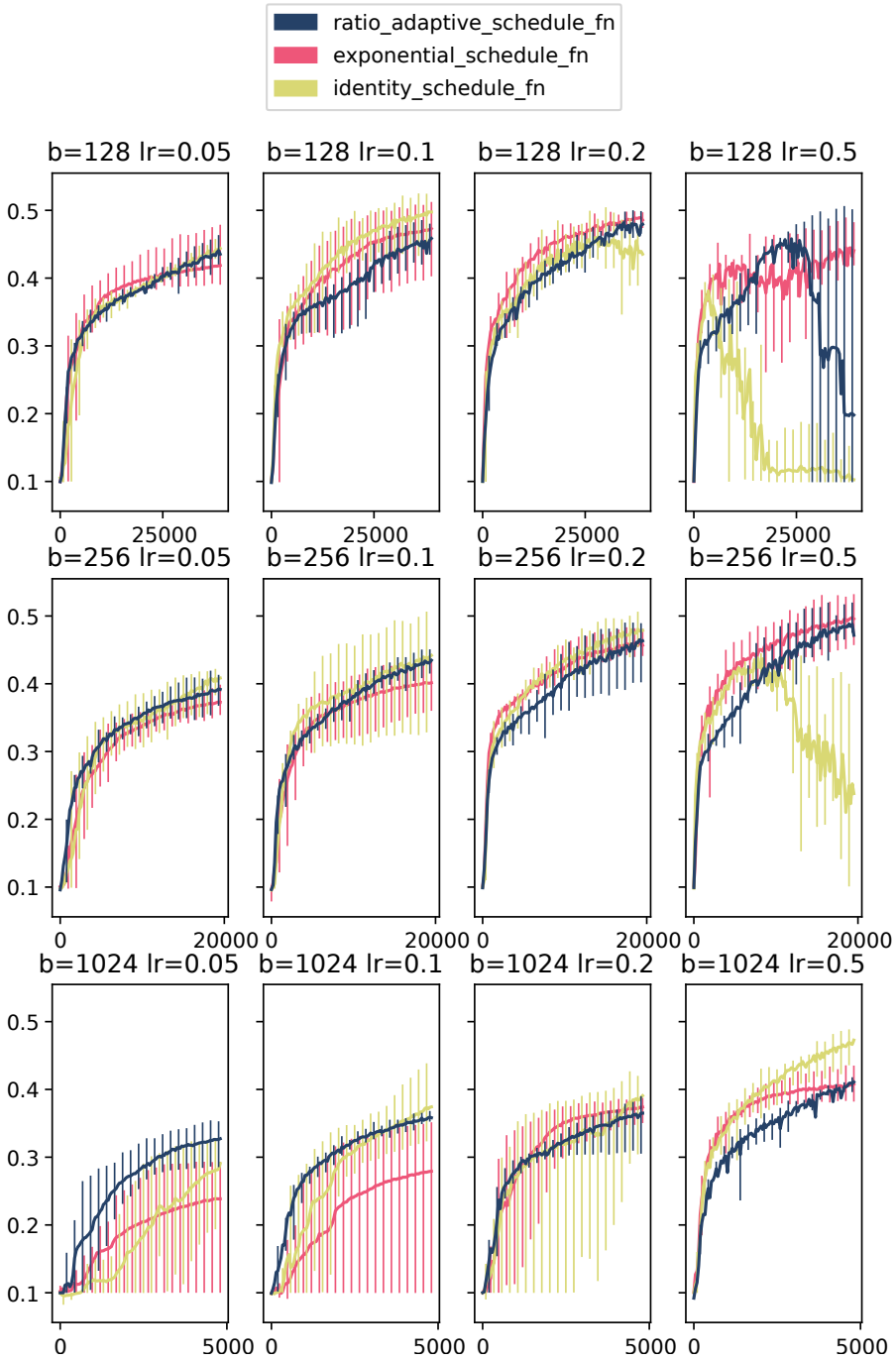


Figure 3.13: Accuracy traces for different schedules on CIFAR10. Error bars show the minimal and maximal values over all runs. Batch size and initial learning rate are shown above each subplot. The exponential decay schedule uses a decay factor of 0.98. Chance level is 0.1. The experiments are run for 100 epochs each.

```
        if n_layers == 0:   # before first batch
            return 1
        else:
            mean_ratio = sum(ratio for ratio in self._ratios.values()
                ) / n_layers
            # prevent numerical issues and keep current LR in that
                case
            if mean_ratio <=  1e-9:
                return 1
            else:
                factor = self._target / mean_ratio
                return min(factor, self._max_factor)

    # invoked by the runtime for each incoming message
    def compute(self, message):
        if message.kind == 'weight_updates_weights_ratio':
            # the 'key' property for these messages will be the
                module/layer
            # here we compute the exponential moving average of
                ratios
            i               = message.key
            R_ik            = message.data
            R_ik_1          = self._ratios[i]
            gamma           = self._smoothing
            self._ratios[i] = gamma * R_ik + (1 - gamma) * R_ik_1
        elif message.kind == 'batch_started':
            # before a batch starts, update the lr multiplier
            self._factor = self._compute_lr_multiplier()

    def __call__(self, epoch):
        return self._factor
```

Listing 3.3: Ratio-Based LR subscriber and scheduler

The subscriber implements the `__call__()` method so it can be dropped into PyTorch's learning rate scheduler (`torch.optim.lr_scheduler.LambdaLR`). This learning rate schedule can thus be used in every model, without modification.

3.3.2 *Effects Of Update-to-Weight-Ratio On Training Loss*

We have seen in the previous section that at least for pathological cases the UW ratio can be used to correct the learning rate to some extent. In this section, we want to examine how this ratio does or should change during training. As discussed in section 3.3.1, it is unlikely that a constantly high rate of change to the weights will be beneficial throughout the entire training. We would therefore like to find a relation between the loss decrease, the current UW ratio and the point in time during training. This could help us improve the learning rate schedule developed above and refine the use of the UW ratio as a signal for inappropriate learning rates.

For this experiment, we learn CIFAR-10 for 75 epochs, again with a batch size of 128, with vanilla SGD and the Adam optimizer and different learning rates. We use the AlexNetMini architecture again, as well as a larger, more powerful VGG network (schema in figure 3.14). We employ `ikkuna` to record losses, accuracies and UW ratios for each layer automatically during training. In order to make larger trends visible, we smooth the loss trace with a gaussian kernel.

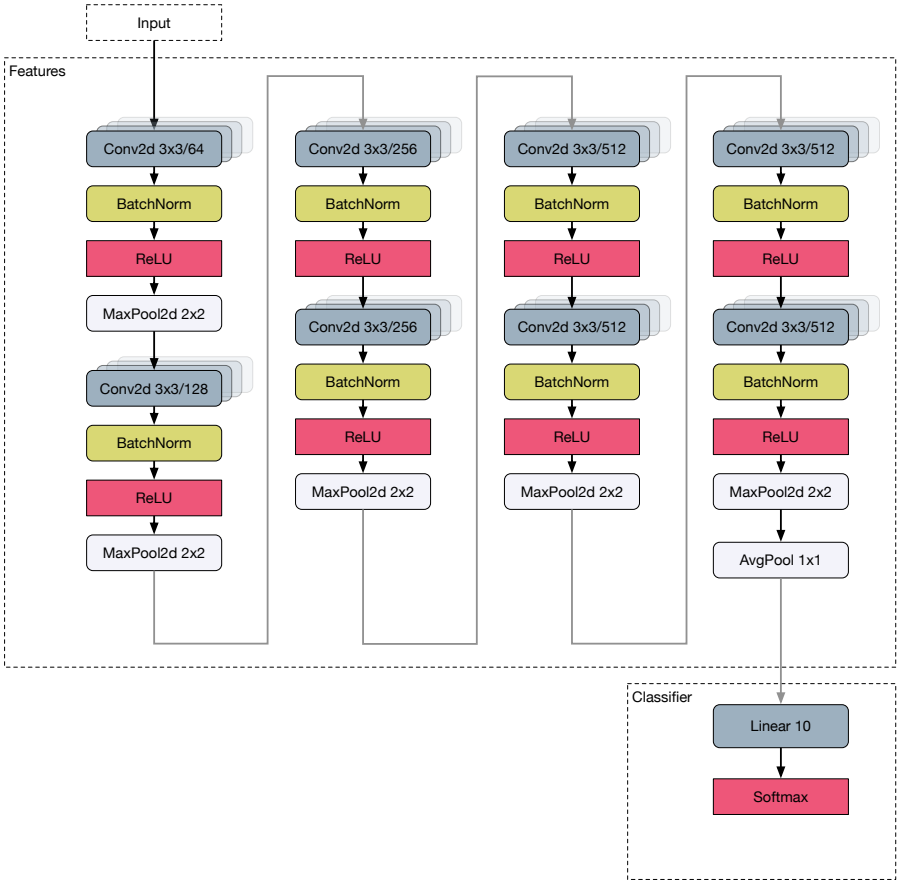


Figure 3.14: VGG network with 8 convolutional layers.

In the following figures, we plot the loss traces and their negative derivative[4] (the amount of decrease) in the upper right, the average weight-update ratio in the lower right, and a scatter plot of ratio versus loss decrease on the left. The scatter plot has the time step colour-coded (blue is early, red is late) and in addition offsets according to time on the z-axis. Due to the number of points (up to 30,000 time steps times the number of runs), the plots have been subsampled where many points overlap. This was necessary to render the data for this document. Care has been taken to not destroy the topology of the data. The density in a given area may thus not be entirely accurate. In the UW ratio plots, the line displayed is the average over multiple runs.

VGG with Stochastic Gradient Descent

The plots in for this network only show the first 10,000 training steps since nothing of note happens afterwards. Since most of the ratio values cluster around the same values towards the end of training, logarithmic subsampling was applied. For the VGG network, we observe a smooth decrease in the training loss alongside a decrease in the average update-to-weight ratio. Training basically stalls after around 10,000 steps (about 25 epochs). We observe the trend that the UW ratio is initially fairly high and falls off subsequently, which correlates with a decrease in loss. However, there is no particular value of the ratio that exhibits any significant correlations beyond other values. The ratio in the beginning of training is also proportional to the learning rate, as is to be expected.

4 Erratum: The plots incorrectly refer to the “inverse“ derivative.

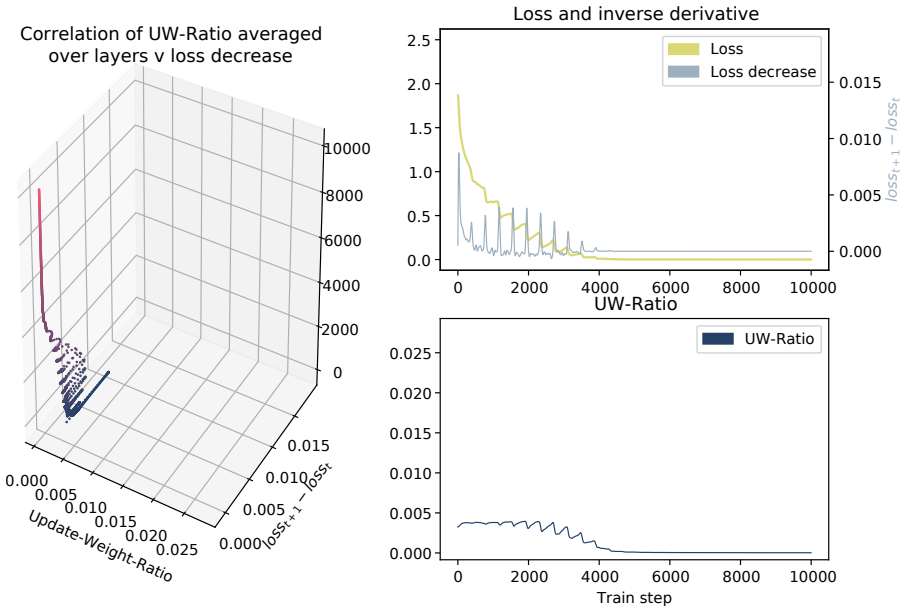
It is curious that in the very beginning of training, the UW ratio has the highest value, but the decrease in loss for these steps increases throughout the first few batches (i.e. learning accelerates) before tapering off. This is barely visible in the line plots, but shows as a prominent feature in the scatter plot. It should be noted that this phenomenon happens on a very small timescale—the number of data points in the arc is orders of magnitude smaller than in the rest of the plot, therefore it is no more than a curiosity. As a preliminary conclusion, we can affirm that there is no linear correlation between the ratio and the decrease in loss.

Furthermore, the smallest of the evaluated learning rates converges fastest. The loss flatlines at 0 after little more than $4,000$ steps, while the larger learning rates need proportionally more time. The difference is marginal however. The phenomenon might relate to the motivation for annealing the learning rate: As we approach a local minimum, we need smaller learning rates to not jump over it in a different direction in every update, but slowly fall into the minimum itself instead. This may be an indication that an appropriate learning rate for this particular problem is $\leqslant 0.01$. None of the configurations exhibit UW ratios close to Karpathy's suggestion of 10^{-3}, but with the smallest learning rate, we get closest, and converge fastest. Perhaps this could motivate Karpathy's constant.

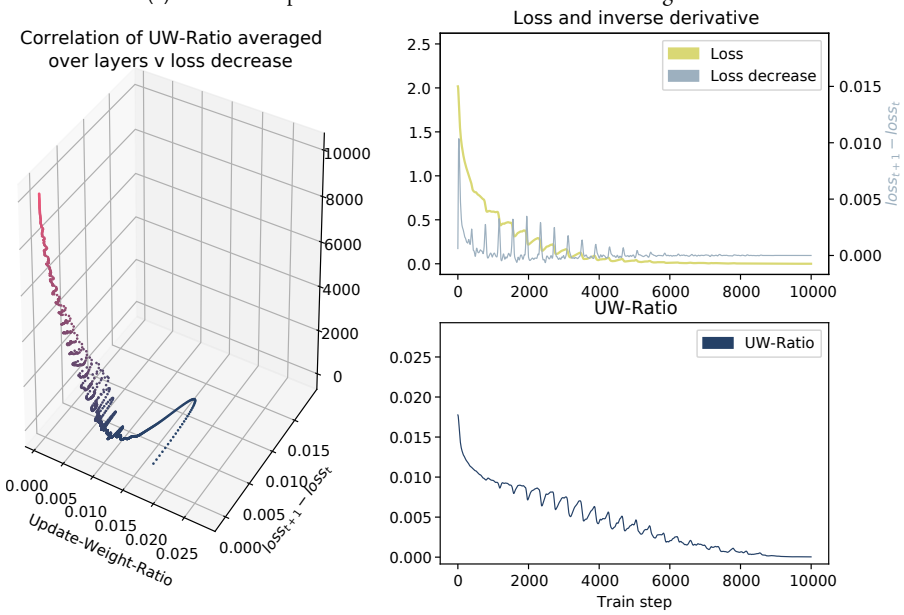
VGG with Adam

The same experiment has been run with the Adam optimizer (Kingma and Ba, 2014) and has been found to produce quite different results. We mainly observe that the steplike nature of the loss function (sharp decreases at epoch boundaries) are much less pronounced with the Adam optimizer. This removes the loops which we see for SGD from the scatterplots. In absolute terms, Adam begets a much higher UW ratio in the beginning of training, which falls off quickly to SGD's values. Convergence takes minimally longer and doesn't occur at all with a learning rate of 0.1 (figure 3.16c). Counterintuitively, the adaptive optimiser is unable to adapt to the high learning rate and is outperformed by vanilla SGD. The Adam optimiser keeps running estimates of the mean and variance of the gradient of each parameter, with exponential smoothing applied. The learning rate is adjusted for each parameter by multiplying it with $\frac{\mu_{\text{grad}}}{\sqrt{\sigma_{\text{grad}}}+\epsilon}$. The argument by Kingma and Ba (2014) is that in locations of high gradient variance, a lower learning rate is appropriate since the estimate of the gradient at this spot is noisy and unreliable, so smaller steps are a safer choice. Furthermore, they claim that the mean of the gradient vanishes closer to an optimum, which is also where we need to anneal the learning rate lest we jump over the minimum. The Adam optimiser thus has learning rate annealing built-in. The step size parameter η is supposed to be of less importance since in Adam's parameter update rule, it gives an upper bound on the size of the steps taken in parameter space, but it can always be adjusted downward if the gradient variance is too high. It can also be adjusted upward up to the upper bound if the gradient mean is high, giving Adam momentum-like behaviour. The primary proposition of Kingma and Ba (2014) is that manually tuning the learning rate per layer or even per parameter is no longer necessary, but we see here that cases can be found where precisely this behaviour of Adam would be needed—we have set a too-high learning rate, but Adam fails to compensate for this negligence. This casts doubt on the universality of the Adam optimiser.[5]

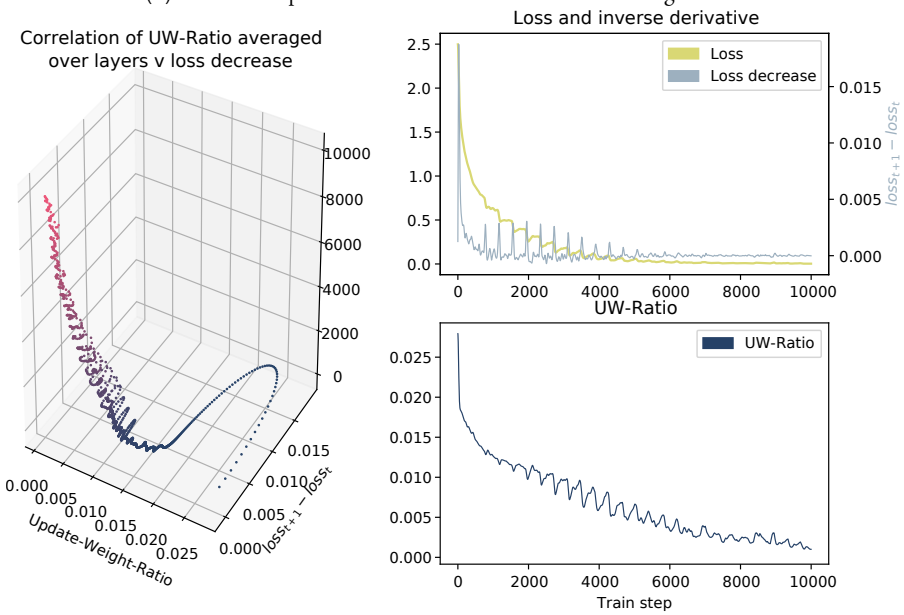
5 The hypothesis that Adam starts to show its merit in the absence of batch normalisation could not be tested on the VGG architecture, since training did not progress at all with any learning rate or



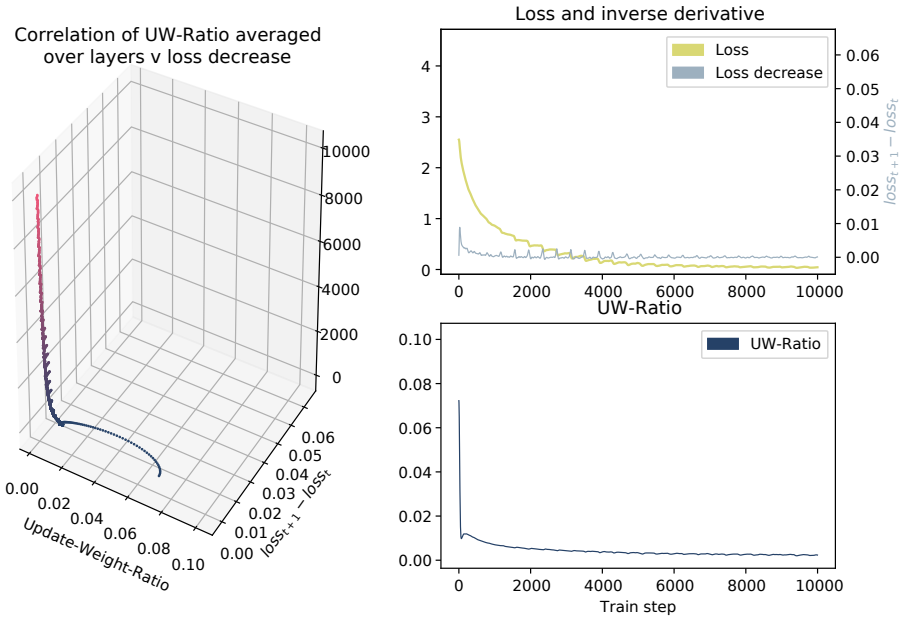
(a) UW ratio experiment for VGG with SGD and learning rate 0.01



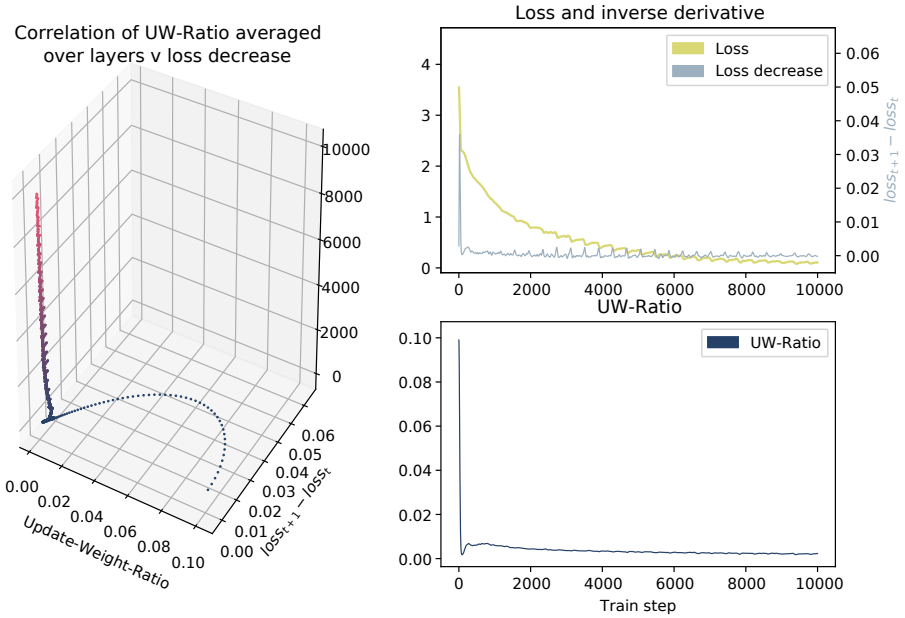
(b) UW ratio experiment for VGG with SGD and learning rate 0.05



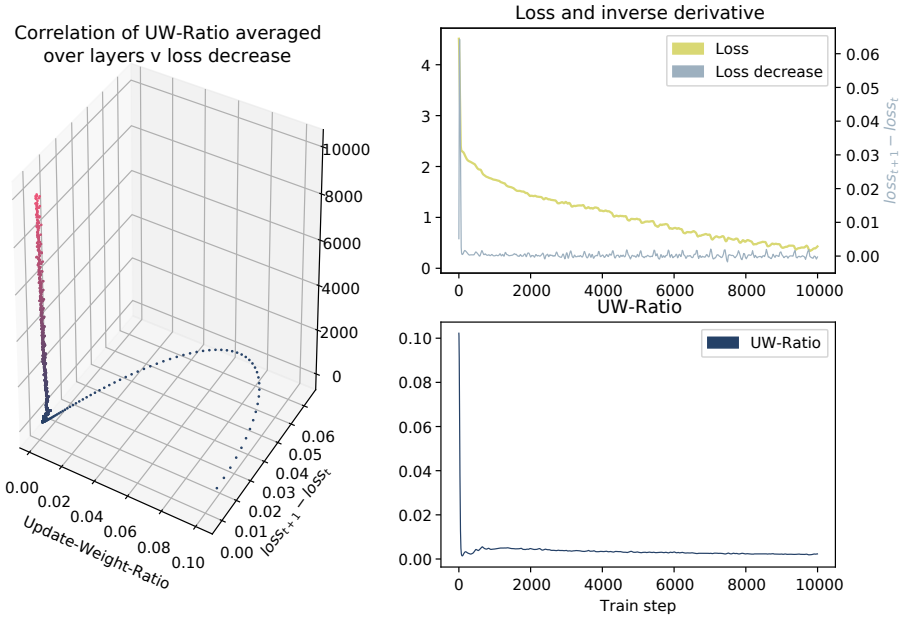
(c) UW ratio experiment for VGG with SGD and learning rate 0.1



(a) UW ratio experiment for VGG with Adam and learning rate 0.01



(b) UW ratio experiment for VGG with Adam and learning rate 0.05



(c) UW ratio experiment for VGG with Adam and learning rate 0.1

Remarkably, the qualitative behaviour is the opposite of SGD, which becomes visible when omitting the first few hundred training steps. While for SGD (plots omitted for brevity), higher learning rates lead to higher initial values in the UW distribution, the opposite seems to hold for the adaptive optimizer. This can be seen in figures 3.17a to 3.17c where the first 500 training steps are omitted to avoid squishing the scatter plot because of the quickly decaying arcs in the beginning of training. A possible explanation is that Adam uses estimates of the mean and variance of gradients, and possibly overcorrects the learning rate.

Other Architectures

Several analyses were conducted on the data, but not displayed here due to inconclusive results. The experiments were run on the AlexNetMini architecture as well, without results, as the network does not converge within 75 epochs. The same experiment on the ResNet18 architecture exhibited the same qualitative behaviour as the VGG network, except for a larger variance in loss decreases for a given ratio value. Slicing the training into pieces and graphing every slice of 5,000 training steps separately reveals nothing beyond a decrease in absolute value of ratio and loss, which is expected. Any correlations between the two quantities do not change after the initial few steps.

Summary

The results from all these experiments do not demonstrate that a ratio value of 0.001 is any more significant than others. While it is intuitive that the loss can only decrease significantly if the weights change accordingly (unless the loss surface is extremely rough), we find no particular insights as to what ratio is appropriate for any given point in training.

Fixing The Learning Rate for Karpathy's Constant

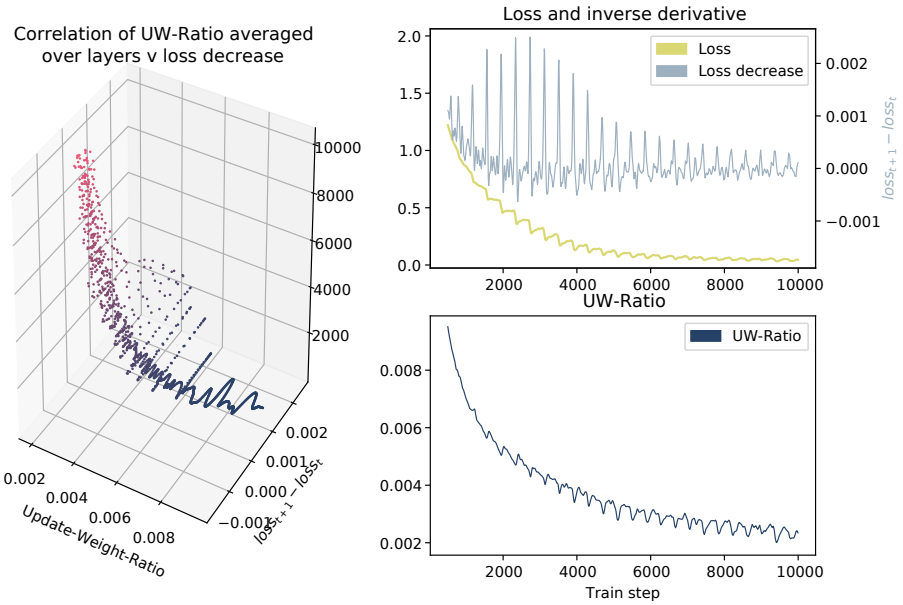
We now want to check how the UW ratio relates to the loss when we attempt to artificially fix the ratio value to a target of 0.001. Since the updates to the weights in vanilla SGD (see section 1.2) are a function of the learning rate and the gradient magnitude, this could be done in one of two ways:

1. Scale the gradients with the constant learning rate in mind
2. Set the learning rate according to the ratio-adaptive schedule from section 3.3.1

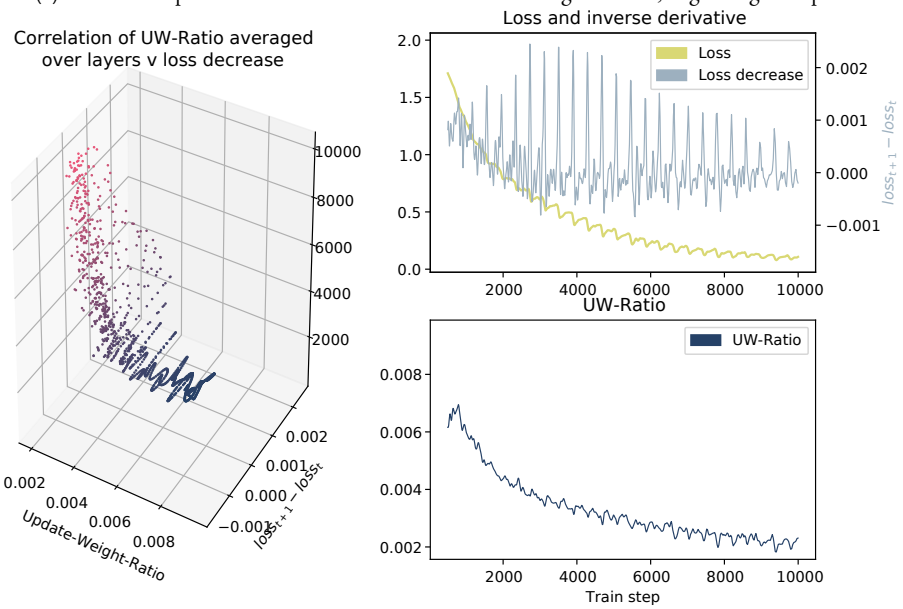
Using the ratio-adaptive schedule results in the behaviour shown in figures 3.18a to 3.18c. In contrast to previous experiments, we also have a configuration with 0.001 learning rate, dropping the 0.05 value. We also plot the learning rate caused by the scheduling over the UW ratio.

Note that for this set of plots, the axis limits are different. This serves to illustrate that the qualitative behaviour is identical for all learning rates (ignoring the fast arc at the beginning for $\mathrm{lr} = 0.1$). The learning rate trajectories are very similar; the learning rate rises as the UW ratio decreases. The primary insight from these

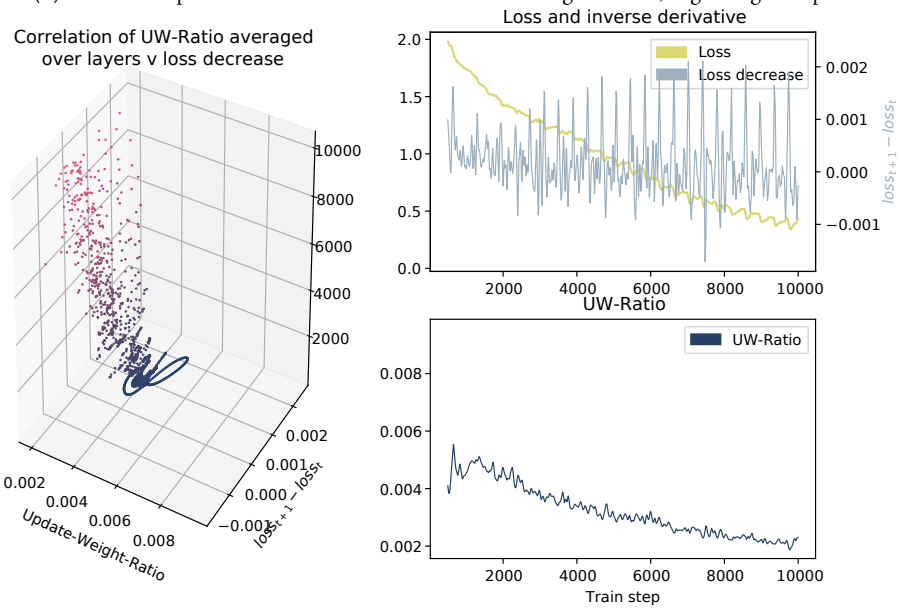
optimiser when batch normalisation layers were removed. For the smaller AlexNetMini architecture with added batch normalisation, Adam did not outperform SGD in the absence of batch norm. To the contrary, Adam even lead to vanishing gradients when using batch normalisation.



(a) UW ratio experiment for VGG with Adam and learning rate 0.01, beginning at step 500



(b) UW ratio experiment for VGG with Adam and learning rate 0.05, beginning at step 500



(c) UW ratio experiment for VGG with Adam and learning rate 0.1, beginning at step 500

experiments is that the empirical UW ratio is not linearly related to the learning rate. Recall from section 1.2 that the weight update is given by

$$\eta \nabla_\theta J, \tag{3.5}$$

so it is a linear function of the learning rate and the gradient magnitude. We would therefore expect that a tenfold reduction in learning rate would result in an equal reduction in the UW ratio. However, the difference is not quite as pronounced in reality. Considering figure 3.18b and figure 3.18c we see the learning rate differing by a factor of ~ 5, whereas the UW ratios only differ by a factor of ~ 2. The cleft between reality and expectation is even more pronounced for learning rates 0.001 and 0.1. This hints at a qualitative difference between the paths through parameter space taken by higher vs. lower learning rates. If we assume the directions are mostly the same, we would also expect the higher learning rate to traverse them quicker (entailing a larger weight change and thus a higher UW ratio, but possibly suffering from the usual problems of too-high learning rate). If instead the path taken by the higher learning rate is actually less steep than the one taken by the smaller rate, we could observe the phenomenon documented here.

Summary

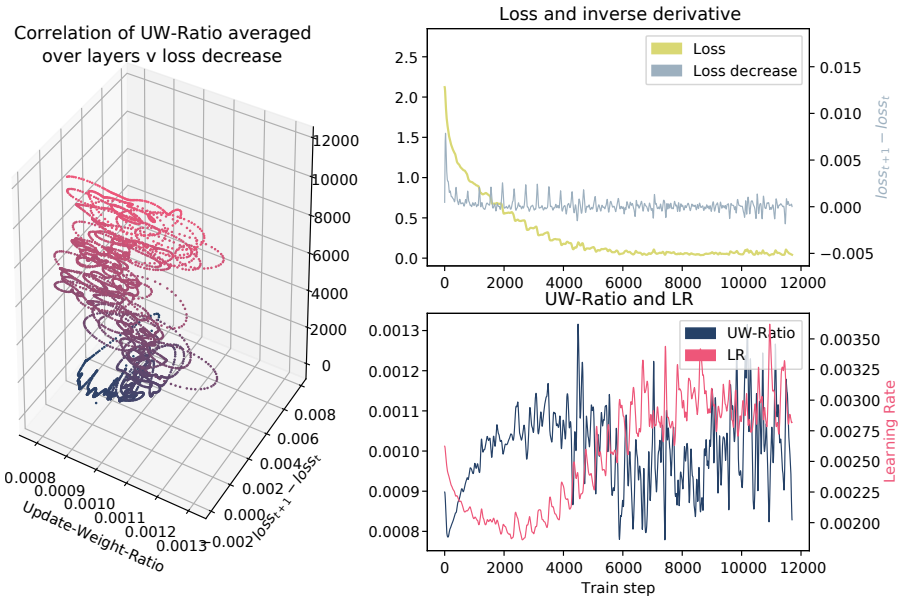
The two experiments with a dynamic learning rate schedule based on the average ratio of updates and parameters do not yield any definitive insights into whether any one value should be considered a target metric. It was obvious from the beginning that as the loss curve approaches its asymptote, the change to the weights needs to decrease, and indeed this has been demonstrated clearly. It is not clear at this point whether the UW ratio provides additional information over other metrics such as the loss curve itself or the norm of the gradients, but the experiments in influencing the ratio have proven that there is no obvious dependency between them—problems can still be learned with a smooth loss decrease even when influencing the UW ratio via learning rate scheduling.

However, more work would be necessary to check whether a moving target could help optimisation. We have seen throughout this section that when learning works well, the ratio decays similarly to an exponential, which opens the possibility for scheduling the learning rate with some decay function and a base value (such as 10^{-3}). While this may not aid convergence in cases that are well-behaved, it could move models out of areas they would otherwise spend too much time in, or reduce parameter oscillations in the face of highly noisy loss surfaces.

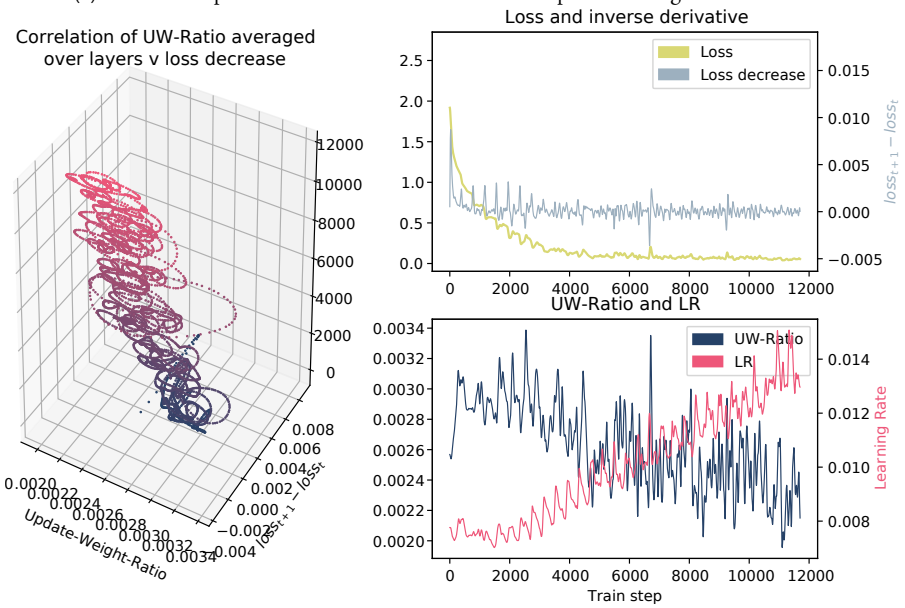
With the help of `ikkuna`, one could easily implement a ratio-based learning rate schedule with e.g. exponential decay built-in and test its performance on a wide range of models and datasets.

3.4 MEASURING LAYER SATURATION FOR EARLY STOPPING

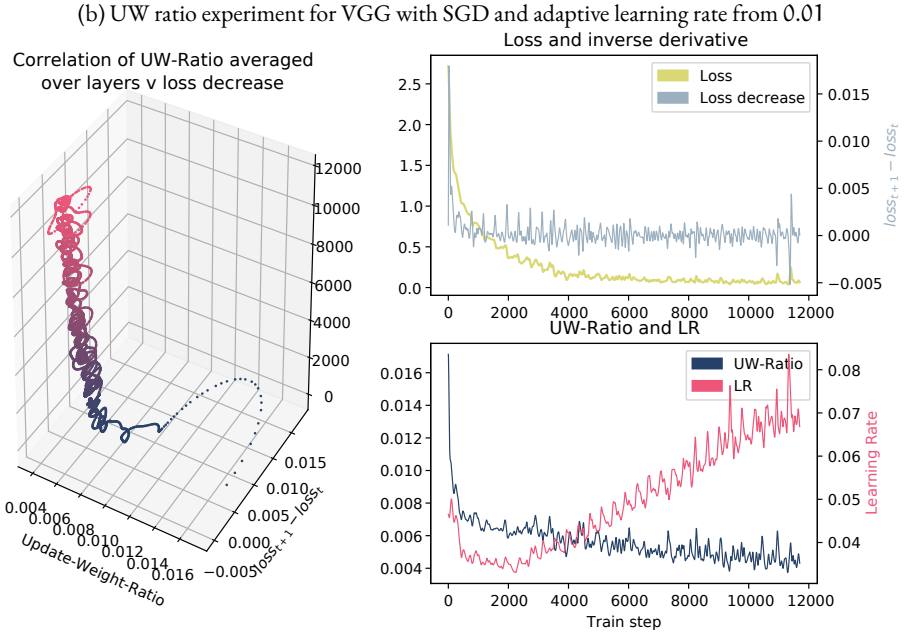
As a last study, we want to investigate an approach for automatically detecting when a good time for stopping parameter updates for a layer comes around. In other words, we are concerned with *freeze training* in which we freeze layers according to some criterion, thus saving computation time during the backward pass. This only makes sense, however, if we do not lose validation accuracy this way.



(a) UW ratio experiment for VGG with SGD and adaptive learning rate from 0.001



(b) UW ratio experiment for VGG with SGD and adaptive learning rate from 0.01



(c) UW ratio experiment for VGG with SGD and adaptive learning rate from 0.1

The term "early stopping" generally refers to halting training completely based on some criterion, often derived from the current training and validation accuracy of the model. It is used to avoid overfitting the model to the dataset by iterating over the dataset too often, until the model essentially memorises the data. In contrast, we use the term for the more fine-grained approach to freezing individual layers at points during training.

3.4.1 *Background*

Recently, the idea of saving computation by progressively freezing layers has been put forth by Brock et al. (2017). They use individual decaying learning rates for each layer so that the rate drops to zero at some point in training. Different schedules for annealing the learning rates are investigated and tradeoffs between validation accuracy and computational savings are reported. The results are that up to 20% of total training time can be saved with less than 3% loss in accuracy for DenseNets (Huang et al., 2017) and ResNets (He et al., 2016) while a VGG variant does not show the same promise. Brock et al. hypothesise that skip-connections play an important role for enabling a good time-accuracy tradeoff, as they are not present in the fully convolutional VGG model.

The approach investigated in this section draws inspiration from (Raghu et al., 2017), where a similarity measure between the representations learned by different layers in the same or different neural networks is defined an evaluated. Raghu et al.'s experiments show that layers generally converge bottom-up by computing their similarity between points during training and the final state. This insight motivates freezing each layer at a percentage of training time, which is a simpler heuristic than the one proposed by Brock et al.. Freezing layer i at time $\frac{i}{n_{\text{layers}}} \cdot n_{\text{iterations}}$ even leads to small improvements in validation accuracy.

After introducing the method of Singular Vector Canonical Correlation Analysis (SVCCA) used by Raghu et al., we will apply it as a metric for measuring layer saturation or convergence by computing the similarity between each layer and itself after the previous epoch. We will compare the usefulness of this metric in informing a freezing schedule with the aforementioned fixed schedule.

3.4.2 *Singular Vector Canonical Correlation Analysis*

Raghu et al. (2017) propose a measure for comparing the similarity of "representations" learned by layers in neural networks. In this context, the representation is defined as the set of activations of a layer's neuron over some dataset D (usually a subset of the training data). Formally, a layer is given as the set of activation vectors of its units, each of which has $n := |D|$ individual activations.

$$L_k := \{(a_{k,i}(\mathbf{x}_0), \ldots, a_{k,i}(\mathbf{x}_{n-1})) \mid i \in [0, n_{\text{units}})\} \tag{3.6}$$

A layer's representation can hence be thought of as a subspace of $\mathcal{R}^n$. However, this primitive concept of a layers learned representation has several problems

- It is not clear whether all directions in the subspace are equally important. Indeed, as the number of parameters often exceeds the number of training examples, most networks are mathematically overparameterised and thus redundant by default (in the absence of regularisation). Previous work (e.g.

LeCun et al. (1990); Srinivas and Babu (2015); Molchanov et al. (2016) report significant computational savings through removing unnecessary units). Using the representations as they are would therefore incorporate many noisy, unstable and unimportant parameters.

- Since the dimension spanned by the activation vectors can be different between layers—as different numbers of units enable more or fewer linearly independent basis vectors—layers of different size could not be straightforwardly compared e.g. by finding an transformation from one space to the other, but it is desirable to compare representations between layers and even networks

The first issue is addressed by reducing the dimensionality of $\operatorname{span} L_k$ by principal component analysis.

(3.7)

First we obtain the data matrix for a layer

$$L_k = (a_{k,i,0}, a_{k,i,1}, \ldots, a_{k,i,n-1})^T \in \mathcal{R}^{n_{\text{units}} \times n} \tag{3.8}$$

$$= \begin{bmatrix} a_{k,0}(\mathbf{x}_0) & \ldots & a_{k,0}(\mathbf{x}_{n-1}) \\ \vdots & \ddots & \vdots \\ a_{k,n_{\text{units}}-1}(\mathbf{x}_0) & \ldots & a_{k,n_{\text{units}}-1}(\mathbf{x}_{n-1}) \end{bmatrix} \tag{3.9}$$

We then get a matrix whose rows are identical—the mean over the first dimension

$$M_k = (1, \ldots, 1)^T \cdot \frac{1}{n} \sum_{i=0}^{n-1} L_k(i, :) \tag{3.10}$$

We can then compute the centered covariance matrix of the data

$$C = (L_k - M_k)^T \cdot (L_k - M_k) \tag{3.11}$$

Diagonalising this matrix through SVD yields

$$C = U\Sigma V \tag{3.12}$$

with Σ containing the singular values and $U\Sigma$ being the transformation matrix to the lower-dimensional space (according to number of singular values and vectors chosen).

In doing so, a new lower-dimensional basis of orthogonal singular vectors is found, a projection to which preserves the most possible variance in the data. The dimensionality can be selected by considering that the magnitude of the singular values is proportional to the amount of variance in the data that is captured by the associated singular direction. A number of singular directions is selected so as to capture 99% of the variance of the data. After projecting the data onto the thusly determined principal components, low-variance directions (which are neurons) are removed.

PCA is performed on both layers to compare, in our case, the same layer at different points in training. The dimensionality-reduced layer representation is then

processed with canonical correlation analysis (CCA). Given vectors of random variables $X = (X_0, \ldots, X_m), Y = (Y_0, \ldots, Y_l)$, CCA finds linear combinations a, b of both sets so that the correlation $\rho = \mathrm{corr}(aX^T, bY^T)$ between the transformed random variables is maximised. When the first transformation is found, a second one can be computed under the constraint that the resulting transformed random variables be uncorrelated with the first pair of transformed variables, then the third and so forth. The result of CCA is a set of $\min\{m, l\}$ correlation coefficients which can be distilled into a similarity measure for learned representations, e.g. by taking the average. The result is a correlation between sets of neuronal activations that is invariant to linear transformations and is not influenced by low-variance or zero activations. The motivation for using this similarity as opposed to simply correlating activations between two time steps is that the CCA correlation allows a layer to change in a linear way between two steps, but we are not interested in how a layer's output distribution is located in space exactly, but what kind of representation it has learned. This additional invariant behaviour could be more robust to noisy and ultimately pointless changes in a layer's weights which we would otherwise conclude from that the layer has changed significantly while it effectively has not.

As a side-effect of CCA, layers of different size can be related, since the input vectors of random variables need not be of the same length. This property is important for comparing different layers and architectures, but immaterial for our experiments.

Todo: Elaborate on difference between linear and conv layers

3.4.3 *Implementation of SVCCA*

Raghu et al. released the code used for the publication[6] under an Apache 2.0 open source license. The implementation uses NumPy and therefore only runs on the CPU. For this work, a fork was created which enables the computations to run transparently on NumPy, PyTorch or CuPy[7][8], so that expensive memory transfers between CPU and GPU can be avoided. Additionally, many of the matrix operations can benefit from GPU acceleration. Making the computations fast enough to compute the similarity often enough on each layer is essential for using it as an on-line metric.

An issue was discovered during benchmarking the CPU and GPU versions against one another: The covariance matrix computed on the activations for many layers turned out to be highly ill-conditioned when using PyTorch for computing it, seemingly due to small numerical deviations. While almost identical to the human observer, the small differences between the covariances often leads to a convergence failure of the SVD routine used in PyTorch. Since it is not unlikely that different units in a layer exhibit the same response to data points – especially in the beginning of training—some redundancy in the activations can be expected and repeated datapoints lead to ill-conditioned covariance matrix. For this reason Raghu et al. implement a robust version of the similarity computation in which noise is progressively added to the activations before the computations until the procedure converges successfully. Other potential workarounds for ill-conditioned covariances include

6 https://github.com/google/svcca

7 CuPy is a GPU-enabled library implementing a subset of the NumPy API; it is developed for the deep learning framework Chainer.

8 The library can be found at https://github.com/themightyoarfish/svcca-gpu

- Adding a small constant to the diagnoal of the covariance matrix (this is known as *nugget regularisation*)
- Adjusting the threshold parameter generally used in SVD algorithms to cut off singular values close to zero

The efficacy of these parameters would need extensive evaluation.

This solution notwithstanding, it turned out that the `torch.mean()` function behaves slightly differently from the the NumPy equivalent, possibly losing precision somewhere, which changes the covariances (see equation (3.7)). The entries of the covariance matrices are often smaller than 0.1, while maximal differences between NumPy and PyTorch computations reach values of 0.05, which is very large in relation to the values. Floating point arithmetic, at least in the IEEE 754 specification generally used in computing, is neither completely commutative nor associative, so the order in which operations are performed can impact the result, so this behaviour is expected[9]. This does not usually lead to problems, but in this use case appears to be highly dangerous. It is unclear in which circumstances the problem arises exactly and a more thorough investigation would be necessary.

For this reason, the experiments performed here use NumPy for computing the SVCCA metric. This incurs high data transfer overhead, but is more stable and sufficient to answer the question of whether the metric can be useful at all. An investigation of the limitations of the GPU impelementation is left for future work.

3.4.4 *Experiments*

The hypothesis to be tested in this seciton is whether the similarity of a layer to itself at previous point in training can be used as a heuristic for determining when to stop training the layer, thus saving all gradient computation for this layer. For this purpose a dedicated subscriber is implemented which periodically feeds a dataset through the network, recording all activations and then computing the SVCCA similarity with the current activations and the ones recorded from the previous time step. The subscriber is shown in listing 3.4.

```
class SVCCASubscriber(PlotSubscriber):

  def __init__(self, dataset_meta, n, forward_fn, freeze_at=10,
               message_bus=get_default_bus(), tag='default',
                   subsample=1,
               ylims=None, backend='tb'):

    # unfortunately, we need access to the model for this subscriber
        in
    # order to initiate evaluation on the dataset
    self._forward_fn     = forward_fn
    self._previous_acts  = dict()
    self._current_acts   = dict()

    # get some random subset of the data
    indices              = np.random.randint(0, dataset_meta.size,
                                             size=n)
    dataset              = Subset(dataset_meta.dataset, indices)
    self._loader         = DataLoader(dataset, batch_size=n,
```

9 https://github.com/pytorch/pytorch/issues/16569

```
                                    shuffle=False,
                                    pin_memory=True)
  # cache input tensors so we don't repeatedly deserialize and copy
  self._input_cache    = []

  # similarity threshold for when to freeze layer
  self._freeze_at      = freeze_at
  self._ignore_modules = set()

  # subscribe 'batch_finished' message and also 'activations' with
      a
  # certain tag chosen by this subscriber to identify messages
      which come up
  # because it called forward() itself
  subscription1 = Subscription(self, ['batch_finished'], tag=tag,
      subsample=subsample)
  subscription2 = Subscription(self, ['activations'], tag='
      svcca_testing',
                               subsample=1)

  # ... call super() and announce publications

# these methods can be overriden by subclasses that implement more
    complex
# forward propagations such as doing it in batches if the entire
    data set
# can't be pushed through at once
def _module_complete_previous(self, module):
  '''Check if activations for module are completely buffered from
      the previous step.'''
  return module in self._previous_acts

def _module_complete_current(self, module):
  '''Check if activations for module are completely buffered from
      the current step.'''
  return module in self._current_acts

def _record_activations_previous(self, module, data):
  '''Record activations into the ``previous`` buffer'''
  self._previous_acts[module] = data

def _record_activations_current(self, module, data):
  '''Record activations into the ``current`` buffer'''
  self._current_acts[module] = data

def _do_forward_pass(self):
  # cache inputs initially
  if not self._input_cache:
      loader = iter(self._loader)
  else:
      loader = iter(self._input_cache)

  # here we make use of the modified forward() function patched in
      the
  # Exporter class. It accepts additional params for controlling
      the tag
  # under which messages are published as a result of this call.
  for i, (X, labels) in enumerate(loader):
      X = X.cuda()
      if len(self._input_cache) < i + 1:
          self._input_cache.append((X, labels))
```

```
      self._forward_fn(X, should_train=False, tag='svcca_testing')

  def _compute_similarity(self, name):
    # now both current and previous acts are complete and we can
        compute

    previous_acts = self._previous_acts.pop(name)
    current_acts = self._current_acts.pop(name)

    # current acts are now previous ones
    self._previous_acts[name] = current_acts

    # convolutional filters need to be reshaped to 2-d
    if previous_acts.ndimension() > 2:
      c = previous_acts.shape[1]  # channel dim
      previous_acts = previous_acts.permute([0, 2, 3, 1]).reshape(-1,
          c)

    if current_acts.ndimension() > 2:
      c = current_acts.shape[1]  # channel dim
      current_acts = current_acts.permute([0, 2, 3, 1]).reshape(-1, c
          )

    previous_acts = previous_acts.detach().cpu().numpy()
    current_acts = current_acts.detach().cpu().numpy()
    result_dict = svcca.cca_core.robust_cca_similarity(
                        previous_acts.T,
                        current_acts.T,
                        epsilon=1e-8,
                        threshold=0.98,
                        verbose=False,
                        compute_dirns=False,
                        rescale=True)
    return result_dict['mean'][0]

  def _freeze_module(self, module):
    print(f'Freezing {module}')
    freeze_module(module)
    self._ignore_modules.add(module)

    # clear any orphaned data
    if module in self._previous_acts:
        self._previous_acts.pop(module)
    if module in self._current_acts:
        self._current_acts.pop(module)

  def compute(self, message):

    if message.tag == 'default' and message.kind == 'batch_finished':
      # this means we are not in a state where we initiated forward
      # passes.
      # This assumes that the subsample initialiser parameter is set
          so
      # that this does not happen after every batch (unless desired)
      self._do_forward_pass()

    elif message.tag == 'svcca_testing':
      # we are receiving activation messages as a result of out call
          to
      forward_fn()
      module, name = message.key
```

```
    if module in self._ignore_modules:
      # maybe already frozen
      return

    if not self._module_complete_previous(module):
      self._record_activations_previous(module, message.data)
    elif not self._module_complete_current(module):
      self._record_activations_current(module, message.data)

    if self._module_complete_current(module) and self.
        _module_complete_previous(module):
      mean = self._compute_similarity(module)
      self._backend.add_data(name, mean, message.global_step)

      # ... publish to message bus

      if mean > self._freeze_at:
        self._freeze_module(module)
```

Listing 3.4: SVCCA Subscriber

The `SVCCASubscriber` holds a dataset and listens for messages that a batch has ended. At this point, it will interrupt the training by passing its dataset through the network, and attaches a custom `tag` to the messages resulting from this call. This allows it to identify activation messages originating from the call. It would be more ergonomic to have the data returned directly from the call to `forward()`, but since the only way PyTorch offers for retrieving intermediate layers' activations and gradients is callback-based, `ikkuna` does not provide a simpler way of doing this at the time of writing.

The activation messages are buffered until activations for both the previous time step and the current one have been received, at which point the similarity can be computed and published to the backend and message bus.

Several hyperparameters need to be decided on, and some parameters for the similarity routine were fixed for theses experiments.

- The dataset on which to evaluate the similarity must be decided. Since we are not interested in absolute predictions but simply in neuronal responses of different layers for the same data, it does not really matter whether the data is from training or validation sets. Ideally, however, it should have a maximally uniform class distribution so as to engage all filters that have been learned up to this point. If the data distribution was highly skewed, it could mean that some neurons never respond for any dataset, thus providing no information and reducing the rank of the covariance matrix (since some variables—columns of the data matrix L_k—would be linearly related) even though with a better dataset we might have had a larger rank and hence a different SV decomposition.

- A similarity threshold at which we determine the layer doesn't change anymore must be found. This value should be determined through thorough experimentation (`ikkuna` can be used to evaluate different thresholds on a wide range of models), but the experiments here use correlation thresholds of 0.99 and 0.995.

- The time scale at which significant layer changes are expected needs to be selected. If one was to compare a layer to itself at the previous training batch,

Table 3.2: Hyperparameters for the saturation experiment

Parameter	Value
Optimiser	Vanilla SGD
Epochs	30
Dataset	CIFAR10
Batch size	512
Datapoints for SVCCA	500
Nugget constant	10^{-8}
Learning rates	0.01, 0.1, 0.5

one would expect very little change and thus a high correlation, whereas changes between the start of an epoch and the end would be much more significant. The experiments use epochs as the resolution and thus compute the measure at the end of every epoch.

The hyperparameters used in these experiment are listed in table 3.2. The experiment was run on the small convolutional architecture (figure 3.11) as well as the larger VGG model (figure 3.14). The self-similarity for each layer was computed at the beginning of each epoch. We select high learning rates has have been found appropriate for standard gradient descent. The different learning rates are split among figure 3.20a ($\eta = 0.01$), figure 3.22a ($\eta = 0.1$) and figure 3.24a ($\eta = 0.5$). All graphs are averages of several runs, noted in parantheses in the legend. For thresholds on the self-similarity score we use 0.99, 0.995 and also report control runs without layer freezing (denoted by "never" in the graphs) and the heuristic from (Raghu et al., 2017) (see section 3.4.1).

Layer Convergence and Validation Accuracy

Focusing on the accuracy graphs (upper row) shows that for both thresholds and models, accuracy is not negatively impacted, testifying that layer freezing can always be appropriate. Except for the largest learning rate on the smaller model (figure 3.24a), the self-similarity-based schedule for freezin is also not worse than the fixed one. For the smaller model, freezing layers even resulted in increases accuracy. One explanation for this could be that the model is more prone to overfit than the VGG network, despite its smaller size. In that case, freezing layers before they start to memorise the data could prevent this from happening. For the higher of the two thresholds, improvents could only be observed for the smaller two learning rates. In contrast, at a learning rate of 0.5, waiting until a layer reaches 0.995 saturation seems to allow the model to overfit too much, while the lower learning rates are not high enough to show this problem.

On the other hand, the VGG model does not benefit from layer freezes performance-wise. It is possible the architecture is less prone to overfit, thus deriving no benefit from regularising it through freezes, besides reducing computational cost.

We can draw the preliminary conclusion that progressively freezing layers is not detrimental to performance, but saves computation.

It is also interesting to look at how the self-similarity for a layer evolves throughout training (bottom row of plots). The similarity graphs for VGG show strikingly, that according to this metric, layers converge bottom-up, without fail, which is in line

with Raghu et al. (2017)'s findings. Freezing according to this metric therefore always results in successive layers getting frozen, not disconnected ones. It is remarkable that in all cases, the first layer of convolutions is converged already after the first five epochs, often needing no more than one. All other layers need substantially longer. This makes sense as the lower layers often learn spatial filters which act as edge detectors. These filters are useful for any input, so the specific class distribution almost does not matter. Because of this, it is not surprising that 50.000 training (one epoch) examples are enough to learn these filters.

As a last point, the high learning rate used in figure 3.24a leads to interesting reactions in the self-similarity metric. While AlexNetMini's performance is not much worse than for the optimal learning rate of 0.1, training is much more unstable, with the accuracy fluctuating. Indeed, the saturation metric reflects this constant change in layer weights, and most of the layers do not converge at all. This observation opens the door for evaluating the SVCCA score as a stability metric and for guiding learning rate scheduling (see section 3.3). Both AlexNetMini and VGG exhibit much slower convergence than would be possible, and we can read the instability from the SVCCA score's trajectory.

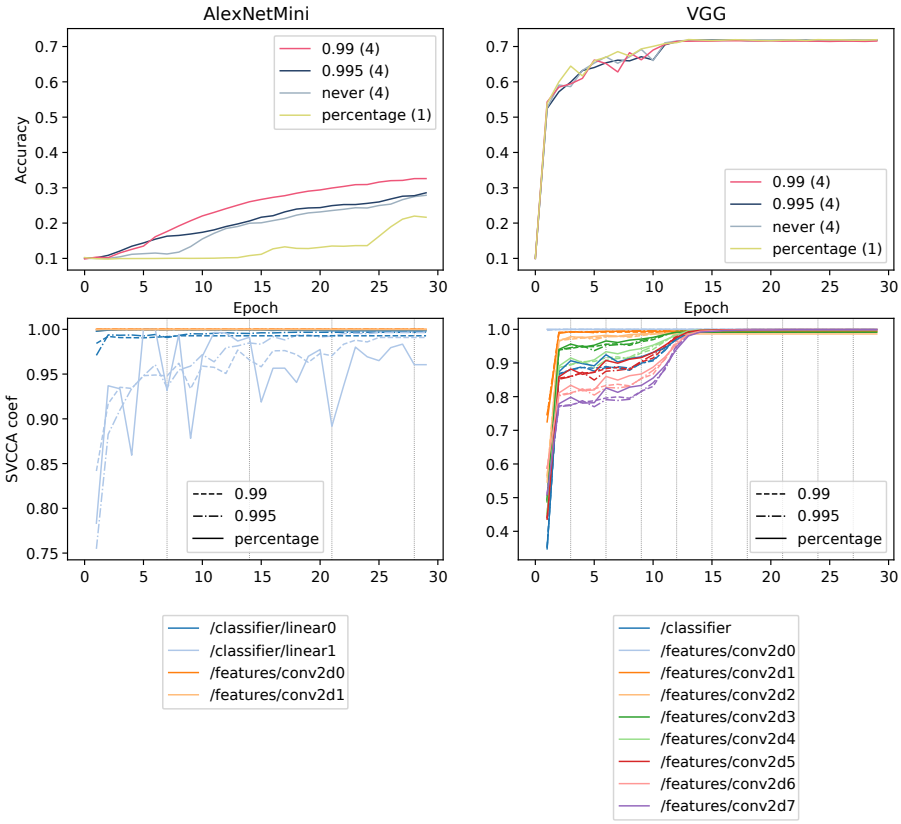
Computational Savings with Self-Similarity vs. Fixed Schedule

Turning to the question of whether this heuristic performs better than the fixed schedule from (Raghu et al., 2017), the point at which each layer was frozen is graphed in figures 3.20b, 3.22b and 3.24b. Layers are differentiated by colour and marker style, and all runs are displayed. A random offset is added in the vertical direction to avoid the runs overlapping.

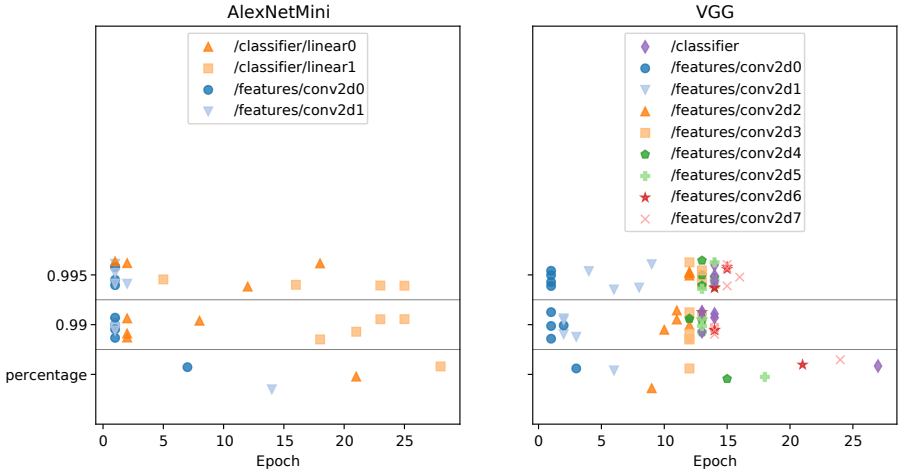
In most cases, using self-similarity—without negatively impacting accuracy – tends to freeze layers earlier, thus saving more computation than freezing layers at fixed percentages of training time. figure 3.20b (both models) and figure 3.22b (VGG model) both show the similarity-based freezing to freeze layers much earlier on average. As the high learning rate of 0.5 leads to unstable training, with likely very high variance in the weight updates, it is unsurprising that layers don't converge and are thus trained until the end. For non-pathological cases, the proposes metric appears to save more update steps. However, additional overhead is incurred in actually computing the metrics. An exhaustive evaluation of the cost-benefit tradeoff is left for future work, but the results are promising, especially in light of a possible GPU implementation of SVCCA, most of which could be done asynchronously.

For the smaller of the two models (left plots), we see again that the two convolutional layers areconverged right after the first epoch or the second, respectively. The precise threshold value does not seem to matter so much. Saturating the first linear layer takes much longer and the different thresholds lead to different times until convergence. A higher learning rate of 0.1 (see figure 3.22b) leads to more spaced-out convergence points for the convolutional layers, while the linear layers are not considered converged during training at all. What can be deduced from this fact is that the 0.01 is a too small learning rate for the problem and architecture in question. The reason why the linear layers converge early for a learning rate of 0.01 could be that the learning rate is simply too small to translate the gradients on this problem into substantial change in the layer's weights, leading to a smaller validation accuracy. By contrast, the larger learning rate will move the weights sufficiently to allow the linear layers to learn more performant representations, and

Figure 3.19: Layer convergence experiment with learning rate 0.01
SGD, 0.01

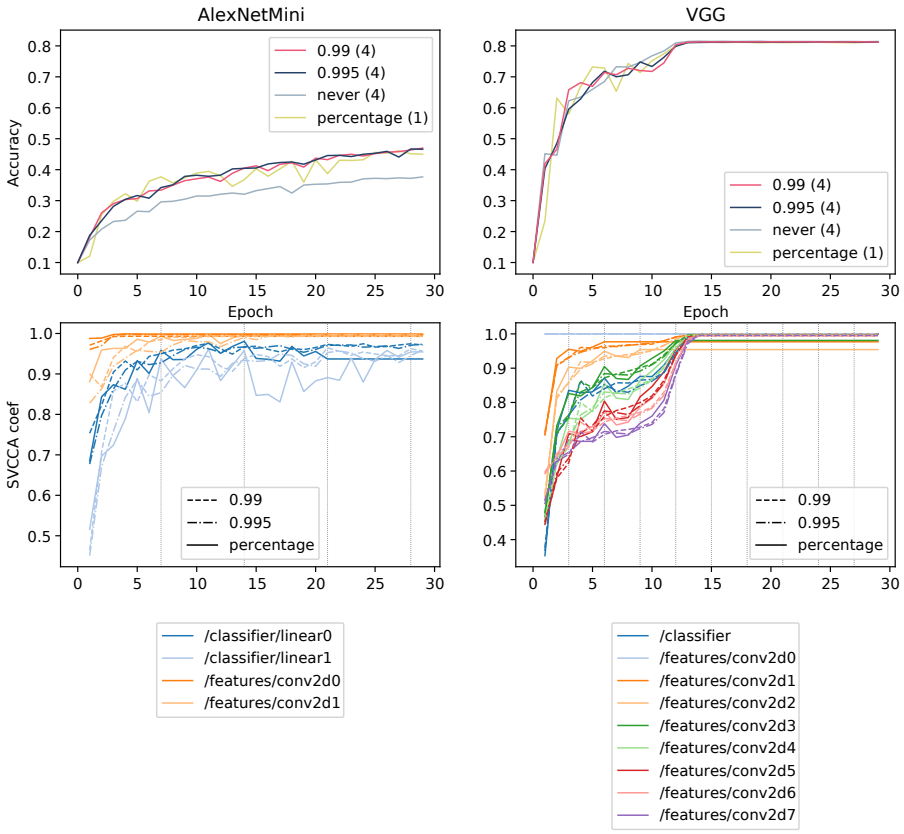


(a) Accuracy and self-similarity for learning rate 0.01
SGD, 0.01

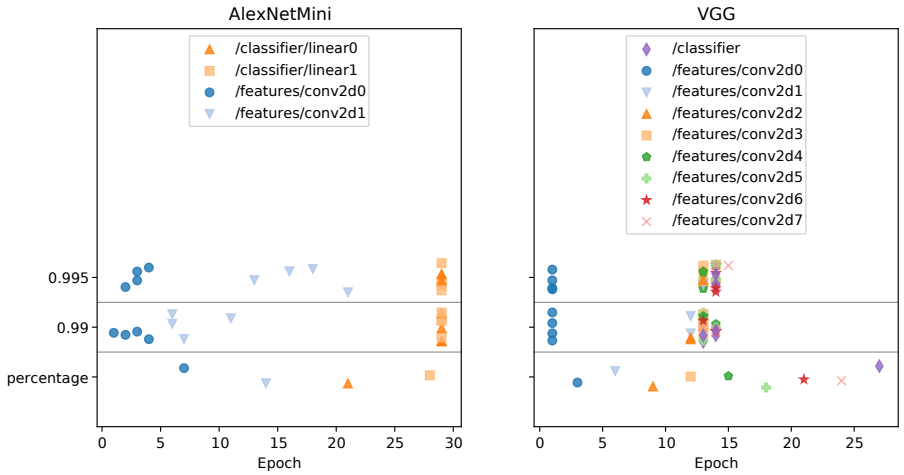


(b) Freeze points for learning rate 0.01

Figure 3.21: Layer convergence experiment with learning rate 0.1
SGD, 0.1

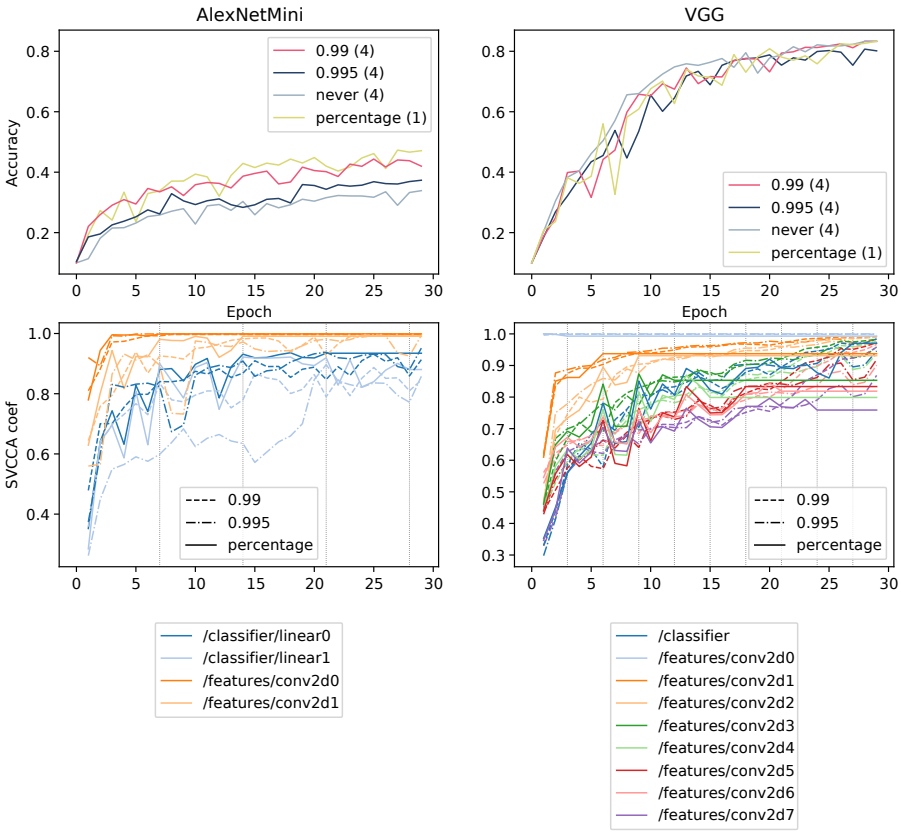


(a) Accuracy and self-similarity for learning rate 0.1
SGD, 0.1

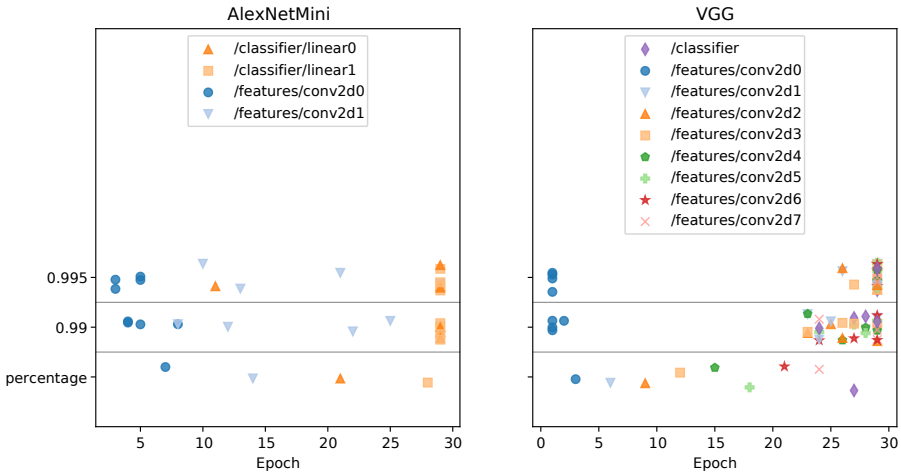


(b) Freeze points for learning rate 0.1

Figure 3.23: Layer convergence experiment with learning rate 0.5
SGD, 0.5



(a) Accuracy and self-similarity for learning rate 0.5
SGD, 0.5



(b) Freeze points for learning rate 0.5

thus need longer to converge. Similar observations can be made for a learning rate of 0.5.

In case of the VGG model, we observe similarly fast convergence of the first convolutional layers, reinforcing the hypothesis that the early feature detectors are mostly architecture- and problem-agnostic. We can also see that convergence for most layers takes longer for higher thresholds (unsurprisingly), except for the medium learning rate (figure 3.22b) for which all layers very consistently converge at the same time as well. Since this is the condition with the best validation accuracy, this phenomenon may merit further investigation.

For the high learning rate of 0.5, most VGG layers do not converge at all according to the self-similarity metric. Nevertheless, the accuracy does reach optimal levels ~ 0.8, but takes much longer than with a smaller learning rate. This tells us that whether or not all layers converge is not necessarily indicative of whether the network can learn the task at all, but more likely how quickly and efficiently it can learn it. An investigation into how the solutions found by networks with converged layers differ from unconverged ones is an interesting application for the SVCCA metric as well. As a matter of fact, Morcos et al. (2018) use it to compare minimisers found by memorising and generalising networks.

3.4.5 *Summary*

This section has demonstrated an alternative use case for the SVCCA similarity metric originally proposed for offline-introspection of neural networks. It was evaluated as a heuristic for detecting layer convergence which resulted in promising observations when compared to freezing layers at a fixed schedule. The SVCCA self-similarity does not usually deteriorate performance and freezes layers earlier than a fixed schedule in many cases. More comprehensive studies should be made to validate this observation on more architectures and datasets. Furthermore, the role of the additional parameters introduced for computing the SVCCA metric remains to be explored—notably, how many datapoints should be used. As computing the metric in a certain interval incurs some computation overhead which we have not yet accounted for, more work is necessary to definitivly answer the question whether this heuristic outperforms the fixed schedule in absolute terms, not only in number of gradient computations saved.

4

FUTURE WORK

This chapter will examine some avenues of inquiry which were beyond the scope of this work and scope out future work needed to solidify the findings from chapter 3.

4.1 EXTENSIONS TO THE EXPERIMENTS

The `ikkuna` library was employed in chapter 3 to simplify experimenting with metrics on a variety of architectures, but those were mostly not state-of-the-art models. To validate all findings, larger and more modern architectures should be tested, alongside a larger variety of hyperparameters, for instance a larger range of learning rates and batch sizes, since they can have a strong influence on the gradient distribution and thus the learning process.

The experiments on learning rate optimisation (section 3.3) showed that a static target ratio between updates and weights may not be the best approach. Therefore, the experiments should be extended with an exploration of whether a moving target ratio leads to faster convergence, before the idea is entirely abandoned.

For the convergence experiment in section 3.4, an extensive search for the best threshold for freezing layers should be conducted in order to optimise the tradeoff between compute time invested for computing the metric and compute time saved by terminating training early.

4.2 TRACKING SECOND-ORDER INFORMATION

While the success of purely gradient-based methods is remarkable in the light of the proliferation of local minima and saddle points in highly non-convex objectives in many dimensions (for a review on the saddle-point problem, see Dauphin et al. (2014)), second-order information about the curvature of the loss function around the current point in parameters space would provide more valuable guides for selecting the direction of the next update step in parameter space.

The central problem in obtaining second-order information about the loss function is the dimensionality of the parameter space. The Hessian H of a scalar function $f(\theta)$ of a parameter vector $\theta \in \mathcal{R}^d$ is of size $d \times d$. Even storing this matrix is infeasible for networks of millions of parameters, let alone computing it. But the Hessian is what e.g. the Newton algorithm requires in higher dimensions.

There exist some second-order algorithms for non-convex optimisation. One such algorithm is Limited Memory BFGS (Liu and Nocedal, 1989), an adaptation of the BFGS algorithm—which estimates the inverse Hessian with respect to all model parameters—that remembers only some history of update steps and gradients. Another method is Conjugate Gradient Descent (originally described by Fletcher and Reeves (1964)), which does not require the Hessian explicitly, but only needs to compute Hessian-vector products, which is much easier.

Still, these methods have so far not demonstrated better performance in practice than variants of first-order gradient descent. It is still an active area of research how

the second derivative can aid in speeding up convergence of the optimisation or avoid some of the guesswork involved in finding good step sizes.

The library developed in this work has been used to track eigenvalues of the Hessian estimated via deflated stochastic power iteration. For a diagonalisable matrix A and an initial estimate $\mathbf{q}_0$ with unit norm, the power method (also known as Von-Mises iteration) computes the iterate

$$\mathbf{q}_k = \frac{A\mathbf{q}_{k-1}}{\|A\mathbf{q}_{k-1}\|_2} \tag{4.1}$$

This series converges to the dominant eigenvector $\mathbf{v}_1$ of A or not at all. The corresponding eigenvalue λ_1 can be computed as

$$\lambda_1 = \frac{(A\mathbf{v}_1)^T \mathbf{v}_1}{\mathbf{v}^T\mathbf{v}} \tag{4.2}$$

per the definition of the eigenvalue. Deflation can then be used to obtain a matrix B whose dominant eigenvalue is the second largest eigenvalue of A.

$$B = A - \lambda_1 \mathbf{v}_1 \mathbf{v}_1^T \tag{4.3}$$

This allows computing the top-k eigenpairs for any $k \leqslant d$. The implementation by Golmant (2018) uses PyTorch's `autograd` functionality to compute the Hessian-Vector product $H\mathbf{q_k}$ with $\mathbf{q}_0 \sim U(0, 1)$. The estimate is stochastic since it used a fixed number of batches from the dataset instead of all samples.

The functionality is realised in the `HessianEigenSubscriber` in the `ikkuna.export.subscr` subpackge. Eigenvalues and eigenvectors could be used for directing the gradient descent process (Alain et al. (2018) tentatively find that largest decreases of the loss can often be made when stepping along the most negative eigendirection, i.e. the most negative curvature), but this would be an active intervention into the training, which is not the goal of the library. More practically however, the Hessian eigenvalues carry information about the sharpness of a local minimum and could be used for diagnosing stability of the current minimiser, which is relevant both for generalisation ability and resilience against adversarial attacks (inputs crafted to fool the network). There has been some recent work arguing for smaller batch sizes—one of the parameters whose choice we want to simplify for the user—as they tend to generalise better (see e.g. (Keskar et al., 2016)). The claim is disputed, but Yao et al. (2018) find that larger batch sizes during training strongly correlate with larger dominant eigenvalues of the Hessian. It is unclear if an absolute value can be determined for a given model and loss function at which the recommendation to reduce the batch size can be made, but this is an interesting area for future research. Analogously to section 3.3.2 one could employ `ikkuna` to track the largest Hessian eigenvalue and a an adversarial susceptibility score. The score could be computed as the average decrease in accuracy over a set of image pairs where one of each pair is adversarially perturbed. We can then discover correlations between metrics such as the dominant Hessian eigenvalue and the vulnerability of a model to adversarial attacks.

4.3 TRACKING ARCHITECTURAL ISSUES

On of the primary difficulties in developing a neural network model to solve a given problem is coming up with an architecture to begin with. Significant advancements of the state of the art have been achieved by novel architectures, such as the VGG, ResNet, ResNeXt, DenseNet or SqueezeNet, which highlights the importance of finding a good number, size and type of layers. section 3.4 discussed an approach for systematically freezing layers, and similar lines of inquiry lead to the question of whether we can track in how far the network is too constrained by its layer sizes, or too powerful and thus harder to train. Shenk (2018) explored how ideas from (Raghu et al., 2017) can be used for identifying such problems. Given we can detect when a layer is too large or too small, we could either abort and restart training with a smaller architecture, thus saving time on iteration, or even resize layers live. It is not obvious how to do this properly, since only the insignificant components of a layer should be removed. We would need to identify a projection of weight matrices onto a lower-dimensional subspace that does not cancel all progress made throughout training.

4.4 IMPROVEMENTS TO THE SOFTWARE

The `ikkuna` library simplifies metric tracking for neural networks, but more work is necessary to make it production-ready, including bug fixes[1].

The following list lists useful features which should be implemented in future versions.

- Subscribers should be asynchronous and interleave their—possibly expensive—computations with the actual training. This should reduce the cost of computing metrics to almost nothing, since the main process is generally waiting while the GPU is working, besides loading and decoding training data—which again involves waiting on the hard disk.
- The API for now is purely callback-based—a result of the PyTorch API. However, for some use cases—notably computing validation accuracy and the SVCCA measure, which require the Subscriber to initiate forward passes—it is desirable to obtain activations or gradients as a return value of the forward pass, and not having to wait for the messages in a callback. An additional abstraction between the model and the subscriber could buffer the messages and return them in response to method call.
- A related problem is that some subscribers need direct access to the model to push data through it. This is undesirable as it couples Subscriber and Publisher. A service accepting data and returning intermediate activations and gradients would be a good complement to the callback-based API.
- The relationship between the `Subscriber` and `Subscription` classes is convoluted at present and can probably be simplified.

1 An updated list of open issues can be found at https://github.com/Peltarion/ai_ikkuna/issues

5

CONCLUSION

This thesis has investigated approaches for illuminating the opaque training process for deep artificial neural networks by tracking certain metrics while training is running. Detecting bad configurations early as opposed to comparing final accuracies after the fact could yield immense savings in computation and cost. Furthermore, finding metrics which predict undesirable network behaviour can be instrumental in guiding research of the theoretical mechanisms underlying the sometimes remarkable performance of neural models, which to this day remain poorly understood.

After introducinng the concepts of artificial neural networks and their optimisation, a software library named `ikkuna` has been designed and implemented for the PyTorch deep learning framework. The library frees the researcher or practitioner from some of the repetitive tasks necessary for experimenting with and applying online metrics. It provides a model-agnostic API based on a Publisher-Subscriber pattern against which portable and reusable network metrics can be implemented. These metrics can be distributed as plugins and thus shared with the community. The library can also be a building block in more comprehensive deep learning applications and libraries, such as the web-based Peltarion platform, thus saving engineering effort.

Following that, the library was employed for its intended use: Aiding researchers and machine learning practitioners in implementing and evaluating online training metrics. Candidate metrics for optimising the learning rate and detecting network convergence have been introduced and evaluated. In doing so, deep learning folklore and research in the context of learning rates and parameter optimisation has been partially debunked and partially validated. Finally, some avenues for further research have been introduced which would build upon this work.

While more work remains to be done to scope out the validity of the findings presented here (a task which is simplified by the software presented here), promising results were obtained for detecting layer convergence, whereas the proposed learning rate metric could not be confirmed to be of immediate use. Nonetheless, some evidence was presented for its potential pending further investigation.

APPENDIX A — OPEN SOURCE ACKNOWLEDGMENTS

The following presents a non-exhaustive list of open source tools used in creating the software, experiments and this document:

1. LaTeX for typesetting this document (Lamport, 1986). This includes contributions by the creators of all the packages which make up the typesetting environment.
2. The various GNU and thrid-party command line tools (`grep`, `parallel` (Tange, 2018), `latexmk` etc.)
3. The Ubuntu operating system and the Linux kernel (Torvalds et al., 2008) providing the computational environment for the experiments conducted for this work, alongside the GNU compiler toolchain for compiling software for the system
4. The TMUX and SSH tools for connecting to the system running the experiments
5. The PyTorch (Paszke et al., 2017), Numpy, TensorBoard and Matplotlib libraries (Jones et al., 01) used for the software, based all on the Python programming language and standard library.
6. The Sacred (Klaus Greff et al., 2017), MongoDB and Pymongo libraries used for logging experiments to a database for later visualisation
7. The Vim editor and associated plugin ecosystem employed for creating all documents, be it code or documentation
8. The Git source control management tool which was used to version both software and documentation artifacts

B

APPENDIX B — CONTRIBUTIONS TO OTHER LIBRARIES

This appendix lists all code and documentation contributions made to other projects during development.

Table B.1: Contributions to third-party projects

Project	Contribution
SVCCA-GPU	Created a GPU-fork of Google's SVCCA code.
Matplotlib	Added a note to the official documentation about the API for `errorbar` (commit hash `3a698d6e7d972d2c18fe4d2524e92ff637326f88`)
PyTorch	Added a document to the official documentation detailing how to make experiments reproducible (see https://pytorch.org/docs/master/notes/randomness.html). The note has been modified since creation.
CuPy	Added NumPy-style ordering and comparison functionality for complex number types, thereby enabling all sorts of operations for complex matrices which did not work previously. This required modifications to the C++ code backing the library and the glue code generating the Python interface. Relevant commits are `6f92c30a6e41a7cd512f17f6a32302f0b8d0970d`, `55d504e4183c271c640d566d8691f763d6276af5`, `bad61578a1f69d46397b7c2556a2787493d44455` and `04506438a1142b98cd1a326fc1d423459aac1433`
TensorBoardX	Documentation improvements

BIBLIOGRAPHY

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org. (Cited on page 10.)

Alain, G., Le Roux, N., and Manzagol, P.-A. (2018). Negative eigenvalues of the hessian in deep neural networks. (Cited on page 74.)

Arora, S. (2018). Toward theoretical understanding of deep learning. `http://unsupervised.cs.princeton.edu/deeplearningtutorial.html`. (Cited on page 3.)

Arpteg, A., Brinne, B., Crnkovic-Friis, L., and Bosch, J. (2018). Software engineering challenges of deep learning. In *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 50–59. IEEE. (Cited on page 8.)

Brock, A., Lim, T., Ritchie, J. M., and Weston, N. (2017). Freezeout: Accelerate training by progressively freezing layers. *arXiv preprint arXiv:1706.04983*. (Cited on page 59.)

Ciresan, D. C., Meier, U., Masci, J., Gambardella, L. M., and Schmidhuber, J. (2011). Flexible, high performance convolutional neural networks for image classification. (Cited on page 4.)

Dauphin, Y. N., Pascanu, R., Gulcehre, C., Cho, K., Ganguli, S., and Bengio, Y. (2014). Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In *Advances in neural information processing systems*, pages 2933–2941. (Cited on page 73.)

Eugster, P. T., Felber, P. A., Guerraoui, R., and Kermarrec, A.-M. (2003). The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131. (Cited on page 15.)

Fletcher, R. and Reeves, C. M. (1964). Function minimization by conjugate gradients. *The computer journal*, 7(2):149–154. (Cited on page 73.)

Golmant, N. (2018). Pytorch-hessian-eigenthings. `https://github.com/noahgolmant/pytorch-hessian-eigenthings/`. (Cited on page 74.)

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778. (Cited on pages 22 and 59.)

Huang, G., Liu, Z., Van Der Maaten, L., and Weinberger, K. Q. (2017). Densely connected convolutional networks. In *CVPR*, volume 1, page 3. (Cited on pages 22 and 59.)

Jacobs, R. A. (1988). Increased rates of convergence through learning rate adaptation. *Neural networks*, 1(4):295–307. (Cited on page 7.)

Jones, E., Oliphant, T., Peterson, P., et al. (2001–). SciPy: Open source scientific tools for Python. (Cited on page 79.)

Karpathy, A. (2015). Convolutional neural networks for visual recognition lecture notes. `http://cs231n.github.io/neural-networks-3/#ratio`. (Cited on pages 44 and 45.)

Keskar, N. S., Mudigere, D., Nocedal, J., Smelyanskiy, M., and Tang, P. T. P. (2016). On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836*. (Cited on page 74.)

Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*. (Cited on pages 7, 32, 33, 36, 38, 40, 44, and 52.)

Klaus Greff, Aaron Klein, Martin Chovanec, Frank Hutter, and Jürgen Schmidhuber (2017). The Sacred Infrastructure for Computational Research. In Katy Huff, David Lippa, Dillon Niederhut, and Pacer, M., editors, *Proceedings of the 16th Python in Science Conference*, pages 49 – 56. (Cited on page 79.)

Krizhevsky, A. and Hinton, G. (2009). Learning multiple layers of features from tiny images. Technical report, Citeseer.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105. (Cited on page 22.)

Lamport, L. (1986). *Latex: A Document Preparation System*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. (Cited on page 79.)

LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., and Jackel, L. D. (1989a). Backpropagation applied to handwritten zip code recognition. *Neural Comput.*, 1(4):541–551. (Cited on page 4.)

LeCun, Y., Denker, J. S., and Solla, S. A. (1990). Optimal brain damage. In *Advances in neural information processing systems*, pages 598–605. (Cited on page 60.)

LeCun, Y. et al. (1989b). Generalization and network design strategies. *Connectionism in perspective*, pages 143–155. (Cited on page 4.)

Liu, D. C. and Nocedal, J. (1989). On the limited memory bfgs method for large scale optimization. *Mathematical programming*, 45(1-3):503–528. (Cited on page 73.)

Molchanov, P., Tyree, S., Karras, T., Aila, T., and Kautz, J. (2016). Pruning convolutional neural networks for resource efficient inference. *arXiv preprint arXiv:1611.06440*. (Cited on page 60.)

Morcos, A., Raghu, M., and Bengio, S. (2018). Insights on representational similarity in neural networks with canonical correlation. In Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., and Garnett, R., editors, *Advances in Neural Information Processing Systems 31*, pages 5732–5741. Curran Associates, Inc. (Cited on pages 44 and 71.)

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. (2017). Automatic differentiation in pytorch. In *NIPS-W*. (Cited on pages 14 and 79.)

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830. (Cited on page 8.)

Pleiss, G., Chen, D., Huang, G., Li, T., van der Maaten, L., and Weinberger, K. Q. (2017). Memory-efficient implementation of densenets. *arXiv preprint arXiv:1707.06990*. (Cited on page 22.)

Raghu, M., Gilmer, J., Yosinski, J., and Sohl-Dickstein, J. (2017). Svcca: Singular vector canonical correlation analysis for deep learning dynamics and interpretability. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R., editors, *Advances in Neural Information Processing Systems 30*, pages 6076–6085. Curran Associates, Inc. (Cited on pages 59, 61, 66, 67, and 75.)

Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, pages 65–386. (Cited on page 4.)

Saad, D. (1998). Online algorithms and stochastic approximations. *Online Learning*, 5. (Cited on page 45.)

Shenk, J. (2018). Spectral decomposition for live guidance of neural network architecture design. Master's thesis, Osnabrück University. (Cited on page 75.)

Simonyan, K. and Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*. (Cited on page 24.)

Smith, S. L., Kindermans, P., and Le, Q. V. (2017). Don't decay the learning rate, increase the batch size. *CoRR*, abs/1711.00489. (Cited on page 8.)

Srinivas, S. and Babu, R. V. (2015). Data-free parameter pruning for deep neural networks. *arXiv preprint arXiv:1507.06149*. (Cited on page 60.)

Tange, O. (2018). *GNU Parallel 2018*. Ole Tange. (Cited on page 79.)

Tokui, S., Oono, K., Hido, S., and Clayton, J. (2015). Chainer: a next-generation open source framework for deep learning. In *Proceedings of workshop on machine learning systems (LearningSys) in the twenty-ninth annual conference on neural information processing systems (NIPS)*, volume 5, pages 1–6. (Cited on page 14.)

Torvalds, L. et al. (2008). The linux kernel. *URL http://www. kernel. org*. (Cited on page 79.)

Werbos, P. (1975). *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. Harvard University. (Cited on page 4.)

Yao, Z., Gholami, A., Lei, Q., Keutzer, K., and Mahoney, M. W. (2018). Hessian-based analysis of large batch training and robustness to adversaries. *arXiv preprint arXiv:1802.08241*. (Cited on page 74.)

ACKNOWLEDGMENTS

I wish to acknowledge the contributions of many people who—directly or indirectly—supported this work.

I thank Justin Shenk for providing the introduction to the Peltarion team and his generosity and company while travelling with me and—along with Veine Haglund—hosting me for my visits to Stockholm.

I am also grateful to Anders Arpteg who has been a constant source of support and advice, for his willingness to let me work on my own terms on what I saw fit.

Thanks as well to Justin, Anders and Mikael Huss for feedback on thesis drafts.

I owe further thanks to Oliver Vornberger who agreed to act as first examiner for this thesis, instead of enjoying retirement, and Ulf Krumnack for acting as co-examiner and providing feedback on this work.

I thank the entire team at Peltarion AB for creating a welcoming and very entertaining environment for working on this thesis and providing computational resources making this work possible at all. Also for the ice cream.

Lastly, I wish to acknowledge the countless unnamed developers of the myriads of open source tools that form the bedrock of any productive scientific endeavour. A highly non-exhaustive list of software used during the creation of this thesis can be found in appendix A.

DECLARATION OF AUTHORSHIP

I hereby certify that the work presented here is—to the best of my knowledge and belief—original and the result of my own investigations, except as acknowledged, and has not been submitted, either in part or whole, for a degree at this or any other university.

Rasmus Diederichsen

Osnabrück, February 13, 2019