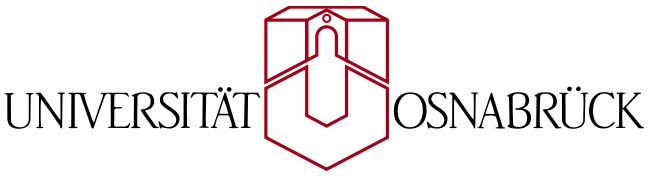


Department of Computer Science
Department of Cognitive Science

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 – tere 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 experimentor.

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 extractd 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. Zuletzt 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

1 INTRODUCTION

While the success 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's rarely one definitive way to solve a certain problem in deep learning. Additionally, the quality of debugging tools available to a 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 parametrisation 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 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 section 1.1.

1.1 ARTIFICIAL NEURAL NETWORKS

The artificial neural network is a class of machine learning model which can be used for regression, classification, generation and a host of other tasks. At its core, neuronal 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). With the introduction of the backpropagation algorithm (Werbos, 1975) and increasing avail-

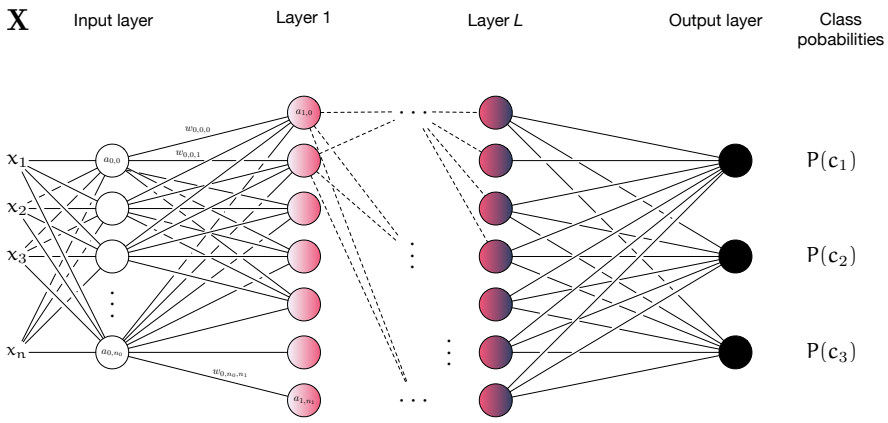


Figure 1.1: Schema of a multi-layer neural network. $\mathbf{x}_i$ are the input values, $a_{l_n,i}$ the i-th activation in layer l_n and $w_{l_n,i,j}$ the weight between the i-th unit in layer l_n and the j-th unit in layer l_{n+1}

ability of computing power, this began to change, but more easily trainable models like Support Vector machines eclipsed neural nets for most applications. Architectural advances, such as the convolutional neural networks in the late 1980s (LeCun et al., 1989) 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 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 1.1 gives a graphical representation of a multi-layer feedforward network.

Training a neural network simply involves a loss function measuring the distance of the networks 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 descent optimisation algorithm. An introduction to the mathematics of deep neural networks is deferred to section 3.1.

1.2 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 by the use of appropriate metrics. 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

1.3 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 be annealed)
- 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 can be prohibitively expensive. It is therefore desirable to notice 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

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))

of useful debugging tools can lead to a lot of wasted time and money in diagnosing problematic behaviour of a neural network.

Without years of training and a lot of mathematical intuition and expertise, it is often very hard to figure out why a network is not learning or how to ensure timely 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 a some of monitoring tools (see section 1.4), 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 enablement to quickly iterate on hypothesised metrics extracted from the training in order to diagnose problems such as those outlined in section 1.2. Most deep learning research 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, decreasing opacity of the training process and a research tool for abstracting away some of the nuisances of machine learning experimentation and simplifying experiments for live metrics in neural networks.

1.4 EXISTING APPLICATIONS

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

TensorBoard

TensorBoard is a visulisation 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.

For exporting data during training, the developer adds operations to the 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 artifact 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[3]).

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 fulfills 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.

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.

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

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 1.4) is that it enables working at a higher level of abstraction, liberating the developer from having to repeat herself, exchanging visualisations and metrics and reduce 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).

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 this purpose. 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.

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 < tensor:
    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, tensor)
# loop body
def body(counter):
    h = tf.add(tf.matmul(W, h), b)
```

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.

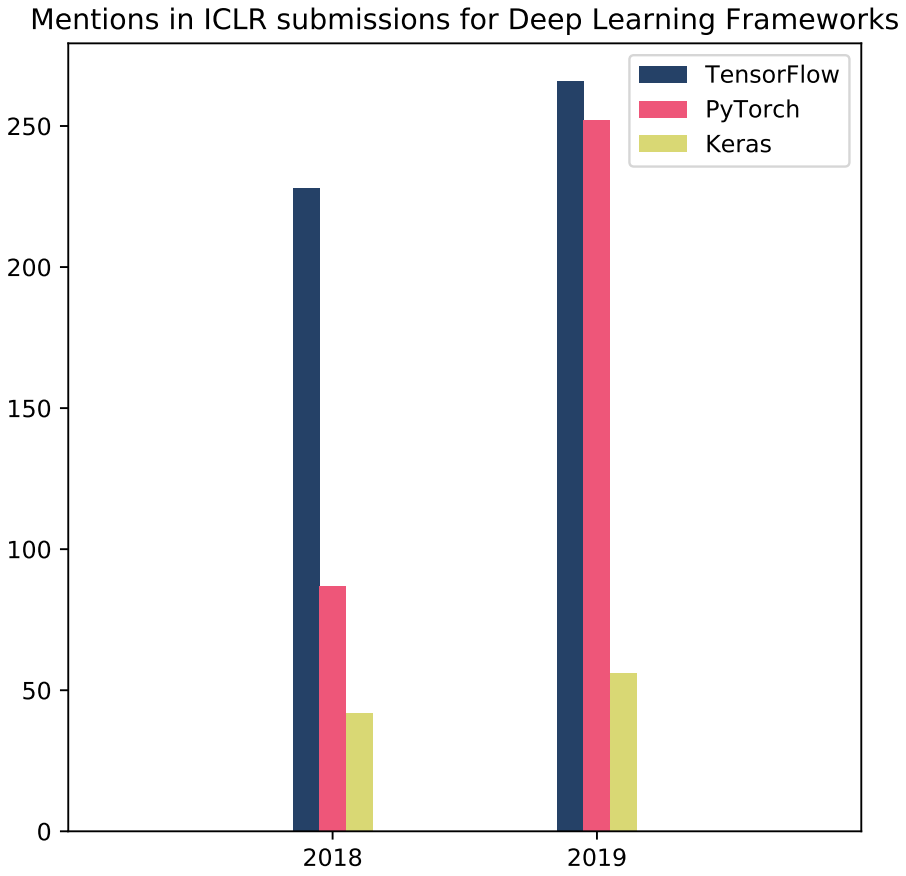


Figure 2.1: Changes in popularity of different deep learning libraries in research. Data was collected by keyword search over ICLR submissions (http://search.iclr2019.smerity.com/search; analogously for 2018)

```
    # 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 she knows 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.

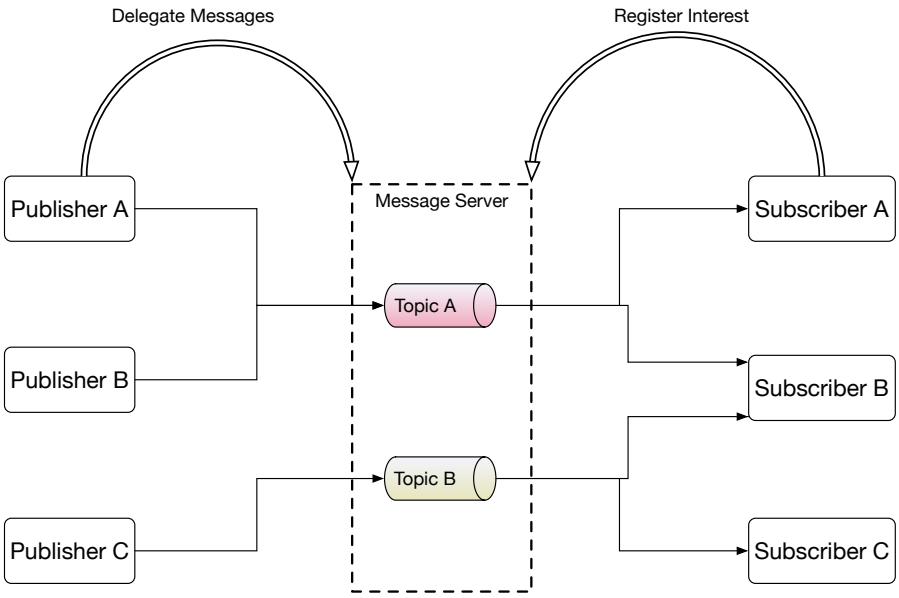


Figure 2.2: One possible implementation of the Publisher-Subscriber pattern.

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 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.

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

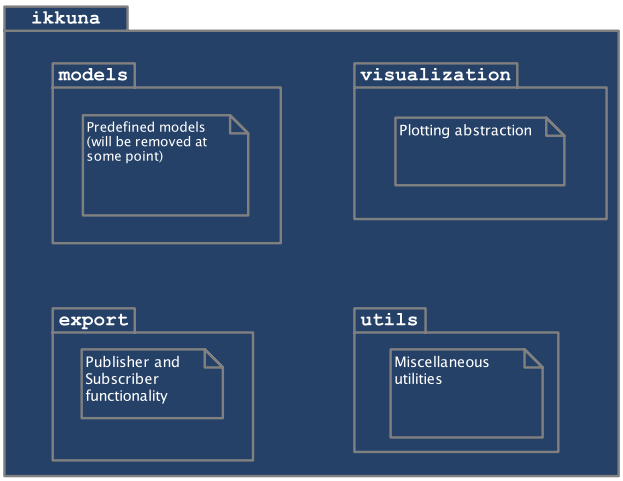


Figure 2.3: `ikkuna` package diagram

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.

2.4 OVERVIEW OF THE LIBRARY

The software is structured into several packages. The root package is `ikkuna` which encapsulates all core functionality. All 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` (see section 2.4.2) subpackage 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` (see section 2.4.3) subpackage 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 `export` subpackage*

The `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

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

Table 2.1: `ikkuna.export` functionalities

Name	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

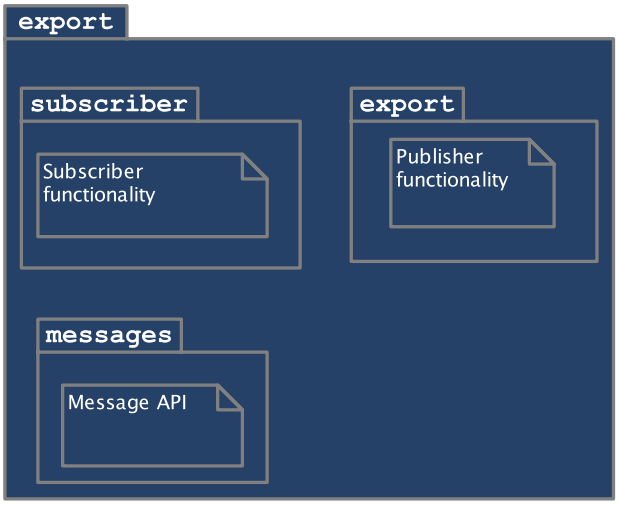


Figure 2.4: `ikkuna.export` package diagram

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

The `ikkuna.export` subpackage

In in slight deviation from the Publisher-Subscriber framework as displayed in figure 2.2, the `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 creation of the model 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 do not want the `Exporter` to generate new messages for this occasion (this could even lead to infinite loops).
- 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, but is easier to do by patching the loss function set with `set_loss()`.

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()`.

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 enters the layer, as could be the case during inference/testing 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.

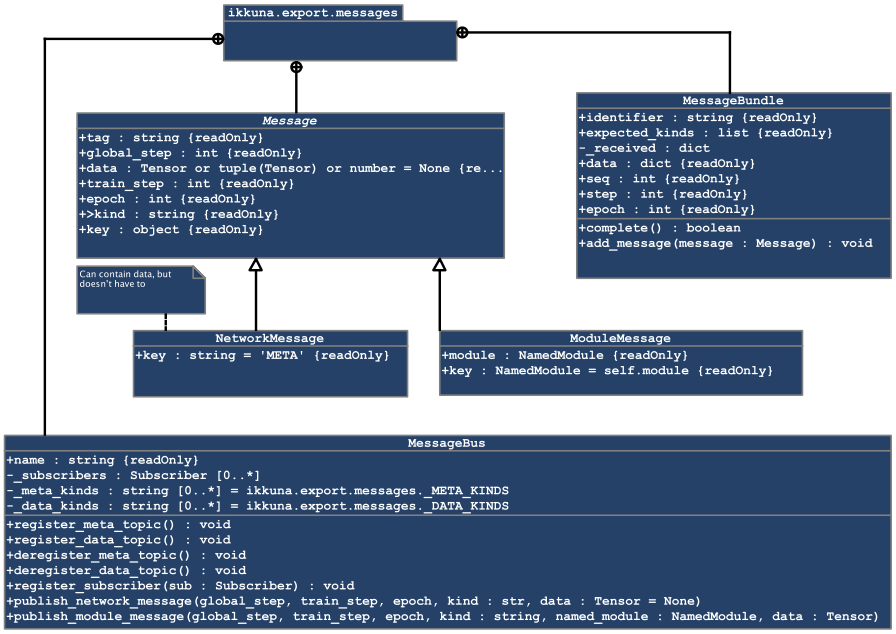


Figure 2.5: Classes in the `ikkuna.export.messages` submodule

- 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 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 training step applying to the entire network. In that case, they appear only once per 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.

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

Messages can be assembled into bundles if a subscribers wants to subscribe several topics at once. The `MessageBundle` class performs all necessary error checking to ensure consistency of the contained messages (only messages of 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 subscribers 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

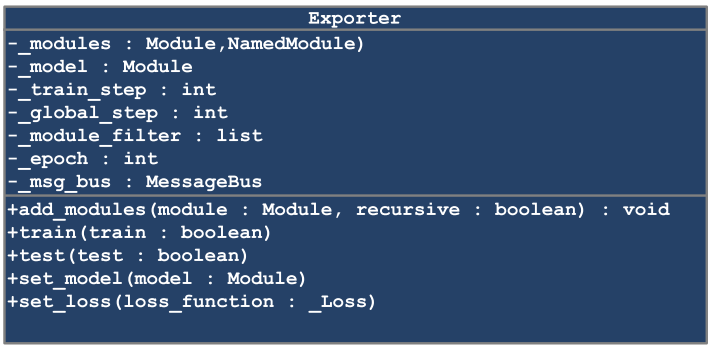


Figure 2.6: `ikkuna.export.Exporter` class diagram

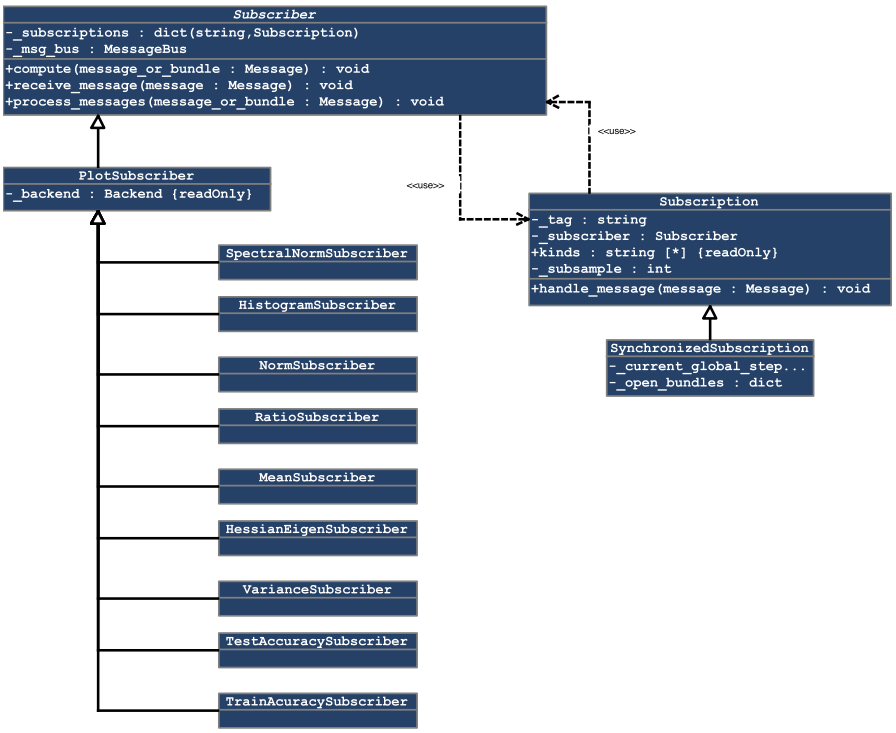


Figure 2.7: Classes defined in `ikkuna.export.subscriber`

message stream, routing only relevant messages to the subscriber and counting the received messages.

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

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

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].

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

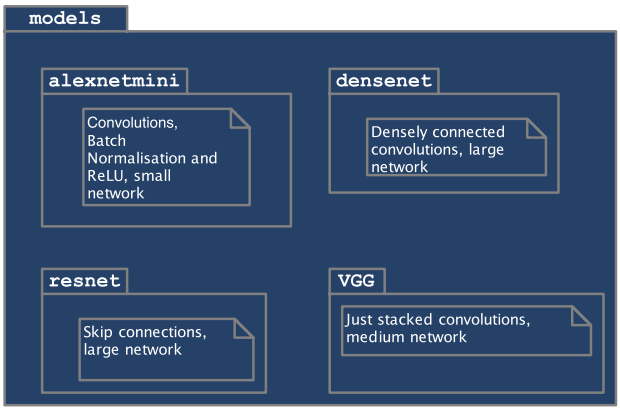


Figure 2.8: `ikkuna.models` package diagram

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.

4. VGG18, a variant of the deep convolutional network introduced in (Simonyan and Zisserman, 2014). The code is adapted from the PyTorch model zoo's implementation (BSD license).

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].

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`

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.

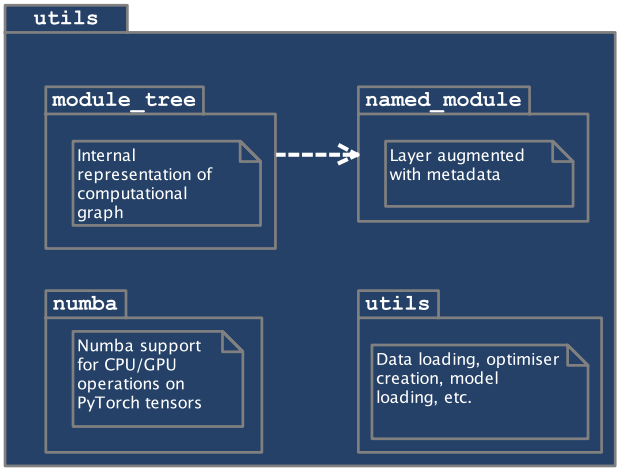


Figure 2.9: `ikkuna.utils` package diagram

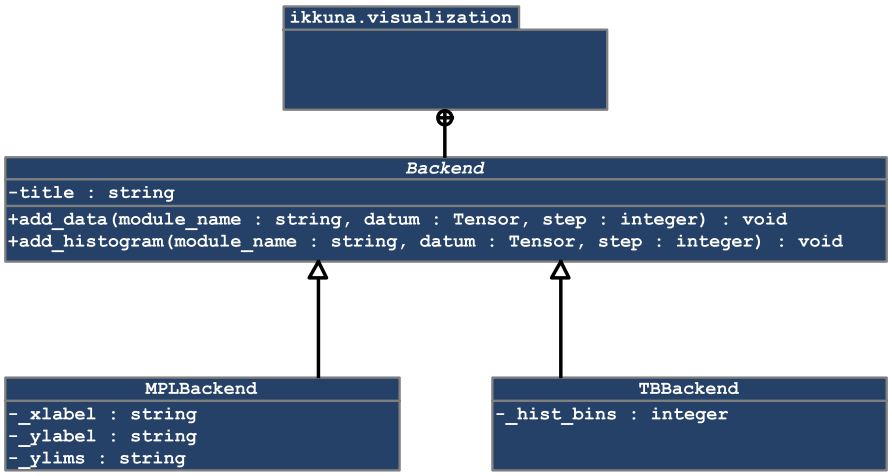


Figure 2.10: Class diagram for classes in `ikkuna.visualization`

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 plotting backend for TensorBoard (see section 1.4) is also provided. The plotting data is generated and processed on the remote system, but served over the web so it

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`

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.

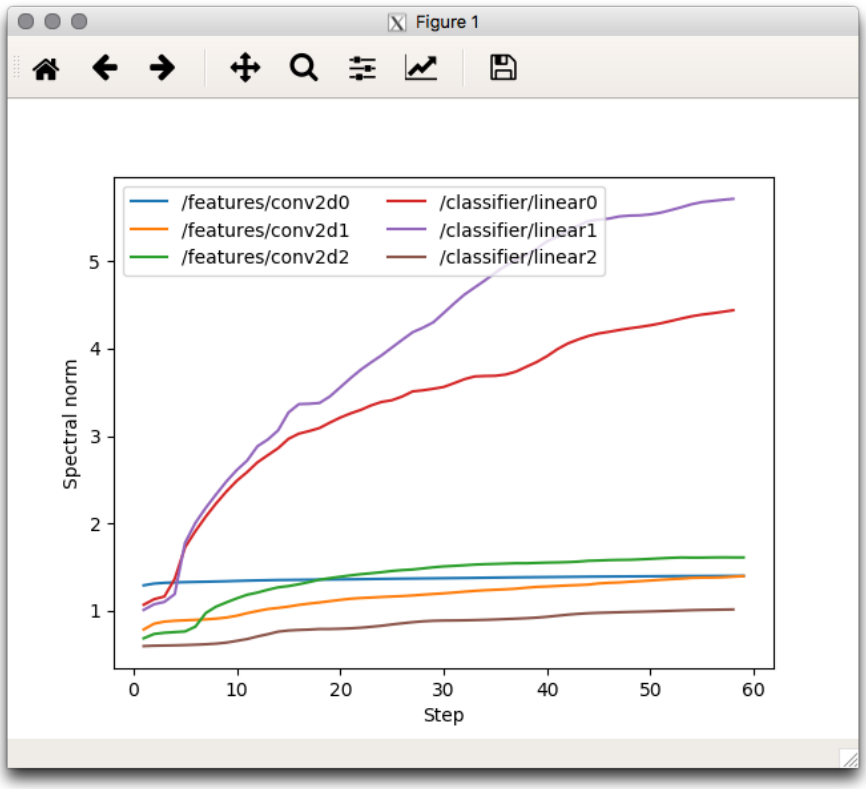
2.4.5 *Miscellaneous tools*

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.

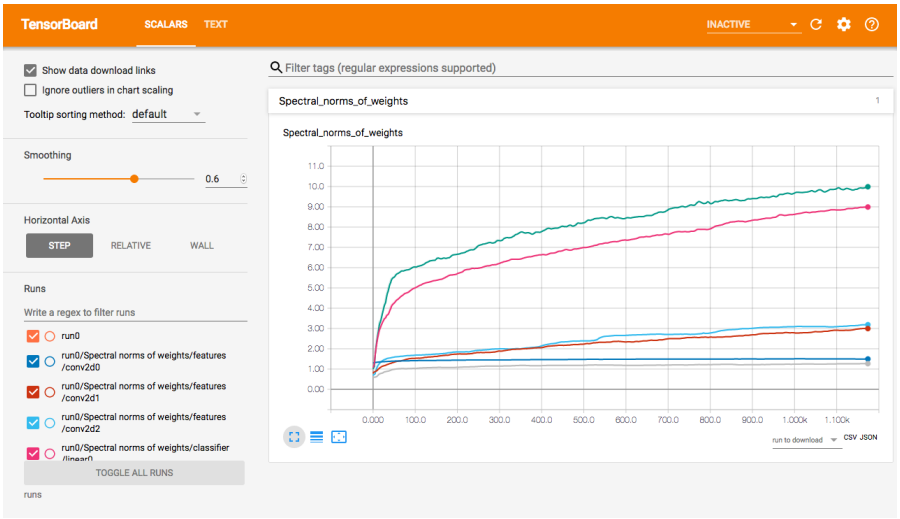
The `train` package defines a `Trainer` class which encapsulates all the logic and parameters needed to train a neural network on one of the datasets provided with PyTorch. The class's capabilities include the following

- 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
- 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.



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



(b) Exemplary view of a TensorBoard session

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 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 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

```
```

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

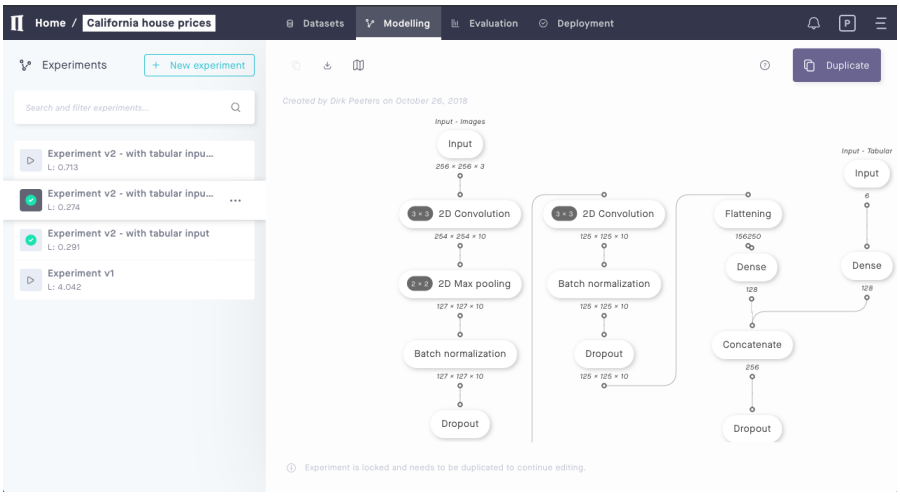


Figure 2.12: The Peltarion platform modeling screen

2.4.7 *Documentation*

The entire codebase is liberally documented using the Sphinx documentation processor[10]. 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[11], 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.

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 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 1.3, 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

10 `http://www.sphinx-doc.org/`
11 `http://www.peltarion.com/`

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 obtain such metrics, not actually use them. This entails that the library is used primarily for easily gathering all the data from the necessary experiments, and not for supervising the actual training.

The chapter is divided between a reproduction part (see ??) and an original research part (see ?? and ??). In the former, experiments are conducted for two popular variations of gradient descent 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 respective publications. 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 preprocess. Recall from section 1.3 the multitude of hyperparameters for a deep learning model and training regimen. In this work, we will concern ourselves with the question of figuring out a good learning rate and when to stop training layers to reduce computation time.

This thesis is not concerned with advancing the state of the art in classification. Instead, toy problems are developed 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 REVIEW: 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 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 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:

$$J(\theta) = \mathbb{E}_{x,y\sim\hat{p}_{\text{data}}} L(\mathbf{x}, y, \theta) \tag{3.1}$$

$$= \frac{1}{m}\sum_{i=1}^{m} L(x^{(i)}, y^{(i)}, \theta) \tag{3.2}$$

The cumulative loss can then be derived for θ

$$\nabla_\theta J = \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta L(x^{(i)}, y^{(i)}, \theta) \tag{3.3}$$

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

$$\theta \leftarrow \theta - \eta \nabla_\theta J \tag{3.4}$$

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.

3.2 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`.

3.3 DETECTING LEARNING RATE PROBLEMS

3.3.1 *Ratio-Adaptive Learning Rate Scheduling*

The first problem to be investigated is that of choosing an appropriate learning rated. We begin with a brief review of gradient descent and then evaluate how the update-to-weight ratio of model weights can be used to identify bad learning rates or automate learning rate selection.

As a first hypothesis, we investigate a claim made by Karpathy (2015) who states that the ratio between updates and weights 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 (the network does learn, but much slower and to smaller final accuracy; plots are omitted for brevity). It is also not clear whether the target should apply to all layers equally. For instance, in networks with the (now outdated) tanh activation function, the later layer's 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.

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.

The experimental setup uses the simplified AlexNet architecture (section 2.4.2) shown in figure 3.1. The dataset used is the well-known CIFAR-10 dataset consisting of 60.000 32×32 colour images from ten object categories (Krizhevsky and Hinton, 2009). The optimisation algorithm used is plain stochastic gradient descent on minibatches of size 128. 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 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 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

$$\eta_{e+1} = 0.98^{e+1}\eta_e, \tag{3.5}$$

e being the epoch index. The final accuracies after 100 epochs of training for constant learning rate, exponential decay are shown in figure 3.2. As can be seen, there is a significant improvement when decaying the learning rate over keeping it constant.

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 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

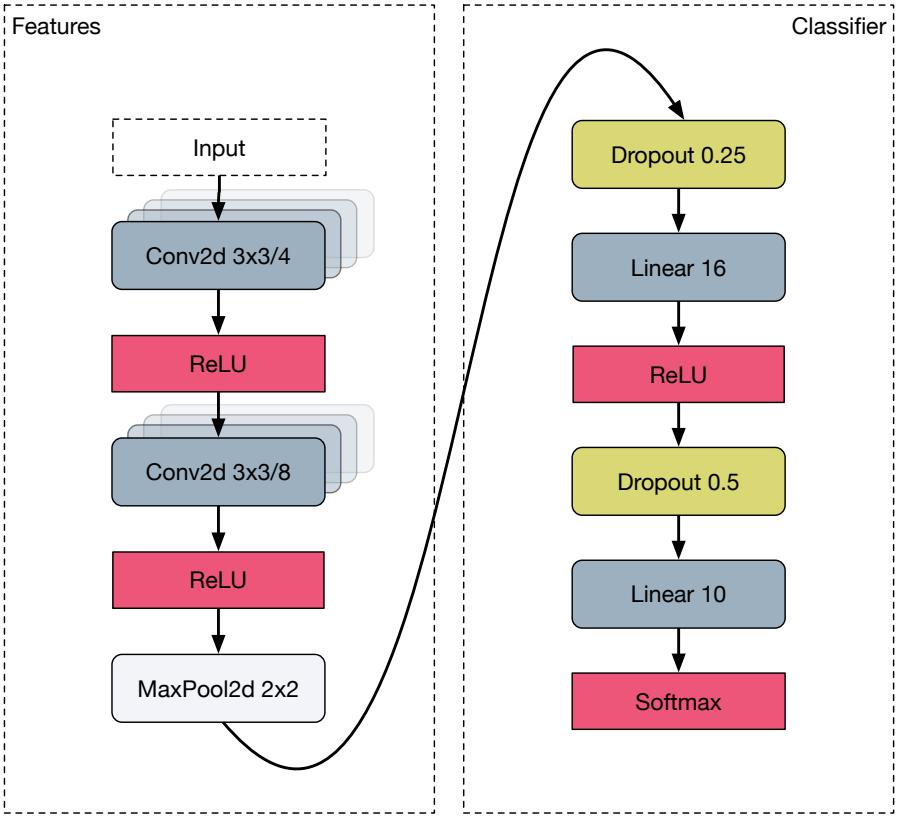


Figure 3.1: The network used in this experiment. Image features are extracted by 4 and 8 convolutional filters, respectively, with ReLU nonlinearities. Maximum pooling is applied with a filter and stride size of 2 leading to a resolution a fourth of the original size. The classifier portion employs dropout layers to reduce co-adaptation of units and a final softmax activation to map outputs to class probabilities in $(0, 1)$.

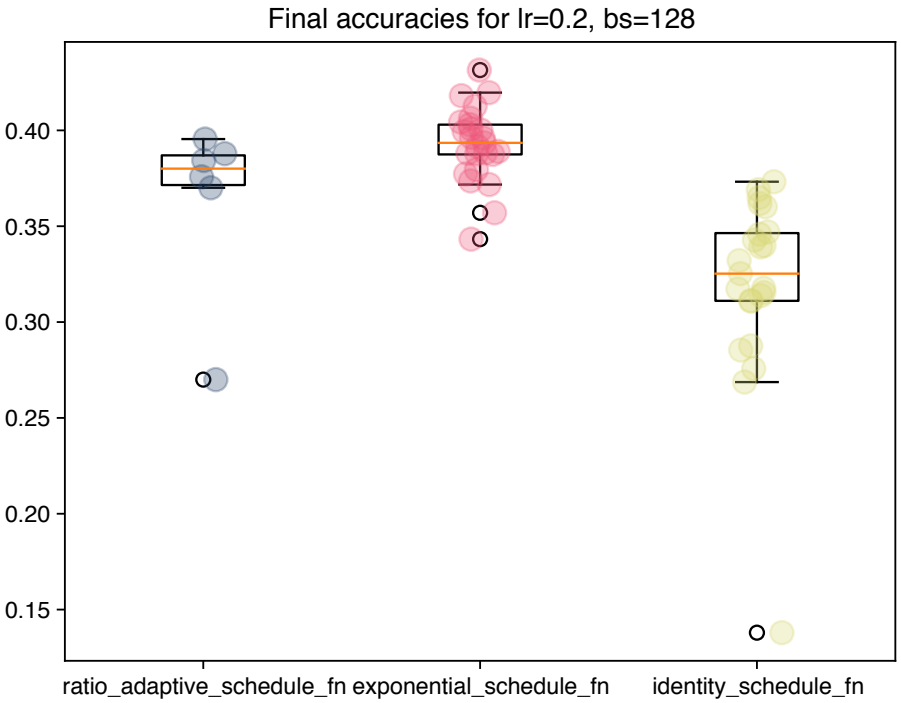


Figure 3.2: Final accuracies after 100 epochs with a learning rate of 0.2 and batch size of 128. The adaptive ratio schedule is discussed insection 3.3.1

$$R_{i,k} = \frac{\|W_{i,k} - W_{i,k-1}\|_2}{\|W_{i,k}\|_2} \tag{3.6}$$

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

$$\eta_{k+1} = \eta_k \left(t \cdot \frac{1}{l} \sum_{i=0}^{l-1} \gamma R_k + (1-\gamma) R_{k-1} \right)^{-1} \tag{3.7}$$

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 3.3 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'})
```

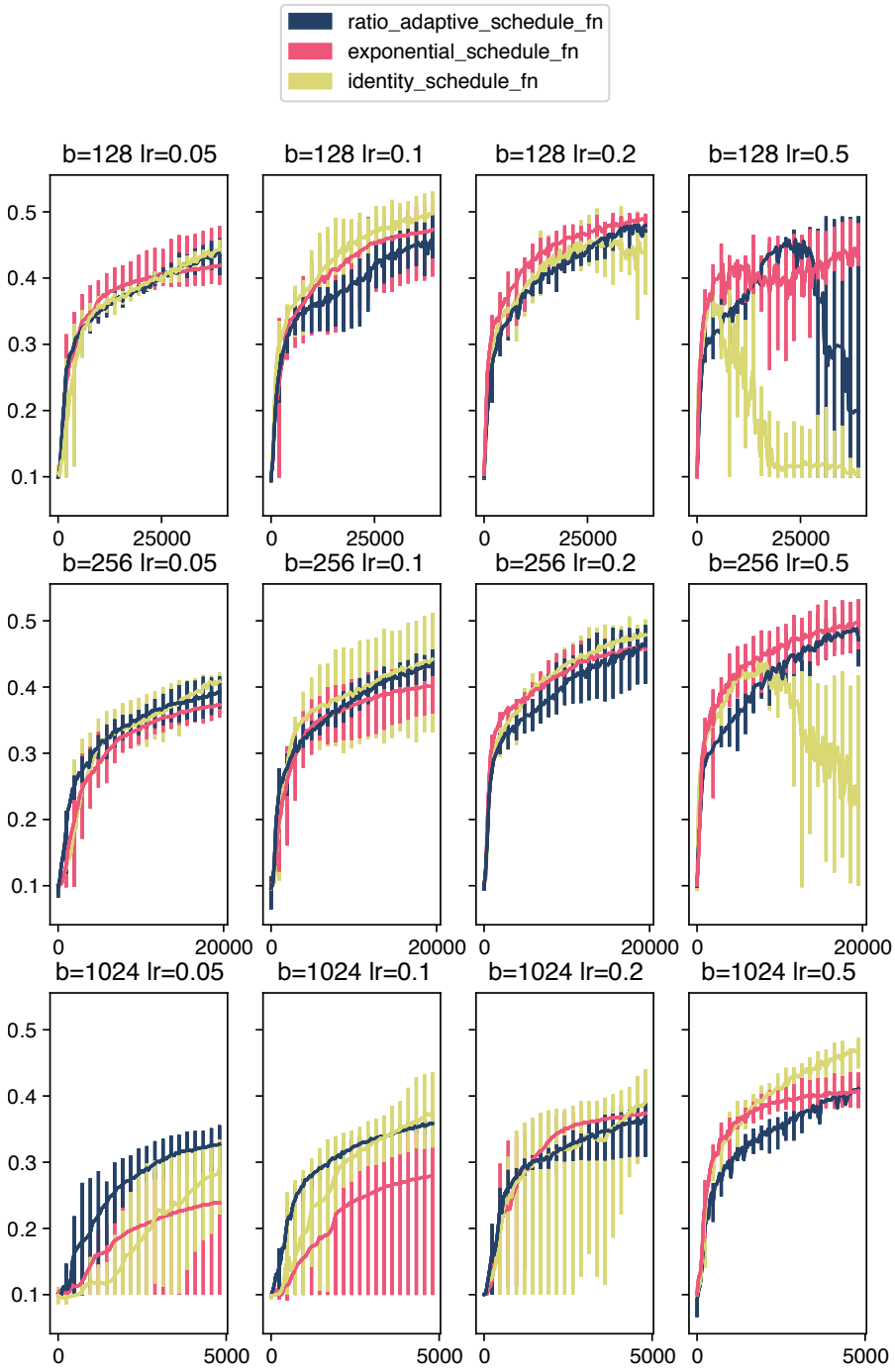


Figure 3.3: Accuracy traces for different schedules on CIFAR-10. Batch size and initial learning rate are shown above each subplot. The exponential decay schedule uses a decay factor of 0.98. Chance level is 0.1. The experiments are run for 100 epochs each.

```
        # 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)
        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
```

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 ??). 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.

In the following figures, we plot the loss traces and their inverse derivative (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

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.

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 4000 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. 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.

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 3.5c). Counterintuitively, the adaptive optimiser is unable to adapt to the high learning rate and is outperformed by vanilla SGD.

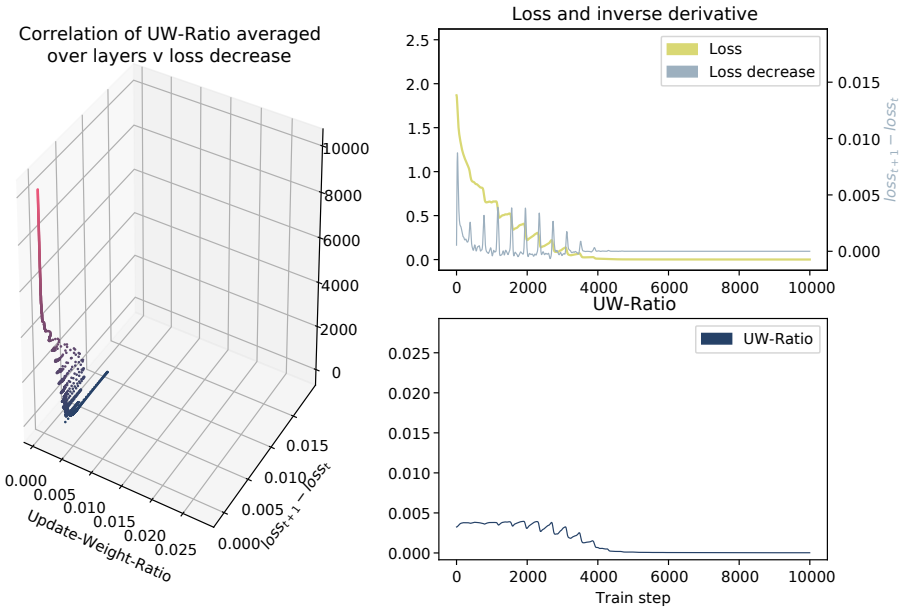
Remarkably, the qualitative behaviour is the opposite of SGD, which becomes visible when omitting the first few hundred training steps. While for SGD (plots omitted for brevity), higher learning rates lead to higher initial values in the UW distribution, the opposite seems to hold for the adaptive optimizer. This can be seen in figures 3.6a to 3.6c 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. We will investigate Adam more closely in ??

Other Architectures

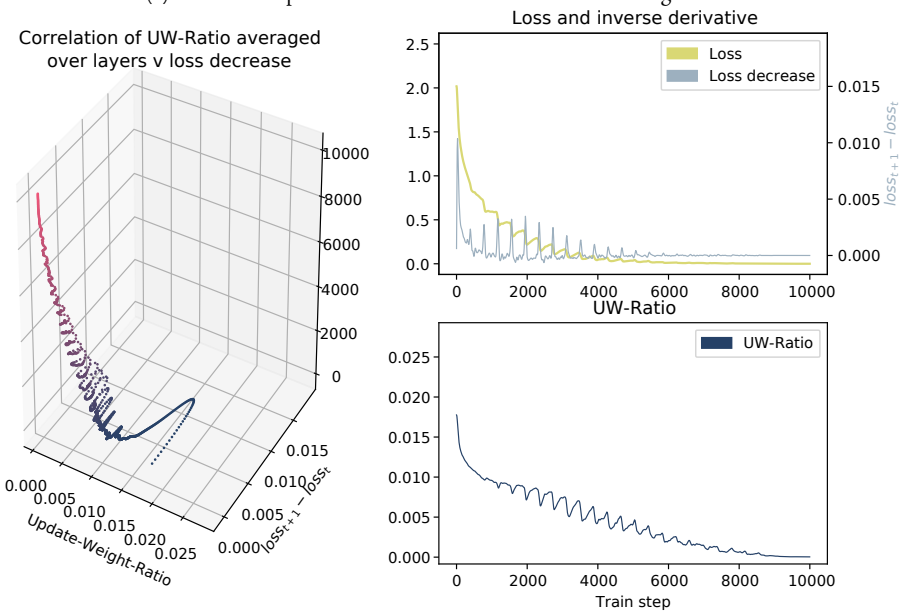
Several analyses were conducted on the data, but not displayed here due to inconclusive results. The experiments were run on the AlexNetMini architecture as well, without results, as the network does not converge within 75 epochs. The same experiment on the ResNet18 architecture exhibited the same qualitative 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 5000 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.

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.

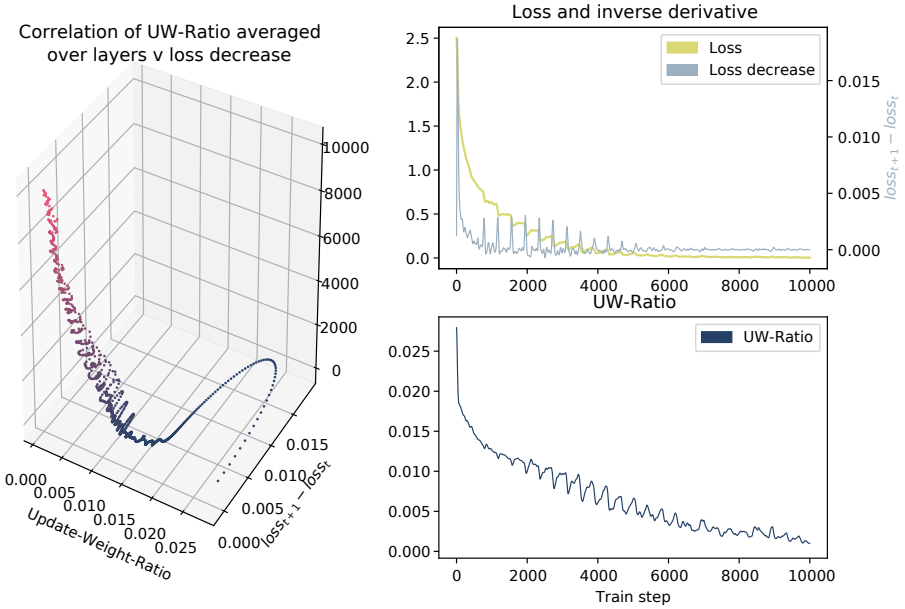
We now want to check how the UW ratio relates to the loss when we attempt to artificially fix the ratio value to a target of 0.001. Since the updates to the weights



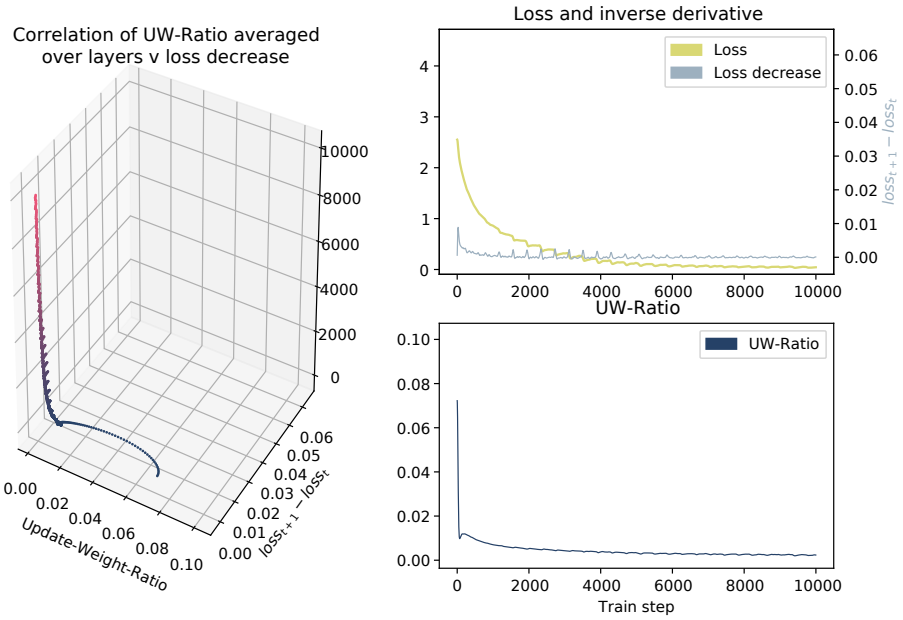
(a) UW ratio experiment for VGG with SGD and learning rate 0.01



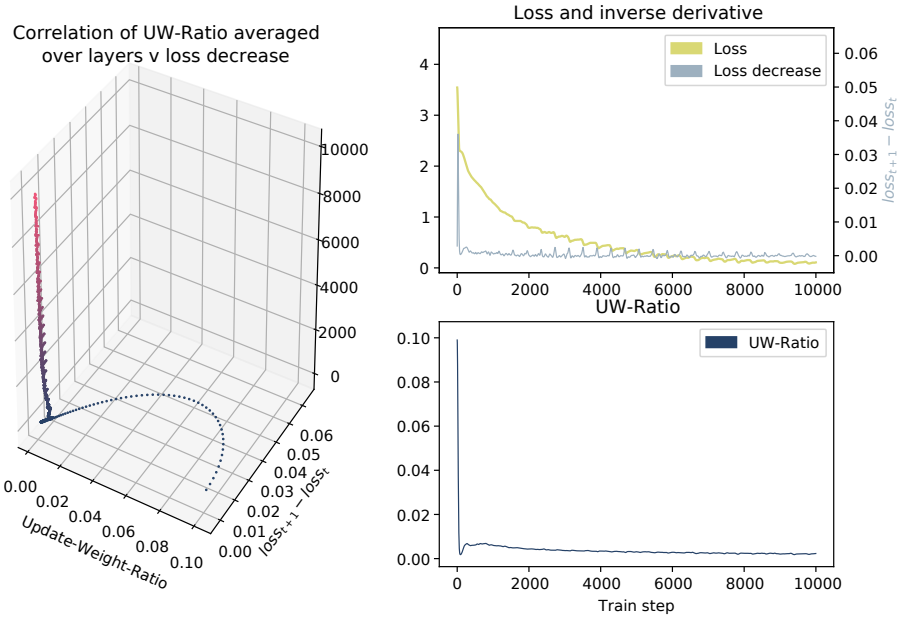
(b) UW ratio experiment for VGG with SGD and learning rate 0.05



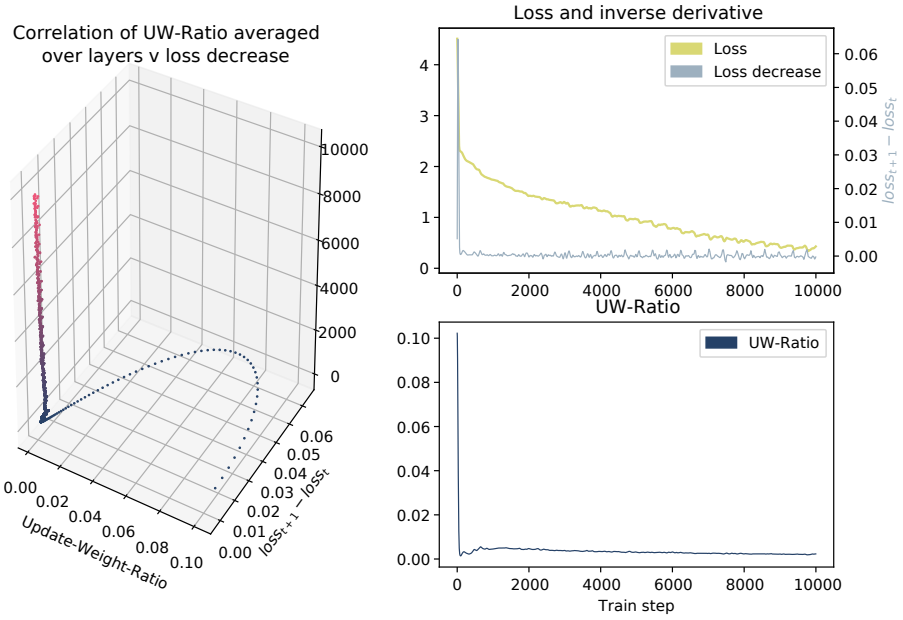
(c) UW ratio experiment for VGG with SGD and learning rate 0.1



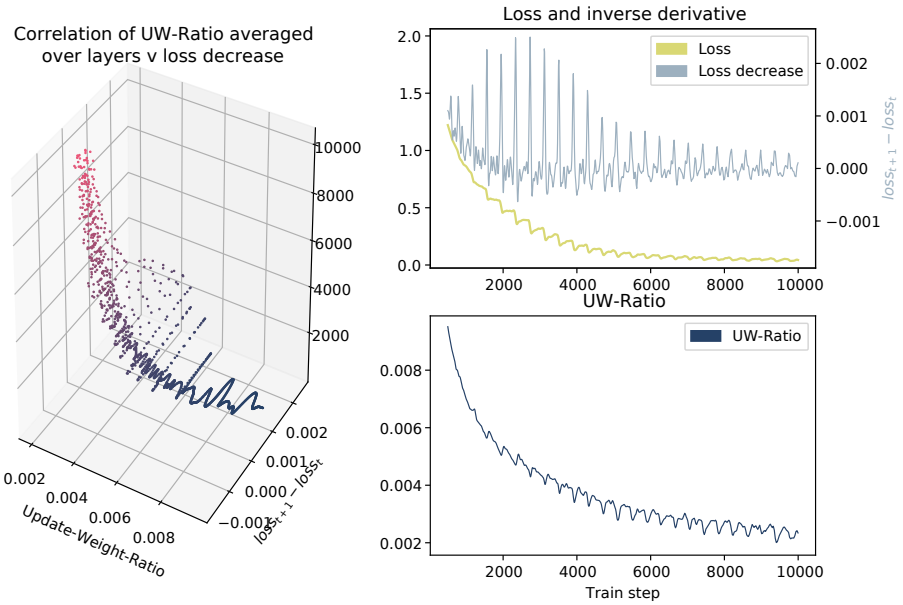
(a) UW ratio experiment for VGG with Adam and learning rate 0.01



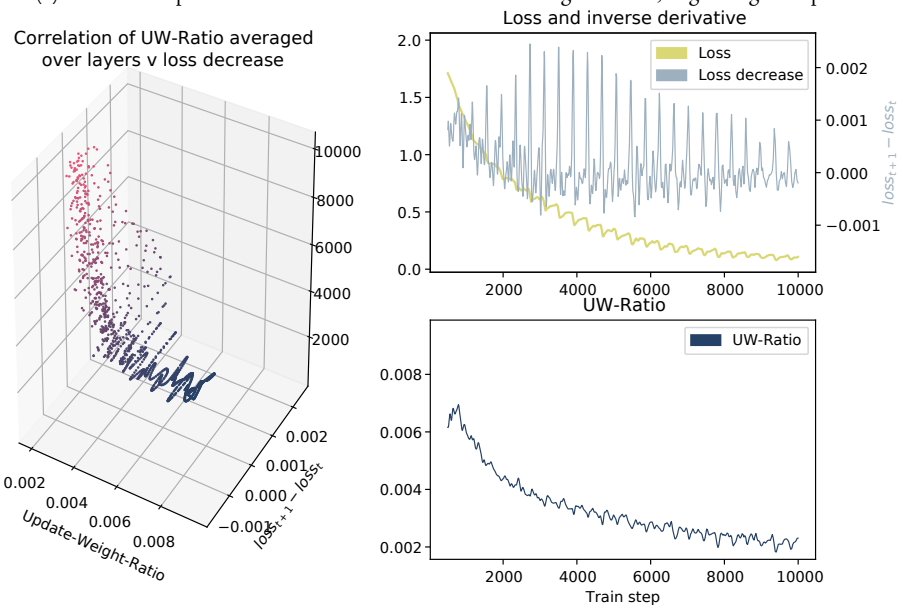
(b) UW ratio experiment for VGG with Adam and learning rate 0.05



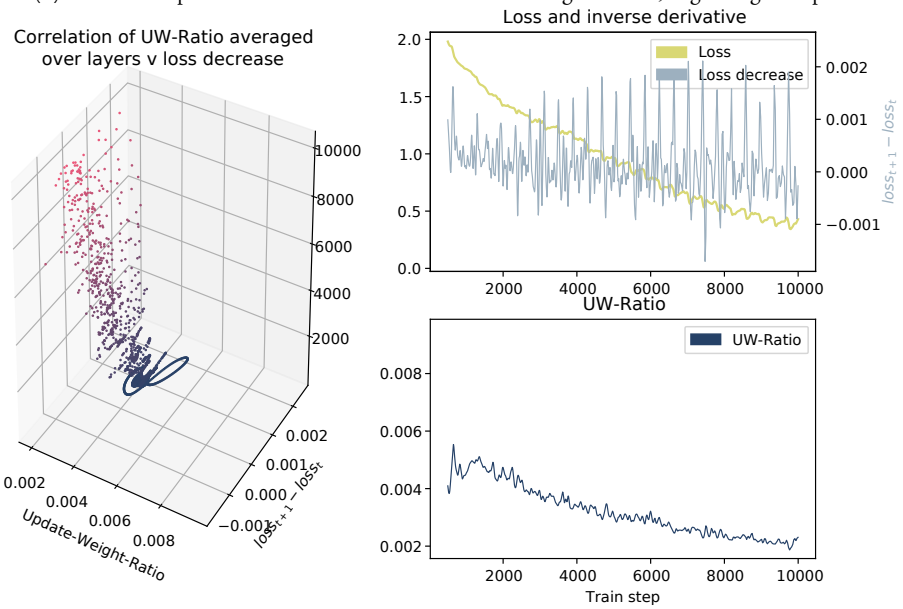
(c) UW ratio experiment for VGG with Adam and learning rate 0.1



(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

in vanilla SGD (see section 3.1) are a function of the learning rate and the gradient magnitude, this could be done in one of two ways

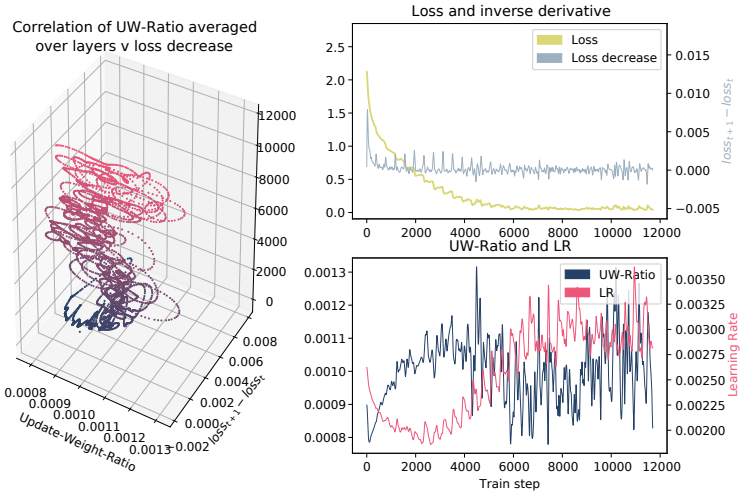
1. Scale the gradients with the constant learning rate in mind
2. Set the learning rate according to the ratio-adaptive schedule from section 3.3.1

Using the ratio-adaptive schedule results in the behaviour shown in figures 3.7a to 3.7c. 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.

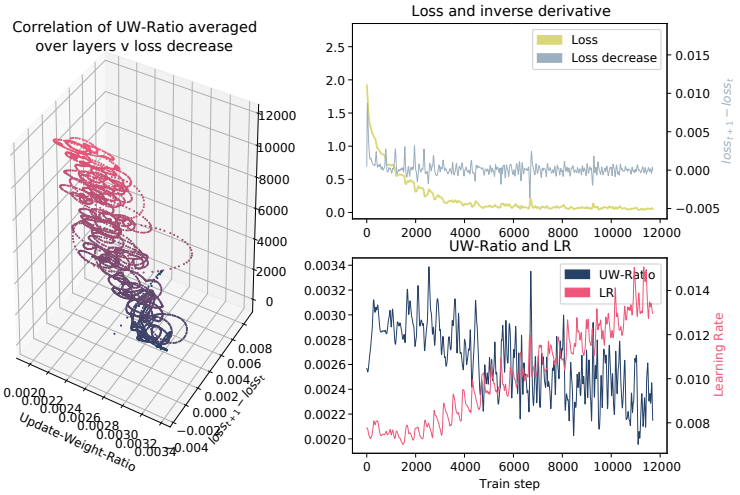
Note that for this set of plots, the axis limits are different. This serves to illustrate that the qualitative behaviour is identical for all learning rates (ignoring the fast arc at the beginning for $\mathrm{lr} = 0.1$). The learning rate trajectories are very similar; the learning rate rises as the UW ratio decreases. The primary insight from these experiments is that the empirical UW ratio is not linearly related to the learning rate. Recall from section 3.1 that the weight update is given by

$$\eta \nabla_\theta J, \tag{3.8}$$

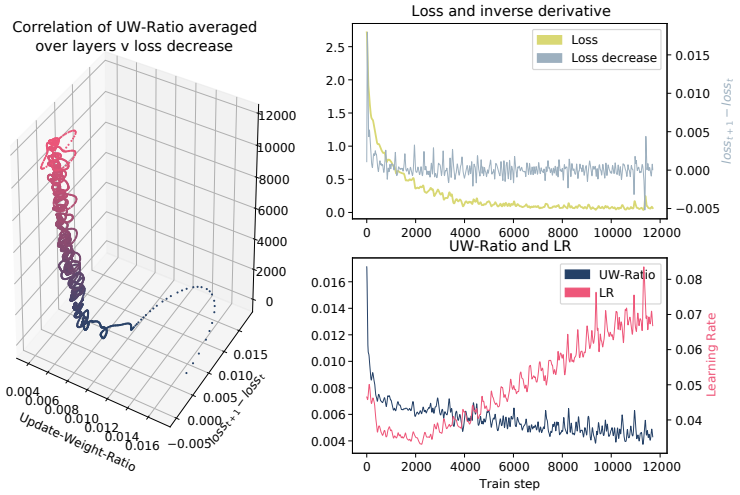
so it is a linear function of the learning rate and the gradient magnitude. We would therefore expect that a tenfold reduction in learning rate would result in an equal reduction in the UW ratio. However, the difference is not quite as pronounced in reality. Considering figure 3.7b and figure 3.7c we 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 assumed 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.



(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

4

FUTURE WORK

This chapter will examine some avenues of inquiry which for several reasons are beyond the scope of this work.

4.1 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.subscriber` 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 mimiser, 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.

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 Ubuntu operating system and the Linux kernel (Torvalds et al., 2008) providing the computational environment for the experiments conducted for this work, alongside the clang (Lattner and Adve, 2004) and GNU compiler toolchains for compiling software for the system

3. The TMUX and SSH tools for connecting to the system running the experiments

4. 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.

5. The Sacred (Klaus Greff et al., 2017), MongoDB and Pymongo libraries used for logging experiments to a database for later visualisation

6. The Vim editor and associated plugin ecosystem employed for creating all documents, be it code or documentation

7. 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
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.

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

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

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.

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

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.

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.

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

Golmant, N. (2018). Pytorch-hessian-eigenthings. https://github.com/noahgolmant/pytorch-hessian-eigenthings/.

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.

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

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

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

Karpathy, A. (2015). Convolutional neural networks for visual recognition lecture notes. http://cs231n.github.io/neural-networks-3/#ratio.

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*.

Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

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.

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.

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

Lattner, C. and Adve, V. (2004). Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA. IEEE Computer Society.

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

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

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*.

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.

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*.

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

Saad, D. (1998). Online algorithms and stochastic approximations. *Online Learning*, 5.

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

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

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.

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

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

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*.

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 traveling with me and 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 willigness to let me work on my own terms on what I saw fit.

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

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.

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 list of software used during the creation of this thesis can be found in the appendix.

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, December 12, 2018