UNIVERSITÄT OSNABRÜCK

Department of Computer Science
Department of Cognitive Science

Rasmus Diederichsen

Live Introspection for Neural Network Training

Master's Thesis

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

ABSTRACT

Artificial neural networks have become the prevalent model class for many machine learning tasks, including image classification, segmentation, video and audio analysis, or time series prediction. With ever increasing computational resources and advances in programming infrastructure, the size of model we can train also increases. Nevertheless, it is not uncommon for training to take days or weeks, even on potent hardware. While there are many obvious causes – e.g. inherent difficulty to parallelize training with the most successful algorithms – there may still be inefficiencies in the framework we typically use for training neural networks. 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 Sanjeev Arora's talk at ICML 2018 [1]). 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 reasearch 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 debugging tools available to every programmer on every level of abstraction far exceed 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 practitioners. A standard approch to choosing a parametrization remains trial-and-error, or only slighty 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 when to tweak certain parameters could thus provide large savings in time and money, besides enabling a more thorough understanding of what is going on.

This thesis addresses the above in two ways: A software library aiding in deep learning debugging and experimentation is designed and implemented. The same library is then used to investigate experimentally, whether heretofore unknown signals can be extracted from the training process in order to validate parameter choices while training is running.

1 Toward Theoretical Understanding of Deep Learning

ZUSAMMENFASSUNG

Künstliche neuronale Netze sind inzwischen die populärste Art von Modell für viele Anwendungen im Maschinellen Lernen, z.B. Bildklassifikation, Segmentierung oder Video- und Audioanalyse sowie Zeitreihenvorhersage. Mit immer zunehmenden Ressourcen und Fortschritten in verfügbarer Software steigt auch die Größe der Trainierbaren Modelle. Dennoch sind Trainingszeiten von Tagen und Wochen – selbst auf potenter Hardware – nicht ungewöhnlich.

Es gibt naheliegende Ursachen – z.B. die Schwierigkeit, das Training zu parallelisieren aufgrund der eingesetzten Optimierungsalgorithmen – jedoch möglicherweise auch weninger offensichtliche Verluste in der Art und Weise, wie Modelle normalerweise trainiert werden.

Während der praktische Erfolg tiefer neuronaler Netze rapide war, konnte die theoretische Erschließung nicht Schritt halten, und viele fortschrittliche Ideen sind mehr von Intuition und Experimenten gestützt als von rigorosen Formalismen. Für die meisten erfolgreichen Ideen haben wir folglich kein klares Verständnis dafür, warum sie in der Praxis gut funktionieren.

Diese Kluft zwischen Praxis und Theorie ist auch ursächlich dafür, dass Deep-Learning-Applikationen mehr als Kunst denn als Wissenschaft gilt. Anders als in der traditionellen Softwareentwicklung, die auf Jahrzenten der Forschung in Ingenieurswissenschaften, Mathematik und theoretischer Informatik basiert, gibt es im maschinellen Lernen oft keinen definitiven Ansatz, um ein konkretes Problem zu lösen. Des Weiteren sind die verfügbaren Debugging-Wekrzeuge in jeder Schicht der Softwareentwicklung denen des Deep Learning bei weitem überlegen. Einfache Fragen wie "Lernt das Modell das, was ich möchte?" sind bislang nicht zu beantworten. Wir können hier also eine Notwendigkeit für mehr nützliche Tools für Anwender konstatieren.

Der Standardweg, eine gute Parameterwahl für das Modell zu treffen ist Trial-and-Error oder etwas intelligentere Variationen. Der Rechenaufwand für das Training großer Modelle verhindert schnelles Experimentieren und schlägt sich häufig auch in finanziellen Kosten nieder. Frühzeitig Sackgassen zu identifizieren oder Zeitpunkte, an denen man bestimmte Parameter ändern sollte, könnten daher große Ersparnisse bedeuten.

Diese Arbeit stellt zum Einen eine Softwarebibliothek vor, die im Debugging von Deep-Learning-Anwendungen hilft und das Experimentieren erleichtert. Zum Anderen wird dieses Werkzeug benutzt, um experimentell festzustellen, ob aus dem Trainingsprozess Signale extrahiert werden können, die Aufschluss über die Wahl von Parametern geben, während das Training läuft.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 INTRODUCTION

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 and classification. At it's core, neuronal models are simply conceptualized as an arrangement of units which receive 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 (Rosenblatt, 1958). With the introduction of the backpropagation algorithm (Werbos, 1975) and increasing availability 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 1980's (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 cornerstone for many AI applications today.

A neural network consist 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.

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
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 to be investigated are
    - bad initializations
    - inappropirate learning rate

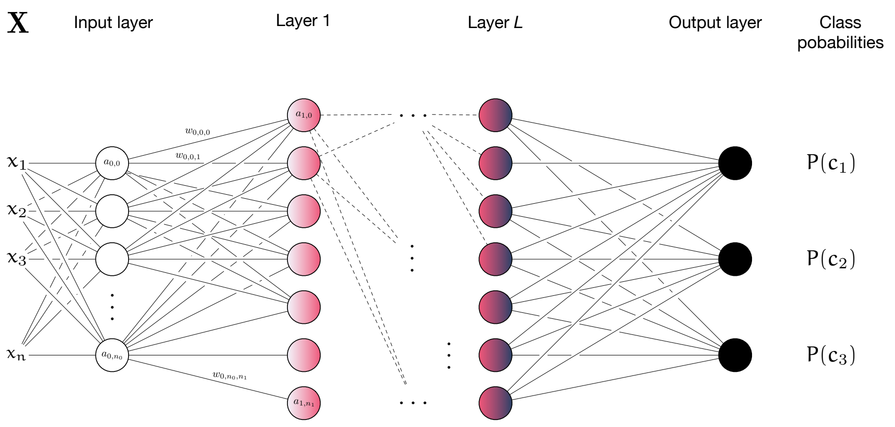


Figure 1.1: Schema of a multi-layer neural network. $\mathbf{x}_i$ are the input values, $a_{l_n,i}$ the i-th activation in layer l_n and $w_{l_n,i,j}$ the weight between the i-th unit in layer l_n and the j-th unit in layer l_{n+1}

- layer/model saturation
- inappropriate network architecture
- bad generalization/overfitting

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, it can be seen that traditional methods such as SVMs, Gaussian Processes, Decision Trees or Gradient Boosting typically require less than 10 algorithmic parameters. [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

Other parameters that need to be set initially are

- Network architecture (how many layers, how many units per layer)
- nonlinearity function
- type of loss
- optimization algorithm
- initial learning rate

1 A look through scikit-learns selection of regressors and classifiers shows most classes require between 5 and 10 parameters. (Pedregosa et al., 2011)

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))

- momentum
- weight decay

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 maximize convergence speed.

This thesis work is motivated by the scarcity of useful tools to debug and monitor deep learning training. Without years of training and a lot of mathematical intuition and expertise, it is often very hard to figure out why a network isn't learning or how to ensure timely convergence. And even with this expertise, visualizations or metrics need to be implemented over and over again because common tools do not abstract from the concrete model architecture.

There exist a some of monitoring tools (see section 1.4), but they are mostly low-level tools which provide visualization primitives (drawing and interacting with graphs). They may enable visualization 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 modularizing 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 hypothesized metrics extracted from the training in order to diagnose problems such as those outlined in section 1.2

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 for evaluating hypotheses about training metrics.

1.4 EXISTING APPLICATIONS

TensorBoard

TensorBoard is a visulization 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 serialized data and displays it in the browser. The server can be used independently from TensorFlow, provided the data is serialized 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.

New kinds of visualizations 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 the programming language for talking to the server is Crayon which so far supports Python and Lua.

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 organizing the display of many visualizations 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, which offer insights into the training after the fact, but do not help guiding the training process while it's running.

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. Minimal configuration, maximum rewards.
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 is that it enables working at a higher level of abstraction, liberating the developer from having to repeat herself, exchanging visualizations and metrics and reduce the friction between development and debugging.

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 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 visualizations is desired. Not only will this aid in extending the framework to different deep learning libraries, but it is also a prerequisite for allowing for modular, self-contained visualizations 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

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 a framework such as PyTorch (Paszke et al., 2017) or Chainer (Tokui et al., 2015) , where there exist no two distinct execution phases – just like in an ordinary matrix library like NumPy, each statement immediately returns or operates on an actual value. By contrast, graph-based 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 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)
      # increment counter
      return [tf.add(counter, 1)]

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

- halting execution at aribtrary 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.

All frameworks have in common that they build a graph representation of the model, wether implicitly or explicitly. Nodes in the graph are operations while edges are data flowing between operations. This allows naturally parallelizing independent computations. To compute gradients, the graph can be traversed backwards from the output node by applying the chain rule of differentiation. 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 and also overloads operators for this purpose. The graph is thus recreated for each propagation through the network. This precludes some optimizations, but makes dynamically changing networks easily achievable.

For this work, the PyTorch framework has been chosen, due to the fact that it is growing quickly in popularity (see fig. 2.1) and relatively new, so the ecosystem is not fully developed and some utilities available for e.g. TensorFlow are not available for

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.

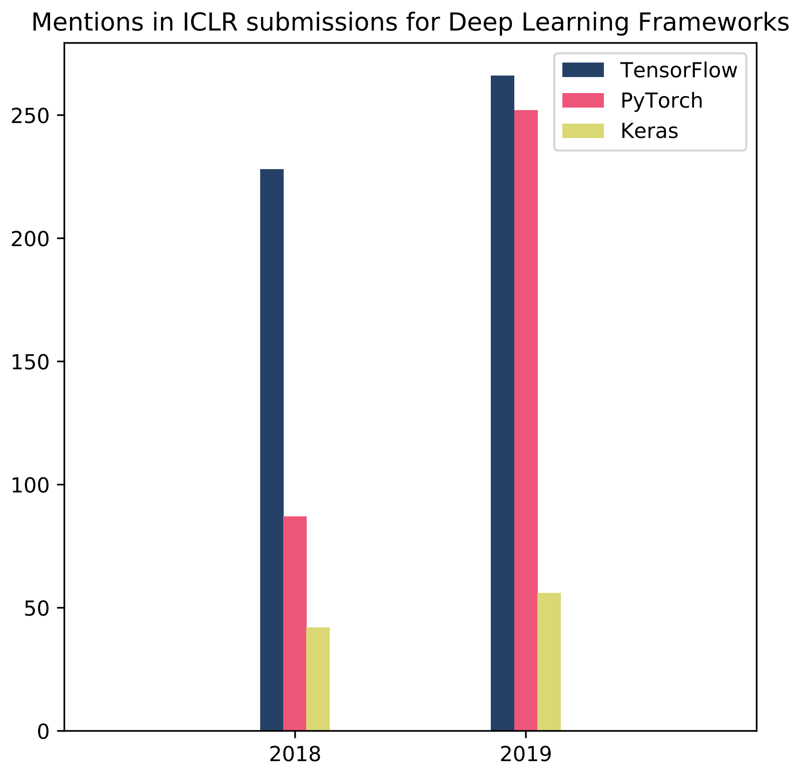


Figure 2.1: Changes in popularity of different deep learning libraries in research. Data was collected by keyword search over ICLR submissions (http://search.iclr2019.smerity.com/search; analogously for 2018)

PyTorch. Because of this, an introspection framework for training momnitoring is judged to present the best value proposition for PyTorch users.

2.3 PUBLISHER-SUBSCRIBER

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

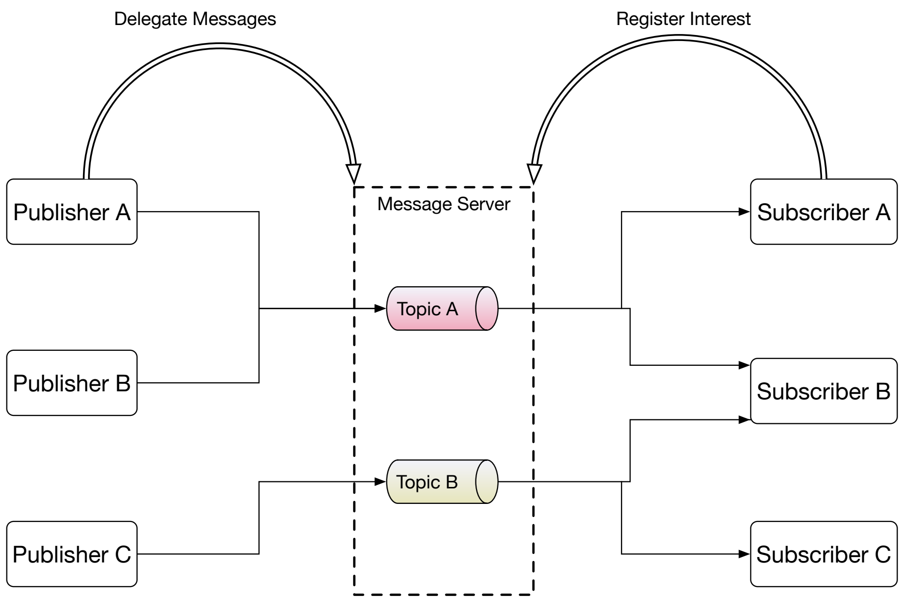


Figure 2.2: One possible implementation of the Publisher-Subscriber pattern.

Since real-world neural networks are trained on the GPU, and communication between host and GPU memory is expensive, making this library truly distributed 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, 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, if more computationally demanding metrics are to be explored.

In the context of neural network training, there is only one source of information and hence only one publisher. Therefore, the message server is folded into the singular publisher which extracts data from the training model and sends messages to any interested subscribers.

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 utilites 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.

The root package diagram is shown in fig. 2.3

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.

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

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

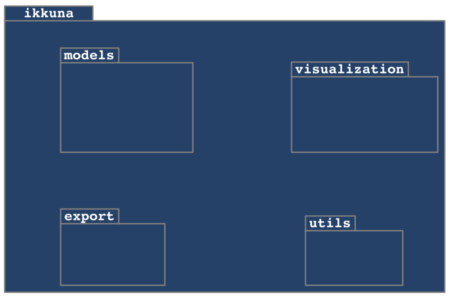


Figure 2.3: `ikkuna` package diagram

Table 2.1: `ikkuna.export` functionalities

Name	Function
`export`	Publish data from an arbitrary model and send messages to registered subscribers
`messages`	Define message interface; i.e. what topics exist and which information a message must contain
`subscriber`	Define the base class for metric subscribers

the training process, defining subscriber functionality and messages used for communication is done here.

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 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 fig. 2.4.

The package comprises three subpackages or modules

In in slight deviation from the Publisher-Subscriber framework as displayed in fig. 2.2, the `export.Exporter` class (fig. 2.5) is the sole publisher of data. There's only one source of data during training, so it is unnecessary to accomodate for

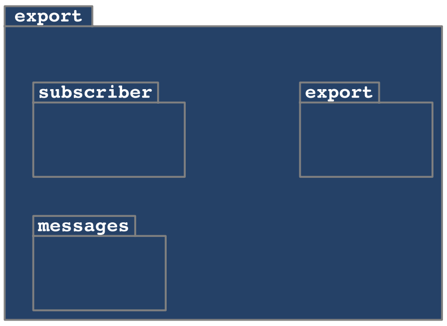


Figure 2.4: `ikkuna.export` package diagram

multiple subscribers. 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. The callbacks also use cached weights – if present – in order to publish updates to the weights. Furthermore, it replaces a few of the model's methods with closure wrappers 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[3])
- increase its own step counter automatically when a new batch is seen
- add a parameter to the model's `forward()` 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 (i.e. feed data through it), but do not want to generate new messages for this occasion.
- intercept labels passed to the loss function during training and publish them as messages so the user need not concern himself with this task
- intercept the final output of the network. This could be realised alternatively by identifying the last module in the network.

The `Exporter` publishes the following information at each training step

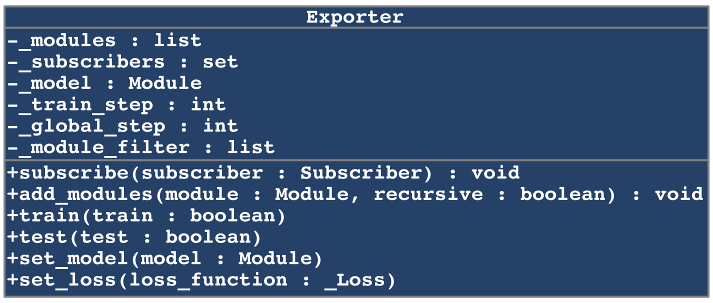
- gradients for each module
- activations for each module
- weights and biases for each module that has these properties (e.g. convolutional or fully-connected layers)
- updates to the weights and biases from the last step to the current one, provided the module has these properties

2.4.2 *The* `models` *subpackage*

This package shown in fig. 2.6 contains model definitions for demonstration purposes and for experimentation. Three 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 the PyTorch implementation.

3 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. 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.



```
Exporter
-_modules : list
-_subscribers : set
-_model : Module
-_train_step : int
-_global_step : int
-_module_filter : list
+subscribe(subscriber : Subscriber) : void
+add_modules(module : Module, recursive : boolean) : void
+train(train : boolean)
+test(test : boolean)
+set_model(model : Module)
+set_loss(loss_function : _Loss)
```

Figure 2.5: `ikkuna.export.Exporter` class diagram

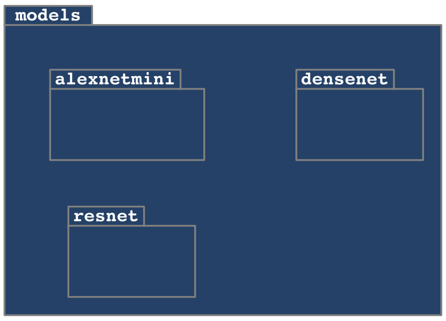


Figure 2.6: `ikkuna.models` package diagram

2. DenseNet (Huang et al., 2017). The implementation is basically the one from (Pleiss et al., 2017)[4] with minor modifications
3. ResNet (He et al., 2016). This implementation comes from GitHub user liukang[5] 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.

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

2.4.3 *The `utils` subpackage*

As shown in fig. 2.7, this package defines classes for traversing a model into a hierarchical tree of layers (called *modules* in PyTorch lingo), structures for adding information to PyTorch's `Module` class, and a set of miscellaneous functions for

1. Seeding random number generators to make experiments reproducible (see section 2.4.7)
2. Creating instances of weight optimizers by named
3. Initialize the weights of any model

4 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.

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

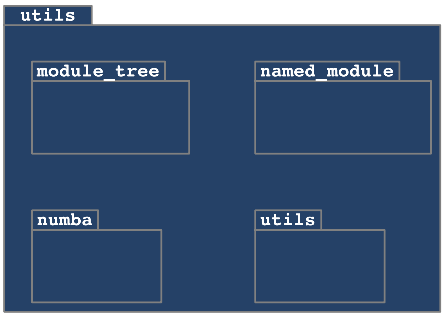


Figure 2.7: `ikkuna.utils` package diagram

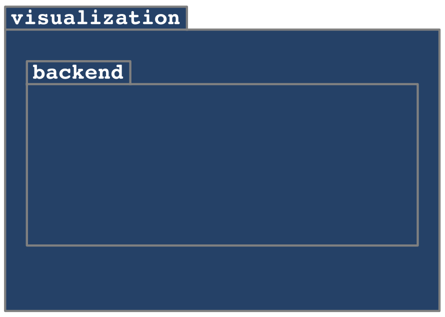


Figure 2.8: `ikkuna.visualization` package diagram

4. Loading datasets

Additionally, it contains the `numba` module which is inteded to allow interoperability with the Numba library[6]. 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.

2.4.4 *The `visualization` subpackage*

This package contains only a single module: `backend`. It defines the classes shown in fig. 2.9. The module serves as an abstraction over plotting libraries so that metrics need not concern themselves with how to actually show the data.

A given metric will compute its value and dispatch it to its visualization backend, which can currently accept scalar and histogram data. The metric class itself need not care about how it is going to be displayed.

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 interactivity. An example is shown in fig. 2.10 To remedy this issue, a plotting backend on TensorBoard is also provided (section 1.4). The plotting data is generated and processed on the remote system, but served over the web so it can be

6 `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.

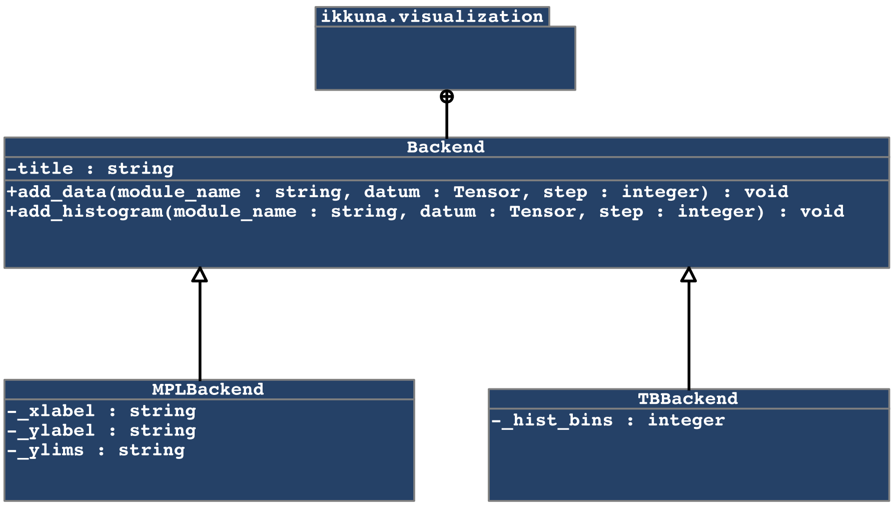

ikkuna.visualization
Backend
-title : string
+add_data(module_name : string, datum : Tensor, step : integer) : void
+add_histogram(module_name : string, datum : Tensor, step : integer) : void
MPLBackend
-_xlabel : string
-_ylabel : string
-_ylims : string
TBBackend
-_hist_bins : integer

Figure 2.9: Class diagram for classes in `ikkuna.visualization`

viewed and interacted with locally (provided the network is configured so that the server responds to HTTP requests). An example is shown in fig. 2.11

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.

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*

2.4.7 *Reproducibility*

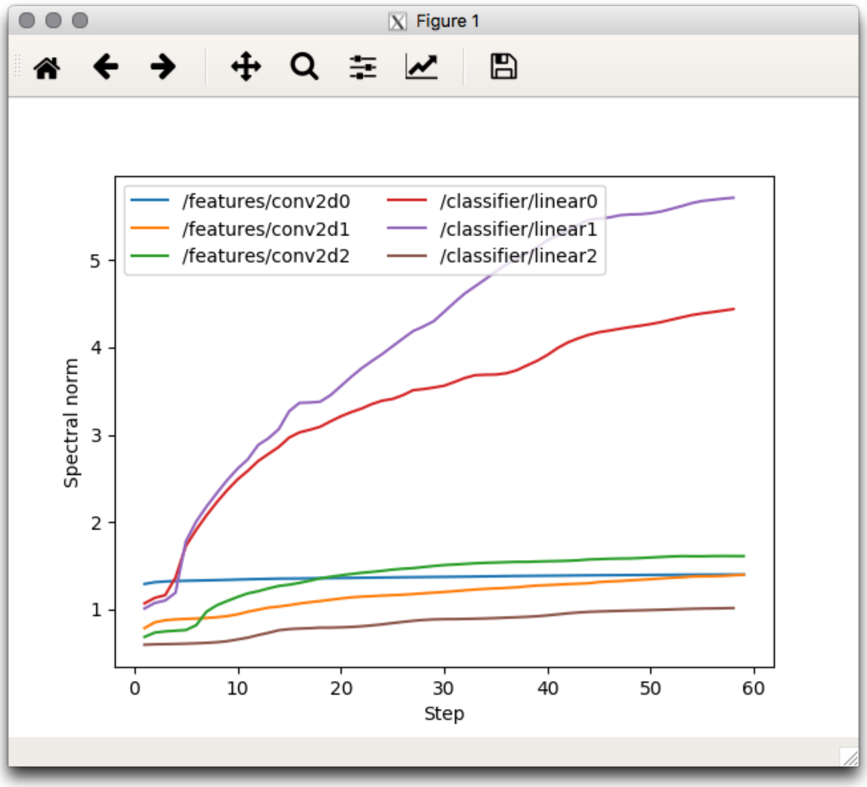
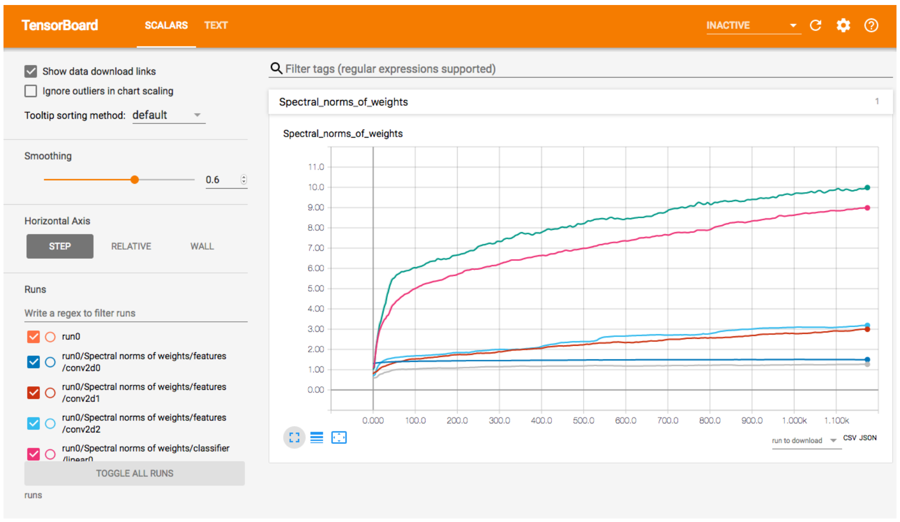


Figure 2.10: Exemplary view of a matplotlib figure forwarded over SSH



[✓] Show data download links
[ ] Ignore outliers in chart scaling
Tooltip sorting method: default

Smoothing 0.6

Horizontal Axis [✓ STEP] RELATIVE WALL

Runs
Write a regex to filter runs
[✓] run0
[✓] run0/Spectral norms of weights/features /conv2d0
[✓] run0/Spectral norms of weights/features /conv2d1
[✓] run0/Spectral norms of weights/features /conv2d2
[✓] run0/Spectral norms of weights/classifier /linear0

runs

Figure 2.11: Exemplary view of a TensorBoard session

Table 2.2: 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`

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.

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

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.

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.

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.

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.

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.

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.

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

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 while traveling with me and hosting me for my visits to Stockholm.

I am also grateful to Anders Arpteg PhD who has been a constant source of supportsupport 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.

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, October 24, 2018