

Image Classification, Deep Learning and Convolutional Neural Networks

A Comparative Study of Machine Learning Frameworks

Rasmus Airola
Kristoffer Hager

Faculty of Health, Science and Technology

Computer Science

C-level thesis 15 hp

Supervisor: Kerstin Andersson

Examiner: Stefan Alfredsson

Opposition date: 170605

**Image Classification, Deep Learning and
Convolutional Neural Networks**
A Comparative Study of Machine Learning Frameworks

Rasmus Airola and Kristoffer Hager

Abstract

The use of machine learning and specifically neural networks is a growing trend in software development, and has grown immensely in the last couple of years in the light of an increasing need to handle big data and large information flows. Machine learning has a broad area of application, such as human-computer interaction, predicting stock prices, real-time translation, and self driving vehicles. Large companies such as Microsoft and Google have already implemented machine learning in some of their commercial products such as their search engines, and their intelligent personal assistants Cortana and Google Assistant.

The main goal of this project was to evaluate the two deep learning frameworks Google TensorFlow and Microsoft CNTK, primarily based on their performance in the training time of neural networks. We chose to use the third-party API Keras instead of TensorFlow's own API when working with TensorFlow. CNTK was found to perform better in regards of training time compared to TensorFlow with Keras as frontend. Even though CNTK performed better on the benchmarking tests, we found Keras with TensorFlow as backend to be much easier and more intuitive to work with. In addition, CNTKs underlying implementation of the machine learning algorithms and functions differ from that of the literature and of other frameworks. Therefore, if we had to choose a framework to continue working in, we would choose Keras with TensorFlow as backend, even though the performance is less compared to CNTK.

We would like to thank our supervisor from Karlstad University, Kerstin Andersson, for her valuable input and guidance throughout the project. We also would like to thank our supervisors at ÅF Karlstad, Mikael Arvidsson and Daniel Andersson, as well as ÅF Karlstad for giving us this opportunity to learn more about an exciting field of study.

Contents

1	Introduction	1
1.1	Project Results	1
1.2	Disposition	3
2	Background	4
2.1	Rationale and Scope	4
2.2	Machine Learning	5
2.2.1	Deep Learning and Neural Networks	9
2.2.2	Convolutional Neural Networks	15
2.2.3	Neural Network Designs and Architectures	19
2.3	Deep Learning Frameworks	22
2.3.1	The Frameworks Used in This Study	23
2.4	Summary	24
3	Project Design	26
3.1	Installation and System Requirements	26
3.2	Features, Functionalities and Documentation	26
3.3	Benchmarking Tests	27
3.4	Implementing an Image Classifier	29
3.5	Summary	30
4	Project Implementation	32
4.1	Installation and System Requirements	32
4.2	Features, Functionalities and Documentation	33
4.3	Benchmarking Tests	35
4.4	Implementing an Image Classifier	36
4.5	Summary	39
5	Results and Evaluation	40

5.1	Installation and System Requirements	40
5.2	Features, Functionalities and Documentation	42
5.3	Benchmarking Tests	45
5.4	Implementing an Image Classifier	48
5.5	Summary	53
6	Conclusion	55
6.1	Future Work	55
6.2	Concluding Remarks	57
	References	58
A	MNIST Keras/TensorFlow source code	62
B	MNIST CNTK source code	65
C	CIFAR-10 Keras/TensorFlow source code	69
D	CIFAR-10 CNTK source code	72
E	CIFAR-100 Keras/TensorFlow source code	76

List of Figures

2.1	Overfitting during training.	8
2.2	An artificial neuron.	10
2.3	A (simple) artificial neural network.	11
2.4	Plot of the sigmoid function.	12
2.5	Plot of the rectified linear unit (ReLU) function.	13
2.6	A hidden layer of neurons with locally receptive fields.	17
2.7	A simple convolutional neural network.	19
2.8	An Inception V1 module.	21
2.9	A residual block with an ordinary skip connection.	22
5.1	Plot of the accuracy for the first run.	48
5.2	Plot of the loss for the first run.	49
5.3	Plot of the accuracy for the second run.	50
5.4	Plot of the loss for the second run.	51
5.5	Plot of the accuracy for the third run.	52
5.6	Plot of the loss for the third run.	53

List of Tables

5.1	Results CNTK MNIST	47
5.2	Results Keras/TensorFlow MNIST	47
5.3	Results CNTK CIFAR-10	48
5.4	Results Keras/TensorFlow CIFAR-10	48

1 Introduction

This project consisted of roughly 20 weeks of part-time work (three days a week) for the company that gave us the assignment, ÅF Karlstad, more precisely the section for industrial IT. ÅF as a whole is an engineering and consulting company with over 9000 employees which works primarily within the energy, industrial and infrastructure sectors of the economy. The section for industrial IT in Karlstad works primarily with projects concerning production systems in a number of different factories and paper mills.

The primary goal of the project was to evaluate two frameworks for developing and implementing machine learning models using deep learning and neural networks, Google TensorFlow and Microsoft Cognitive Toolkit (CNTK). A secondary goal that followed from the first was to explore and learn about deep learning, both the theory and the practice, through the implementation of an image classifier. The evaluation was supposed to consist of as many relevant factors as possible (for more detail see chapters 3 and 4), with the greatest emphasis on the performance of the frameworks in terms of speed and hardware usage. The company's motivation behind the project was to explore the possibility of using machine learning in the work that is being done today regarding production systems and industrial automation, and that led into a need to look into possible tools for development and implementation.

The project thus became a comparative study of the frameworks, a study broken down into parts where different parts were evaluated according to a set of criteria, combined with an exploratory part consisting of learning about deep learning and neural networks through the implementation of an image classifier.

1.1 Project Results

Below a summary of the project results is provided.

At the time of this writing, the frameworks are equal in terms of ease of installation. Regarding setting up the computations on GPU (Graphics Processing Unit), we think the frameworks

are equal in this regard as well, seeing as they required the same amount of steps and dependencies to be installed. When it comes to system requirements and support, the decision which framework is better in this regard largely comes down to which operating system and programming language one is going to use; seeing as CNTK and TensorFlow both support languages the other does not, and that TensorFlow supports Mac OS X and CNTK does not.

We found that the frameworks provide an equivalent set of features and functionalities, and the frameworks are more than capable of constructing neural networks. The frameworks' documentation were both found to be lacking in both quality and quantity, however Keras has the slight advantage of having its documentation gathered in one place, whereas CNTK has its documentation distributed on different sites. Keras was found to be more beginner friendly and easier to work with, as compared to CNTK. It was found that CNTK does not conform to the literature, having instead its own implementation, which in turn requires relearning if one has studied the literature, it also requires recalculating the parameters in order to function according to the literature; Keras on the other hand, requires no such readjusting, which is a big plus in our view.

CNTK was found to give a shorter training time of the networks compared to Keras with TensorFlow as backend, see tables 5.1-5.4, given the uncertainties listed in chapter 5.3 and the experimental setup presented in chapter 4.3. GPU was found to be so superior over CPU (Central Processing Unit) in terms of training speed, that if one has a GPU available one should use it instead of the CPU. GPU and VRAM (Video RAM) usage were both similar in both CNTK and Keras with TensorFlow as backend.

Regarding the implementation of an image classifier; the model was found to quite rapidly overfit on the training set, a learning rate schedule was found to be beneficial, and the number of epochs seems to have been detrimental to the performance of the model. A number of possible improvements were found and discussed as follows: more regularization, a learning rate schedule with more steps, early stopping, and higher spatial resolution in the latter parts

of the network.

1.2 Disposition

Chapter 2 will introduce the rationale behind this study and its scope, as well as providing the necessary theoretical background for someone completely new to machine learning to understand the project's implementation. The deep learning frameworks used in this study will be presented and introduced as well.

Chapter 3 will present an overview of the different parts of the project and the evaluation method used for the different parts. Chapter 3.1 covers the installation and the system requirements of the two frameworks and their dependencies; as well as the frameworks' support of programming languages. Chapter 3.2 covers the frameworks' features, functionalities and documentation; as well as their support for third-party application programming interfaces, or APIs. Chapter 3.3 covers the frameworks' performance using two widely used data sets for benchmarking machine learning models; Mixed National Institute of Standards and Technology [1] dataset of handwritten numbers (MNIST) and Canadian Institute for Advanced Research's dataset of tiny images in color (CIFAR-10) [2]. Chapter 3.4 presents the design of a custom built image classifier.

Chapter 4 will describe the actual implementation of the project's parts described in chapter 3.

Chapter 5 will present and discuss the results from each part of the evaluation as described in chapter 3 and 4.

Chapter 6 will provide and motivate the conclusion of the evaluation of the frameworks, as well as suggestions for future work, and finally some concluding remarks by the authors.

2 Background

This chapter will introduce the rationale behind this study and its scope, as well as providing the necessary theoretical background for someone completely new to machine learning to understand the project's implementation. The deep learning frameworks used in this study will be presented and introduced as well.

Chapter 2.1 will introduce the rationale behind this study and its scope. Chapter 2.2 will provide the necessary theoretical background, where chapter 2.2.1 will focus on deep learning and neural networks, chapter 2.2.2 will focus on convolutional networks, and lastly chapter 2.2.3 will present and discuss different neural network designs and architectures. Chapter 2.3 will present and introduce the deep learning frameworks used in this study. At the end of the chapter a summary is provided.

2.1 Rationale and Scope

In this study the two deep learning frameworks Google TensorFlow and Microsoft Cognitive Toolkit (CNTK) will be evaluated based on their performance and user friendliness. The rationale behind this study is that the company that gave us the assignment wants to integrate machine learning in their work. The company therefore wants to explore the potential of machine learning and to find the best framework to work with (more on such frameworks in chapter 2.3). The motivations for the authors are that machine learning is a very interesting field, in which a lot of possible future breakthroughs in IT can be made, and this study provides a great opportunity to learn more about this field.

Machine learning is a vast field and in order for this study to be completed within the given time span, the scope of the study needs to be limited to some subset of machine learning. The theoretical scope of the study is therefore limited to neural networks, with a focus on convolutional neural networks. The practical scope will be limited to creating and training three equivalent neural networks in both TensorFlow and CNTK for three already prepared and well known datasets in order to evaluate the frameworks' performance on the same neural

network. Gathering and preparing training data is therefore beyond the scope of this study. See chapter 3 for further details on the study.

2.2 Machine Learning

Machine learning as a field of research is somewhat hard to describe succinctly, due to the breadth of the field, the amount of ongoing research, the interdisciplinary aspects of the field and a multitude of other factors. For the purposes of this report the following definition will be used as a starting point, as it is sufficiently precise:

"A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P , if its performance at tasks in T , as measured by P , improves with experience E ." [3]

From a mathematical standpoint the problem can be formulated as applying a machine learning algorithm to find an unknown mathematical function with a known domain, that of the input, and a known co-domain, that of the output. In more informal terms the program is supposed to find a mathematical connection between the data and the target input. The connection depends on the task at hand. [4] A relevant example of the aforementioned classes of tasks is to find a connection between a vectorized representation of the input to an output consisting of a set of discrete categories or a probability distribution. Concrete examples of such a task can consist of developing a program for image classification and object detection, e.g. recognizing faces in pictures and finding bounding boxes for them. Other examples of types of tasks are anomaly detection, e.g. spotting credit fraud and machine translation, e.g. translation software. [4]

The performance of an implementation of a machine learning algorithm is usually either expressed in the accuracy of the implementation, e.g. what percentage of the examples were classified correctly, or in the error rate, e.g. what the ratio of incorrectly to correctly classified

examples is. The difficult part here is not so much the choice of measure rather than which aspect(s) of the output that should be measured in the first place. Should one, for example, choose the percentage of correctly transcribed whole sequences as the only measure or take the correctly transcribed parts of sequences into account when measuring the performance? [4] Another complicating factor is that it can be hard to measure the chosen quantity due to practical difficulties, and in those cases another measure must be chosen.

There are basically two kinds of learning, or experiences, a machine learning algorithm can have during a learning or training process; unsupervised and supervised learning. [4] A training process is based upon a dataset consisting of a given amount of examples or data points. Unsupervised learning is based upon datasets containing features where the goal is to learn useful properties about of the structure of the dataset or, more precisely, to find the underlying probability distribution of the dataset. Supervised learning, in contrast, is based upon datasets consisting of many features but with labels attached to each example. [4] The goal in supervised learning is to learn to predict the label, the associated value, from an arbitrary example in the dataset after the training session is over. The terms supervised and unsupervised comes from the fact that the learning algorithm has a teacher that shows the algorithm what to do in the former case, but is supposed to learn from the data without a guide in the latter case.

The most challenging part of machine learning, regardless of the algorithm(s) applied in the learning process, is to make programs that generalize well; that is, programs that perform as well or almost as well on unobserved inputs as those observed during training. The main thing that separates machine learning from optimization is that the goal consists of minimizing both the training error and the test error. [4] The aforementioned datasets are split in two predetermined sets to be used in the training process, one for training and one for testing. The goal of machine learning can now be defined more precisely; to minimize the training error and to minimize the gap between the training error and the test error. [4] In attempting to achieve the aforementioned goal two more challenges appear; underfitting and overfitting on the training set.

Underfitting occurs when the model cannot reach a sufficiently low training error and overfitting (see figure 2.1) occurs when the gap between the training error and the test error is too large. [4] The former problem is caused by a model with low representational capacity that cannot fit the training set properly, the latter can be caused by a model with high capacity that memorizes properties from the training set too well to perform sufficiently on the test set. [4] Overfitting can also be caused by a training set that is too small to generalize properly from. Underfitting can be fixed rather easily with a model with sufficient capacity, but overfitting can be harder to fix due to the fact that acquiring greater amounts of data is not always feasible. A prominent example of a technique that can be introduced during the training is regularization, which is used to limit the space of potential functions (since the goal is to find the best one). [4] There is not much more space to go into detail here, and the problems that can occur and their corresponding solutions will be discussed in later sections.

The parameters of the machine learning algorithm that are not adapted or changed by the algorithm itself during training are called hyperparameters, parameters that can be tuned to change the behavior of the learning algorithm. [4] It is fitting to introduce a further partition of the dataset here, that of the original training data into a training set and a validation set. The training set is used purely for adjusting the internal parameters, e.g weights and biases (see chapter 2.2.1), during the training and the validation set is used to measure the current generalization error and to adjust the hyperparameters accordingly. The validation set is not used to tune the internal parameters. [4] The important distinction between the validation set and the test set is that the latter is not used during the training process at all and does not give any input to the training process - it is simply used to measure performance.

As a final observation in this chapter; there are a lot of machine learning algorithms to choose from, far to many to describe in detail here or even to give a cursory oversight. Since the frameworks in the study are implementations of the components, the functionality and the algorithms necessary to train neural networks and implement deep learning there is no need

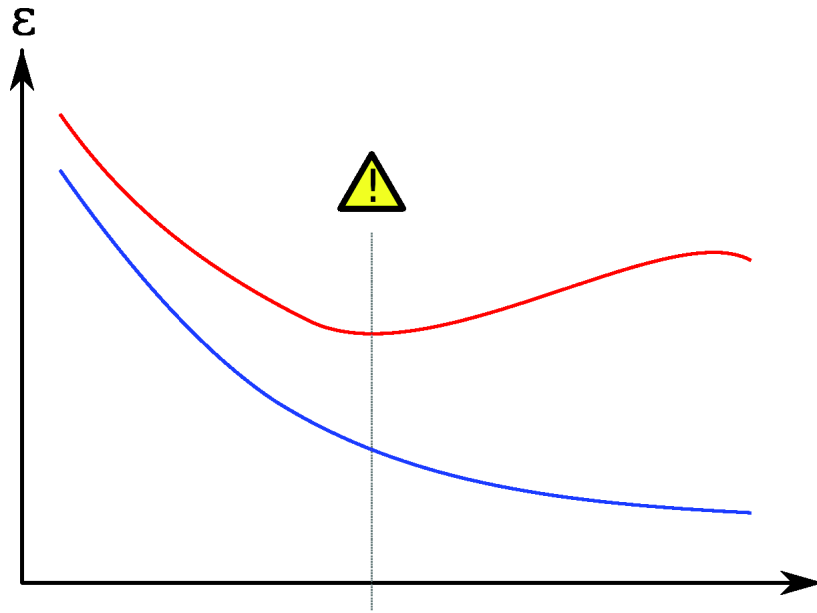


Figure 2.1: Overfitting during training.

As seen in the image the training loss (blue) steadily decreases while the test loss (red) starts to increase after a while.

Author: Gringer

Source: https://commons.wikimedia.org/wiki/File:Overfitting_svg.svg

License: 

to introduce nor discuss other algorithms in the theoretical background.

2.2.1 Deep Learning and Neural Networks

Although the very first step towards neural networks was taken in 1943 with the work of Warren McCulloch and Walter Pitts, the first practical application using artificial neurons came with Frank Rosenblatt's invention, the perceptron. A perceptron is the simplest possible version of an artificial neuron (see figure 2.2), and it has the basic essential attributes as follows [5]:

- One or more numeric inputs with corresponding weights, positive or negative, for each input.
- A bias that can be either positive or negative. Can informally be described as the neurons resistance to "firing off".
- An activation function (in the case of the perceptron, the unit step function).
- A single output value, the activation function applied to the sum of the weighted inputs and the bias.

More informally stated the perceptron outputs 1 if the sum of the weighted inputs and the bias is bigger than 0, and 0 if not. Even though perceptrons are not used in practice they led to the next logical step, the multilayer perceptron (MLP) or the feedforward neural network (see figure 2.3). A feedforward neural network is simply artificial neurons in layers, with all the outputs from each neuron in the preceding layer fed forward, not backwards, into each neuron in the following layer, the exceptions being the input layer (consisting of passive neurons that do not transform the input) and the output layer. [5] The layers between the first and last are called the hidden layers, which gives the network depth and therefore leads to the first part of the name of this chapter, deep learning, which is also a common name for the use of deep neural networks as a whole and associated techniques. Since each hidden layer, and the output layer, consist of neurons who individually are connected with the output from each neuron in the previous layer, those layers in a feedforward network are called fully connected

layers. [5] All neurons in the network have a unique set of weights and the activation functions are non-linear functions. That latter part will be expounded upon later in the chapter.

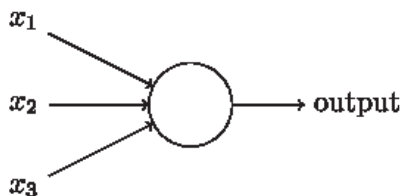



Figure 2.2: An artificial neuron.

Author: Michael Nielsen

Source: <http://neuralnetworksanddeeplearning.com/chap1.html#perceptrons>

License: 

Before the more technical details of feedforward neural networks are dealt with, a more fundamental property of feedforward neural networks needs to be introduced first; that feedforward neural networks work as universal function approximators. [4] [5] An single perceptron or any other artificial neuron is not of much use, but a network with at least one hidden layer can approximate any continuous function, which in practice means that any discontinuous function can be approximated as well. [5] A description of the formal proof is outside of the scope of this report, but conceptually an artificial neuron can be compared to a NAND or NOR logic gate in that it works as an universal building block. [5] The main difference is that an artificial neuron has parameters that can be tuned and can therefore be trained.

While speaking of parameters it is fitting to introduce the activation functions that are being used in practice. The unit step function mentioned above is not used in practice due to the fact that a small change in input can lead to a big change in output (0 to 1), an unwanted property since a continuous change in the output is preferable. [5] The sigmoid function (see figure 2.4) and the hyperbolic tangent function, smoother versions of the unit step function, have been used in practice but the activation function of choice today is the rectified linear unit function (ReLU), which is defined by returning only positive values, and variants of that function (see figure 2.5). [4] The output layer, or the classification layer, uses the softmax

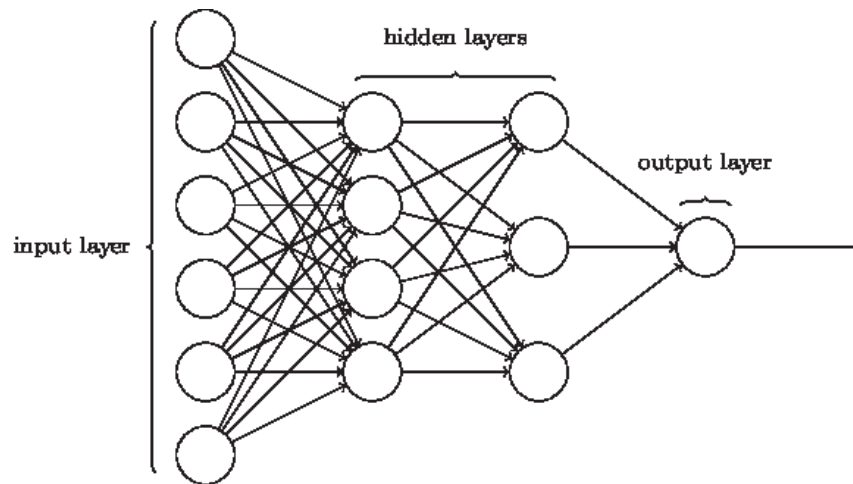



Figure 2.3: A (simple) artificial neural network.

Author: Michael Nielsen

Source: http://neuralnetworksanddeeplearning.com/chap1.html#the_architecture_of_neural_networks

License: 

function, which outputs a probability distribution over all the categories. More informally the softmax function outputs the most likely category, the classification that the network found most probable.

To train a neural network we need some other kind of measure of how big the current error is outside of the training error, some kind of measure of how well off the weights and biases in the network are as whole. To solve this a cost or objective function is introduced, a function that measures the total current error. The two important properties such an objective function must have is that it is non-negative for all inputs, and that it is zero or close to zero if the error is small. [5] The direct goal of the training is thus to minimize the objective function. A simple approach here would be to choose the mean squared error as the cost to minimize, but in practice the cross entropy function is used instead due to a intrinsically better performance. [5]

Neural networks can contain millions, tens of millions and even billions of parameters and there is no feasible way to find the minimum with methods from ordinary calculus. Instead

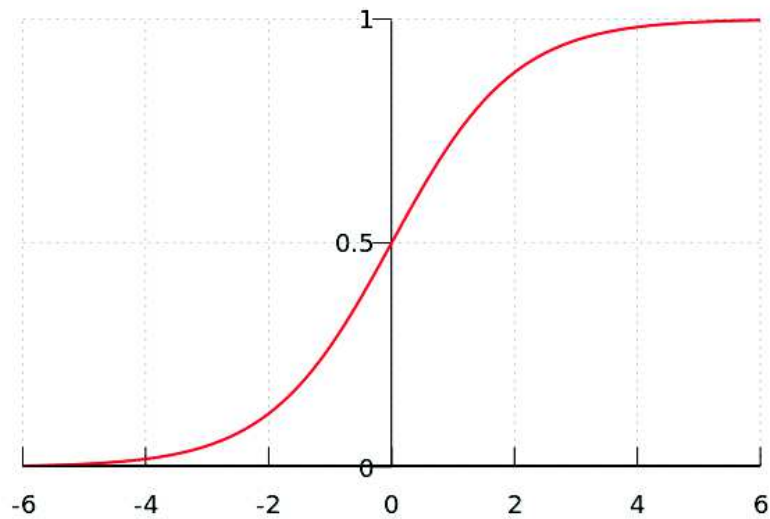


Figure 2.4: Plot of the sigmoid function.

a algorithm called gradient descent, more precisely stochastic gradient descent, is used to minimize the objective function. [5] The method uses the gradient or derivative in a random starting point to find the current "slope" of the objective function, and decreases the value of the objective function with a fixed multiple of the absolute value of the gradient, which in more informal terms means that the algorithm moved down the "slope" a fixed amount towards the minimum. Applying gradient descent to a neural network means that each weight and each bias in the network is adjusted with a fixed multiple of the partial derivative of the objective function with regards to that specific weight or bias during each pass of the algorithm. The fixed multiples mentioned so far are multiplied with the learning rate, a hyperparameter of the network that can be tuned to boost performance. [5] Calculating the gradient over the whole training set would take too much time and isn't used in practice. Instead the gradient is calculated for a randomly chosen subset of the training set, a so called minibatch, and used to update the network. This process is repeated for each minibatch in the training set until all minibatches have been processed. The time taken to process the entire training set is called an epoch, and training sessions are usually defined by the number of epochs. The fact that the training set is shuffled also explains the full name of the algorithm, stochastic gradient descent. [5]

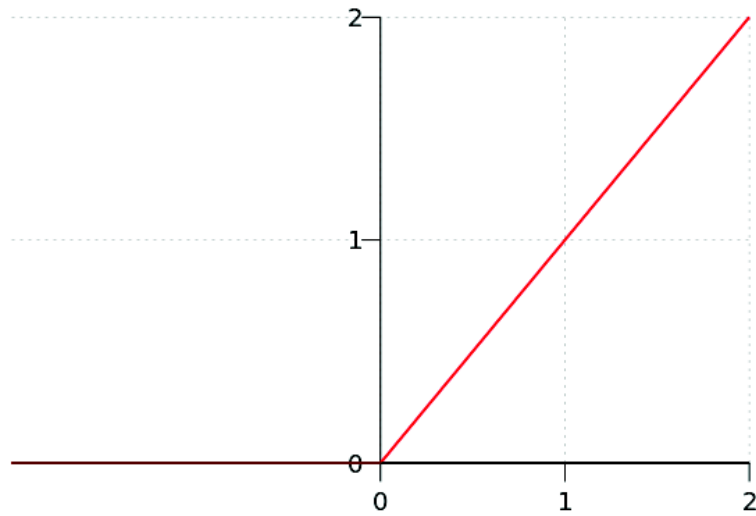


Figure 2.5: Plot of the rectified linear unit (ReLU) function.

Even though stochastic gradient descent, as described above, is an algorithmic solution to a mathematically unsolvable problem it still needs further work to be useful in practice. Once again, neural networks have a huge amount of parameters and computing each partial derivative of the objective function as in the naive implementation above is numerically unfeasible. The solution to the problem that makes stochastic gradient descent useful in practice is backpropagation, an algorithm that, simply put, calculates the error of one layer based upon the error of the preceding layer and updates its parameters accordingly. [5] The name of the algorithm comes from that property, that the error, and the correction of the error, propagates backwards through the entire network from the output layer and backwards. The elegance of the algorithm is that it mirrors the path the activations took forward in the network and carries roughly the same computational cost. As a final note on gradient descent there are more advanced variants in use today that builds upon the standard version with backpropagation, variants that adds additional elements such as dynamic learning rates and other tweaks. [4] Explaining those algorithms in detail are outside of the scope of this report, but it is worth mentioning that they exist.

As mentioned in the previous section there are inbuilt challenges connected to the machine learning process, and neural networks are no exception. The two problems that will be explored here are overfitting in the case of neural networks and the unstable gradient problem, a problem specific to the stochastic gradient descent algorithm applied to the training of neural networks. [5] Overfitting in this context is remedied by methods such as gathering more and better data, something that has been mentioned earlier in this chapter, and regularization. It has been mentioned in the former section, but it is worth repeating that regularization can be described as limiting the amount of functions the machine learning algorithm can generate. In the case of neural networks there are many kinds of regularization but the discussion here will restrict itself to these following methods: L1 regularization, L2 regularization, dropout and data augmentation. [5]

L1 and L2 regularization are two variants of the same theme and are built upon adding an extra term to the cost function, a term consisting of an weighted average of the sum of all the weights in the network. [5] The difference between the two lies in that the sum is of the absolute values of the weights in the former case and of the squares of the weights in the latter case. The reason why this term is added is to penalize networks with too large weights, a penalty whose effect, informally stated, is to make the network generalize better by forcing it to choose less complex functions linking input to output. [5] Dropout is a technique that is based upon dropping a random number of neurons in the hidden layers during each round of minibatches with a final adjustment of the weights in the end of a training run. The intended effect is, as with L1 and L2, to enforce better generalization. The last method, data augmentation, is built upon artificially expanding the available data by introducing random changes in the examples, e.g flipping an image horizontally, shifting it slightly in a direction or rotating it slightly. [5] Since the best cure for overfitting is more data this technique works in that direction to make the network generalize better.

The unstable gradient problem occurs due to the fact that the gradient, or the parameters' rate of change, in a layer is a product of the rate change of all the layers before it. [5] This

can cause the rate of change to vanish entirely, the vanishing gradient problem, or go up drastically, the exploding gradient problem. The unwanted effects accumulate more rapidly the earlier in the network the layer in question is. This problem was, historically at least, a major hindrance to training networks beyond a certain depth. Luckily the problem seems to have been partially solved, partly due to the fact that the aforementioned rectified linear function and variants thereof have become the standard activation functions. A major cause of the problem was the use of saturating activation functions like the sigmoid and hyperbolic tangent functions, saturating in this context meaning the rate of change or derivative of the function goes to zero as the input becomes too large of a positive or negative number. [5] In contrast the ReLU function has a derivative that is either 0 or 1, which means that it does not saturate in its positive region and always propagate the gradient backwards. There have been other breakthroughs in this area, breakthroughs that will be discussed later.

During this exposition the terms neural networks and feedforward neural networks have been used synonymously, which is not surprising since it is the latter that has been discussed and explained during the majority of this chapter. The terms are not entirely synonymous though, since there are neural networks with loops and feedback connections. That subclass of neural networks is called recurrent neural networks, and makes use of more complex layers than the ones mentioned so far, layers such as long short-term memory units (LSTMs). [4] How recurrent neural networks work in detail is outside the scope of this report, but they are important and worth mentioning due to the fact that they are behind some of the latest breakthroughs in text and speech processing. There are other kinds of neural networks, but the focus of this work lies upon feedforward neural networks and their derivatives.

2.2.2 Convolutional Neural Networks

One of the weaknesses of an ordinary feedforward neural network with fully connected layers is that it has no prior inbuilt assumption about the data it is supposed to learn from. It is agnostic about the structure of the data and treats all data the same. [5] This can easily

become a problem partly due to the resulting redundant parameters, partly due to the size of the resulting redundancy since a neural network can, as mentioned several times before, grow very large. It is also counterintuitive to use this approach in the case of, for example, data consisting of images since it would mean that the spatial structure of the image would be ignored and the training would begin with the assumption that all pixels are equally related. [5] Another approach is clearly needed, which is why this sub-chapter deals with a special kind of feedforward neural networks, convolutional neural networks. Convolutional neural networks are designed with certain assumptions about the data, assumptions that fit image data but also other data with a similar internal structure.

The most fundamental operations of convolutional neural networks is, perhaps not surprisingly, convolutions. The concept of convolutions in the context of neural networks begins with the idea of layers consisting of neurons with a local receptive field, i.e. neurons which are connected to a limited region of the input data and not the whole. [5] In the case of image data that means that each neuron is connected to only a limited region of pixels, e.g. a square of 3 x 3 pixels in the upper left corner of an image. The receptive fields of the neurons are also overlapping to a certain degree, in that (continuing the example from the sentence before) two adjacent neurons have receptive fields that are shifted by one or more pixels horizontally or vertically in relation to each other. One can visualize a sliding window of a given size, sliding from left to right and top to bottom (the data can be assumed to have a two-dimensional structure from now on), connecting the output of the window in each pass to a neuron. The result will be a two-dimensional structure of its own, a map or image of the activations from each neuron. The "sliding window" in this context is more properly called a kernel, a set of weights and a bias. That means that each neuron in such a layer or map shares the same parameters and thus could be said to learn the same feature, which leads to the proper name for such structures, feature maps. The operation described above, applying a kernel to an input map and generating a feature map, is a convolution. [5] The usage of convolutions is, intuitively, highly fitting for highly spatially correlated data since one can argue that, for example, two pixels in and around the eye in picture of a face has a more meaningful relationship than two

pixels chosen at random from the picture.

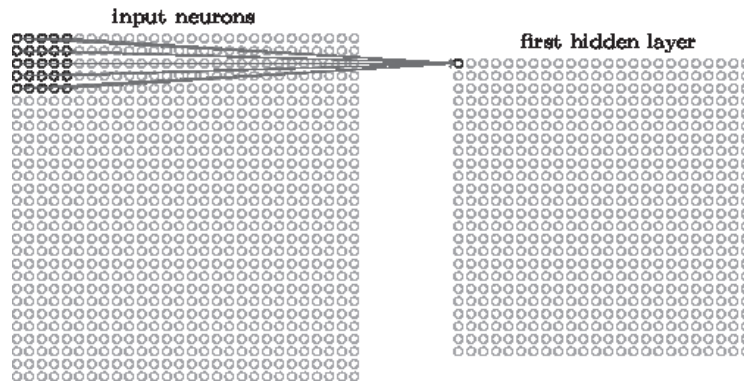


Figure 2.6: A hidden layer of neurons with locally receptive fields.

Author: Michael Nielsen

Source: http://neuralnetworksanddeeplearning.com/chap6.html#introducing_convolutional_networks

License: 

The convolutional layers in convolutional neural networks consists of multiple, stacked feature maps with associated neurons. This is also why the kernels associated with the feature maps are also called filters or channels. [6] Convolutional layers can also be stacked upon other convolutional layers, which in more informal terms can be described as building feature maps upon feature maps or learning higher-level features. One other important aspect of convolutional neural networks is that the parameter sharing involved decreases the absolute number of parameters needed, in addition to a major relative decrease of parameters in comparison to a network using only fully connected layers to do the same task. In general, the formula for the number of parameters in a convolutional layer performing two dimensional convolutions is $(M * X * Y + 1) * N$, where M is the number of feature maps in the input (counting the red-green-blue channels in colored images), X and Y the height and the width of the convolution (3 x 3, 5 x 5, etc.) and N the number of feature maps in the output, the one added to account for the bias. [7] The two final aspects of convolutions to be discussed in this section are stride and padding. In the previous paragraph it was implicit that the size of the step, or stride, during the convolution was one, but higher strides can be used as well. Padding of the input

is used to preserve the spatial resolution during the convolution, since it shrinks without it by a constant factor in both dimensions depending on the size of the convolution. [5]

Another kind of important layer in convolutional neural networks are pooling layers which apply the pooling operation. Pooling in general consists of transforming feature maps to smaller, more aggregated feature maps. [5] The most common kind of pooling is max-pooling, which takes non-overlapping regions of a given size and outputs the largest activation in that region, e.g. splitting the input map into sections of size 2×2 , choosing the maximum value and creating an output map consisting of those values, but only a fourth of the original size. A pooling layer thus pools the input from a preceding convolutional layer, preserving the number of feature maps. The rationale behind pooling is to shrink the spatial dimensionality in the network while preserving spatial information, using the strong correlation between data points that are close to each other. [4] As with convolutions, pooling layers can use stride to achieve even greater aggregation.

With pooling layers introduced, the structure of a convolutional neural network can be made clearer. The overarching goal of a convolutional neural network is to transform the spatial representation of the input to a representation rich in features, a large feature space from which to classify. The basic structure of a convolutional neural network is thus convolutional layers alternated as fitting with pooling layers, combined with using the techniques discussed in the previous section, and usually a couple of fully connected layers with a classifier at the end. [5] This structure has been around for years, at least since Yann LeCuns LeNet5 architecture in 1998 [8], but the same ideas are still useful today. It merits to mention that the basic approach still works, especially since the research around convolutional neural networks is intense and that many major breakthroughs have been made and are being made, especially in the areas of image classification and object detection.

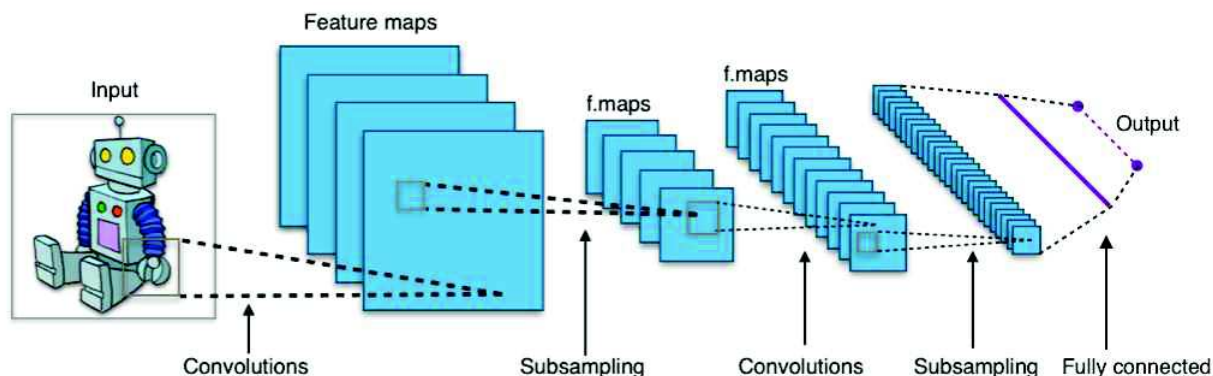


Figure 2.7: A simple convolutional neural network.

Author: Aphex34

Source: https://commons.wikimedia.org/wiki/File:Typical_cnn.png

License: CC BY-NC-SA

2.2.3 Neural Network Designs and Architectures

In this section some of the more significant neural network designs and architectures, more particularly those of convolutional neural networks, will be introduced in appropriate detail. The common task all of the following architectures are geared towards is image classification, to be able to classify images as correctly as possible into a number of predetermined categories.

The first major breakthrough after the LeNet5 model mentioned in the previous section was Alex Krizhevsky's AlexNet in 2012 [9], a contribution to the ImageNet competition. ImageNet is a database consisting of a million images sorted into a thousand categories being used for research into image recognition and object detection. [10] AlexNet built upon the approach of LeNet5, used rectified linear units and ran on two GPU:s. [9] This approach was refined further with the VGG (Visual Geometry Group) networks, networks with a much greater depth and accuracy. [11] The main innovation was the use of smaller convolutions of size 3x3 and 5x5 instead of the much larger convolutions used in AlexNet, and the stacking of layers using these smaller convolutions upon each other. [6] The networks grew quite large, however, and the training had to be done in parts due to the sheer difficulty of it. [6] The run-time performance

of the VGG networks during inference, or classification after training is done, was also quite costly. [12]

The approach mentioned above has, as mentioned, some obvious drawbacks, especially regarding the computational burden of training and serving the model. A research team at Google took another approach with those considerations in mind, an approach that resulted in the 2014 winner of the ImageNet competition, GoogleNet, a network design based upon Inception modules and the first network to use the Inception architecture. [13] The basic idea behind an Inception module is to have layers working on input in parallel, both convolutional layers and pooling layers, layers with different sizes of kernels, and then concatenating the outputs into a single layer. GoogleNet began with an ordinary stack of layers as those in the aforementioned AlexNet and VGG models, a large middle part of Inception modules stacked upon each other and finally a global averaging layer with a softmax classifier, the last part inspired by the Network In Network paper. [14] Another important aspect of the Inception architecture is the use of 1x1 convolutions. 1x1 convolutions serves two useful purposes in convolutional neural networks; they can be used to cheaply, computationally speaking, add nonlinearity via extra activations and to reduce the number of features in the outgoing layer cheaply. [6] These convolutions are used to build so called bottleneck layers, constructs consisting of a layer using 1x1 convolutions to downsample the outgoing features by usually a factor of four, a second layer using a larger kernel on the smaller number of features and a final layer of 1x1 convolutions to upsample the features again. These bottlenecks can reduce the numbers of computations needed by nearly a factor of ten. [6] The combination of bottlenecks and Inception modules has led to the success of the architecture and the following network designs, Inception V2 [15] and V3 [16]. The Inception networks has, in comparison to the VGG networks and the approach behind them, increased the accuracy in the ImageNet Challenge while decreasing the number of parameters and the number of operations significantly. [6] [12]

The final approach to be introduced here is that of the Microsoft team behind ResNet, the winner of the ImageNet competition 2015 and an architectural approach that achieved an

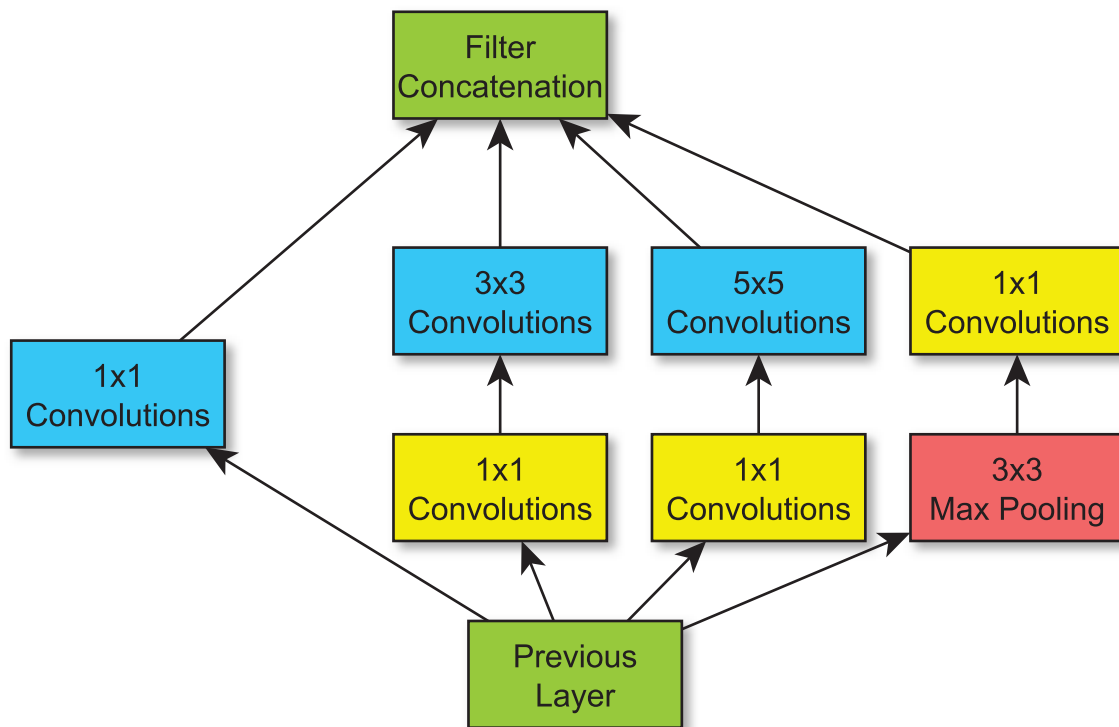


Figure 2.8: An Inception V1 module.
 Recreation of an image from: [6]

unprecedented network depth by using a new technique: residual blocks consisting of several layers, and networks consisting of those blocks, residual neural networks. [17] The basic idea of residual blocks is to run the input through two or more layers and then, crucially, adding the initial input to the output of the second layer, neuron for neuron, and applying the activation function to get the output of the block. The block can be said to consist of an residual connection, the layers the input is run through, and a skip connection, the bypassing of the initial input. [18] The residual connection is usually a bottleneck layer, and the skip connection is usually only the input, but other variants exist, primarily to cut down the spatial dimensionality. [17] The team behind ResNet managed to use the residual blocks to design and train a convolutional neural network with over a thousand layers, a record in deep learning. [6]

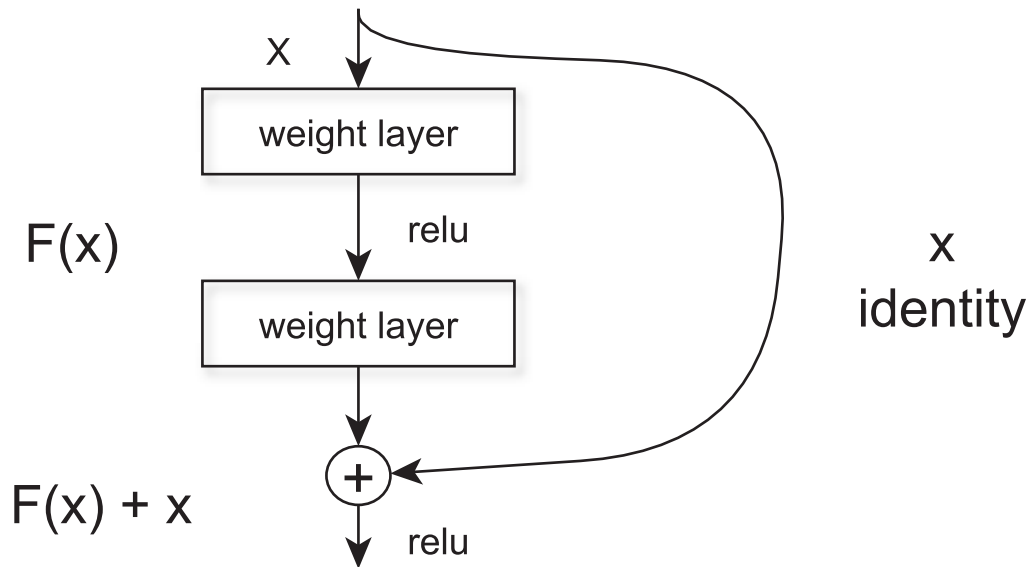


Figure 2.9: A residual block with an ordinary skip connection.
 Recreation of an image from: [6]

Research concerning new designs and new architectural approaches is being published continuously and rapidly, and we have not prioritized presenting the most current designs here. What has been presented here is some of the more significant approaches in later years. There are of course a plethora of other interesting designs that could be mentioned, such as a design with the same performance as AlexNet but with 50 times less parameters [19], a design with the alternating pooling layers replaced with convolutional layers [20] and a design which uses a variant of ResNet to achieve state of the art performance on certain problems. [21].

2.3 Deep Learning Frameworks

This chapter introduces the reader to the deep learning frameworks used in this study, as well as some other other deep learning frameworks, but firstly: a short introduction to the

purposes and functions of deep learning frameworks in general.

The algorithms and functions used in machine learning, and especially deep learning, involves a lot of mathematics; it can therefore be difficult and time consuming to implement the neural network from the ground up. Deep learning frameworks provide a high level API in order to make the implementation of neural networks simpler and more efficient. The frameworks make the implementation simpler and more efficient by abstracting away the underlying mathematics and by providing premade modules and code. By abstracting away the mathematical implementation the frameworks remove the requirement for the programmer to have an extensive mathematical background, thus making deep learning easier to work with and more available.

2.3.1 The Frameworks Used in This Study

The frameworks used in this study are: Google TensorFlow [22] and The Microsoft Cognitive Toolkit (CNTK) [23]. A third party API called Keras was used for TensorFlow. Keras provides a higher level, and more user friendly API, and is capable of running both TensorFlow and another deep learning framework called Theano [24], as backend. [25] There are also plans to develop a CNTK Keras backend. [26]

Google TensorFlow is an open source deep learning and machine learning framework developed by Google and was initially released on November 9, 2015; the stable version was released on February 15, 2017. TensorFlow is written i C++ and Python, and provides interfaces for Python, C++, Java, Haskell, and Go. [27] TensorFlow was originally developed for the purpose of conducting machine learning and deep neural networks research. TensorFlow supports computation over multiple GPUs or CPUs, both locally and distributed. [22] [28]

Microsoft CNTK is an open source deep learning framework developed by Microsoft and was initially released January 25, 2016; and has, as of this writing, not had a stable release. CNTK is written in C++ [29], and provides interfaces for Python, C++, C# [30], and Microsoft's own scripting language for deep learning: BrainScript. [31] CNTK was initially

developed for Microsoft themselves for fast training on large datasets. [32] Many of Microsoft's own products uses CNTK, e.g. Cortana, Bing, and Skype. [23] CNTK supports computation on CPU [33] and multiple GPUs, both locally and distributed [34], as well as training and hosting models on Azure. [35] [36]

There are a lot of other frameworks to choose from including: Theano [24], Torch [37], Caffe [38], and Deeplearning4j [39]

2.4 Summary

In this chapter the project has been introduced as well as the rationale behind it and its scope, to compare and evaluate deep learning frameworks. The general concepts behind machine learning, the goal of training a program to generalize well, the difference between optimization and machine learning as well as some of the challenges have been introduced to an appropriate degree. The machine learning algorithm this project is based upon, neural networks, and a major derivative, convolutional neural networks, have been introduced, explained and illustrated. Some of the more noteworthy designs and architectural choices have also been covered. The frameworks that we will be using in the project have also been introduced as well as other deep learning frameworks.

The project as a whole is partly based upon evaluating two deep learning frameworks, Google TensorFlow and Microsoft CNTK, and partly based upon learning about and applying machine learning in the form of neural networks, more specifically convolutional neural networks. Neural networks is one of the most prominent machine learning algorithms in use today, especially in the context of training networks with many layers, deep learning. The focus lies in particular upon convolutional neural networks, a variant of neural networks adapted for data with prominent spatial relations, the type of data related to problems as image classification and object detection. The designs of and the architectural approaches to convolutional neural networks have been rapidly developing in later years, and the results have been groundbreak-

ing. The tools to implement deep learning have also been rapidly developing in later years, and there are a lot of frameworks and libraries to choose from.

3 Project Design

The purpose of this project is to evaluate two frameworks for machine learning: Microsoft CNTK and Google TensorFlow. This chapter will present an overview of the different parts of the evaluation and the evaluation method used for the different parts. Chapter 3.1 covers the installation and the system requirements of the two frameworks and their dependencies; as well as the frameworks' support of programming languages. Chapter 3.2 covers the frameworks' features, functionalities and documentation; as well as their support for third-party APIs. Chapter 3.3 covers the frameworks' performance using two widely used data sets for benchmarking machine learning models; Mixed National Institute of Standards and Technology [1] dataset of handwritten numbers (MNIST) and Canadian Institute for Advanced Research's dataset of tiny images in color (CIFAR-10) [2]. Chapter 3.4 presents the design of a custom built image classifier; its development process and performance will be used in the evaluation of the frameworks' performance and user friendliness. At the end of the chapter a summary is provided.

3.1 Installation and System Requirements

In this part of the evaluation the system requirements of Microsoft CNTK and Google TensorFlow will be compared, including their dependencies. In the evaluation the following is taken into consideration: ease and speed of installation, system requirements, software and hardware support, and programming language support. Additionally, the system requirements to be able to perform the necessary calculations on GPU instead of CPU, as well as the ease of setting that up, will be evaluated. A development environment with the necessary tools will be set up as well.

3.2 Features, Functionalities and Documentation

In this part of the project CNTK:s and TensorFlow's features, functionalities, and documentation will be evaluated and set against each other. The evaluation will begin by studying the frameworks' documentation to ascertain the following: which machine learning algorithms

the frameworks provide, how intuitive the frameworks' API:s are; how well documented the frameworks are, and other features and functionalities the frameworks provide; as well as their support for third party API:s. The results of the evaluation will be used in comparing the frameworks.

3.3 Benchmarking Tests

In a comparative study of software frameworks such as TensorFlow and CNTK one may make a distinction between the softer criteria, e.g. personal experiences and perceived ease of use, from the harder criteria, e.g. numerical benchmarking data and the objective failure of a program developed with these frameworks as aid to perform a task in a given time span. The distinction is not always easy to make in practice, nor always relevant for the task at hand. Data must be interpreted and put into proper context and other subjective factors almost always come into play to muddy the picture. It is nonetheless, in our opinion, a fruitful approach to identify as many objective, or "hard", aspects of our study as possible, with results that mostly speak on their own.

The part of the project that is to be designed here is the laboratory environment where we to the best of our ability remove, or turn into constants, as many variables and parameters as possible in all aspects of the process, from the software to the hardware, in order to reliably benchmark the performance in terms of time taken, GPU/CPU usage, memory usage and other relevant factors. Part of that design consists of developing the same exact model, down to each individual layer and each parameter, in the API:s of both frameworks. Another part is making sure that the data sets in question are preprocessed identically in the pairs of models that are learning from them, and that the data sets in question are well chosen concerning quality and availability.

The datasets that will be used to benchmark the frameworks are the MNIST database of 70000 images, and the CIFAR-10 database of 60000 images. The MNIST database consists of 70000 black and white images of handwritten numbers, 0 to 9 and 28x28 pixels large, and the

task the corresponding model is supposed to learn is to classify the images into ten classes, one for each number. The CIFAR-10 database consists of 60000 images in color, 32x32 pixels large, and the task the corresponding model is supposed to learn is to classify the images into ten classes as follows: airplane, automobile, bird, cat, deer, dog, frog, horse, ship and truck.

A large part of the work in this chapter of the project will consist of learning to pick apart and redesigning existing models. Since the datasets mentioned above, MNIST and CIFAR-10, are common academic benchmarks for performance there already are examples shipped with the frameworks, examples that are ready to run with a single command. The substantial work that is to be done here is the refactoring of the examples, identifying what each part of the code does and rewriting each model until the architecture, the parameters and all other design factors are the same. The work that is to be done here is, firstly, to create two designs and, secondly, to implement each design in each of the two frameworks.

The final part; the actual benchmarking and gathering of measurement data, will be done with a combination of system monitoring tools to monitor GPU/CPU usage and memory usage. What tools to be used during the actual benchmarking will be decided during the implementation. The data that is the most important for the purpose of benchmarking are the following:

- The time it takes to finish a complete training run with a given model on a dataset.
- The GPU usage and, in special cases, the CPU usage in operations per second and in the percentage of the total capacity over time.
- The memory usage, again measured both in gigabytes and in the percentage of the total capacity over time.

The data above needs to be averaged over multiple training runs; e.g. five training runs per benchmarking session and model. Visualizing the data in the form of diagrams will be

necessary in due course, but the type of diagrams and the software that will be used for the drawing lies outside of the scope of the project design.

3.4 Implementing an Image Classifier

Since the project specification did not specify a concrete problem to solve; a problem first needs to be found and decided on. The problem needs to be not too complex, while still not being too trivial. We know that we want to implement some kind of image classifier; what kind of images however, and how specific the image classification; e.g. classifying different objects or more specifically classifying different types of the same object, that remains a part of the implementation. A sufficient amount of data of sufficient quality is also needed to be able to train and test the image classifier; what sufficient means in numerical terms and in practice; may fall on the implementation chapter as well.

The first task in this part of the project is therefore to find and look at different sources of data sets; decide on a problem of reasonable complexity, where a sufficient amount of data can be acquired for training and testing the image classifier. After that the data of course needs to be downloaded.

The second task is to begin implementing the image classifier. The image classifier will be implemented in both Microsoft CNTK and Google TensorFlow, using TensorFlow as back end with Keras, a third party API for deep learning, as front end. Keras is usable as front end to TensorFlow today, the process to add Keras to the TensorFlow core is ongoing as of this January [40] and will probably be able to use CNTK as back end at some point in the future as well [26]. All the different models of the image classifier in the different frameworks will be implemented and developed in the same programming language and development environment, to make the models more comparable. The programming language that will be used for the implementation is Python 3 [41] and the development environment that will be used is Microsoft Visual Studio 2015 Enterprise [42] with the Python Tools for Visual Studio plug-in [43] installed. In addition to using the same programming language and IDE

(Integrated Development Environment), we will also strive to make the models as similar as possible code wise; i.e. as similar as the frameworks allow; again, to make the models as comparable as possible.

The third and final task is to start training and testing the different models developed in the two frameworks, Microsoft CNTK and Google TensorFlow with Keras as front end. In the end the models development process, performance and their test accuracy will be used as part of the evaluation and comparison of the frameworks' performance and user friendliness. In this part of the project, the more softer aspects of the frameworks are of more interest, since the frameworks' raw performance will be tested in the benchmarking tests, see chapter 3.3. The more softer aspects that will be considered are: how intuitive the frameworks API:s are, ease of development, and speed of development. The previously mentioned criteria are of course of a more subjective nature, but we will strive to motivate our conclusions with as well grounded arguments as possible. In addition to the comparative aspect described above this part of the projects also contains an exploratory aspect in the form of designing and implementing the models themselves, choosing and evaluating different techniques and analyzing the performance of the models.

3.5 Summary

In this chapter the general design of the project has been laid out. The project as a whole is a larger evaluation and comparison of the two machine learning frameworks Microsoft CNTK and Google TensorFlow. In the evaluation both softer and harder aspects are considered, such as: ease of installation, system requirements, programming language support, user friendliness; performance, test accuracy, as well as ease and speed of development. The frameworks' documentation will be studied and evaluated.

As a part of the evaluation, a number of tests will be performed in order to evaluate the more harder aspects, such as the frameworks' performance and test accuracy. These tests include training and testing models developed in each framework on two well known machine

learning benchmarking tests: MNIST and CIFAR-10. Finally, a custom image classifier will be developed in each framework in order for the authors to get a first hand grip of the frameworks' user friendliness, the ease and speed of development in each framework and the process of designing the models and evaluating their performance.

4 Project Implementation

In this chapter the actual implementation of the project's parts described in chapter 3 will be described. The sub-chapters are mirrored after chapter 3's ditto, for ease of reference.

4.1 Installation and System Requirements

In order to gather the necessary information to evaluate the system requirements, software and hardware support, and programming language support; TensorFlow's and Keras', and CNTK's documentation was studied. In order to evaluate the ease and speed of installation, the frameworks were downloaded and installed. This is the more subjective part of this part of the evaluation. The aspects that the conclusions are based on are: the amount of steps required to be able to use the framework, and the perceived ease to follow the aforementioned steps. Written below are the steps the authors used to install each respective framework.

Firstly, the development environment needed to be set up. Since the development is to be done in Python, Python 3.5.3 was downloaded and installed from Python's homepage <https://www.python.org/>. The IDE used was Microsoft Visual Studio 2015 Enterprise [42], which was already installed on the computers used in this study. To be able to use Python in Visual Studio the Python Tools for Visual Studio (PTVS) extension [43] needed to be installed. To install PTVS, the Visual Studio installation was modified through the Windows Control Panel with the added PTVS extension.

Google TensorFlow was downloaded and installed through Visual Studio, with PTVS, using the built-in tool pip. To be able to use the GPU, TensorFlow's GPU version 0.12.1 was installed, pip handles and installs all python related dependencies. When using TensorFlow's GPU version two additional downloads were required: the NVIDIA CUDA Toolkit 8.0, and the NVIDIA cuDNN v5.1 (CUDA Deep Neural Network) library, which were downloaded from <https://developer.nvidia.com/cuda-downloads>, and <https://developer.nvidia.com/cudnn> respectively. The cuDNN's dll-file was placed in the CUDA-folder created after installing the CUDA Toolkit.

Keras was downloaded and installed through Visual Studio, with PTVS, using the built-in tool pip. The version installed was 1.2.2. Pip handles and installs all python related dependencies, however the scipy and numpy versions installed through pip were wrong, and needed to be downloaded and installed manually. The correct versions of scipy and numpy needed by Keras were downloaded from this site: <http://www.lfd.uci.edu/~gohlke/pythonlibs/>. The downloaded whl-files of the correct versions of scipy and numpy were installed using pip through the Windows Command Prompt.

Microsoft CNTK was downloaded from <https://github.com/Microsoft/CNTK/wiki/Setup-Windows-Binary> and installed manually. The version installed was the Beta GPU version 10 using the binary installation with scripts. Installation using pip was not available at the time of installation, but has since been added. The CNTK binary installation with scripts includes all dependencies, including NVIDIA CUDA Toolkit 8.0 and NVIDIA cuDNN v5.1, which were installed along with CNTK itself; the installation also included an Anaconda environment [44], in which CNTK is supposed to run.

4.2 Features, Functionalities and Documentation

The work that was done to compare the two frameworks, CNTK and TensorFlow, according to the criteria in the corresponding sub-chapter of the project design, consisted of an in-depth reading of the documentation (in the case of TensorFlow primarily that of Keras') to understand the workings of the frameworks, notes taken during the development and testing process and a final evaluation of the gathered information based upon a refined set of criteria. The following four criteria were chosen: features and functionalities, quantity and quality of documentation, extensibility and interoperability and cognitive load. The criteria the authors chose and the evaluation process will be elaborated upon below.

The motivation behind the choice of using features and functionalities as a criteria in the

evaluation process followed from the rationale of the project. Features and functionalities in this context means, more precisely, the tools available in the frameworks to implement neural networks and deep learning. Another aspect evaluated in relation to this criteria was also how the tools are implemented in the form of classes, methods and functions, but not in specific detail.

Since proper documentation is a necessary tool in the development process the frameworks were evaluated on the quantity and quality of the documentation available. The steps needed to find the desired information in the documentation, the amount of information available, the ease of access to tutorials, the frequency of updates to the documentation and the ease of navigating the source code were all aspects of the documentation that the frameworks were evaluated by in the process. The amount of missing and incomplete information in the documentation of the frameworks were also considered in the context of this criteria.

The extensibility and interoperability of the frameworks was not a criteria that was tested directly in either the benchmarking process, nor in the process to design and implement the image classifier. It was, however, found prudent to include it as a criteria since the details of actually deploying the model in application is important. The aspects that were evaluated here were how the deployment process might work and how the frameworks integrate with other languages, libraries and frameworks. The information acquired here were taken from the documentation of the frameworks since, once again, the criteria did not come up in the other work done within the project.

Cognitive load is a concept originated in cognitive psychology, referring to the total amount of mental effort being used in the working memory. [45] In the context of this report the term is referring to the amount of mental effort it took to learn, use and develop in the frameworks. The motivation behind the criteria is that it in general, all other things being equal, is better with a framework that is easy to use than a framework that demands a lot from the developer. Other aspects grouped under this criteria that was used during the evaluation

process are how much the frameworks conformed to the principle of least astonishment [46], how much boilerplate that was required and how well the implementations in the frameworks conformed to the literature.

4.3 Benchmarking Tests

The System used in this part of the project

Operating system: 64-bit Windows 8.1

CPU: Intel Core i7-4800MQ @ 2.80GHz

RAM: 32GB

GPU: NVIDIA Quadro K2100M 2GB

Other: TensorFlow-gpu 1.0.1, Keras 2.0.2, CNTK 2.0.beta15.0, Numpy 1.12.1 + mkl, Scipy 0.19.0, PyYaml 3.12, NVIDIA graphics driver version 376.51, NVIDIA CUDA Toolkit 8.0, NVIDIA cuDNN v5.1

We had updated TensorFlow, Keras, and CNTK since we first installed them before running the benchmarking tests. The frameworks' versions during this part of the project were: TensorFlow GPU 1.0.1, Keras 2.0.2, CNTK GPU beta 15. Both Keras and CNTK provided premade code examples for training neural networks on the MNIST and CIFAR-10 datasets. We used the given code examples for each respective dataset to get started and tried to make them as similar as possible in both frameworks, given our knowledge of the frameworks' underlying implementation and the theoretical background. We started working with the given code examples for the MNIST dataset, followed by CIFAR-10s ditto once the MNIST ones were complete. We followed the same working procedure during the implementation of the neural networks for both MNIST and CIFAR-10. The working procedure was as follows: studying the code and the documentation to understand each step in the code, finding where the code examples differ, and studying the documentation to find a substitute solution which can be made as similar as possible in both frameworks.

The prime factor in this part of the project is not the models' testing accuracy but evaluating the frameworks' performance, especially training time. The system was monitored during the training process using ASUS GPU Tweak. The training time for each epoch was printed to standard output using built-in functions in the frameworks. We ran five separate training and testing runs for each model in each framework, totaling four models and twenty runs. The models trained on MNIST went through twenty training epochs in each run. The models trained on CIFAR-10 went through forty training epochs in each run.

4.4 Implementing an Image Classifier

The System used in this part of the project

Operating system: 64-bit Windows 10

CPU: Intel Core i7-4790k @ 4.4GHz

RAM: 16GB

GPU: NVIDIA GTX 970 4GB

Other: TensorFlow-gpu 1.1.0, Keras 2.0.3, Numpy 1.12.1 + mkl, Scipy 0.19.0, PyYaml 3.12, Matplotlib 2.0.0, NVIDIA graphics driver version 376.51, NVIDIA CUDA Toolkit 8.0, NVIDIA cuDNN v5.1

The main difference between the approach set out in the corresponding chapter of the project design and the work that was actually done is that the resulting image classifier was written in Keras, with TensorFlow as backend, but not in CNTK. The problems encountered while working with CNTK, as outlined in chapter 5.2, combined with the time constraints made it impossible to implement the classifier in both frameworks. This means that the comparative aspect in this part of the project had to be left out, with only the exploratory aspect left. The rest of the process followed the project design and consisted of two parts, finding and deciding upon a dataset and then implementing the image classifier, through coding and training the resulting network.

The dataset we decided upon using was a variant of the CIFAR dataset, the same dataset we used during the benchmarking process as documented in the previous chapter, but with a different labeling. The CIFAR-100 dataset has 100 classes, classes that are grouped into 20 superclasses. Each example in the dataset is labeled with both the class it belongs and the superclass it belongs to. The motivation behind the choice of using CIFAR-100 as the dataset to train an image classifier on can be divided into several reasons. The first and foremost reason is that the dataset is widely used in research and development, which means that the quality of the data was not an unknown and could be counted upon. Another reason is that the use of an existing, easily available dataset saved considerable time and effort by eliminating the work that it takes to gather, prepare och preprocess the data needed to train the image classifier. The third and final reason is that the design and implementation of the network could make use of the work done in the benchmarking part of the project, since related problems had been worked through and tested there.

The network was designed and implemented using a variety of different techniques as follows: exponential linear units, residual blocks, global average pooling, dropout and data augmentation. The choice of activation function, the exponential linear unit (ELU) function, is a variant of the ReLU function that allows a small gradient to flow through if the input is negative, a property of the function that is purported to have a normalizing effect on output by making the activations closer to the mean. [47] The residual blocks that we used as the main tool in the design have been introduced in chapter 2.2, and two variants were used, the first consisting of an unaltered skip connection and an ordinary bottleneck layer and a second one using 1x1 convolutions with stride 2 to shrink the spatial dimensionality. The reason why residual blocks were chosen is partly due to the impressive results achieved by the use of networks using residual connections, partly due to the relative ease of understanding och implementing them in the classifier. Alternatives such as the Inception architecture were found both too complex and too inflexible in terms of flexibility and scalability.

Instead of using fully connected layers before the classifier layer in the end of the network a global average pooling layer was used instead. The global average pooling operation is a variant of pooling that averages over all feature maps and outputs a one-dimensional tensor with the size being the number of features in the input. The output can be fed directly into the classifying layer. The reason behind using it is partly, as mentioned in chapter 2.2, that the number of parameters in fully connected layers tend to grow very quickly and partly that the pooling operation utilizes the spatial correlation in the data, also as mentioned in chapter 2.2. Regularization in the form of dropout was used once per residual block and in the initial convolutional layers. Data augmentation in the form of random vertical and horizontal shifts as well as horizontal flips were also used in the training of the network. The reason behind the use of data augmentation was to regularize the network and to artificially inflate the number of training samples. Instead of ordinary stochastic gradient descent a version of the algorithm, called Adam, was tried and used. [48] It was complemented with a learning rate scheduler that decreased the learning rate in steps during the training session, the whole training session being two hundred epochs.

The design of the finalized network or image classifier, as seen in appendix E, consisted of a small amount of ordinary, sequential convolutional layers in the beginning and the majority of the network, consisting of residual blocks with increasing number of features as the network went deeper, and finally a global average pooling layer followed by a softmax layer for classification. An important aspect of the design was to make it as small as possible from the perspective of the number of parameters and operations. Since the computers at the project's disposal had comparatively weak graphic cards in comparison to the hardware used in proper research and development, the resulting network had to be trainable on at least one of the computers available. The challenge here was thus to make as good of a design as possible with limited resources.

The network was implemented with the libraries listed above. The accuracy, measured in percent and the loss on both the training set and the test set were visualized with the use of

Matplotlib, an open source library written in Python. [49]

4.5 Summary

In this chapter the details of the project implementation and the process behind it have been described in depth. The first major part of the project became a evaluation in three parts: the evaluation of the installation and system requirements, the evaluation of the features, functionalities and documentation of the frameworks and the evaluation of the frameworks' performance, the benchmarking part. The final part became the implementation of an image classifier trained on the CIFAR-100 data set.

The installation and system requirements were documented in detail and the ease of installation was evaluated in both the frameworks. The evaluation of the features, functionalities and documentation were made according to four refined criteria: features and functionalities, quality and quantity of documentation, extensibility and interoperability and cognitive load. During the benchmarking the test scripts were rewritten, analyzed and made as equal as possible. The resulting models were trained on the two chosen datasets, MNIST and CIFAR-10, and the relevant data were gathered and compiled. CIFAR-100 was chosen as the dataset for the image classifier and the image classifier was written in Keras and trained using TensorFlow as backend. The resulting network used a variety of different techniques, all with the purpose of making a good design as possible with constrained resources.

5 Results and Evaluation

In this chapter the results from each part of the evaluation will be presented and discussed. The sub-chapters are mirrored after chapter 3 and 4, for ease of reference. A summary of the results is provided at the end of the chapter.

5.1 Installation and System Requirements

Google TensorFlow's System Requirements [50]

Operating system: 64-bit Windows. Ubuntu 14.04+. Mac OS X.

Using GPU: CUDA Toolkit 8.0. cuDNN v5.1. GPU with CUDA Compute Capability 3.0+. In addition concerning Ubuntu: the libcupti-dev library.

Supported Languages: Fully supported: Python. Partially supported: C++, Java, Go. Community supported: C#, Haskell, Julia, Ruby, Rust. [51]

Microsoft CNTK:s System Requirements [52] [53] [54]

Operating system: Windows 64-bit 8.1, 64-bit 10. Windows Server 2012 R2+. Ubuntu 64-bit 14.04+.

Using GPU: CUDA Toolkit 8.0. cuDNN v5.1. NVIDIA CUB v. 1.4.1. GPU with CUDA Compute Capability 3.0+.

Other: Windows: Visual Studio 2015 Update 3+, Microsoft MPI (Message Passing Interface) version 7.0. Linux: GNU C++ 4.8.4+, Open MPI v. 1.10.3+.

Supported Languages: Fully supported: Python, C++, BrainScript. For model evaluation only: C# + other .NET languages. [55]

As can be seen from the frameworks' system requirements above, the frameworks have similar requirements regarding using the GPU, they also have similar operating system support,

except for TensorFlow having support for Mac OS X, which is a strong plus for TensorFlow. Regarding programming language support, both frameworks share support for Python and C++, while TensorFlow seem to have more community support, and therefore supports more languages overall, however, CNTK have more languages which it fully supports. Lastly, CNTK have more other system requirements.

Ease of installation The frameworks have changed a lot since we first installed them and their installation, especially CNTK:s, have been made much easier; the original installation comparison is therefore no longer valid. Also worth noting: we only installed and used the frameworks in Windows.

Originally CNTK:s installation required a lot more steps, and the steps were also more difficult to follow, due to having to enter a lot of commands in the command prompt; in addition the required download was large, approximately 2 GB, the only alleviating factor was that CNTK:s software dependencies came in the same package and were installed at the same time as CNTK. TensorFlow's installation has remained much the same, and requires only a simple pip command, TensorFlow's dependencies (CUDA Toolkit 8.0 and cuDNN v5.1) however, need to be installed manually. Now however, CNTK:s installation is much simpler and faster, and can be installed using pip, also the required download is much smaller, approximately 250 MB. Setting up to do the computations on GPU is easy in both frameworks, the GPU-version of each framework simply needed to be installed as well as the the above listed system requirements for using GPU. We got CNTK:s GPU-version to work without manually installing NVIDIA CUB (CUDA Unbound), so either it is not needed or it is included in the NVIDIA Toolkit.

In summary we would say that at the time of this writing, the frameworks are equal in terms of ease of installation. Regarding setting up the computations on GPU, we think the frameworks are equal in this regard as well, seeing as they required the same amount of steps and dependencies to be installed. When it comes to system requirements and support, the

decision which framework is better in this regard largely comes down to which operating system and programming language one is going to use; seeing as CNTK and TensorFlow both support languages the other does not, and that TensorFlow supports Mac OS X and CNTK does not.

5.2 Features, Functionalities and Documentation

Features and functionalities The authors found that, in terms of tools, capabilities and overall functionality, that the deep learning frameworks evaluated in the study were essentially equivalent in the capability to implement neural networks and deep learning. Both CNTK and TensorFlow/Keras have all the essential functionality to use, modify and implement the building blocks of neural networks such as activation functions, layers of neurons, cost functions, stochastic gradient descent and variants thereof, regularization in its different forms and others. Simply put, all the techniques and concepts introduced and explained in chapter 2.2 can be implemented using either CNTK, TensorFlow or Keras using TensorFlow as the engine.

It was further found that the API:s of both CNTK and Keras are predominantly object-oriented, with the main difference being that Keras encapsulates the network into a Model object. [56] The Model object is instantiated on a network configuration, a configuration that can be made in a sequential or functional way. [57] [58] The training process is configured and started through method calls to the encapsulated network object, and there are method calls for other functionalities. CNTK, in contrast, was found to have split the implementation of the training process into substantially more classes than Keras, however the network configuration can still be made in a sequential or functional way, as in Keras. [30]

One notable difference the authors found was that datasets such as MNIST and CIFAR-10/100 are included in the Keras framework as modules, but not in CNTK. The processing of data in regards to the aforementioned datasets were found to be cumbersome and dependent on specially designed scripts in the case of CNTK, to the degree that it adversely affected later

parts of the project due to time constraints (see chapter 4.4).

Quantity and quality of documentation The overall quality and quantity of the documentation in both frameworks were found to be lacking in several aspects, but the overall assessment made was that the documentation of TensorFlow and Keras was better than that of CNTK at the time of writing. The documentation of both frameworks was found lacking in how the information were kept up to date in comparison to the software updates and how thorough and complete the information of each section in the documentation were. One advantage that both Keras and TensorFlow had over CNTK was that the documentation in both frameworks were collected in one place each, the respective websites of each framework. CNTKs documentation was in contrast spread between the website for the Python API and the Github repository, a factor that both increased the steps necessary to find the information needed in the documentation and made it harder to find.

The tutorials provided in the documentation of either of the frameworks were found to not contain as much relevant information as the examples provided in the source code repositories. Regarding the source code it was found that Keras had the more readable and accessible source code of the two frameworks directly used by the authors. During the benchmarking process the need to look into the implementations behind the frameworks arose on occasion, and finding the code of interest was substantially harder in the CNTK repository. As a last observation the authors also noted that more effort seemed to have been made to make the documentation of Keras and TensorFlow more presentable and easy to navigate in contrast to CNTK.

Extensibility and interoperability The extensibility of the frameworks were found to be quite broad, mainly due to the fact that the main API:s of the frameworks are written in Python and therefore can be extended with chosen packages and libraries from the entire Python ecosystem. The only caveat found in that regard is that the Python API of both CNTK and TensorFlow are implemented on top of an underlying C++ implementation with types native to the implementations being used, types that necessitate conversions to and from. Keras makes use of types from both Numpy and the backend, in this case TensorFlow. The

amount of other languages supported in both frameworks, as explained in detail in the previous chapter, was also found to be an improvement in this regard. In regard to deployment it was found that models developed both in Keras and TensorFlow can make use of TensorFlow Serving, a serving system for machine learning, on both Windows and Linux. [59] Models developed with CNTK can be deployed using the CNTK NuGet Package for Visual Studio and the API it provides on Windows. Azure, Microsoft's cloud computing service, can also be used to deploy models trained with CNTK and the binaries can be used to deploy the models on Linux. [60]

Cognitive load In terms of being easy to learn, use and to develop in, the authors found that Keras were significantly more user-friendly than CNTK. The main factors that was found to favor Keras in the authors view was the amount of flexibility in choosing between convention and configuration, the amount of code that was efficiently and properly encapsulated and the self-explanatory names of functions, classes, methods and modules. The authors found it favorable that both convention, in the form of strings to implicitly instantiate default behavior, and configuration, in the form of explicitly created objects, could be used and interchanged as necessary while developing in Keras. It was also found that lower level modules and functions from TensorFlow could be used in and interact with code written in Keras in addition to its use as a computational backend, even though the authors did not make much practical use of those capabilities in the project. As mentioned earlier in this chapter Keras encapsulates the neural network configuration into a Model class, a class that the authors found easy to work with due to self-explanatory names of methods and keyword parameters. Since a lot of functionality were found to be implemented within the Model class the amount of boilerplate necessary were minimized too.

As mentioned before, the functionalities in CNTK was found to spread out in multiple classes, many of which the authors found to not conform to the principle of least surprise regarding names and sometimes behavior. The authors also found that the multiple classes mentioned above, in combination with names which weren't always self-explanatory, made it sometimes

hard to understand what was actually happening in the code. The constant need for explicit configuration and instantiating of multiple objects that in many cases were immediately used to instantiate other objects made the implementation of parts of the training process, in the authors view, look like a Russian doll of objects inside other objects. The authors would like to remark that neither explicit configuration nor functionality spread out in multiple classes are necessarily bad, but that the implementation in the case of CNTK were deemed not good enough. Regarding how well the implementations conformed with the literature the authors found that Keras won in that regard as well, due to the fact that CNTK has a explicitly different implementation that requires recalculation of certain parameters. [61]

In summary we found that the frameworks provide an equivalent set of features and functionalities, and the frameworks are more than capable of constructing neural networks. The frameworks' documentation were both found to be lacking in both quality and quantity, however Keras has the slight advantage of having its documentation gathered in one place, whereas CNTK has its documentation distributed on different sites. Keras was found to be more beginner friendly and easier to work with, as compared to CNTK. It was found that CNTK does not conform to the literature, having instead its own implementation, which in turn requires relearning if one has studied the literature, it also requires recalculating the parameters in order to function according to the literature; Keras on the other hand, requires no such readjusting, which is a big plus in our view.

5.3 Benchmarking Tests

Below the training time results are presented in tables 5.1-5.4, one for each dataset and framework. See appendices A-D for the source code used for each dataset and framework. The source code is mostly the same as the example code for the datasets, given by the frameworks. Changes were made to the example code to make the neural networks in the examples as similar as we could make them. We were unable to find the necessary information in the frameworks' documentation to ascertain some aspects of the example code's function and implementation; therefore there are still some aspects of the code that we are unsure about,

these aspects are listed below for the sake of transparency.

- 'randomize' in CNTK; what is Keras' equivalent function? Is it done by default in Keras, or not at all?
- Shuffle is done by default in Keras; how is it done in CNTK?
- The parameters to the optimizers, epsilon (ϵ) in Adagrad [62] for example; where is it set in CNTK? Is it set at all? What is it called in CNTK, and what is its default value?
- In CNTK you can set both training minibatch size and testing minibatch size; is that possible in Keras, and how and where is it done?
- In CNTK the minibatch size changes dynamically during training; where is it done in the code, or is it done automatically? What are the default values? Can it be done in Keras?

Given the uncertainties listed above and the experimental setup presented in chapter 4.3, the training time using CNTK is consistently faster than using Keras with TensorFlow as backend, as can be seen in tables 5.1-5.4. The variation in training time is relatively low in both frameworks, although the variation is slightly higher using Keras with TensorFlow as backend, the last run on CIFAR-10 using Keras with TensorFlow as backend especially stands out, having 30 seconds to its nearest neighbour, see table 5.4. Interestingly, the first epoch was consistently the epoch that took the most time to finish, see the Maximum Epoch Time column in tables 5.1-5.4. After some testing after the results presented in tables 5.1-5.4 were compiled, we came to the conclusion that the first epochs took more time because we ran the scripts with debugging on in Visual Studio. When we ran the scripts without debugging, the first epochs took approximately the same time as the rest of the epochs.

Regarding the comparison of CPU versus GPU, the GPU was found to be so superior in terms of training speed that further investigation in the matter was found superfluous. Regarding the frameworks' GPU usage and VRAM usage, we quickly noticed during the implementation

that these aspect were not interesting to evaluate; they were similar in both frameworks and mostly constant. We therefore chose to not look at the GPU usage and VRAM usage in further detail, and chose to focus on the more important part of the evaluation: comparing the training time of the two frameworks. One interesting aspect however, regarding the memory usage, is that Keras by default uses all available VRAM, while CNTK does not, however it is possible to configure Keras to only use the memory it needs, by a few lines of code.

In summary CNTK was found to give a shorter training time of the networks compared to Keras with TensorFlow as backend, see tables 5.1-5.4, given the uncertainties listed and the experimental setup presented in chapter 4.3. GPU was found to be so superior over CPU in terms of training speed, that if one has a GPU available one should use it instead of the CPU. GPU and VRAM usage were both similar in both frameworks.

Run (nr.)	Total Time (s)	Mean Epoch Time (s)	Maximum Epoch Time (s)
1	328	16.4	18.1
2	324	16.2	18.1
3	324	16.2	18.7
4	324	16.2	18.1
5	322	16.1	18.1
1-5	1622	16.2	18.7

Table 5.1: Results CNTK MNIST

Run (nr.)	Total Time (s)	Mean Epoch Time (s)	Maximum Epoch Time (s)
1	444	22.2	26
2	451	22.6	26
3	447	22.4	26
4	446	22.3	26
5	444	22.2	26
1-5	2232	22.3	26

Table 5.2: Results Keras/TensorFlow MNIST

Run (nr.)	Total Time (s)	Mean Epoch Time (s)	Maximum Epoch Time (s)
1	2776	69.4	74.7
2	2766	69.2	74.2
3	2756	68.9	73.4
4	2756	68.9	73.1
5	2760	69.0	74.6
1-5	13814	69.1	74.7

Table 5.3: Results CNTK CIFAR-10

Run (nr.)	Total Time (s)	Mean Epoch Time (s)	Maximum Epoch Time (s)
1	3731	93.3	98
2	3745	93.6	99
3	3759	94.0	98
4	3758	94.0	101
5	3701	92.5	97
1-5	18694	93.5	101

Table 5.4: Results Keras/TensorFlow CIFAR-10

5.4 Implementing an Image Classifier

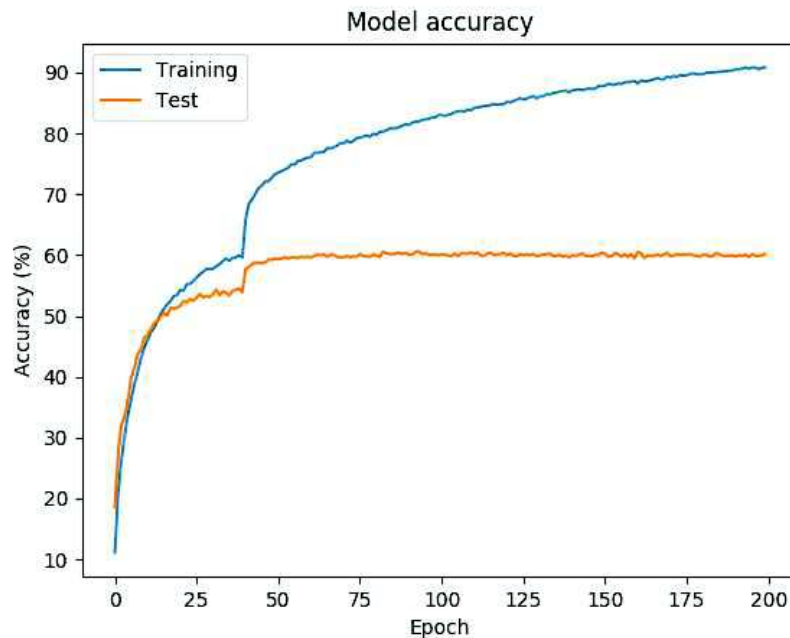


Figure 5.1: Plot of the accuracy for the first run.

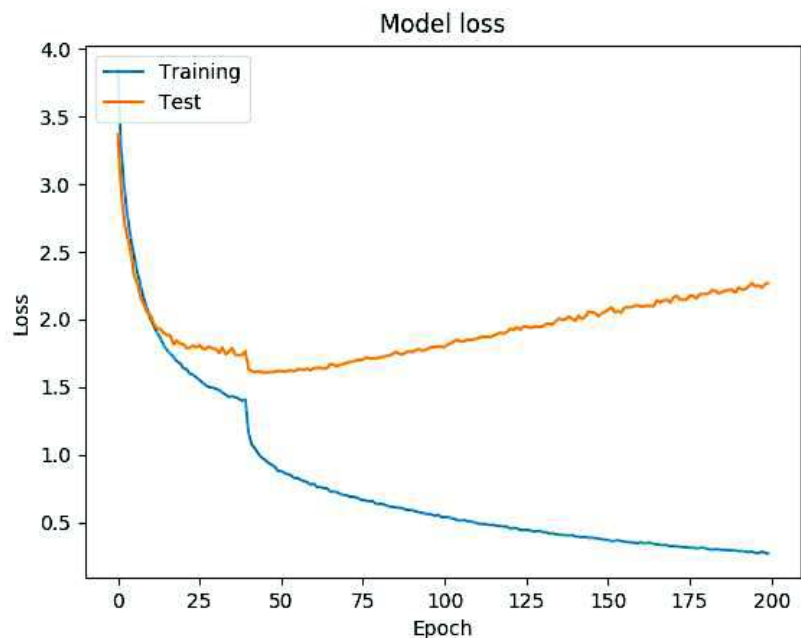


Figure 5.2: Plot of the loss for the first run.

The diagrams presented here were results from three training sessions or runs in the form of two diagrams per run, one diagram depicting the training and test accuracy and one depicting the training and test loss. The accuracy in this context is referring to the ratio of correctly classified examples to the number of examples in the set as a whole, and the loss is referring to the size of the current error or the size of the objective function (see chapter 2.2 for more detail). The goal here, as with all kinds of machine learning, consisted of minimizing the loss and maximizing the accuracy on both the training set and the test set. The training process, as seen in the diagrams, resulted in a final accuracy of around 60 percent on the test set and around 90 percent of the training set. The training loss decreased smoothly and the training accuracy increased smoothly in all three runs, with a temporary increase in the convergence regarding both metrics around epoch 40 after a slowdown starting around epoch 12-13. The accuracy on the test set did not, in contrast, increase in either of the three runs after epoch 50, an epoch after which the test loss started to increase steadily. The aforementioned slowdown

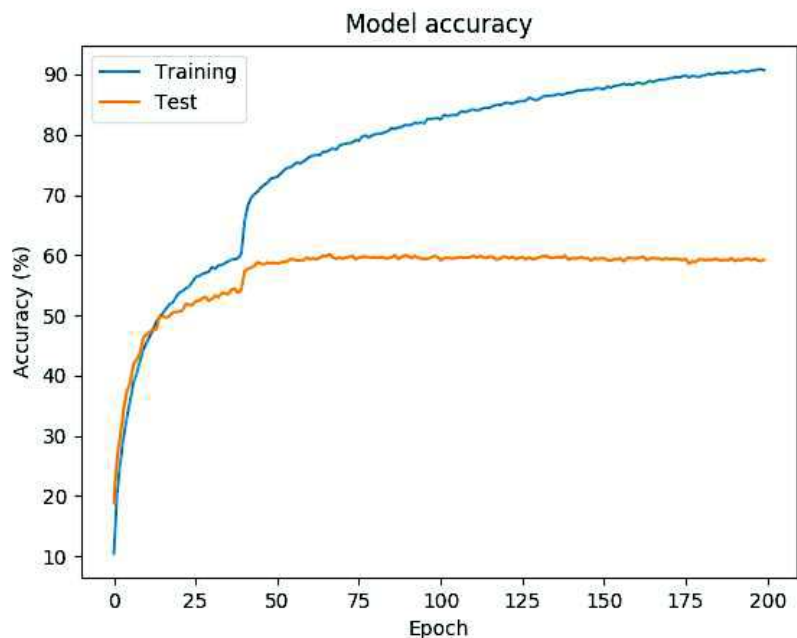


Figure 5.3: Plot of the accuracy for the second run.

was also more pronounced in the case of both the test accuracy and loss.

The behavior described above is a classical example of the model overfitting on the training set, losing the ability to generalize about data it has not seen during the training session. The diagrams of the training and test losses particularly illustrative in that regard, as the curves grow clearly apart after epoch 50 in all three runs. The decrease in the learning rate after epoch 40 markedly improved the performance, in terms of loss and accuracy, on the training set and temporarily improved the performance the test set, providing evidence of the efficiency of the learning rate schedule (see chapter 4.4). The rationale behind the use of such a schedule is that it improves the convergence of the loss towards the minimum, an assertion the authors found proof for here. An unwanted effect of the learning rate schedule might have been to increase the overfitting, since the divergence of the performance on the training test respective to the test set accelerated after the decrease in the learning rate. The last observation the authors would like to make here is that all three of the training sessions ran for all of the

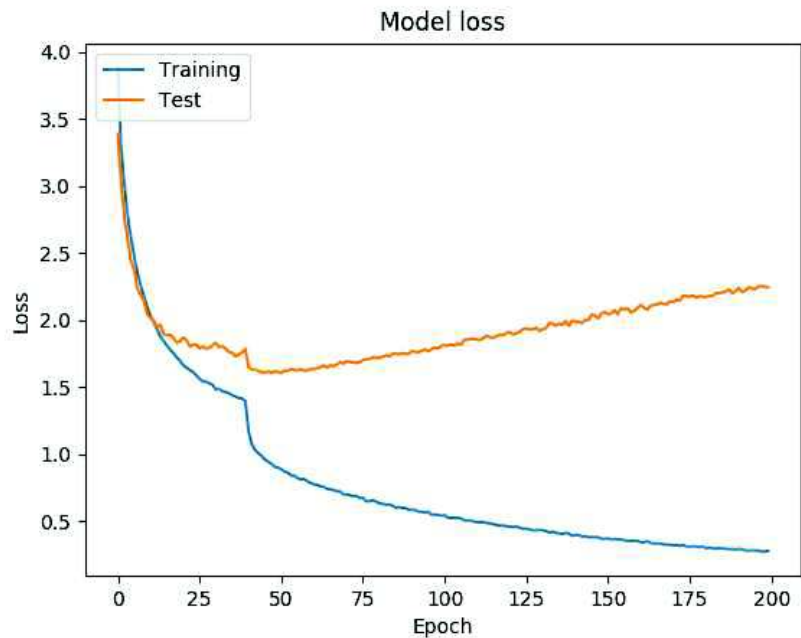


Figure 5.4: Plot of the loss for the second run.

200 epochs, a number of epochs that seems to have been redundant or even harmful to the performance of the model in regards to the overfitting seen.

The results in terms of accuracy on the test set are pretty far from those of the top contenders (the state of the art being an accuracy of 75.72 percent) [63], but the authors would like to argue that it could not be expected that state of the art results could be achieved with the resources at the project's disposal, more precisely the hardware, the time allocated and the prior knowledge of the authors. A top performing model in terms of accuracy was not the goal in this part of the project, the goal was to make as good of a model as possible with the given constraints. If the implementation of the design itself, and the coding behind it, is taken into account the authors found several ways it could be improved, improvisations that with enough time and better hardware could be easily implemented and tested. One example of the flaws that could be fixed is that, due to the time and hardware constraints, the spatial resolution of the feature maps in the later stages of the network might have been too small. Another

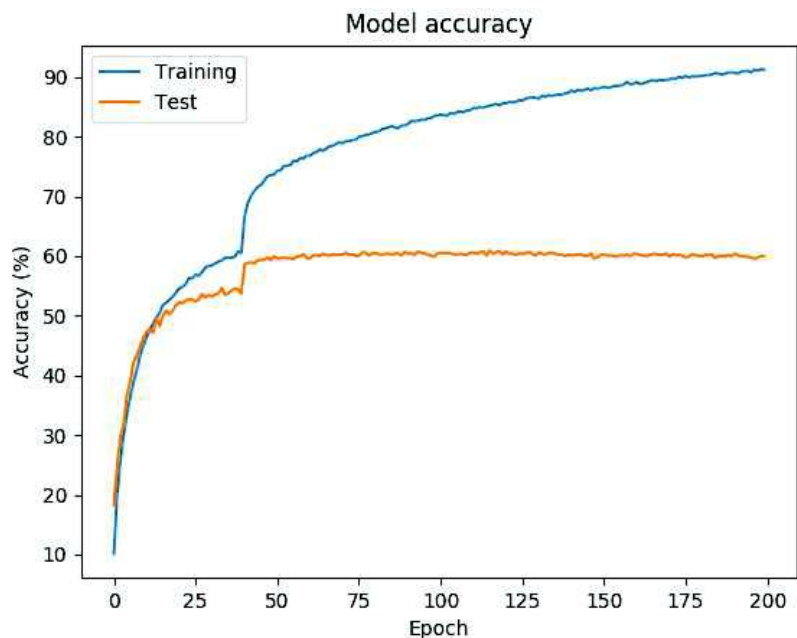


Figure 5.5: Plot of the accuracy for the third run.

improvement might have been to use early stopping, a technique that stops the training after a number of epochs without improvement in regards to a given metric, such as the test loss or accuracy. Other improvements that could be made include a more elaborate learning rate scheduler and heavier regularization, both in form of the existing techniques and others (see appendix E for the implementation that produced the results).

To conclude this analysis a short summary can be made; the model was found to quite rapidly overfit on the training set, a learning rate schedule was found to be beneficial, and the number of epochs seem to have been detrimental to the performance of the model. A number of possible improvements were found and discussed as follows: more regularization, a learning rate schedule with more steps, early stopping, and higher spatial resolution in the later parts of the network.

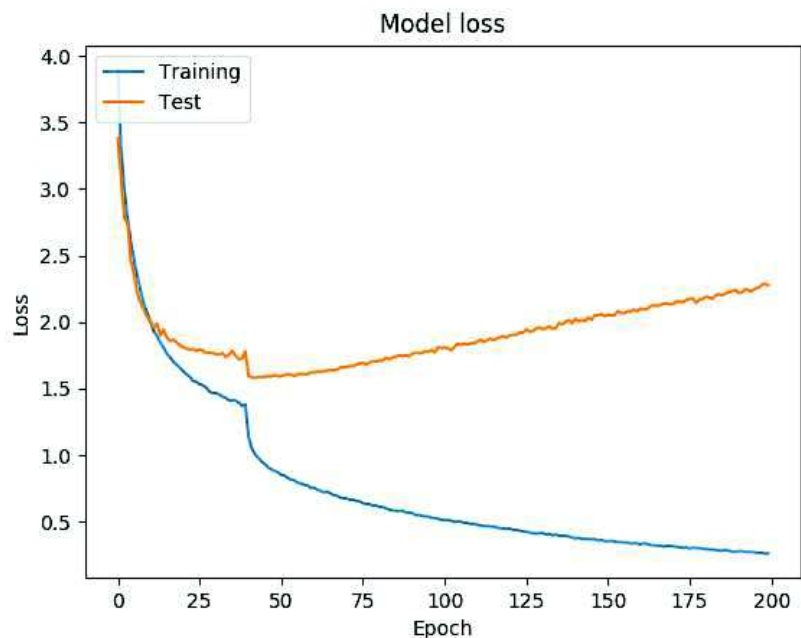


Figure 5.6: Plot of the loss for the third run.

5.5 Summary

In this chapter the results from the evaluation of Microsoft CNTK and Google TensorFlow using Keras' API, as well as the results from the implementation of an image classifier using Keras with TensorFlow as backend, was presented and analyzed.

At the time of this writing, the frameworks are equal in terms of ease of installation. Regarding setting up the computations on GPU, we think the frameworks are equal in this regard as well, seeing as they required the same amount of steps and dependencies to be installed. When it comes to system requirements and support, the decision which framework is better in this regard largely comes down to which operating system and programming language one is going to use; seeing as CNTK and TensorFlow both support languages the other does not, and that TensorFlow supports Mac OS X and CNTK does not.

We found that the frameworks provide an equivalent set of features and functionalities, and

the frameworks are more than capable of constructing neural networks. The frameworks' documentation were both found to be lacking in both quality and quantity, however Keras has the slight advantage of having its documentation gathered in one place, whereas CNTK has its documentation distributed on different sites. Keras was found to be more beginner friendly and easier to work with, as compared to CNTK. It was found that CNTK does not conform to the literature, having instead its own implementation, which in turn requires relearning if one has studied the literature, it also requires recalculating the parameters in order to function according to the literature; Keras on the other hand, requires no such readjusting, which is a big plus in our view.

CNTK was found to give a shorter training time of the networks compared to Keras with TensorFlow as backend, see tables 5.1-5.4, given the uncertainties listed in chapter 5.3 and the experimental setup presented in chapter 4.3. GPU was found to be so superior over CPU in terms of training speed, that if one has a GPU available one should use it instead of the CPU. GPU and VRAM usage were both similar in both CNTK and Keras with TensorFlow as backend.

Regarding the implementation of an image classifier; the model was found to quite rapidly overfit on the training set, a learning rate schedule was found to be beneficial, and the number of epochs seem to have been detrimental to the performance of the model. A number of possible improvements were found and discussed as follows: more regularization, a learning rate schedule with more steps, early stopping, and higher spatial resolution in the later parts of the network.

6 Conclusion

In this chapter the conclusion of the evaluation of the frameworks is provided and motivated, as well as suggestions for future work, and finally some concluding remarks by the authors.

The main goal of this project was to evaluate the two deep learning frameworks Google TensorFlow and Microsoft CNTK, primarily based on their performance in the training time of neural networks. In the aforementioned aspect CNTK performed better, than TensorFlow with Keras as frontend, see chapter 5.3 for a more detailed presentation of the benchmarking results. We chose to use the third-party API Keras instead of TensorFlow's own API when working with TensorFlow, we made this choice because we found TensorFlow's own API too cumbersome to work with, given the project's time span and our inexperience in the field of machine learning when the project began. When using Keras with TensorFlow as backend we found the development process in it to be easy and intuitive, see chapter 5.2 for our more detailed opinions on the frameworks.

In conclusion; even though CNTK performed better on the benchmarking tests, we found Keras with TensorFlow as backend to be much easier and more intuitive to work with, two aspects we think are more important when choosing a deep learning framework to work with. In addition; the fact that CNTKs underlying implementation of the machine learning algorithms and functions differ from that of the literature and of other frameworks, makes the development process tedious if you, like us, have just studied the literature and are new to machine learning. Therefore; based on the reasons just mentioned, if we had to choose a framework to continue working in, we would choose Keras with TensorFlow as backend, even though the performance is less compared to CNTK.

6.1 Future Work

Regarding the possibility of future work in this area we found four interesting areas to explore, evaluate and to work on further development in: data gathering and preparation, integration

with other applications, further exploration of CNTK and production environments.

The quantity and the quality of data available is one of the most critical factors, if not the most critical, influencing the performance of a trained model. The first step in getting the necessary quantity of data is to gather the raw data, a process that, all the practical difficulties aside, will take a considerable amount of time. The practical setup of the data gathering process would necessitate developing tools for automatic gathering of data, tools that would have to be specialized depending on the kind of data required, e.g. production data from different processes in a paper mill. The raw data would then have to be cleansed, labeled and divided into training and test sets, a process that will entail additional costs in terms of time and money. The legal and business aspects of the process would also be needed to be taken into account. Despite the work needed here we think that working with acquiring data could become an interesting project in its own right.

The question of how to integrate the finished, trained models into applications for practical use is a question we found interesting, but did not have time for in the project. One major obstacle here would be how to make the predictions or the classifications of the model as fast as possible, an obstacle that must be overcome with both the proper hardware and the proper software. The application must in other words perform the chosen task, e.g. finding faces in pictures, without too much delay. The idea of developing such an application, with all the architectural choices, setup of hardware and other aspects that such a development process would entail, is an idea that we both found interesting. The topic of production environments is closely related to that of developing an application, but merits discussion in its own right. Having high-performing hardware is key to making the training and serving of models feasible with regards to time and latency, a fact we learned the hard way during the project. The hardware that is needed, particularly high-end GPU:s, is often too expensive to justify owning and administering it yourself. There are platforms that could be used to lease or purchase the necessary computing power, platforms offering virtual machines for that purpose. Setting up an environment for production, development and research with the proper resources is both

something we would like to do and could not do during the time span of the project.

The final area of potential future work, using CNTK in more depth, had to be cut from the project due to time constraints. Making CNTK work as well for us as we made Keras and Tensorflow do in implementing more advanced functionalities could be a good learning experience in several aspects. Learning to implement the building blocks behind deep learning and neural networks in several frameworks might be knowledge worth having since both the field at large and the development tools change rapidly. The frameworks that are being used for development today could be obsolete tomorrow - it is good to be prepared.

6.2 Concluding Remarks

Neural networks and deep learning is an area of research with a lot of unanswered questions, an area with such intensive research that it can be difficult to keep pace with all the new findings that are discovered constantly. A practitioner need to be well acquainted with the underlying theory to able to work with neural networks and deep learning and also need to be familiar with their building blocks to be able to use them effectively. A very important thing to emphasize here that in the process of developing machine learning applications the quality of the available data, not just the quality of the code, affects the end quality of the product, unlike other areas of software development. The performance of a machine learning model can almost always be improved with more and better data.

A final and perhaps uplifting remark is that not a lot of code is needed to develop a working prototype with the use of modern deep learning frameworks. The time saved due to the ease of scripting and designing models in these frameworks is valuable, especially when the time needed to tune and train the model is taken into account. Since a lot of the work that is involved with machine learning consists of testing, validating and discarding multiple hypotheses and ideas the turnaround time while developing the model should be as small as possible, and good frameworks help with that.

References

- [1] Wikipedia. MNIST database — wikipedia, the free encyclopedia, 11 April, 2017. [Online; accessed 13-April-2017].
- [2] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. 2009.
- [3] Tom M Mitchell et al. Machine learning. wcb, 1997.
- [4] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. <http://www.deeplearningbook.org>.
- [5] Michael A Nielsen. *Neural networks and deep learning*. 2017. <http://neuralnetworksanddeeplearning.com/>.
- [6] Eugenio Culurciello. Neural network architectures, 2016. [Online; accessed 19-April-2017].
- [7] Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning. *arXiv preprint arXiv:1603.07285*, 2016.
- [8] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.
- [9] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.
- [10] ImageNet. Imagenet. <http://image-net.org/>, 2017. Online; accessed 18 May, 2017.
- [11] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.
- [12] Alfredo Canziani, Adam Paszke, and Eugenio Culurciello. An analysis of deep neural network models for practical applications. *arXiv preprint arXiv:1605.07678*, 2016.
- [13] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–9, 2015.
- [14] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. *arXiv preprint arXiv:1312.4400*, 2013.
- [15] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.
- [16] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2818–2826, 2016.

- [17] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.
- [18] Andreas Veit, Michael J Wilber, and Serge Belongie. Residual networks behave like ensembles of relatively shallow networks. In *Advances in Neural Information Processing Systems*, pages 550–558, 2016.
- [19] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and < 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016.
- [20] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller. Striving for simplicity: The all convolutional net. *arXiv preprint arXiv:1412.6806*, 2014.
- [21] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. *arXiv preprint arXiv:1605.07146*, 2016.
- [22] Google. Tensorflow homepage. <https://www.tensorflow.org/>, 2017. Online; accessed 13 April, 2017.
- [23] Microsoft. CNTK homepage. <https://www.microsoft.com/en-us/research/product/cognitive-toolkit/>, 2017. Online; accessed 13 April, 2017.
- [24] Theano. Theano documentation. <http://deeplearning.net/software/theano/index.html>, 31 March, 2017. [Online; accessed 13-April-2017].
- [25] Various. Keras documentation. <https://keras.io/>, 2017. Online; accessed 13 April, 2017.
- [26] Various. CNTK keras backend. <https://github.com/Microsoft/CNTK/issues/797>, 2016. Online; accessed 13 April, 2017.
- [27] Wikipedia. Tensorflow wikipedia. <https://en.wikipedia.org/wiki/TensorFlow>, 2017. Online; accessed 13 April, 2017.
- [28] Google. Distributed tensorflow. <https://www.tensorflow.org/deploy/distributed>, 8 March, 2017. Online; accessed 13 April, 2017.
- [29] Wikipedia. CNTK wikipedia. https://en.wikipedia.org/wiki/Microsoft_Cognitive_Toolkit, 29 March, 2017. Online; accessed 13 April, 2017.
- [30] Microsoft. CNTK library API. <https://github.com/Microsoft/CNTK/wiki/CNTK-Library-API>, 13 April, 2017. Online; accessed 13 April, 2017.
- [31] Microsoft. Brainscript. <https://github.com/Microsoft/CNTK/wiki/Using-CNTK-with-BrainScript>, March, 2017. Online; accessed 13 April, 2017.
- [32] Microsoft. CNTK FAQ. <https://github.com/Microsoft/CNTK/wiki/CNTK-FAQ>, April, 2017. Online; accessed 13 April, 2017.

- [33] Microsoft. CNTK CPU-version installation. <https://github.com/Microsoft/CNTK/wiki/Setup-CNTK-on-your-machine>, 11 April, 2017. Online; accessed 13 April, 2017.
- [34] Microsoft. Distributed CNTK. <https://github.com/Microsoft/CNTK/wiki/Multiple-GPUs-and-machines>, April, 2017. Online; accessed 13 April, 2017.
- [35] Microsoft. Azure CNTK. <https://github.com/Microsoft/CNTK/wiki/CNTK-on-Azure>, March, 2017. Online; accessed 13 April, 2017.
- [36] Microsoft. Azure. <https://azure.microsoft.com/en-us/>, 2017. Online; accessed 13 April, 2017.
- [37] Torch. Torch homepage. <http://torch.ch/>, n.d. [Online; accessed 13-April-2017].
- [38] Evan Shelhamer Yangqing Jia. Caffe homepage. <http://caffe.berkeleyvision.org/>, n.d. [Online; accessed 13-April-2017].
- [39] Skymind. Deeplearning4j homepage. <https://deeplearning4j.org/>, 2017. [Online; accessed 13-April-2017].
- [40] François Chollet. Twitter - the author of keras confirms keras' integration into tensorflow. <https://twitter.com/fchollet/status/820746845068505088>, January 15, 2017. Online; accessed 27 April, 2017.
- [41] Python. Python. <https://www.python.org/>, 2017. Online; accessed 13 April, 2017.
- [42] Microsoft. Visual studio. <https://www.visualstudio.com/downloads/>, 2017. Online; accessed 13 April, 2017.
- [43] Microsoft. Python tools for visual studio. <https://www.visualstudio.com/vs/python/>, 2017. Online; accessed 13 April, 2017.
- [44] Continuum Analytics. Anaconda documentation, 2017. [Online; accessed 19-April-2017].
- [45] Wikipedia. Cognitive load — Wikipedia, the free encyclopedia, 3 May, 2017. [Online; accessed 03-May-2017].
- [46] Eric S Raymond. *The art of Unix programming*. Addison-Wesley Professional, 2003.
- [47] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289*, 2015.
- [48] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.
- [49] The Matplotlib development team. Matplotlib homepage. <https://matplotlib.org/>, 2017. Online; accessed 17 May, 2017.

- [50] Google. Tensorflow's system requirements. <https://www.tensorflow.org/install/>, April 26, 2017. Online; accessed 27 April, 2017.
- [51] Google. Tensorflow's api documentation. https://www.tensorflow.org/api_docs/, April 26, 2017. Online; accessed 27 April, 2017.
- [52] Microsoft. CNTK:s windows system requirements. <https://github.com/Microsoft/CNTK/wiki/Setup-CNTK-on-Windows>, April 26, 2017. Online; accessed 27 April, 2017.
- [53] Microsoft. CNTK:s linux system requirements. <https://github.com/Microsoft/CNTK/wiki/Setup-CNTK-on-Linux>, April, 2017. Online; accessed 27 April, 2017.
- [54] Microsoft. CNTK:s tested configurations. <https://github.com/Microsoft/CNTK/wiki/Test-Configurations>, January 13, 2017. Online; accessed 27 April, 2017.
- [55] Microsoft. CNTK:s tested configurations. <https://github.com/Microsoft/CNTK/wiki/CNTK-Library-API>, April 20, 2017. Online; accessed 27 April, 2017.
- [56] Various. About keras models. <https://keras.io/models/about-keras-models/>, 2017. Online; accessed 04 May, 2017.
- [57] Various. Getting started with the keras sequential model. <https://keras.io/getting-started/sequential-model-guide/>, 2017. Online; accessed 04 May, 2017.
- [58] Various. Getting started with the keras functional API. <https://keras.io/getting-started/functional-api-guide/>, 2017. Online; accessed 04 May, 2017.
- [59] Google. Tensorflow serving. <https://tensorflow.github.io/serving/>, 2017. Online; accessed 04 May, 2017.
- [60] Mark Hillebrand. CNTK evaluation overview. <https://github.com/Microsoft/CNTK/wiki/CNTK-Evaluation-Overview>, 2017. Online; accessed 04 May, 2017.
- [61] Goran Dubajic. Brainscript SGD block. <https://github.com/Microsoft/CNTK/wiki/BrainScript-SGD-Block#converting-learning-rate-and-momentum-parameters-from-other-toolkits>, 2017. Online; accessed 11 May, 2017.
- [62] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for on-line learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.
- [63] Rodrigo Benenson. Classification datasets results. http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html, 2017. Online; accessed 10 May, 2017.

A MNIST Keras/TensorFlow source code

```
# Copyright (c) François Chollet and other contributors. All rights reserved.
# Modifications by Rasmus Airola and Kristoffer Hager.
# Licensed under the MIT license.
# =====

from __future__ import print_function
import numpy as np
np.random.seed(1337) # for reproducibility

from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten
from keras.layers import Convolution2D, MaxPooling2D
from keras.utils import np_utils
from keras import backend as K, optimizers

batch_size = 128
nb_classes = 10
nb_epoch = 16

# input image dimensions
img_rows, img_cols = 28, 28
# number of convolutional filters to use
nb_filters = 32
# size of pooling area for max pooling
pool_size = (2, 2)
# convolution kernel size
kernel_size = (3, 3)

# the data, shuffled and split between train and test sets
(X_train, y_train), (X_test, y_test) = mnist.load_data()

if K.image_dim_ordering() == 'th':
    X_train = X_train.reshape(X_train.shape[0], 1, img_rows, img_cols)
    X_test = X_test.reshape(X_test.shape[0], 1, img_rows, img_cols)
    input_shape = (1, img_rows, img_cols)
else:
    X_train = X_train.reshape(X_train.shape[0], img_rows, img_cols, 1)
    X_test = X_test.reshape(X_test.shape[0], img_rows, img_cols, 1)
```

```

input_shape = (img_rows, img_cols, 1)

X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
X_train /= 255
X_test /= 255
print('X_train_shape:', X_train.shape)
print(X_train.shape[0], 'train_samples')
print(X_test.shape[0], 'test_samples')

# convert class vectors to binary class matrices
Y_train = np_utils.to_categorical(y_train, nb_classes)
Y_test = np_utils.to_categorical(y_test, nb_classes)

model = Sequential()

model.add(Convolution2D(nb_filters, 5, 5,
                        border_mode='same',
                        input_shape=input_shape))
model.add(Activation('relu'))
model.add(MaxPooling2D((3,3), (2,2)))

model.add(Convolution2D(48, 3, 3))
model.add(Activation('relu'))
model.add(MaxPooling2D((3,3), (2,2)))

model.add(Convolution2D(64, 3, 3))
model.add(Activation('relu'))

model.add(Flatten())
model.add(Dense(96))
model.add(Activation('relu'))
model.add(Dropout(0.5))

model.add(Dense(nb_classes))
model.add(Activation('softmax'))

model.compile(loss='categorical_crossentropy',
              optimizer='adagrad',
              metrics=['accuracy'])

model.fit(X_train, Y_train, batch_size=batch_size, nb_epoch=nb_epoch,

```

```
        verbose=2, validation_data=(X_test, Y_test))

score = model.evaluate(X_test, Y_test, verbose=0)
print('Test_score:', score[0])
print('Test_accuracy:', score[1])
```


B MNIST CNTK source code

```
# Copyright (c) Microsoft. All rights reserved.
# Modifications by Rasmus Airola and Kristoffer Hager.
# Licensed under the MIT license.
# =====

from __future__ import print_function
import numpy as np
import sys
import os
import cntk

# Paths relative to current python file.
abs_path = os.path.dirname(os.path.abspath(__file__))
data_path = os.path.join(abs_path, "MNIST")
model_path = os.path.join(abs_path, "Models")

# Define the reader for both training and evaluation action.
def create_reader(path, is_training, input_dim, label_dim):
    return cntk.io.MinibatchSource(cntk.io.CTFDeserializer(path, cntk.io.StreamDefs(
        features = cntk.io.StreamDef(field='features', shape=input_dim),
        labels = cntk.io.StreamDef(field='labels', shape=label_dim)
    )), randomize=is_training, epoch_size = cntk.io.INFINITELY_REPEAT if is_training else
        cntk.io.FULL_DATA_SWEEP)

# Creates and trains a feedforward classification model for MNIST images
def convnet_mnist(debug_output=False):
    image_height = 28
    image_width = 28
    num_channels = 1
    input_dim = image_height * image_width * num_channels
    num_output_classes = 10

    # Input variables denoting the features and label data
    input_var = cntk.ops.input_variable((num_channels, image_height, image_width), np.
        float32)
    label_var = cntk.ops.input_variable(num_output_classes, np.float32)

    # Instantiate the feedforward classification model
    scaled_input = cntk.ops.element_times(cntk.ops.constant(0.00392156), input_var)
```

```

with cntk.layers.default_options(activation=cntk.ops.relu, pad=False):
    conv1 = cntk.layers.Convolution2D((5,5), 32, pad=True)(scaled_input)
    pool1 = cntk.layers.MaxPooling((3,3), (2,2))(conv1)
    conv2 = cntk.layers.Convolution2D((3,3), 48)(pool1)
    pool2 = cntk.layers.MaxPooling((3,3), (2,2))(conv2)
    conv3 = cntk.layers.Convolution2D((3,3), 64)(pool2)
    f4     = cntk.layers.Dense(96)(conv3)
    drop4  = cntk.layers.Dropout(0.5)(f4)
    z      = cntk.layers.Dense(num_output_classes, activation=None)(drop4)

ce = cntk.ops.cross_entropy_with_softmax(z, label_var)
pe = cntk.ops.classification_error(z, label_var)

reader_train = create_reader(os.path.join(data_path, 'Train-28x28_cntk_text.txt'), True,
                             input_dim, num_output_classes)

# training config
epoch_size = 60000                # for now we manually specify epoch size
minibatch_size = 128

# Set learning parameters
lr_schedule = cntk.learning_rate_schedule([0.01], cntk.learner.UnitType.sample,
                                           epoch_size)

# Instantiate the trainer object to drive the model training
learner = cntk.learner.adagrad(z.parameters, lr_schedule)
trainer = cntk.Trainer(z, (ce, pe), learner)

# define mapping from reader streams to network inputs
input_map = {
    input_var : reader_train.streams.features,
    label_var : reader_train.streams.labels
}

cntk.utils.log_number_of_parameters(z) ; print()
max_epochs = 20
progress_printer = cntk.utils.ProgressPrinter(tag='Training', log_to_file='log.txt',
                                              num_epochs=max_epochs)

# Get minibatches of images to train with and perform model training
for epoch in range(max_epochs):    # loop over epochs

```

```

sample_count = 0
while sample_count < epoch_size: # loop over minibatches in the epoch
    data = reader_train.next_minibatch(min(minibatch_size, epoch_size - sample_count
    ), input_map=input_map) # fetch
        minibatch.
    trainer.train_minibatch(data) # update model with
        it
    sample_count += data[label_var].num_samples # count samples
        processed so far
    progress_printer.update_with_trainer(trainer, with_metric=True) # log progress

progress_printer.epoch_summary(with_metric=True)
z.save(os.path.join(model_path, "ConvNet_MNIST_{}.dnn".format(epoch)))

# Load test data
reader_test = create_reader(os.path.join(data_path, 'Test-28x28_cntk_text.txt'), False,
    input_dim, num_output_classes)

input_map = {
    input_var : reader_test.streams.features,
    label_var : reader_test.streams.labels
}

# Test data for trained model
epoch_size = 10000
minibatch_size = 128

# process minibatches and evaluate the model
metric_numer = 0
metric_denom = 0
sample_count = 0
minibatch_index = 0

while sample_count < epoch_size:
    current_minibatch = min(minibatch_size, epoch_size - sample_count)

    # Fetch next test min batch.
    data = reader_test.next_minibatch(current_minibatch, input_map=input_map)

    # minibatch data to be trained with
    metric_numer += trainer.test_minibatch(data) * current_minibatch
    metric_denom += current_minibatch

```

```
# Keep track of the number of samples processed so far.
sample_count += data[label_var].num_samples
minibatch_index += 1

print("")
print("Final_Results:_Minibatch[1-{}]:_errs_={:0.2f}%*_{}".format(minibatch_index+1, (
    metric_numer*100.0)/metric_denom, metric_denom))
print("")

return metric_numer/metric_denom

if __name__=='__main__':
    convnet_mnist()
```

C CIFAR-10 Keras/TensorFlow source code

```
# Copyright (c) François Chollet and other contributors. All rights reserved.
# Modifications by Rasmus Airola and Kristoffer Hager.
# Licensed under the MIT license.
# =====

from __future__ import print_function
import keras
from keras.datasets import cifar10
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten
from keras.layers import Conv2D, MaxPooling2D

batch_size = 32
num_classes = 10
epochs = 40
data_augmentation = False # True

# input image dimensions
img_rows, img_cols = 32, 32
# The CIFAR10 images are RGB.
img_channels = 3

# The data, shuffled and split between train and test sets:
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
print('x_train_shape:', x_train.shape)
print(x_train.shape[0], 'train_samples')
print(x_test.shape[0], 'test_samples')

# Convert class vectors to binary class matrices.
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)

model = Sequential()

model.add(Conv2D(32, (3, 3), padding='same',
                input_shape=x_train.shape[1:]))
model.add(Activation('relu'))
model.add(Conv2D(32, (3, 3)))
```

```

model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Conv2D(64, (3, 3), padding='same'))
model.add(Activation('relu'))
model.add(Conv2D(64, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Flatten())
model.add(Dense(512))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes))
model.add(Activation('softmax'))

# Let's train the model using Adagrad
model.compile(loss='categorical_crossentropy',
              optimizer='adagrad',
              metrics=['accuracy'])

x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255

model.summary()

if not data_augmentation:
    print('Not using data augmentation.')
    model.fit(x_train, y_train,
            batch_size=batch_size,
            epochs=epochs,
            verbose=2,
            validation_data=(x_test, y_test),
            shuffle=True) # True
else:
    print('Using real-time data augmentation.')
    # This will do preprocessing and realtime data augmentation:
    datagen = ImageDataGenerator(

```

```

featurewise_center=False, # set input mean to 0 over the dataset
samplewise_center=False, # set each sample mean to 0
featurewise_std_normalization=False, # divide inputs by std of the dataset
samplewise_std_normalization=False, # divide each input by its std
zca_whitening=False, # apply ZCA whitening
rotation_range=0, # randomly rotate images in the range (degrees, 0 to 180)
width_shift_range=0.1, # randomly shift images horizontally (fraction of total width)
height_shift_range=0.1, # randomly shift images vertically (fraction of total height)
horizontal_flip=True, # randomly flip images
vertical_flip=False) # randomly flip images

# Compute quantities required for featurewise normalization
# (std, mean, and principal components if ZCA whitening is applied).
datagen.fit(x_train)

# Fit the model on the batches generated by datagen.flow().
model.fit_generator(datagen.flow(x_train, y_train,
                                batch_size=batch_size),
                    steps_per_epoch=x_train.shape[0] // batch_size,
                    epochs=epochs,
                    validation_data=(x_test, y_test))

```

D CIFAR-10 CNTK source code

```
# Copyright (c) Microsoft. All rights reserved.
# Modifications by Rasmus Airola and Kristoffer Hager.
# Licensed under the MIT license.
# =====

from __future__ import print_function
import numpy as np
import sys
import os
import cntk
import _cntk_py

# Paths relative to current python file.
abs_path = os.path.dirname(os.path.abspath(__file__))
data_path = os.path.join(abs_path, "CIFAR10")
model_path = os.path.join(abs_path, "Models")

# Define the reader for both training and evaluation action.
def create_reader(path, is_training, input_dim, label_dim):
    return cntk.io.MinibatchSource(cntk.io.CTFDeserializer(path, cntk.io.StreamDefs(
        features = cntk.io.StreamDef(field='features', shape=input_dim),
        labels = cntk.io.StreamDef(field='labels', shape=label_dim)
    )), randomize=is_training, epoch_size = cntk.io.INFINITELY_REPEAT if is_training else
        cntk.io.FULL_DATA_SWEEP)

# Creates and trains a feedforward classification model for MNIST images
def convnet_cifar10(debug_output=False):
    _cntk_py.set_computation_network_trace_level(0)

    image_height = 32
    image_width = 32
    num_channels = 3
    input_dim = image_height * image_width * num_channels
    num_output_classes = 10

    # Input variables denoting the features and label data
    input_var = cntk.ops.input_variable((num_channels, image_height, image_width), np.
        float32)
```



```

label_var = cntk.ops.input_variable(num_output_classes, np.float32)

# Instantiate the feedforward classification model
input_removemean = cntk.ops.minus(input_var, cntk.ops.constant(128))
scaled_input = cntk.ops.element_times(cntk.ops.constant(0.00392156), input_removemean) #
    0.00390625 =>
    0.00392156

with cntk.layers.default_options(activation=cntk.ops.relu, pad=False):
    z = cntk.models.Sequential([
        cntk.layers.Convolution2D((3,3), 32, pad=True),
        cntk.layers.Convolution2D((3,3), 32),
        cntk.layers.MaxPooling((2,2), (2,2)),
        cntk.layers.Dropout(0.25),

        cntk.layers.Convolution2D((3,3), 64, pad=True),
        cntk.layers.Convolution2D((3,3), 64),
        cntk.layers.MaxPooling((2,2), (2,2)),
        cntk.layers.Dropout(0.25),

        cntk.layers.Dense(512),
        cntk.layers.Dropout(0.5),
        cntk.layers.Dense(num_output_classes, activation=None)
    ])(scaled_input)

ce = cntk.ops.cross_entropy_with_softmax(z, label_var)
pe = cntk.ops.classification_error(z, label_var)

reader_train = create_reader(os.path.join(data_path, 'Train_cntk_text.txt'), True,
    input_dim, num_output_classes)

# training config
epoch_size = 50000 # for now we manually specify epoch size
minibatch_size = 32 # 64->32

# Set learning parameters
lr_per_sample = [0.01]
lr_schedule = cntk.learning_rate_schedule(lr_per_sample, cntk.learner.
    UnitType.sample, epoch_size)

# Instantiate the trainer object to drive the model training
learner = cntk.learner.adagrad(z.parameters, lr_schedule)

```

```

trainer = cntk.Trainer(z, (ce, pe), learner)

# define mapping from reader streams to network inputs
input_map = {
    input_var : reader_train.streams.features,
    label_var : reader_train.streams.labels
}

max_epochs = 40
cntk.utils.log_number_of_parameters(z) ; print()
progress_printer = cntk.utils.ProgressPrinter(tag='Training', log_to_file='log.txt',
    num_epochs=max_epochs)

# Get minibatches of images to train with and perform model training
for epoch in range(max_epochs):      # loop over epochs
    sample_count = 0
    while sample_count < epoch_size: # loop over minibatches in the epoch
        data = reader_train.next_minibatch(min(minibatch_size, epoch_size - sample_count
            ), input_map=input_map) # fetch
            minibatch.
        trainer.train_minibatch(data) # update model with
            it
        sample_count += trainer.previous_minibatch_sample_count # count samples
            processed so far
        progress_printer.update_with_trainer(trainer, with_metric=True) # log progress

    progress_printer.epoch_summary(with_metric=True)
    z.save_model(os.path.join(model_path, "ConvNet_CIFAR10_{}.dnn".format(epoch)))

# Load test data
reader_test = create_reader(os.path.join(data_path, 'Test_cntk_text.txt'), False,
    input_dim, num_output_classes)

input_map = {
    input_var : reader_test.streams.features,
    label_var : reader_test.streams.labels
}

# Test data for trained model
epoch_size = 10000
minibatch_size = 32 # 16->32

```

```

# process minibatches and evaluate the model
metric_numer    = 0
metric_denom    = 0
sample_count    = 0
minibatch_index = 0

while sample_count < epoch_size:
    current_minibatch = min(minibatch_size, epoch_size - sample_count)
    # Fetch next test min batch.
    data = reader_test.next_minibatch(current_minibatch, input_map=input_map)
    # minibatch data to be trained with
    metric_numer += trainer.test_minibatch(data) * current_minibatch
    metric_denom += current_minibatch
    # Keep track of the number of samples processed so far.
    sample_count += data[label_var].num_samples
    minibatch_index += 1

print("")
print("Final_Results:_Minibatch[1-{}]:_errs=_{:0.2f}%*_{}".format(minibatch_index+1, (
    metric_numer*100.0)/metric_denom, metric_denom))
print("")

return metric_numer/metric_denom

if __name__=='__main__':
    convnet_cifar10()

```

E CIFAR-100 Keras/TensorFlow source code

```
# Rasmus Airola and Kristoffer Hager, 2017.
# Inspired by the ResNet architecture.
# =====

from __future__ import print_function
from keras.layers import Conv2D, GlobalAveragePooling2D, Dropout, Flatten, add, MaxPool2D,
    Input, Dense, Activation, AveragePooling2D
from keras.layers.advanced_activations import ELU
from keras.datasets import cifar100
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Model
from keras.optimizers import Adam
from keras.utils import to_categorical
from keras.regularizers import l2
from keras import backend
from keras.callbacks import LearningRateScheduler
from matplotlib import pyplot

def residual_identity_block(input_tensor, kernel_size, filters, dropout=None):
    x = Conv2D(filters // 4, (1, 1), padding='same')(input_tensor)
    x = ELU()(x)
    x = Conv2D(filters // 4, kernel_size, padding='same')(x)
    x = ELU()(x)
    x = Conv2D(filters, (1, 1), padding='same')(x)
    x = add([x, input_tensor])
    output = ELU()(x)
    if dropout:
        output = Dropout(dropout)(output)
    return output

def residual_conv_block(input_tensor, kernel_size, filters, dropout=None):
    x = Conv2D(filters // 4, (1, 1), strides=2)(input_tensor)
    x = ELU()(x)
    x = Conv2D(filters // 4, kernel_size, padding='same')(x)
    x = ELU()(x)
    x = Conv2D(filters, (1, 1), padding='same')(x)
    shortcut = Conv2D(filters, (1, 1), strides=2)(input_tensor)
    x = add([x, shortcut])
    output = ELU()(x)
```

```

    if dropout:
        output = Dropout(dropout)(output)
    return output

def plot_accuracy(history, run):
    pyplot.plot([i * 100 for i in history.history['acc']])
    pyplot.plot([i * 100 for i in history.history['val_acc']])
    pyplot.title('Model_accuracy')
    pyplot.ylabel('Accuracy_(%)')
    pyplot.xlabel('Epoch')
    pyplot.legend(['Training', 'Test'], loc='upper_left')
    pyplot.savefig("accuracy_trial" + run + ".png")
    pyplot.close()

def plot_losses(history, run):
    pyplot.plot(history.history['loss'])
    pyplot.plot(history.history['val_loss'])
    pyplot.title('Model_loss')
    pyplot.ylabel('Loss')
    pyplot.xlabel('Epoch')
    pyplot.legend(['Training', 'Test'], loc='upper_left')
    pyplot.savefig("losses_trial" + run + ".png")
    pyplot.close()

def cifar100_model():
    # Parameter initialization
    num_classes = 100

    # Defining the model
    input = Input(shape=(32, 32, 3))

    x = Conv2D(64, (3, 3), padding='same')(input)
    x = ELU()(x)
    x = Conv2D(128, (3, 3), padding='same')(x)
    x = ELU()(x)

    x = residual_conv_block(x, (3, 3), 128)
    x = residual_identity_block(x, (3, 3), 128)
    x = residual_identity_block(x, (3, 3), 128, dropout=0.25)

    x = residual_conv_block(x, (3, 3), 256)
    x = residual_identity_block(x, (3, 3), 256)

```

```

x = residual_identity_block(x, (3, 3), 256, dropout=0.25)

x = residual_conv_block(x, (3, 3), 512)
x = residual_identity_block(x, (3, 3), 512)
x = residual_identity_block(x, (3, 3), 512, dropout=0.25)
x = GlobalAveragePooling2D()(x)

x = Dense(num_classes)(x)
x = Activation('softmax')(x)

model = Model(inputs=input, outputs=x)

return model

def print_model(model):
    model.summary()

def learning_rate_schedule(epoch):
    learning_rate = 0.001
    if epoch < 40:
        return learning_rate
    if epoch >= 40:
        return learning_rate * 0.1

def cifar100_keras(model):
    # Parameter initialization
    batch_size = 128
    num_classes = 100
    num_epochs = 200

    # Fetching the data
    (x_train, y_train), (x_test, y_test) = cifar100.load_data()
    print('x_train_shape:', x_train.shape)
    print(x_train.shape[0], 'train_samples')
    print(x_test.shape[0], 'test_samples')

    # Encode the labels properly
    y_train = to_categorical(y_train, num_classes)
    y_test = to_categorical(y_test, num_classes)

    # Compiling the model
    adam = Adam(lr=0.001)

```

```

model.compile(loss='categorical_crossentropy',
              optimizer=adam,
              metrics=['accuracy'])

# Preparing the data
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255

# This will do preprocessing and realtime data augmentation
print('Using real-time data augmentation.')
datagen = ImageDataGenerator(width_shift_range=0.1,
                             height_shift_range=0.1,
                             horizontal_flip=True)

datagen.fit(x_train)

# The training of the model
lr_schedule = LearningRateScheduler(learning_rate_schedule)
history = model.fit_generator(datagen.flow(x_train, y_train, batch_size=batch_size),
                             callbacks=[lr_schedule],
                             steps_per_epoch=x_train.shape[0] // batch_size,
                             epochs=num_epochs,
                             validation_data=(x_test, y_test),
                             verbose=2)

return history

if __name__ == '__main__':
    print_model(cifar100_model())
    for i in range(3):
        history = cifar100_keras(cifar100_model())
        plot_accuracy(history, str(i + 1))
        plot_losses(history, str(i + 1))

```