TKP4555

Advanced Simulation

TensorFlow

Julie Rasmussen and Oliver Sale Haugberg

November 2017

Abstract

This reports can be viewed as an introduction tutorial to TensorFlow, where its main principles are explained and illustrated with different examples.

Contents

1 Introduction

TensorFlow is an open source multipurpose software library used for numerical computation using data flow graphs [4]. It is designed for machine learning applications using mathematical concepts such as neural networks. It is a library that it provides a vast number of functions for simple implementation of complex algorithms, such as automatic computation of model gradients or fully implemented model training algorithms.

TensorFlow, and its predecessor DistBelief, were developed by Google Brain, a research project at Google. The project has it's roots in using deep learning, a subfield of general machine learning to handle problems related to artificial intelligence. From this project DistBelief, a large-scale deep learning software system was made. As the number of applications of the DistBeliefs system grew within the Google conglomerate, TensorFlow was created as a faster and a more robust application-grade library. TensorFlow is also open source, which its predecessor was not, and has a flexible architecture that allows the same programs to work on a wide range of different hardware configurations. A single API, (Application Programming Interface), can be run on different numbers of CPUs, (Central Processing Units), and GPUs (Grapic Processing Units). Google have also recently announced the use and development of TPUs(Tensor processing units). These computer circuits are custom made for TensorFlow applications and vastly increases their computational effectiveness. At the moment TensorFlow applications can be written in both Python and C++, whereas Python has most supported features.

Even though TenorFlow was originally developed for internal use, it was in late 2015 released under the Apache 2.0 open source license. This licence allows users to freely use the library for any purpose, modify of the library, and even distribution of these modified versions. As a result of this TensorFlow has a large community with over 24,000 commits on github and over 18,000 posts tagged with 'tensorflow' on StackOverflow [5][6]. There are also a wide range companies that use TensorFlow, including CocaCola, AirBnb and Airbus.

The main use of TensorFlow is the training and optimization of Artificial Neural Networks. These networks are mathematical models that have shown excellent performance in patterns recognition problems, and have been implemented for use in a wide range of areas. Within Google several of their products are already using TensorFlow to improve performance. For example Google Translate can now, in real time, translate visual images of text to another language for the user [1], By using more complicated neural networks, automatic generation of email responses can be made [2]. Google search has also started using Rankbrain, an algorithm developed with TensorFlow for improving search results. Outside Google there has also been research into how these neural network models can be used for drug discovery [3].

This report is written as an introductory tutorial for how models can be made and trained in TensorFlow. The examples are written in Python where TensorFlow is imported as a library. As TensorFlow was inherently designed for experienced programmers many concepts might, unfortunately, seem complex and counter intuitive. Therefore this report will start with a brief explanation in section 2 and 3 of TensorFlows main applications; Machine Learning and Artificial Neural Networks. Section 4 presents the main concepts and terms in TensorFlow. In section 4.7 these concepts are used to create a small program solving a problem with linear regression, before a more complicated problem is presented and solved in Section 5. Lastly, the solution along with some TensorFlows pros and cons will be discussed, before the report is concluded in section 6.

2 Machine Learning

Machine learning is a field of computer science that gives the computer the ability to learn without being explicitly programmed. It makes it possible to solve problems without much insight in to the particular problem and to solve input/output responses where it is hard to define an explicit algorithm. The drawback is that you often need a lot test/training data and a lot of computation time. But because of the explosive growth in computer clock-speed and computer memory in recent years, the performance of machine learning based programs has increased rapidly [8].
Machine learning can be categorized in three main tasks:

1. **Supervised learning:** The program is provided with example inputs and their desired outputs, and the goal is to develop a general rule that maps the input with the output. In TensorFlow this is called the training data. A typical example of this is the mail spam detector that sorts spam mails form other mails. This training continues when you mark mail as spam or not spam.
2. **Unsupervised learning:** When the program receives input but not the desired output, and the goal is to find the pattern itself. For example if you have a set of photos of 6 people, but with no information about who is on which one. The task is then to divide this data-set into 6 piles, each with photos of one individual.
3. **Reinforcement learning:** A program interacts with a dynamic environment in which it must achieve a certain goal. It learns by punishments and rewards. For example train an model to play tetris by only giving it its main goal which is to get a higher score.

In this report we only show examples of supervised learning.

3 Artificial Neural Networks

An Artificial neural network (ANN) is a mathematical model that is inspired by biological neural networks. In a biological neural network, neurons receive, react to, and transmit electrical signals depending on external stimuli. Depending on the received stimuli it will then turn on and send out an output signal. These neurons will then themselves send signals to other neurons forming a network. This behaviour is basis for how an ANN are modelled.
Because of the wide range of applications for ANN's there are several different types or structures that can be used. Not only can the topology of the network vary greatly, but also how information or data flow is processed. This chapter starts in section 3.1 by introducing the theory behind ANN architecture and general setup. Section 3.2 explain how the information flow is structured in a ANN, and how the structure is mathematical represented. Further, this theory is used to show how they can be trained and optimised in section 3.3.

3.1 Network Architecture

To understand how ANN's work it is important to have a good understanding of what components they are made of, and what these components actually do. They turn out to be quite simple when looked at individually, and the complexity of neural networks instead comes from the scale of the model. The networks are made up of neurons, which can be thought of as nodes, and edges, or connections, between these nodes. The easiest way to think of these nodes are as "something that stores a number". The edges between them also hold a value, which represents the strength of the connection.

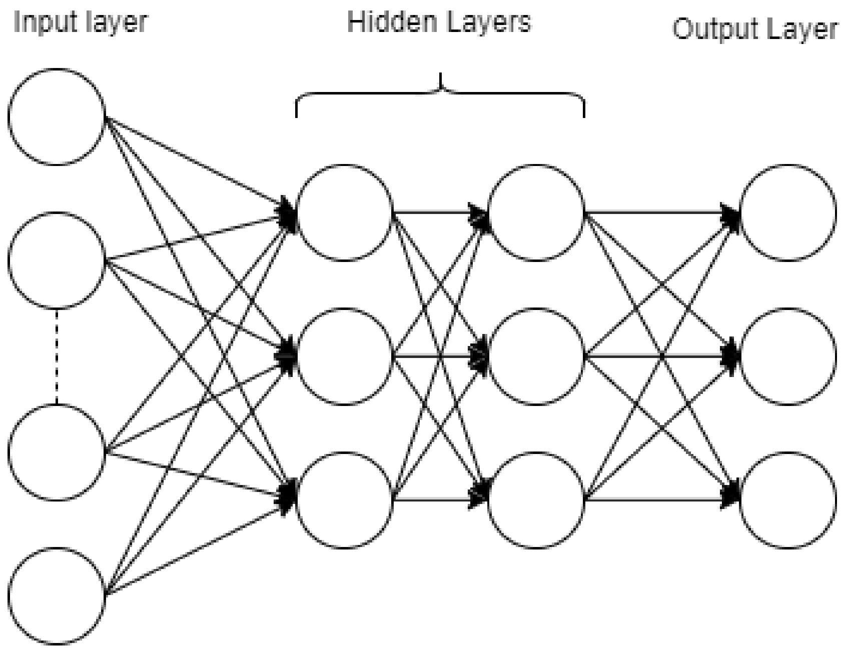


Figure 1: Example of how a neural network is structured

As shown in figure 1 the nodes are sorted into different layers, each with their own number of nodes. In this network all nodes have connections to all nodes in the next layer, which is called a dense layer. The number of nodes in each layer, and the number of layers will off course depend on the problem to be solved. The input layer will have one node for each input, and the output layer have one for each output. All layers between are known as the hidden layers. The chosen structure of the hidden layers will depend on the problem to be solved and what type of neural network is being made. More complicated neural networks can have a large number of hidden layers, they can have different layer types, and different connection patterns.

3.2 Forward Propagation and Node Activation

To build a quick and efficient mathematical model of what is going on in the network, matrices are used. In figure 2 the value for the node is represented by a_n^1, which would be the n'th node in layer 1.

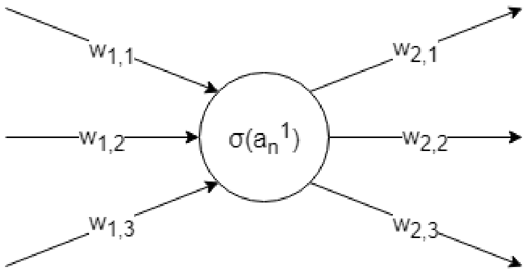


Figure 2

The input of this node will be the sum of all previous node times their connection strength, or their weight $w_{i,j}$.

$$a_n^1 = w_{1,1}a_1^0 + w_{1,2}a_2^0 + w_{1,3}a_3^0 + ... + w_{1,n}a_n^0 \tag{1}$$

The value stored in the node, and therefore the value that is send to its edges is then $\sigma(a_n^1)$. σ is here the activation function of the node.

This function is applied because the value of a_n^1 could be anything, so the data can quickly become quite unstructured. Because of this it is normal to use an activation function on every node. The activation function will define what the output of the node will be, given the input. Various of activation functions can be used, depending on the problem. Often we want the range of each node value to be (0,1), and to do this the sigmoid activation function can be used. This function is expressed as

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{2}$$

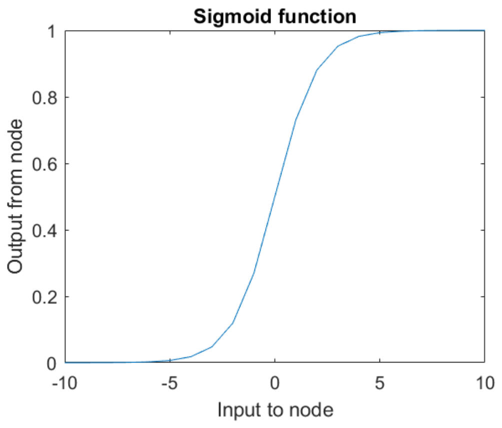


Figure 3: Sigmoid function

As the figure shows, higher positive values will return 1, and low negatives return 0. This gives the node a turned on or turned off behaviour. If only the input from the previous layers is given, the functions "switching point" will be around 0. To shift this point a bias can be subtracted from the input. This bias could shift the turning point to say 10, resulting in only higher inputs turning on the node. We can then rewrite equation 1 with the activation function, and the bias included.

$$a_n^1 = \sigma(w_{1,1}a_1^0 + w_{1,2}a_2^0 + w_{1,3}a_3^0 + ... + w_{1,n}a_n^0 - b_1) \tag{3}$$

It can now be seen that the output value of node a_n^1 will be "turned on" or set to 1 if the weighted sum of the inputs is above the bias value. Now matrices can be used to get a function for every node output, as a function of node input in a layer.

$$\begin{bmatrix} a_1^2 \\ a_2^2 \\ a_3^2 \end{bmatrix} = \sigma\left(\begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} \\ w_{2,1} & w_{2,2} & w_{2,3} \\ w_{3,1} & w_{3,2} & w_{3,3} \end{bmatrix} \begin{bmatrix} a_1^1 \\ a_2^1 \\ a_3^1 \end{bmatrix} - \begin{bmatrix} b_1^1 \\ b_2^1 \\ b_3^1 \end{bmatrix} \right) \tag{4}$$

Which can be rewritten for a more general case as

$$a^{n+1} = \sigma(Wa^n - b^n) \tag{5}$$

3.3 Gradient Computation for Neural Networks

A general cost function for a neural network can look like this:

$$J = \frac{1}{2}\sum(y - \hat{y})^2 \tag{6}$$

where y is the desired model output and $\hat{y}$ is the estimate model output. The values of $\hat{y}$ are calculated using equation [5] for the final layer. The desired output is taken from our labelled training data.

The gradient of the cost function is a function of every weight and bias $\nabla J(\mathbf{W}, \mathbf{b})$. The number of weights and biases if off course determined by the structure of the network. So what we are looking for are all the partial derivatives of the cost function.

$$\nabla J = (\frac{\delta J}{\delta w_{1,1}}, \frac{\delta J}{\delta b_1}, ..., \frac{\delta J}{\delta w_{j,k}}, \frac{\delta J}{\delta b_j}) \tag{7}$$

Now these derivatives are in themselves not that hard to calculate for simple systems, but when there are many nodes in a network and especially when there are many layers, the exact formula for the derivatives becomes very large. This is because every node is a function of every node in a previous layer. In other words, because nodes are connected to a large number of edges, they will have an effect on each weight. But when using tools like TensorFlow the exact algorithms for calculating these derivatives are not required as TensorFlow already has built in algorithms. The functions find the path from each variable to the cost function, then they backtrack from the cost function to each variable. When backtracking the partial gradients are calculated along the path using the chain rule. Then when each partial gradient has been computed, the gradients for each variable can be computated.

4 TensorFlow Basics

4.1 Tensors

Tensors are the central unit of data in TensorFlow and the creators define tensors as a "typed multidimentional array" [9]. That is, an object to store data elements of different types; e.g. strings, integers floats etc. The rank of the tensor is its number of dimensions, and the shape is its size in each of its dimensions. Examples of tensors of different sizes with integer elements are shown below.

4	scalar, a rank 0 tensor with shape []
$[1, 2, 3]$	vector, a rank 1 tensor with shape [3]
$[[1, 2, 3], [7, 8, 9]]$	matrix, a rank 2 tensor with shape [2, 3]
$[[[1, 2, 3], [1, 2, 3]], [[4, 5, 6], [7, 8, 9]]]$	a rank 3 tensor with shape [2, 2, 3]

4.2 Computational Graph and Operations

TensorFlow programs can be divided into two sections:

1. Building the computational graph.
2. Running the computational graph.

A computational graph in TensorFlow is a series of TensorFlow operations, where each operation is denoted as a node in the graph. The example below shows how a simple computational graph can be implemented, with the nodes a, b and c.

```
import tensorflow as tf

# Bulid graph
a = tf.constant([[-1.0,-1.0,-1.0],[-1.0,-1.0,-1.0]])
b = tf.constant(1.0, shape=[3, 2]) # an other way of defining a
    tensor
c = tf.matmul(a,b)

# Print output
print(c)
```

This will produces the output

```
Tensor("MatMul:0", shape=(2, 2), dtype=float32)
```

tf.matmul is a built-in TensorFlow operation that multiplies its input matrices. Table 1 shows a list of some operations that are provided in the TensorFlow library. Some of them will be used in the examples in this report.

Table 1: Example of some TensorFlow operators

Operator types	**Examples**
Mathematical operations	add, sub, mul, div, exp, log, greater, square, less, equal, ...
Array operations	concat, slice, split, constant, rank, shape, shuffle, ...
Matrix operations	matmul, matrix_inverse, matrix_determinant, ...
Stateful operations	Variable, assign, assign_add, global_variables_initializer, local_variables_initializer, variables_initializer
Activation Functions	softmax, sigmoid, relu, convolution2D, maxpool, ...
Checkpointing operations	Save, restore
Reduction operators	reduce_mean, reduce_max, reduce_prod, reduce_all
Random tensor generators	random_normal,truncated_normal, random_uniform, random_crop, multinomial...

4.3 Sessions

The output in the previous section was not printing a numerical tensor as expected. A special remark with TensorFlow is that the graph has to be *evaluated*, in order to get a numerical value. **Session** is class for running TensorFlow operations. A Session object encapsulates the environment where operation objects is executed and the tensor objects are evaluated [12]. The following code example first build the graph, and then creates a Session object which uses the run method to evaluate the graph - i.e first build the graph and then run the graph.

```
import tensorflow as tf

# Bulid graph
a = tf.constant([[-1.0,-1.0,-1.0],[-1.0,-1.0,-1.0]])
b = tf.constant(1.0, shape=[3, 2]) # an other way of defining a
    tensor
c = tf.matmul(a,b)

# Create a session object
sess = tf.Session()

# Run the graph
print(sess.run(c))
```

This produce the output:

```
[[-3.  -3.]
 [-3.  -3.]]
```

It is also possible to use an **InteractiveSession**. The InteractiveSession object installs itself as the default session on construction and it is not needed to pass an explicit session object to evaluate it.

```
sess = tf.InteractiveSession()

# Evaluate the tensor c,
# can just use 'c.eval()' without passing 'sess'
print(c.eval())
```

This will give the exact same output

```
[[-3.  -3.]
 [-3.  -3.]]
```

4.4 TensorBoard

The computational graphs in TensorFlow can quickly become quite complicated. To get a better understanding on what's actually going on, the tool TensorBoard can be used to visualize the graph. The visualization is interactive, and the user can zoom, pan, expand or collapse different groups for a more detailed description. In order to be able to view the TensorBoard page the following line must be added to the end of your script.

```
writer = tf.summary.FileWriter('output_folder', sess.graph)
```

tf.summary provide a way to save and export information about the graph to TensorBoard. The FileWriter class writes the summary to a folder in the current directory, which in this case is called 'output_folder'. To then launch TensorBoard the following must be done:

- Run the command: **tensorboard –logdir=path/to/log-directory**
- In a web browser, navigate to: **localhost:6006**

Figure 4 shows the TensorBoard page for the code example in section 4.3.

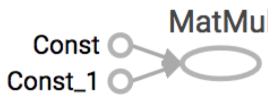


Figure 4: Screenshot showing how TensorBoard visualizes the graph.

To make the graph more readable in TensorBoard, it is possible to name the different operators in the graph. The a, b and c operators in the previous example can be named by inserting 'name = 'x'' in the operator declaration.

```
import tensorflow as tf
a = tf.constant(-1.0, shape=[2, 3], name = 'A')
b = tf.constant(1.0, shape=[3, 2], name = 'B')
c = tf.matmul(a,b, name = 'C')

sess = tf.Session()
print(sess.run(c))
writer = tf.summary.FileWriter("output", sess.graph)
```

This changed the look of the TensorBoard page to the following:

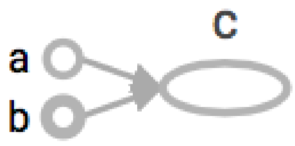
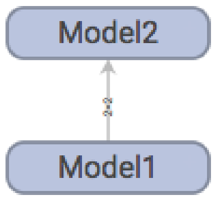


Figure 5: Screenshot showing how TensorBoard visualizes the named graph.

It is also possible to sort your graph into different name scopes in TensorBoard. This increases the readability even more, especially for more complicated graphs. One simple example:

```
import tensorflow as tf
with tf.name_scope('Model1'):
    a = tf.constant(-1.0, shape=[2, 3], name = 'A')
    b = tf.constant(1.0, shape=[3, 2], name = 'B')
    c = tf.matmul(a,b, name = 'C')
with tf.name_scope('Model2'):
    d = tf.matmul(c,c, name = 'D')
sess = tf.Session()
print(sess.run(d))
writer = tf.summary.FileWriter("output", sess.graph)
```



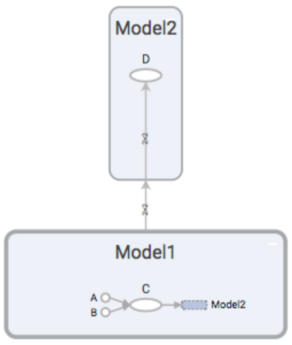


Figure 6: TensorBoard page

Figure 7: TensorBoard page when expand for the different scopes

4.5 Placeholders and Feed Dictionaries

In order to make graphs that can be parameterized to accept external inputs, the **placeholder** operation must be used [11]. The placeholder must be declared with a type, with options to declare its shape and name. If a parameter is declared with a placeholder it must be provided with values later, using the **feed_dict** optional argument to Session.run(). If the graph consist of placeholder, the run method must be provided with inputs in order to be evaluated. Example of implementation:

```
import tensorflow as tf
import numpy as np
# create placeholder
x = tf.placeholder(dtype=tf.float32)

# define session object in order to evaluate
sess = tf.Session()

# run and print placeholder
#print(sess.run(x)) # will fail since x is not provided with
    values

# make random numbers with numpy, 4X4 tensor
rand_array = np.random.rand(4,4)

print(sess.run(x,feed_dict={x: rand_array})) # will work
```

And produces the random output of a 4X4 matrix:

```
[[ 0.18460925  0.1135103   0.4232578   0.60328126]
 [ 0.46244735  0.2923753   0.37354666  0.79248434]
 [ 0.9640283   0.62143719  0.43418732  0.17993963]
 [ 0.15920405  0.9960075   0.81037658  0.19702186]]
```

4.6 Variables

Objects of the Variable class makes it possible to add trainable parameters to a graph [13]. The constructor, Variable(), must be provided with an initial value that define its shape and type. The type and shape of the variable is fixed after construction. By contrast to the constant nodes a and b, shown in the examples in section 4.2, 4.3, 4.4, variables needs to be initialised explicitly. TensorFlow has an operation called **global_variables_initializer()** that initials all variables in the program.

In the example code below, variable objects are used to fit a liner model to some data. We use the model:

$$y_est = W \cdot x + b$$

We want to find the parameters W and b, that maps the relationship between x and y. In order to evaluate a model, we must have a cost function that we want to minimize. In this example the sum of all square deltas is used as the cost function:

$$cost = \sum(y - y\_est)^2$$

The cost function tells how far the estimated output, y_est, is from the provided data, y. The following code show the implementation in using TensorFlow.

```
import tensorflow as tf
# define model
W = tf.Variable([1.5])
b = tf.Variable([-1.5])
x = tf.placeholder(tf.float32)
y_est = tf.add(tf.multiply(W,x),b)
```

```
# create session object
sess = tf.Session()

# initialize variables
init = tf.global_variables_initializer().run()

# define cost function
y = tf.placeholder(tf.float32)
square_deltas = tf.square(y_est-y)
cost_function = tf.reduce_sum(square_deltas)

# evaluate cost
print(sess.run(cost_function, feed_dict={x: [1,2,3,4],y:[0,1,2,3]
    }))
# print out the model estimation
print(sess.run(y_est,feed_dict={x: [1,2,3,4]}))
```

That gives the following output for the loss and the estimated output, *y_est*, respectively:

```
3.5
[ 0.   1.5  3.   4.5]
```

The initial values for W and b are used to calculate the the value for y_est, and the loss is quite high. This is because our model variables W and b are completely wrong. We have just given the model two random variables. In order to get a model with a better fit to the data, we must train the model. This is done by changing the variables to minimize the cost function.

4.7 Training and Optimization

In order to create a useful model in TensorFlow, we need training data that our model can be used on, to see what it is doing, and to check it's current performance. Using this data we can then "train" our model to perform better in regard to some cost function. The way our model changes and improves its performance during training, is by adjusting its variables. This is why constants, placeholders and variables are fundamentally quite different in TensorFlow. Their way TensorFlow organises and handles them are quite different. TensorFlow will find the model and its variables, and then adjust them to minimize the cost function. The variables are changed by a certain rate, called the learning rate, for a certain amount of iterations. These parameters, the learning rate and the number of training steps, are defined by the user.

The module **tf.train** provide a set of classes and functions that help train models. TensorFlow has several classes built in that define and implement different optimization algorithms, such as GradientDescentOptimizer, AdagradOptimizer, AdamOptimizer, FtrlOptimizer, ProximalGradientDescentOptimizer, ProximalAdagradOptimizer, ect. All these classes are sub classes of the base class **Optimizer**. The Optimizer class provides the gradient of the cost function with respect to the variables it depends on. The **GradientDecentOptimizer** is one of the simpler optimizers, as the algorithm is easier to use and less complex. In this report we only use this optimizer. The general procedure for the gradient decent optimizer is as follows:

1. It starts of with initial values for the variables, V_0.
2. The derivative of the cost function are calculated with respect to the variable,

$$\Delta = \frac{\partial cost}{\partial V_0}$$

   The derivative describes which way we need to change the variable in order to get a lower cost in the next iteration. In other words, it is the gradient of the cost function.
3. A learning rate parameter, α, must be specified in order to know how much the variable change for each training step. The variable for the next iteration is then given by:

$$V_1 = V_0 - (\alpha \cdot \Delta)$$

This process is repeated for a given number of steps.

The next example code below shows how GradientDecentOptimizer can be used to solve the linear regression problem, shown in section 4.6. It does this by trying to minimizes the cost function over 1000 iterations. In this example we use the same training date in every training step, as we only want our model to fit to these data points. In the example, the parameters are named and name scopes are defined for the different part of the program. This is done to make the TensorBoard page more readable.

```
import tensorflow as tf
# define model scope
with tf.name_scope('model'):
    W = tf.Variable([1.5], name = 'W')
    b = tf.Variable([-1.5], name = 'b')
    x = tf.placeholder(tf.float32, name = 'x')
    y_est = tf.add(tf.multiply(W,x),b,name = 'y_est')

# define a scope for cost function
with tf.name_scope('cost_func'):
    y = tf.placeholder(tf.float32, name = 'y')
    square_deltas = tf.square(y_est-y,name ='sqaredeltas')
    cost_function = tf.reduce_sum(square_deltas, name = 'cost')

# define scope for training
with tf.name_scope('training'):
    learning_rate = 0.01
    optimizer = tf.train.GradientDescentOptimizer(learning_rate,
   name = 'optimizer')
    train_step = optimizer.minimize(cost_function)

# create session object
sess = tf.InteractiveSession()
# initialize variables
init = tf.global_variables_initializer().run()

# training data
x_train = [1, 2, 3, 4]
y_train = [0, -1, -2, -3]

# training loop
for i in range(1000):
    sess.run(train_step, {x: x_train, y: y_train})

writer = tf.summary.FileWriter("folder_name", sess.graph)
```

Gives TensorBoad page:

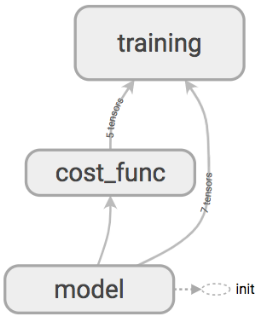


Figure 8: TensorBoard page

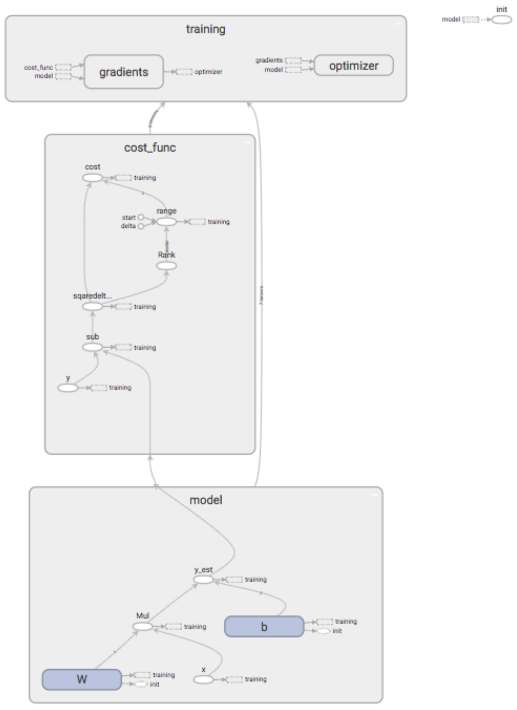


Figure 9: Tensorboard page when zoomed in for the different scopes

Figure 10 shows that the Variable objects, W and b, are changed for each iteration and that the cost is minimized. The variables initial value and the learning rate will have an effect on how fast the model improved performance, and is shown in Figure 11 and 12.

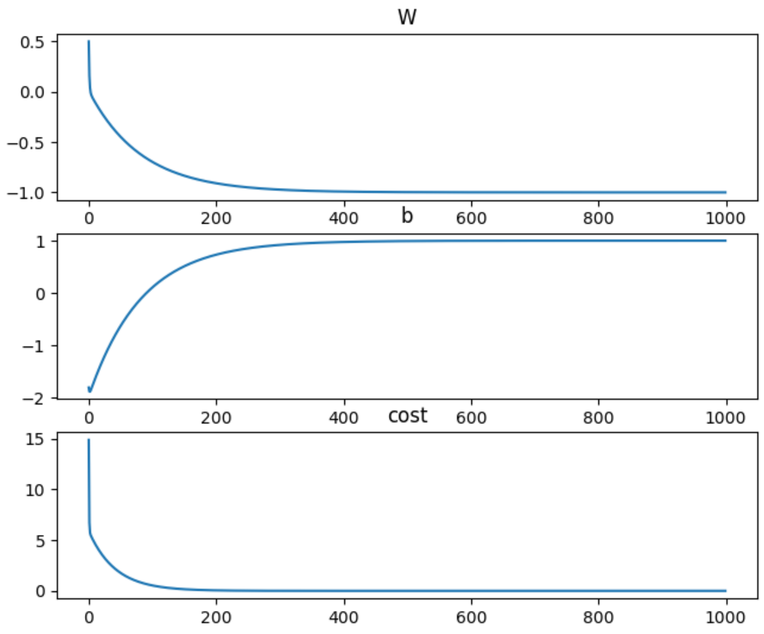


Figure 10: Plot shows how the variables, W and b, and the cost function changes for every iteration in the training loop

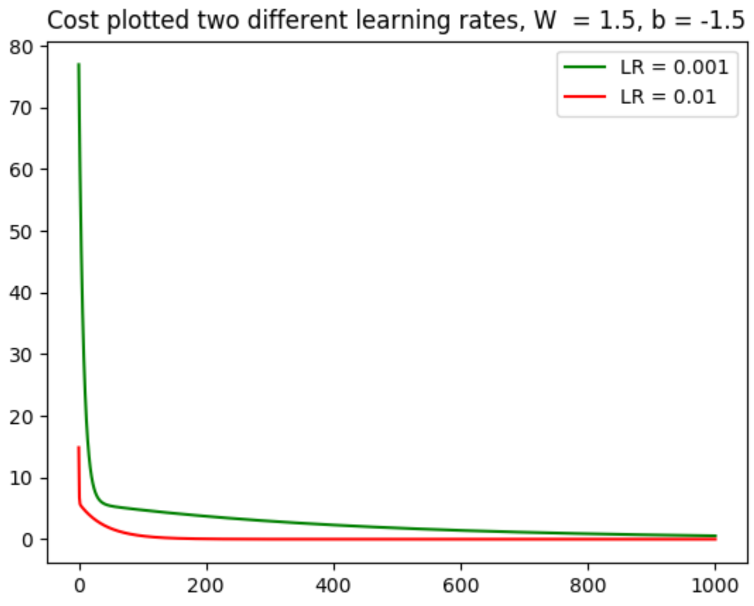


Figure 11: Plot shows how the cost function get reduced for every iteration for two different learning rates.

Figure 11 shows that a higher learning rate converges faster. It were also tested for learning rates higher than 0.01, but they did not converge at all.

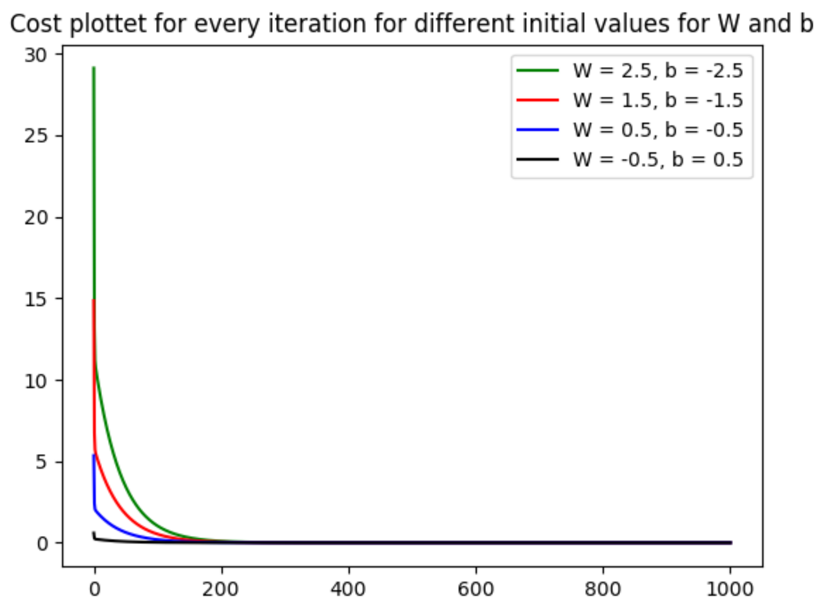


Figure 12: Plot show the cost for every iteration with different initial values for W and b.

Figure 12 shows that a initial value closer to the desired parameters, W = -1 , b = 1, converges faster.

Even though this is an easy example, it gives some insight into how TensorFlow programs are structured. The basic building blocks that are required in TensorFlow programs have been introduced. These will be used in the next section on a more complicated problem.

5 Image Recognition Example Using TensorFlow

One of the major issues when dealing with machine learning models is the need for large sets of training data. Not only is a large amount of data needed, but a large set of labeled and structured data. Because of this in this section by using TensorFlow, a detailed explanation will be given on how to create a model to accurately recognise handwritten digits. In section 5.2 a detailed explanation is given on a set of example code. The entire code can also be found in the appendix.

5.1 The MNIST Database

The MNIST database is a collection of handwritten digits that are commonly used to test different models for image recognition. The main benefits of the database are its size and simplicity of its structure. There are 60,000 training images and 10,000 testing images. Each image in the set is a 28x28 pixel image with grey-scale values for each pixel. Instead of working with a 28x28 matrix, these values are flattened into a 784 long vector. The labels for each image represent what the correct digit is is displaying. These labels are stored as ”one-hot vectors”, this means that a label indicating 4 as the wright digit would be represented as [0,0,0,0,1,0,0,0,0,0]. This notation will make it easier to mathematically check for accuracy.

5.2 Example Code

After importing TensorFlow and downloading the mnist dataset the structure of the model is defined.

```
import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
```

In this basic example there are three layers in the artificial neural network. There is the input layer, which will be one node for each pixel, the hidden layer which can be any number of nodes, and the output layer. The output layer is 10 nodes, one for each digit.

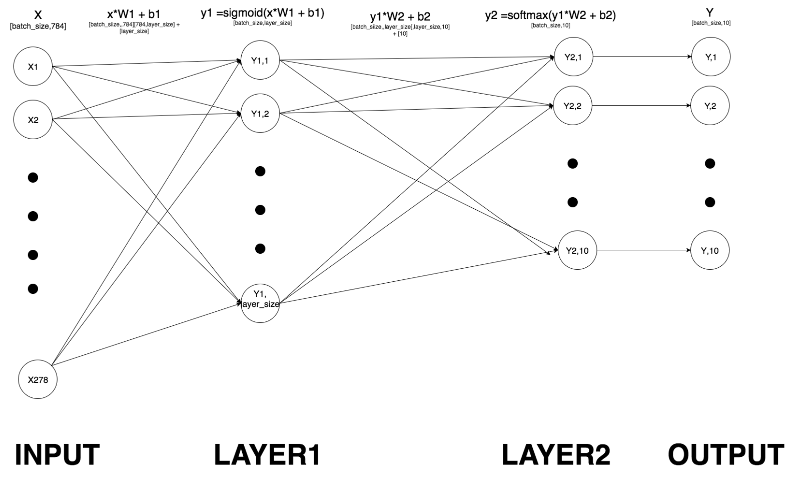


Figure 13: Vizualization of the ANN structure

The idea behind the model is, when run on a sample image, we want the output node corresponding the correct digit to have the highest value. This will create a probability distribution, where the highest number is the best guess. For example when run on an image showing 0, we want the first output node to be 1, and all others to be 0. The model will off course never have a perfect guess, but it should return a significantly higher number in the first node.

Next the structure of the graph is defined. These operations are named and defined under a common name scope called "model". When displayed in TensorBoard, all operations are located together to simplify the structure.

```
with tf.name_scope('model'):
    #input
    x = tf.placeholder(tf.float32, [None, 784],name = 'x')

    #layer 1
    layer_size = 200
    W1 = tf.Variable(tf.random_normal([784, layer_size]),name = '
W1')
    b1 = tf.Variable(tf.random_normal( [layer_size]),name = 'b1')
    y1 = tf.nn.sigmoid(tf.matmul(x, W1) + b1,name = 'y1')

    #layer 2
    W2 = tf.Variable(tf.random_normal([layer_size,10]),name = 'W2
')
    b2 = tf.Variable(tf.random_normal([10]),name = 'b2')
    y2 = tf.nn.softmax(tf.matmul(y1, W2) + b2,name = 'y2')

    #output
    y = y2
```

In the dimension of of the x operation [None, 784], none is written because when declaring it's values we want to be able to modify the number of images. None declarers that any number is valid. This will be used to vary the batch sizes later when training the model. For the first set of weights and biases the new dimension is 200. This is then the number of nodes in the hidden layer. For the second set there are only 10, as this is the number of nodes in the output layer.

The variables are initialized with the $tf.random\_normal$ operation, for a given shape. It returns a tensor of a specified shape with normal distributed values. In our example we use the default values of standard deviation = 1.0 and mean = 0.0. TensorFlow provides several other types of operations that produce different tensors often used for initialization from random values[15].

We are using the sigmoid activation from function 2 in the first hidden layer, and an activation function called softmax in the secound layer.

$$\sigma(\mathbf{x})_i = \frac{exp(x_i)}{\sum_i^N exp(x_i)} \quad for\ i = 1, ..., N \tag{8}$$

The softmax activation function is often used in ANN classification problems. These problems have the advantage that the desiered outcome, the one hot vector in this case, are mutually exclusive. The softmax activation function works in a similar way to the sigmoind, with the difference being that the sum of all the node values will be 1. This works out great, as it then is on the same format as our one hot vector.

From TensorBoard we can now get a check that the graph of the model section is correct.

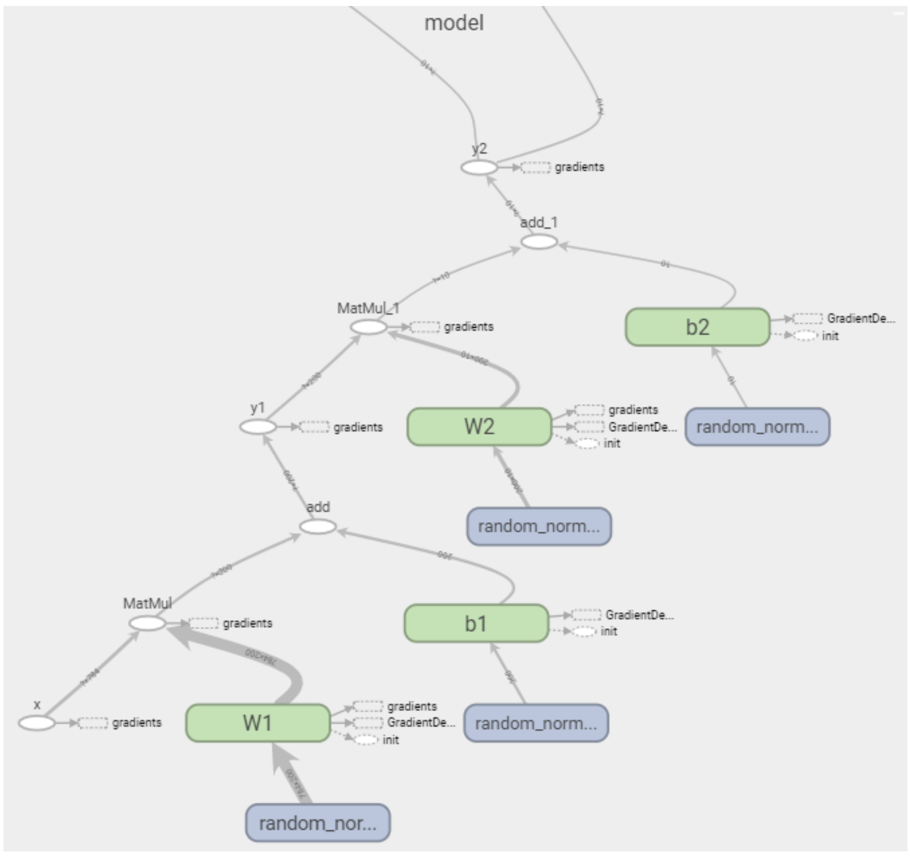


Figure 14: Model section of the graph

This is actually the entire model, and if the weights and biases where known we could simply put them to their right values and start using the model to classify images. But because we have initialised them with random values we need to adjust them using the training data. To do this we need to first define the cost function for our model. In this example we will use the cross entropy function. Which is defined as

$$H_y(y_{est}) = -\sum_i y_i log(y_{est}) \tag{9}$$

The cross entropy function is used instead of mean error, because in our model the output is a probability distribution. Cross entropy measures the difference between a estimated probability distribution and the true one. Next the train step operation is defined. This is where the benefits of using Tensorflow really are displayed. The entire algorithm for calculating the gradient of the cost function, and adjusting all variables is done in one line. All operations needed to do this in our graph are then put in by TensorFlow. In this example, the operation train_step uses gradient decent to minimize the cross_entropy function. The train_step operation use the graph to find the variables connected to cross_entropy, then it can use the algortihms for adjusting them.

```
y_ = tf.placeholder(tf.float32, [None, 10])

with tf.name_scope('Training'):
    cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y),
    reduction_indices=[1]),name ='Cross_entropy')
    learning_rate = 1
    train_step = tf.train.GradientDescentOptimizer(learning_rate,
   name ='Train_step').minimize(cross_entropy)
```

The .minimize() function simply combines two other functions of GradientDecentOptimizer namely compute_gradients() and apply_gradients().

The accuracy of the model is calculated by the next two operators, *correct_prediction* and *accuracy*.

```
with tf.name_scope('Accuracy'):

    correct_prediction = tf.equal(tf.argmax(y,1), tf.argmax(y_,1)
   ,name ='Correct_prediction')
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.
   float32),name ='Accuracy')

sess = tf.InteractiveSession() # create session object
tf.global_variables_initializer().run() # initialize variables
tf.summary.FileWriter("graph", sess.graph) # TensorBoard
    visualization
```

correct_prediction first uses argmax to return the index of the highest number. Remember that for the correct answers y_, the index of the correct digit is 1, with all others 0. The model y, will have the highest value in the index it thinks is the right digit. The equal function will then return a list of booleans, with True for a correct guess and false for a wrong guess. Next the accuracy operation takes this list, turns the booleans into integers 1 and 0, then calculated the average. This value is then the accuracy of the model.

Next the actual training of the model is done. Here we are doing 10000 training steps with each of them containing a batch of 100 random samples. Here it is important to understand how TensorFlow works for the code to make proper sense.

Because the placeholders x and y_, are connected to 'train_step' trough the cost function, we need to supply the sess.run method with a feed_dict. The feed_dict is a batch of 100 randomly sampled training images and their corresponding one hot vectors. In every iteration all variables in the graph, all weights and biases are slightly adjusted to minimise our cost function. In the code bellow the accuracy is calculated at the end, when the all the training steps are done.

```
batch_size = 100
for i in range(10000):
    batch_xs, batch_ys = mnist.train.next_batch(batch_size)
    sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys})

print(sess.run(accuracy, feed_dict={x: mnist.test.images, y_:
      mnist.test.labels}))
```

Because name_scope has been used to order our operations, it is now easier to look into how the graph is structured, and thereby understand the program better.

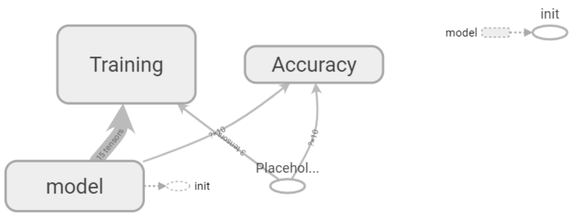


Figure 15: The graph

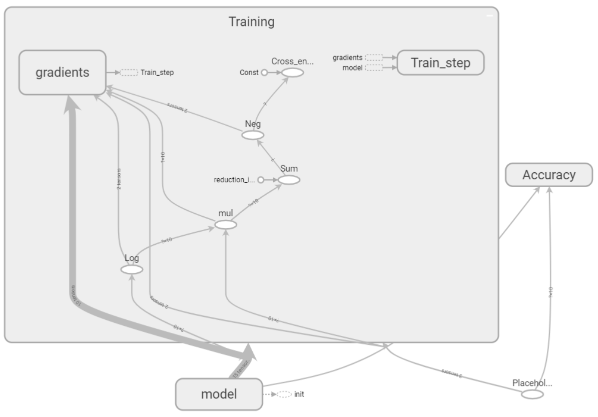


Figure 16: Training section of the graph

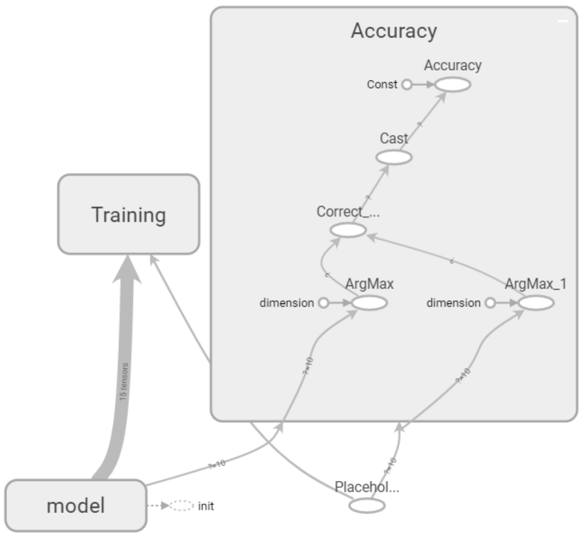


Figure 17: Accuracy section of the graph

5.3 Analysing The Model

In this section the effect different parameters have on the training and optimisation of the model will be shown. Hopefully it should give a better understanding of what these parameters actually do.

The three parameters that where changed where the learning rate given to the gradient decent optimiser, the training data batch size, the number of nodes in the hidden layer, and the number of hidden layers. If not otherwise specified, the default values used where: Learning rate: 1, Batch size: 100, number of nodes: 200, and 1 hidden layer. Note that for all the following figures in this section, the y-axis represents the model performance, 0.9 representing 90% accuracy at correct recognition. The x-axis is training steps, or iterations.

5.3.1 Learning Rate

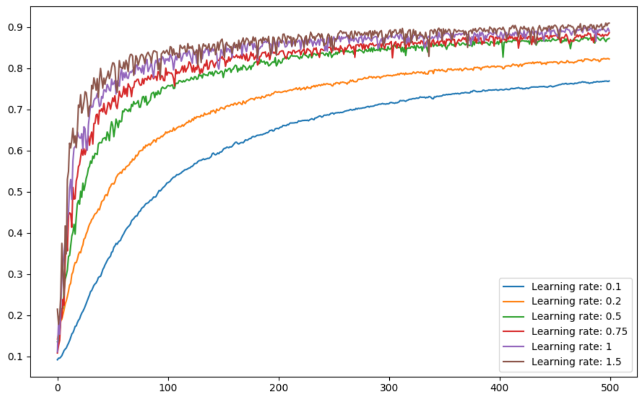


Figure 18: Learning rate for first 500 steps

Figure 18 shows how the learning rate impacts the model for the first 500 training steps. It can clearly be seen that for higher learning rates, the model accuracy is more quickly tuned to a higher values at the cost of some noise on the way. If instead looking at the first 10 000 training steps, the accuracy is as follows:

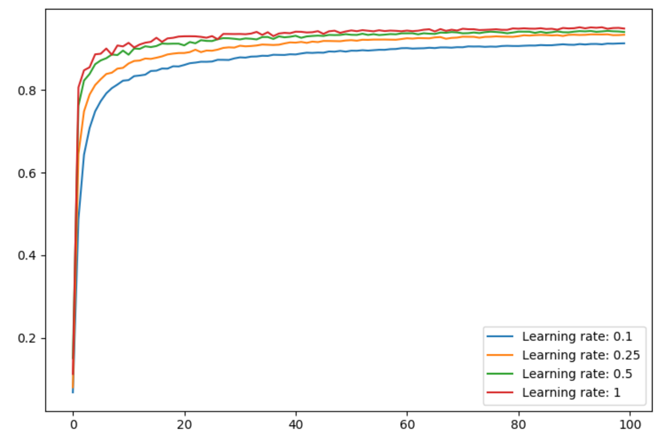


Figure 19: Learning rate for 10 000 steps, plotting every 100th.

This figure was quite surprising, as the expected behaviour was that lower learning rates would eventually converge to a higher accuracy, but this does not seem to be the case. The optimal rate seems to be 1, as anything above this could result in divergence.

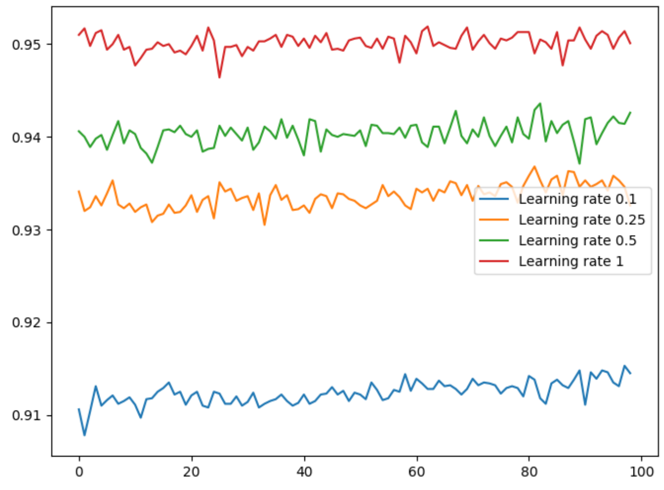


Figure 20: Learning rate for the final 100 steps

Looking at the final 100 steps (Figure 20), the final accuracy levels can be seen. It is also quite surprising that the noise seems to hardly be any higher for the higher learning rate. One cause of this could be that at this level, because the model is close to the desired outcome, the gradients are very small that the learning rate is no longer the determining factor in the noise.

5.3.2 Number of Nodes in Hidden Layer

The number of nodes in the hidden layer is looked at next. Unlike the input and output layer, the hidden layer (or layers) has no real physical meaning it can be quite hard to determine how it should be structured.

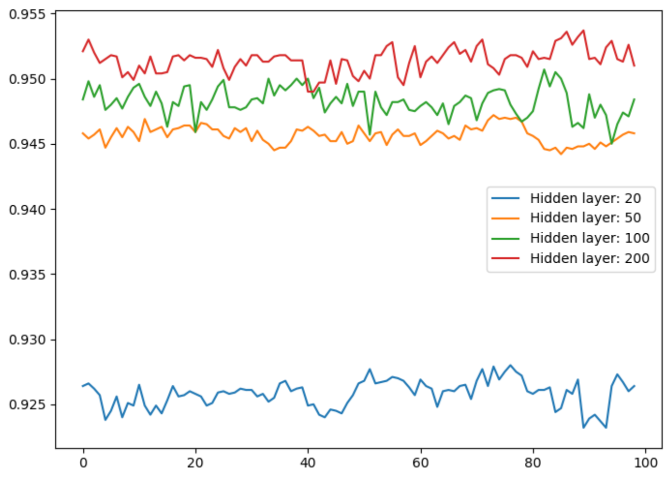


Figure 21: Different layer sizes for the final 100 steps

Going from 20 to 200 nodes there is a clear improvement at the end of the training. But now the catch is that the number of weights and biases are significantly higher. This results in longer computational time for optimisation. For this model the running time is not that much of a worry as is is generally quite low. But for more complicated models that use extremely large training sets, computation time becomes very important.
Increasing the number beyond 200 did not result in any better performance

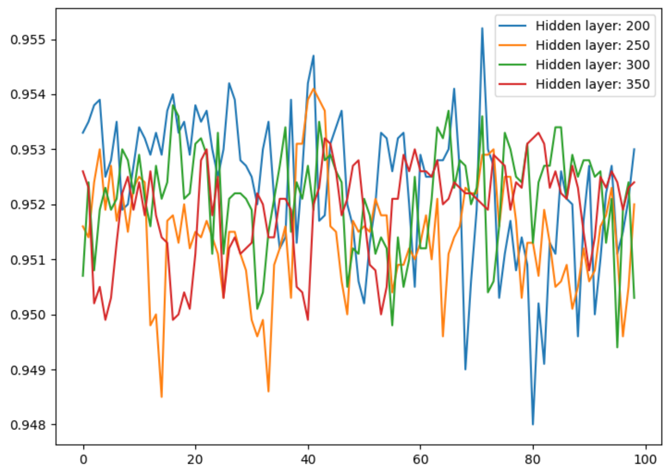


Figure 22: Different layer sizes for the last 100 steps

5.3.3 Batch Size

As previously explained, the model uses gradient decent to tune the model. It is an example of a stochastic gradient decent optimizer, using only a batch of training images, or an approximation of the true full data set. The effect varying the size of this batch is here shown.

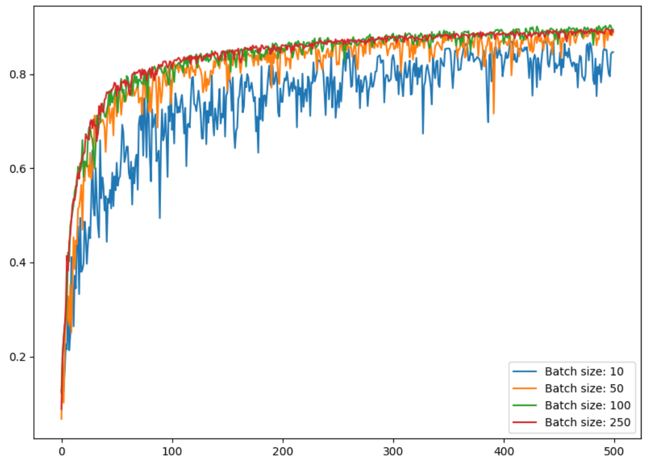


Figure 23: Batch size for first 500 steps

Clearly using a to small batch sizes is not good for training. Because the optimiser can only tunes in the direction of the minimum for the current batch, it can clearly go in the wrong direction when the batch is to small. This is very clear from the plot for a batch size of 10.
For higher batch sizes the accuracy becomes better, but the drawback here is again that the computation time gets longer. Using a batch size of 100 compared to a batch size of 200 gave an increase in accuracy of 0.1%. But the computation time was 50% longer. A too large batch size can also result in 'over-fitting'. If you take in the whole mnist training set of 60,000 images for every iteration. The model would give an almost perfect accuracy for those images. But since its been adjusted over and over again for the same images, it might cause problems when it's tested for other images outside the training batch.

5.3.4 Different Numbers of Hidden Layers

Because of the "black box" nature of the hidden layers it is not always intuitive how one should set up the neural network. In this part the model was extended with extra hidden layers and the accuracy's after training was plotted for models using 0 to 4 hidden layers. The code for the generation of this plot can be found in the appendix. This model was structured with 200 nodes in the first layer, 100 in the second, 60 in the third and 30 in the fourth layer.

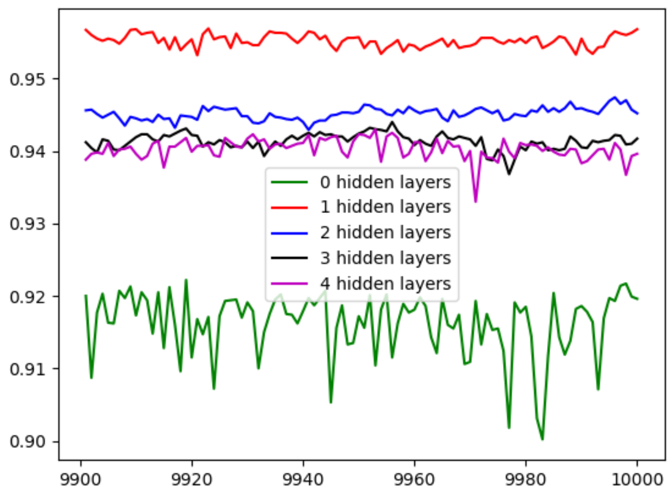


Figure 24: Accuracy for models with 5 different number of layers, shown for the last 100 training steps.

Number of layers	0	1	2	3	4
Accuracy [%]	91.22	95.34	94.09	94.22	94.16

Table 2: Final accuracy for 10,000 iteration for different number of hidden layers.

As seen in figure 24 and table 2 the accuracy do not increase with a higher numbers of hidden layer in this example. With zero hidden layers the accuracy is at its lowest, and its fluctuating quite aggressively. The model with one hidden layer gave the best result.

At first glance, this is not the result one might expect. One might think that more layers would give a better accuracy, but here there is instead a decrease in performance. One explanation for this might be that more layers "over complicates" the model. At its core, the model is only trying to sort images into 10 different boxes. Adding more layers increases the number of weights and biases significantly, which probably are not needed. For each new layer added, the information must also flow through yet another activation function. This might acts as a filter simply removing information. The model is probably filtering out important information to classify images as different. Computational time for the model will also increase significantly with each added layer.
So the best accuracy for this model was found with one hidden layer, 200 nodes in the hidden layer, a learning rate of 1.0, and a batch size of 100.

6 Conclusion

Hopefully this report has given an insight into what TensorFlow is and what it can be used for. An in depth tutorial of all the areas of TensorFlow is out of the scope for this report, anyhow the report presents the main concepts behind the software library and it's building blocks.

Working with TensorFlow for some months we where quite impressed what the library could do. It is possible to very quickly develop complex models that solve quite complicated tasks, using only a few lines of code. The large amount of build in functions and tools allow the user to skip a large part of the groundwork usually required for using machine learning to solve problems. Such as data structures and the development of algorithms.

Since ML and ANNs are used to solve problems that can be difficult to express mathematically. This makes it difficult to define how your input map to your expected output, that is, defining the model. TensorFlow is inherently structured around neural networks, and expressing a problem correctly in neural network terms is often the hardest part. But when this has been done, and a clear understanding of what the user wants the neural network to do TensorFlow is extremely powerful.TensorFlow make the problems easier to solve by using some of its "blackbox" functionality. The downside to this is that it also distances the programmer to actually understand the whats going on in the network, and this could make it harder to improve the performance of the model. Some of the main challenges with ANNs is to know how to structure the network in terms of activation functions, layer sizes, connections between the layers and number of layers. Our experience with this so far, is that a lot of trial and error is involved to be able to build a usable model.

TensorFlow was originally developed for internal use within Google. Because of this it might be hard for non experienced programmers to use it to its full potential. For example one of the main benefits with TensorFlow is its ability to fully utilise different hardware for its capabilities, such as multiple CPUs, GPUs and even TPUs. This side of TensorFlow has not been explored in this report at all, as the examples given have not been limited by computational time.

On the other hand as TensorFlow is very new, and in constant development it also has a large and active community. This makes it possible for inexperienced users to quickly find help, and examples from more experienced users. The library is also well documented on the TensorFlow homepage, where there also are multiple examples and tutorials.

Our advice if someone wants to start using TensorFlow to solve problems is to first make sure it actually is the right tool for the problem. Supervised machine learning, which everything in this report, is an example of, requires large amounts of training and testing data. We also suggest that before jumping right into TensorFlow one should have a good understanding of the concepts related to Artificial Neural Networks and optimization in general. This is because they often are at the core of TensorFlow programs.

References

[1] https://research.googleblog.com/2015/07/how-google-translate-squeezes-deep.html Accessed: 18.11.2017

[2] https://research.googleblog.com/2015/11/computer-respond-to-this-email.html Accessed: 18.11.2017

[3] Massively multitask networks for drug discovery B Ramsundar, S Kearnes, P Riley, D Webster, D Konerding, V Pande arXiv preprint arXiv:1502.02072

[4] https://en.wikipedia.org/wiki/TensorFlow

[5] https://github.com/tensorflow/tensorflow

[6] https://stackoverflow.com/tags/tensorflow/info

[7] https://www.coursera.org/learn/machine-learning

[8] https://en.wikipedia.org/wiki/Machine_learning

[9] http://download.tensorflow.org/paper/whitepaper2015.pdf

[10] https://www.tensorflow.org/

[11] https://www.tensorflow.org/api_docs/python/tf/placeholder

[12] https://www.tensorflow.org/api_docs/python/tf/Session

[13] https://www.tensorflow.org/api_docs/python/tf/Variable

[14] http://yann.lecun.com/exdb/mnist/

[15] https://www.tensorflow.org/versions/r1.2/api_guides/python/constant_op

A Example code in full

```
import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)

def xLayerModel(nLayer):
    with tf.name_scope('model'):
        #input
        x = tf.placeholder(tf.float32, [None, 784],name = 'x')

        #layer 1
        layer1_size = 200
        W1 = tf.Variable(tf.random_normal([784, layer1_size]),
    name = 'W1')
        b1 = tf.Variable(tf.random_normal( [layer1_size]),name =
    'b1')
        y1 = tf.nn.sigmoid(tf.matmul(x, W1) + b1,name = 'y1')

        #layer 2
        layer2_size = 100
        W2 = tf.Variable(tf.random_normal([layer1_size,
    layer2_size]),name = 'W2')
        b2 = tf.Variable(tf.random_normal( [layer2_size]),name =
    'b2')
        y2 = tf.nn.sigmoid(tf.matmul(y1, W2) + b2,name = 'y2')

        #layer 3
        layer3_size = 60
        W3 = tf.Variable(tf.random_normal([layer2_size,
    layer3_size]),name = 'W3')
        b3 = tf.Variable(tf.random_normal( [layer3_size]),name =
    'b3')
        y3 = tf.nn.sigmoid(tf.matmul(y2, W3) + b3,name = 'y3')

        #layer 4
        layer4_size = 30
        W4 = tf.Variable(tf.random_normal([layer3_size,
    layer4_size]),name = 'W4')
        b4 = tf.Variable(tf.random_normal( [layer4_size]),name =
    'b4')
        y4 = tf.nn.sigmoid(tf.matmul(y3, W4) + b4,name = 'y4')

        # switch case deciding layer structure
        if nLayer == 0:
            ls = 784
            yx = x
        elif nLayer == 1:
            ls = layer1_size
            yx = y1
        elif nLayer == 2:
            ls = layer2_size
            yx = y2
        elif nLayer == 3:
            ls = layer3_size
            yx = y3
        elif nLayer == 4:
            ls = layer4_size
            yx = y4
        else :
            print('nLayer should be between 0 and 4')


        #layer 5
        W5 = tf.Variable(tf.random_normal([ls,10]),name = 'W5')
        b5 = tf.Variable(tf.random_normal([10]),name = 'b5')
        y5 = tf.nn.softmax(tf.matmul(yx, W5) + b5,name = 'y5')

        #output
        y = y5

    y_ = tf.placeholder(tf.float32, [None, 10])

    #Operations for training
    with tf.name_scope('Training'):
        cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log
    (y),
        reduction_indices=[1]),name ='Cross_entropy')
        learning_rate = 1
        train_step = tf.train.GradientDescentOptimizer(
    learning_rate,name ='Train_step').minimize(cross_entropy)

    #Operations for calculating accuracy
    with tf.name_scope('Accuracy'):
        correct_prediction = tf.equal(tf.argmax(y,1), tf.argmax(
    y_,1),name ='Correct_prediction')
        accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.
    float32),name ='Accuracy')

    sess = tf.InteractiveSession()
    tf.global_variables_initializer().run()
    tf.summary.FileWriter("graph", sess.graph)

    acc = []
    iters = []
    batch_size = 100
    #Training loop using new batch each itteration
    for i in range(10000+1):
        batch_xs, batch_ys = mnist.train.next_batch(batch_size)
        sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys
    })
        #Store accuracy data for last 100 training steps
        if i > 10000-100:
            acc.append(sess.run(accuracy, feed_dict={x: mnist.
    test.images, y_:
            mnist.test.labels}))
            iters.append(i)

    lastAcc = sess.run(accuracy, feed_dict={x: mnist.test.images,
     y_:
        mnist.test.labels})
    return acc,iters,lastAcc

# fetch data
```

```
acc0,iters0,lastAcc0 = xLayerModel(0)

acc1,iters1,lastAcc1 = xLayerModel(1)

acc2,iters2,lastAcc2 = xLayerModel(2)

acc3,iters3,lastAcc3 = xLayerModel(3)

acc4,iters4,lastAcc4 = xLayerModel(4)
print(acc0)
print(iters0)

print(lastAcc0,lastAcc1,lastAcc2,lastAcc3,lastAcc4)
# plot data
import matplotlib.pyplot as plt
plt.figure(1)
plt.plot(iters0,acc0,'g', label= '0 hidden layers')
plt.plot(iters1,acc1,'r', label= '1 hidden layers')
plt.plot(iters2,acc2,'b', label= '2 hidden layers')
plt.plot(iters3,acc3,'k', label= '3 hidden layers')
plt.plot(iters4,acc4,'m', label= '4 hidden layers')

plt.legend(loc='best')
plt.show()

plt.savefig('xLayer_sigmoid.png')
```