

TensorFlow - Deep learning framework

Hubl Lukas, Grüneis Dominik



WAF3 ABGABE-TUTORIAL

Contents

1	Einführung	1
2	Installation	2
2.1	Python	2
2.2	Pandas und Matplotlib	3
2.3	Tensorflow	3
2.4	Optional: Openvc	3
3	MNIST	4
3.1	Implementierung: MNIST Daten einlesen	5
3.2	Testen: MNIST Daten einlesen	6
4	Model	7
4.1	Logistic Regression	7
4.2	Softmax Regression	7
4.3	Implementierung	8
5	Bewertungsschema	10
6	Trainieren	12
6.1	Initiale Gewichte	12
6.2	Learning Rate	12
6.3	Implementierung	13
7	Evaluiere und Visualisieren	15
7.1	Accuracy	15
7.2	Graph	15
7.3	Skalare	16
7.4	Projektor	16
8	Verwenden des gelernten Modells: Realtime Klassifizierer	18
8.1	Implementation	18

Chapter 1

Einführung

Dieses Dokument ist die Abgabe bzw. das Tutorial für das Wahlfach 3. Dieses Dokument behandelt das Framework Tensorflow und generell neuronale Netzwerke. Das Ziel dieses Dokuments ist, dem Leser Schritt für Schritt zu erklären wie man mit Tensorflow ein neuronales Netzwerk aufbauen, einlernen und verwenden kann. Dazu wird ein Beispiel programmiert, welches handschriftliche Ziffern erkennen soll. Es wird die MNIST Datenbank verwendet um Trainingsdaten bereitzustellen. Am Ende sollte man ein Modell einlernen und verwenden können.

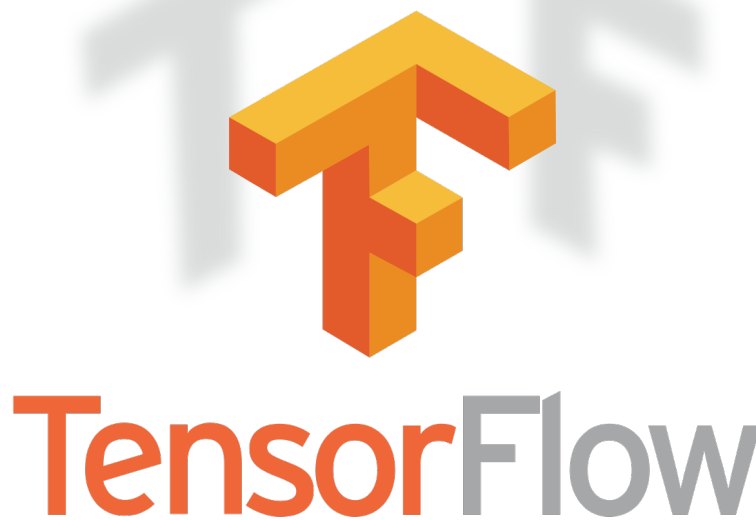


Figure 1.1: Tensorflow Logo

Chapter 2

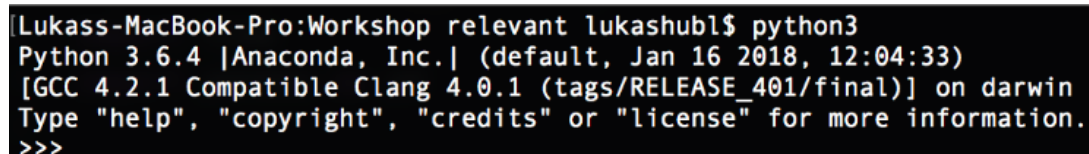
Installation

2.1 Python

Um dieses Tutorial durchführen zu können muss zuerst die Entwicklungsumgebung aufgesetzt werden. Als Programmiersprache wird in diesem Tutorial Python zum Einsatz kommen. Python gilt als höhere Programmiersprache die besonders auf kurzen und gut lesbaren Programmierstil setzt. Charakteristisch ist zum Beispiel das Nichtverwenden von geschwungenen Klammern um Blöcke zu kennzeichnen. Diese werden in Python durch Einrückungen gekennzeichnet.

In diesem Tutorial wird die Python Version 3.6.5 oder höher verwendet. Diese muss, falls noch nicht installiert, von der offiziellen Website <https://www.python.org> heruntergeladen werden. Es gibt von Python sowohl eine Windows, MacOS als auch eine Linux Version. Je nach Betriebssystem sollte die richtige Version verwendet werden. Hat man den Installer heruntergeladen kann er auch einfach ausgeführt werden, wodurch dann Python installiert wird. Wichtig für die Windows Version ist hierbei, dass Python zu den Umgebungsvariablen hinzugefügt werden muss. Dies wird aber bei der Installation nachgefragt und kann bzw. sollte ausgewählt werden. Vergisst man dies, muss man Python manuell den Umgebungsvariablen hinzufügen.

Hat man Python installiert kann man überprüfen ob die Installation erfolgreich war. Dazu öffnet man einfach die Kommandozeile bzw. das Terminal und gibt bei Windows den Befehl *python* bzw. bei MacOS und Linux den Befehl *python3* ein. Dann sollte eine ähnliche Ausgabe wie bei Bild 2.1 ausgegeben werden. Im weiteren Verlauf dieses Tutorials wird nur noch der Befehl *python3* verwendet. Windows Nutzer müssen aber im fortlaufenden Tutorial trotzdem *python3* verwenden



```
[Lukass-MacBook-Pro:Workshop relevant lukashubl$ python3
Python 3.6.4 |Anaconda, Inc.| (default, Jan 16 2018, 12:04:33)
[GCC 4.2.1 Compatible Clang 4.0.1 (tags/RELEASE_401/final)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

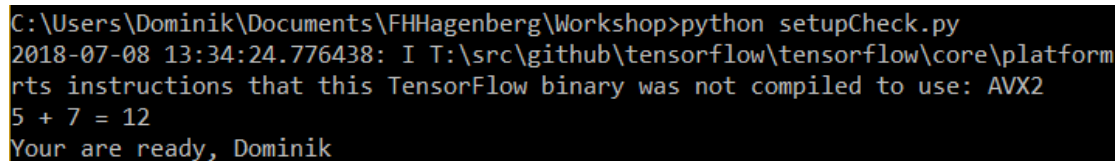
Figure 2.1: Ausgabefenster Kommandozeile bzw. Terminal bei erfolgreicher Installation

2.2 Pandas und Matplotlib

Als nächstes werden zwei python Packages zum Visualisieren von Daten benötigt. Diese Packages können mit dem in Python integrierten Paketverwaltungsprogramm pip heruntergeladen und automatisch installiert werden. Das erste Package ist Pandas und kann mit dem Befehl *pip install pandas* von der Kommandozeile bzw. vom Terminal aus installiert werden. Genau gleich kann man das Package Matplotlib installieren, nämlich mit dem Befehl *pip install matplotlib*.

2.3 Tensorflow

Ähnlich verhält es sich mit der Installation unseres eigentlichen Frameworks Tensorflow. Dies kann auch mit pip heruntergeladen und installiert werden. Dazu muss der Befehl *pip install tensorflow* auf der Kommandozeile bzw. auf dem Terminal ausgeführt werden. Dadurch wurde die Standardversion von Tensorflow installiert. Das bedeutet ohne GPU Unterstützungen und ohne bestimmte prozessorspezifische Instruktionen wie AVX2. Um zu testen ob Tensorflow und alle Packages richtig installiert wurden, wurde ein Skript geschrieben welches alles Packages testet. Dafür muss vom Workshop-Repository das Skript *setupCheck.py* heruntergeladen werden. Wird das Skript mit dem Befehl *python3 setupCheck.py* ausgeführt gibt das Skript aus ob die Entwicklungsumgebung mit allen Packages richtig installiert wurde. Die Ausgabe des ausgeführten Skripts sollte dann der Ausgabe des Bildes 2.2 ähneln.



```
C:\Users\Dominik\Documents\FHHagenberg\Workshop>python setupCheck.py
2018-07-08 13:34:24.776438: I T:\src\github\tensorflow\tensorflow\core\platform
rts instructions that this TensorFlow binary was not compiled to use: AVX2
5 + 7 = 12
Your are ready, Dominik
```

Figure 2.2: Ausgabefenster bei erfolgreichem ausführen des Skripts *setupCheck.py*

2.4 Optional: Openvc

Das Package *opencv-python* wird später in unserer Realtime Klassifizierung benötigt um mit der Kamera des Notebooks oder PCs zu interagieren. Theoretisch gehört dieses package aber nicht zur Entwicklungsumgebung und wird deshalb hier als optional gekennzeichnet. Um die Realtime Klassifizierung programmieren bzw. ausführen zu können, wird dieses Package aber UNBEDINGT benötigt.

Das Package selbst kann mit *pip install opencv-python* installiert werden. Leider verhält es sich mit diesem Package anders als mit den anderen Packages und es enthält viele externe Abhängigkeiten zu anderen Packages. Da von PC zu PC unterschiedlich ist was bereits installiert ist und in unserem Fall relativ wenige Packages fehlten, können wir hier nicht angeben welche Packages nachinstalliert werden müssen.

Damit wäre die Entwicklungsumgebung installiert.

Chapter 3

MNIST

Die MNIST (Modified National Institute of Standards and Technology) Datenbank ist eine große Datenbank mit handgeschriebenen Ziffern (0..9). Die Datenbank enthält 60.000 Trainingsbilder und 10.000 Testbilder. Als Beispiel siehe Bild 3.1

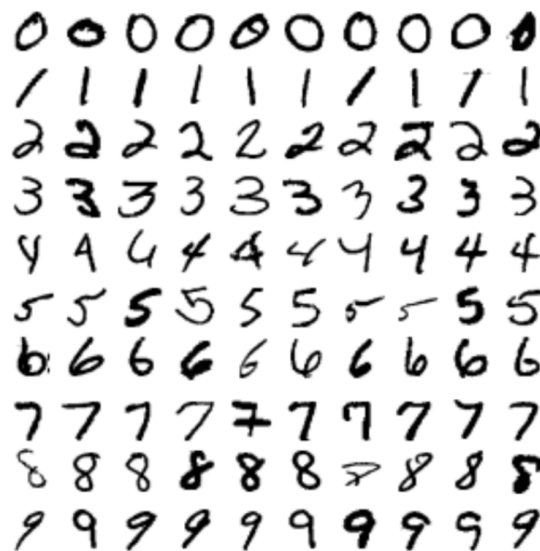


Figure 3.1: Ausschnitt aus MNIST Datenbank

Das Format der Bilder ist dabei 28x28 und besteht aus schwarzer Schrift auf weißem Hintergrund. Ein Pixel kann dabei Werte zwischen 0(weiß) und 1(schwarz) annehmen. Als Beispiel siehe Bild 3.2 Diese Daten werden häufig zum Trainieren und Testen verschiedenster Bilderkennungsmodellen bzw. Algorithmen verwendet. Wir benutzen sie in unserem Beispiel um unserem Modell beizubringen handschriftliche Ziffern zu erkennen.

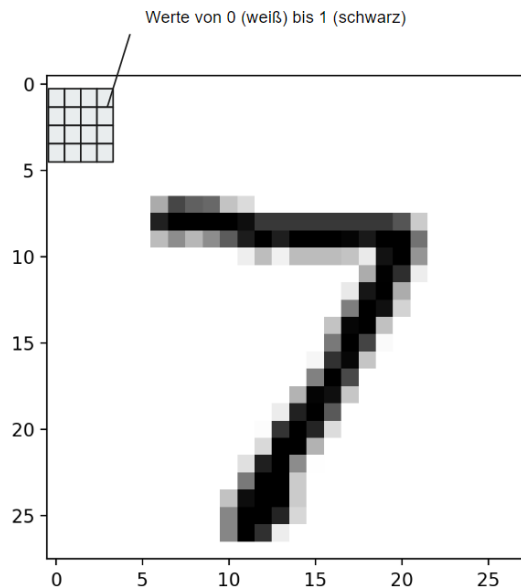


Figure 3.2: Betrachtung eines einzelnen MNIST Datensets

3.1 Implementierung: MNIST Daten einlesen

In unserem Codebeispiel ist das einlesen der MNIST Daten auch der erste Schritt der programmiert werden muss. Dazu müssen wir aber generell vorher alle für das Programm benötigten Abhängigkeiten importieren. In unserem Programm werden folgende Abhängigkeiten benötigt:

```
1 import tensorflow as tf
2 import os
3 import matplotlib as matplotlib
4 import matplotlib.pyplot as plt
5 import pandas as pd
6 from tensorflow.contrib.tensorboard.plugins import projector
```

Diese Imports werden ganz oben an den Anfang des Skripts geschrieben.

Die MNIST Daten selbst bekommen wir von Tensorflow, da diese die MNIST Daten in ihre Beispiele integriert haben. Diese Möglichkeit des Ladens der Daten könnte in Zukunft ein Problem darstellen, da Tensorflow derzeit ihre Beispielstruktur ändert. Das Laden der Daten kann mit folgendem Befehl realisiert werden.

```
1 from tensorflow.examples.tutorials.mnist import input_data
```

Dadurch werden die Trainings und Testdaten automatisch als Archiv heruntergeladen. Es gibt zwei Möglichkeiten die Labels der Daten auszulesen. Man kann sie ganz normal binär interpretieren also zum Beispiel `2=0000000010` oder mit one_hot encoding zum Beispiel `2=0010000000`. Dies lässt sich mit einem mit dem Übergabeparameter der Funktion `input_data.read_data_sets()` einstellen. Setzt man `one_hot=false` wird die normale binäre Version genutzt, setzt man `one_hot=true` verwendet man one_hot encoding. Wir werden unsere Daten mit beiden Möglichkeiten auslesen.

```
1 mnist = input_data.read_data_sets("image_data", one_hot=True)
2 mnistTwo = input_data.read_data_sets("image_data", one_hot=False)
```

Um die Daten nachher in Tensorboard verwenden und anzeigen zu können, speichern wir sie noch in eine Variable mit

```
1 images = tf.Variable(mnist.test.images, name='images')
```

3.2 Testen: MNIST Daten einlesen

Um zu testen ob die Daten richtig heruntergeladen und eingelesen wurden lassen wir uns einfach ein Bild aus den MNIST Daten anzeigen. Der hier folgende Codeblock kann nach dem Testen wieder auskommentiert werden.

Dafür lesen wir die Werte eines Bildes aus(hier Bild Nr.240) und reshappen sie von einem 784x1 Vektors zu einer 28x28 Matrix, damit die Pixel wieder an der richtigen Stelle sind.

```
1 testImage = mnist.test.images[240].reshape(28,28)
```

Diese Matrix lassen wir dann mit dem Package Matplotlib auf einen Plot printen mit folgendem Befehl:

```
1 plt.imshow(testImage, cmap = matplotlib.cm.binary, interpolation = "nearest")
```

Um zu wissen welche Ziffer ausgegeben wird lassen wir auch noch das Label ausgeben

```
1 print(mnist.test.labels[240])
```

Und dann müssen wir noch den Plot ausgeben

```
1 plt.show()
```

Funktioniert dies, sollte diese Codestück wieder auskommentiert oder gelöscht werden.

Chapter 4

Model

Nun da man mit dem Begriff MNIST vertraut sein sollte und auch weiss, wie man die Daten in ein eigenes Python Programm einbinden kann, kann mit dem nächsten Schritt, dem Model, fortgefahren werden. Um den Begriff des Models möglichst knapp zusammenzufassen, eignet sich der Begriff des Systems. Das Model bildet dabei ein System, welches einen Input entgegennimmt, mit diesem Berechnungen durchführt und schlussendlich einen entsprechenden Output liefert. Die wichtigsten Bausteine im Model sind dabei die künstlichen Neuronen, die Gewichte, welche diese verbinden und der Bias, welcher als externer Input dient. Je nach Anordnung und Aufbau dieser Elemente kann zwischen etlichen verschiedenen Models unterschieden werden. Für diesen Workshop wird ein Model namens Softmax Regression verwendet. Um dieses allerdings möglichst einfach zu beschreiben, macht es Sinn, erstmals die Logistic Regression zu erläutern.

4.1 Logistic Regression

Wie die Grafik 4.1 zeigt, wird der Input bei der Logistic Regression durch die Gewichte propagiert und schlussendlich die Summe der einzelnen Neuronen gebildet. Dieser Wert wiederum wird in die Aktivierungsfunktion weitergegeben, welche zwischen zwei möglichen Ausgangswerten entscheidet. Dies macht beispielsweise Sinn, wenn man sich zum Ziel setzt mit Hilfe eines neuronalen Netzes festzustellen, ob es sich bei einem gewissen Input um ein Element eines gewissen Typs handelt oder nicht. Es wird also zwischen Ja und Nein unterschieden.

4.2 Softmax Regression

Wie aus der Grafik 4.2 zu entnehmen ist, verfügt die Softmax Regression prinzipiell über einen sehr ähnlichen Aufbau wie die Logistic Regression. Der wesentliche Unterschied zur Logistic Regression liegt jedoch im Output des Models. Im Gegensatz zur Logistic Regression werden die Ausgänge der einzelnen Neuronen nicht in einer Summe an eine Aktivierungsfunktion weitergeleitet, es gibt N Summen der Neuronen, welche in die Softmax Funktionseinheit propagiert werden. Diese entscheidet nun nicht einfach zwischen Ja und Nein, sondern gibt eine Wahrscheinlichkeit für jeden der N möglichen Ausgänge ab, dass es sich um diesen handelt. Dieses Muster eignet sich besonders gut

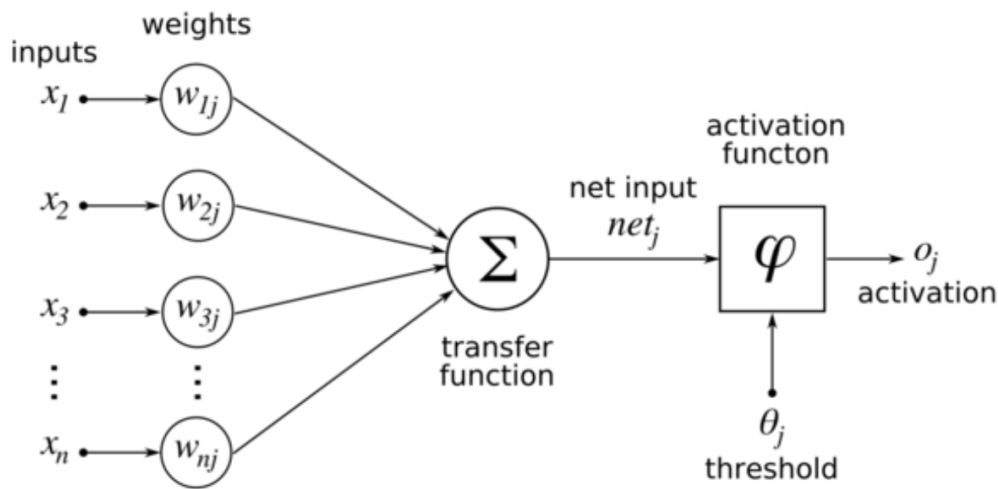


Figure 4.1: Logistic Regression

für die Problemstellung der Ziffernerkennung. Sie liefert jeweils für jede Ziffer zwischen 0 und 9 einen Wahrscheinlichkeitswert, der angibt, wie wahrscheinlich es für das Model ist, dass der Input, in diesem Fall ein Bild, eine 0,1,2,..., 9 zeigt. Sucht man jetzt den höchsten Wahrscheinlichkeitswert aus diesem Vektor, erhält man jene Ziffer, von der das Model glaubt, sie auf dem Bild erkannt zu haben.

4.3 Implementierung

Um das Model im Python Skript zu implementieren, müssen zuerst einmal die benötigten TensorFlow Variablen und Placeholder angelegt werden. Der erste Placeholder der gebraucht wird, ist jener der Input-Images. Jedes Bild der MNIST Daten besteht aus 28x28 Pixeln. Reiht man diese Pixel hintereinander, erhält man einen Zeilenvektor der Länge 784. Da die Bilddaten auf diese Weise gespeichert werden sollen, wird mit folgender Zeile ein passender Placeholder angelegt.

```
1 x = tf.placeholder(tf.float32, [None, 784], name='InputData')
```

Dieser kann eine beliebige Anzahl (None) an Bilddaten (1x784) im Format Float (Grauwerte zwischen 0 und 1) beinhalten und trägt den Namen InputData. Als nächstes wird ein Placeholder gebraucht, der zu jedem Bild das dazugehörige Label abspeichert. Da wir die Labels als One-Hot Vektor geladen haben und 10 mögliche Ausgänge existieren, bildet jedes Label einen Zeilenvektor der Länge 10 ab. Um so einen Placeholder anzulegen, wird folgender Befehl verwendet.

```
1 y = tf.placeholder(tf.float32, [None, 10], name = 'LabelData')
```

Analog zum InputData Placeholder kann auch dieser eine beliebige Anzahl (None) an Labels (1x10) im Format Float (Wahrscheinlichkeitswert zwischen 0 und 1) beinhalten

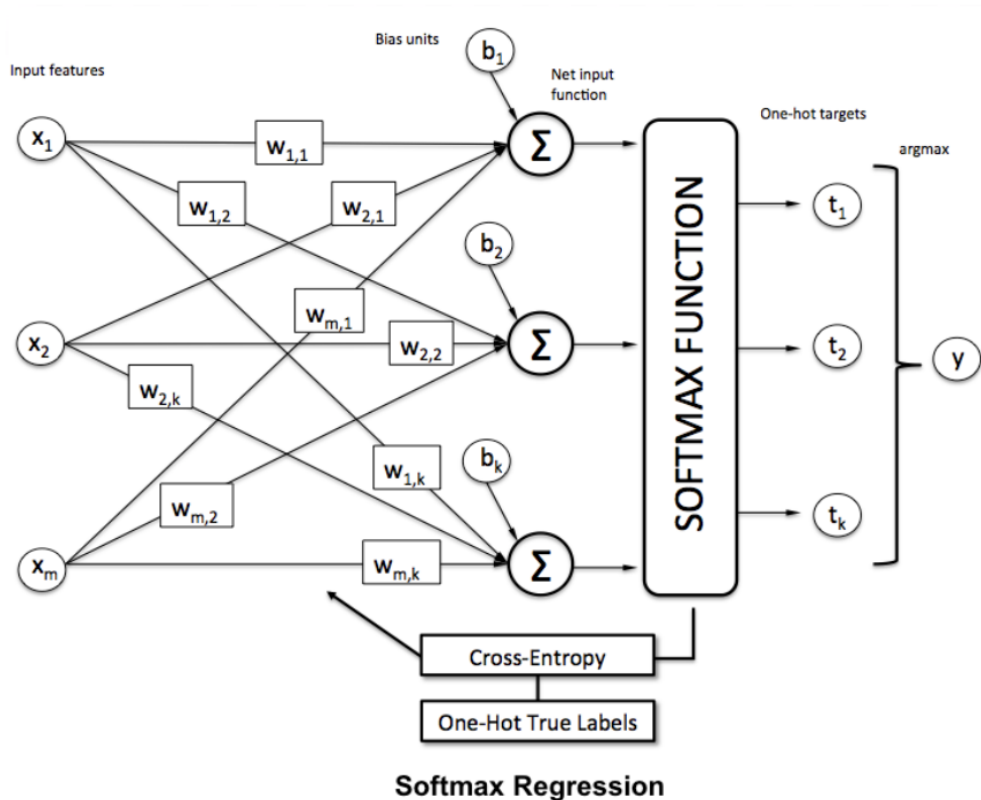


Figure 4.2: Softmax Regression

und trägt den Namen `LabelData`. Nachdem nun die Placeholder für den Input angelegt wurden, kann mit den nötigen, modellspezifischen Variablen fortgefahren werden. Der Befehl:

```
1 W = tf.Variable(tf.zeros([784, 10]), name='Weights')
```

erzeugt eine Variable, welche alle Gewichte zwischen Input und Output beinhaltet. Dies ergibt bei 784 Eingängen und 10 Ausgängen 7840 Gewichte insgesamt. Die letzte benötigte Variable ist jene der Bias-Werte. Für jeden der möglichen Ausgänge wird jeweils ein Wert benötigt, was zu einem Zeilenvektor der Länge 10 führt. Dieser wird folgendermaßen implementiert.

```
1 b = tf.Variable(tf.zeros([10]), name = 'Bias')
```

Das letzte Stück Code welches benötigt wird, ist jenes des Models an sich.

```
1 with tf.name_scope('Model'):
2     pred = tf.nn.softmax(tf.matmul(x,W) + b)
```

Die erste Zeile hat prinzipiell keine funktionalen Auswirkungen auf das Model, jedoch hilft es TensorBoard dabei den Graphen übersichtlicher darzustellen, wie man später bemerken wird. Die zweite Zeile erzeugt nun das eigentliche Model, welches auf Basis der zuvor angelegten Placeholder und Variablen eine Softmax entsprechende Schätzung liefert, um welche Ziffer es sich handelt.

Chapter 5

Bewertungsschema

Nach dem Einbinden der MNIST Daten und dem Erstellen des Models mitsamt allen benötigten Placeholdern und Variablen ist es nun an der Zeit festzustellen, wie nahe die Prädiktionen des Models an der Realität sind. Dafür wird in diesem Workshop eine mathematische Methode namens Cross-Entropy verwendet. In kürzester und einfachster Form beschrieben, liefert Cross-Entropy einen Zahlenwert, welcher die Distanz zwischen dem One-Hot Vektor des Labels mit jenem der Prädiktion liefert. Diese Distanz wird Loss genannt und beschreibt den Fehler des Models bei der Prädiktion. Die Formel von Cross-Entropy lautet dabei: $-\sum y' * \ln y$.

Cross-Entropy

$$\begin{array}{rcl} 000000100 & \text{Label} \\ 000000100 & \text{Prediction} \\ \hline -[0 \times \ln(0) + 0 \times \ln(0) + \dots + 1 \times \ln(1) + \dots] & = 0 & \text{Error} \end{array}$$

$$\begin{array}{rcl} 0000000100 & \text{Label} \\ ,1,1,1,1000,600 & \text{Prediction} \\ \hline -[4 \times 0 \times \ln(0,1) + 5 \times 0 \times \ln(0) + 1 \times \ln(0,6)] & = 0,51 & \text{Error} \end{array}$$

$$\begin{array}{rcl} 0000000100 & \text{Label} \\ 1000000000 & \text{Prediction} \\ \hline -[0 \times \ln(1) + 8 \times 0 \times \ln(0) + 1 \times \ln(0)] & = \infty & \text{Error} \end{array}$$

Figure 5.1: Cross-Entropy Beispiel

Nun, da ein passendes Bewertungsschema gefunden wurde, ist es an der Zeit, dieses in den Code zu integrieren. Gleich wie beim Model, macht es wiederum Sinn, dieser Funktion einen eigenen *name_scope* zu geben.

```
1 with tf.name_scope('Loss'):  
2     cost = tf.reduce_mean(-tf.reduce_sum(y*tf.log(pred)))
```

Betrachtet man die zweite Zeile genauer, scheint es eventuell erstmals nicht ganz klar, was hier berechnet wird. Dies ist allerdings schnell geklärt. Der Teil innerhalb von *tf.reduce_mean* beinhaltet lediglich die mathematische Formel von Cross-Entropy $-\sum y' * \ln y$. Da wir allerdings beim Model (pred) möglicherweise nicht nur ein einzelnes Bild evaluieren lassen möchten (die Anzahl der Bilder in InputData ist nicht begrenzt), wird über *tf.reduce_mean* der Mittelwert über alle Einzelfehler aller zu evaluierenden Bilder berechnet.

Chapter 6

Trainieren

Nachdem nun auch die Möglichkeit der Bewertung des Models gegeben ist, kann damit begonnen werden das Model, basierend auf seinen Fehlern, zu trainieren. Der Ansatz, der in diesem Workshop dazu verwendet wird, lautet Gradient Descent und lässt sich am einfachsten mit einem simplen Beispiel erläutern. Dazu kann man sich einen Bergsteiger vorstellen, der sich auf einem Berggipfel befindet und sich ins Tal begeben will. Dazu kommt, dass der Bergsteiger eine Augenbinde trägt und somit den Weg ins Tal nicht sehen kann. Den Ansatz, den der „Gradient Descent Bergsteiger“ nun wählt, ist folgender: Zuerst fühlt er mit seinen Füßen den Boden sich selbst herum ab, um das steilste Gefälle rundum ihn zu finden, um daraufhin einen Schritt in diese Richtung zu machen. Diesen Vorgang wiederholt er so lange, bis der zaehler

- a) keine Schritte mehr gehen kann.
- b) kein negatives Gefälle mehr fühlen kann und somit annimmt im Tal zu sein.

Um dieses Beispiel auf das Training des Models umzumünzen, muss man sich den Berg als Fehlerfunktion des Models (Cross-Entropy), die Position des Bergsteigers als aktuellen Fehler des Models (welcher sich durch das aktuelle Weight-Setup ergibt) und den Schritt des Bergsteigers den Berg hinab als Lernschritt beim trainieren des Models vorstellen. Zwei wichtige Parameter beim Training sind dabei zum einen die initialen Gewichte des Models und zum anderen die Learning-Rate.

6.1 Initiale Gewichte

Die initialen Gewichte des Models spiegeln dabei die Startposition des Bergsteigers auf seinem Weg ins Tal wieder. Wählt man die initialen Gewichte geschickt aus, beginnt die Reise des Bergsteigers relativ nahe über dem Tal, werden sie ungeschickt gewählt, muss der Bergsteiger seinen Abstieg am Gipfel des Berges, am maximalen Fehler, beginnen.

6.2 Learning Rate

Die Learning-Rate gibt an, wie viel das Model auf einmal lernen kann. Dies ist wiederum vorstellbar mit der Schrittlänge, die der Bergsteiger hat. Wird die Learning-Rate zu

groß gewählt, kann es dazu kommen, dass der Fehler nicht wie erwünscht immer kleiner wird mit jeder Iteration, sondern dass sich dieser mit jeder Iteration immer weiter aufschauelt. Wird sie allerdings zu klein gewählt, sind die Lernschritte äußerst klein und der Lernvorgang dauert somit sehr lange und benötigt eine enorme Menge an Trainingsdaten.

6.3 Implementierung

Um das Modell zu trainieren müssen folgende Zeilen Code hinzugefügt werden. Zunächst müssen drei Parameter definiert werden:

```
1 learning_rate = 0.01
2 #Gibt an wie schnell gelernt werden kann.
3
4 training_epochs = 30
5 #Gibt an wie oft das Model trainiert werden soll. Vergleichbar mit der Anzahl der
   Schritte die der Bergsteiger gehen darf.
6
7 batch_size = 100
8 #In jeder Epoche wird das Model mit allen MNIST Trainings-Datensätzen trainiert. Die
   Batch Size gibt dabei an wie groß die einzelnen Chargen an Datensätzen sein
   sollen.
```

Als nächstes muss wiederum eine Funktion angelegt werden, die den eigentlichen Lernvorgang, in diesem Fall mit Hilfe von Gradient Descent, abbildet:

```
1 with tf.name_scope('Optimizer'):
2     optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)
```

Wiederum in einem eigenen *name_scope* wird hier ein Gradient Descent Optimizer erstellt, der mit der vorhin definierten Learning-Rate versucht, die Kosten-cost (Cross-Entropy)-zu minimieren. Nun kann begonnen werden das Model zu trainieren, dazu müssen zuerst alle TensorFlow Variablen initialisiert werden. Dies geschieht mit dem Befehl.

```
1 init = tf.global_variables_initializer()
```

Im nächsten Schritt wird eine TensorFlow Session eröffnet, mit deren Hilfe erst die einzelnen Funktionen ausgeführt werden können.

```
1 with tf.Session() as sess:
2     sess.run(init)
3     #training
4     for epoch in range(training_epochs):
5         avg_cost = 0.0
6         total_batch = int(mnist.train.num_examples/batch_size)
7         #loop over all batches
8         for i in range(total_batch):
9             batch_xs, batch_ys = mnist.train.next_batch(batch_size)
10            opt, c = sess.run([optimizer, cost], feed_dict={x: batch_xs, y: batch_ys})
11            avg_cost += c / total_batch
12
13            if (epoch+1) % 1 == 0:
14                print(avg_cost)
```

Dieser Code initialisiert zuerst alle Variablen und fährt dann fort mit dem eigentlichen Training. Dazu wird ***training_epochs*** mal das gesamte Trainingsdaten-Set vom MNIST in ***batch_size*** große Stücke zerlegt und jeweils nacheinander in die Placeholder Input-Data und LabelData geladen. Daraufhin wird auf Basis der Daten in den Placeholdern sowohl der Fehler des Models berechnet (cost), als auch der Optimizer gestartet, welcher das Model iterativ verbessert. Darüber hinaus wird aus Visualisierungsgründen der durchschnittliche Fehler berechnet und auf der Konsole ausgegeben. Durch die Ausgabe auf der Konsole kann über jede Epoche hinweg mitverfolgt werden, wie der Fehler stetig kleiner wird.

Chapter 7

Evaluieren und Visualisieren

7.1 Accuracy

Fertig! Die eigentliche Arbeit ist somit erledigt und das Model wurde trainiert. Nun möchte man möglicherweise evaluieren, wie gut das Model performt. Dies kann relativ einfach durch das Hinzufügen einer weiteren Funktion erledigt werden.

```
1 with tf.name_scope('Accuracy'):
2     acc = tf.equal(tf.argmax(pred,1), tf.argmax(y, 1))
3     acc = tf.reduce_mean(tf.cast(acc, tf.float32))
```

Die Accuracy vergleicht dabei das Ergebnis des Models mit dem Label des Datensatzes und notiert sich eine 1, wenn diese übereinstimmen und eine 0, wenn nicht. Im nächsten Schritt wird ein Mittelwert über alle Prädiktionen gebildet, was einen Wert ergibt, der eine prozentuelle Übereinstimmung zwischen Prädiktion und Label widerspiegelt. Um diesen Wert in der Konsole anzeigen zu lassen, muss einfach folgender Befehl an das Ende des Session Bereichs hinzugefügt werden:

```
1 print("Accuracy: ", acc.eval({x: mnist.test.images, y: mnist.test.labels}))
```

Will man außerdem visuell Einsicht in die geleistete Arbeit werfen, eignet sich TensorBoard perfekt dafür. In diesem Workshop werden folgende Visualisierungen mit Hilfe von TensorBoard implementiert:

7.2 Graph

Will man sich den Graphen, welcher den Datenfluss dieses Scripts zeigt, anzeigen lassen, sind folgende Codeteile einzufügen:

```
1 logs_path = 'tf_logs'
```

Diese Zeile muss zu den anderen Parametern am Anfang des Scripts hinzugefügt werden und gibt an, in welches Verzeichnis TensorFlow die serialisierten Graph-Daten für TensorBoard speichern soll.

Danach muss am Beginn des Session Bereichs im Script folgende Zeile eingefügt werden um einen FileWriter zu erzeugen, welcher den Graphen serialisiert und an den definierten Pfad (*logs_path*) speichert:

```
1 summary_writer = tf.summary.FileWriter(logs_path, graph= tf.get_default_graph())
```

Am Ende des Session Bereichs im Script muss zusätzlich folgendes Statement eingefügt werden, um den FileWriter wieder ordnungsgemäß zu schließen:

```
1 summary_writer.close()
```

Wird nun das Script ausgeführt und TensorBoard gestartet, kann unter dem Tab „Graph“ der Graph eingesehen werden.

7.3 Skalare

Eine weitere Möglichkeit TensorBoard sinnvoll einzusetzen ist, bestimmte Skalare zu definieren, die im Laufe des Trainings geloggt werden, um deren Verlauf in TensorBoard nachvollziehbar zu machen. In diesem Fall werden Skalare für den Fehler und die Accuracy gebildet:

```
1 tf.summary.scalar("loss", cost)
2 tf.summary.scalar("accuracy", acc)
3 merged_summary_op = tf.summary.merge_all()
```

Dieser Code muss vor dem Beginn des Session Bereichs implementiert werden und erstellt jeweils einen Skalar für den Fehler und die Accuracy. Der dritte Befehl wird benötigt, damit alle Summaries im Graph verwendet werden. Um nun diese Skalare zu loggen, muss folgende Zeile im Session Bereich:

```
1 opt, c = sess.run([optimizer, cost], feed_dict={x: batch_xs, y: batch_ys})
```

durch diese ersetzt werden:

```
1 opt, c, summary = sess.run([optimizer, cost, merged_summary_op], feed_dict={x:
    batch_xs, y: batch_ys})
```

und darüber hinaus jene Zeile gleich danach eingefügt werden:

```
1 summary_writer.add_summary(summary, epoch*total_batch + i)
```

Analog zum Graph muss auch hier das Script und TensorBoard ausgeführt werden, um den Trace der beiden Skalare über den Trainingsprozess hinweg nachverfolgen zu können.

7.4 Projektor

Der Projektor bietet die Möglichkeit, die 784 Dimensionen eines MNIST Bildes auf 3 Dimensionen zu reduzieren und somit die einzelnen Bilder der MNIST Datenbank im Dreidimensionalen Raum gegenüberzustellen. Dazu müssen im Code zweierlei Daten zu Verfügung gestellt werden: Metadaten und Embeddings. Metadaten werden dazu benötigt, den Bezug eines Bildes zum entsprechenden Label herstellen zu können. Dies kann gemacht werden, indem man mit folgendem Code eine Datei erstellt, die Zeilenweise die Labels, jedoch nicht im One-Hot Format, zu jedem Bild der Trainingsdaten enthält.

```
1 metadata = os.path.join(logs_path, 'metadata.tsv')
2 with open(metadata, 'w') as metadata_file:
3     for row in mnistTwo.test.labels:
4         metadata_file.write('%d\n' % row)
```

Die Position dieses Codestücks liegt wiederum vor dem Session Bereich. Innerhalb des Session Bereichs werden nun die Embeddings konfiguriert:

```
1 saver = tf.train.Saver([images])
2 saver.save(sess, os.path.join(logs_path, 'images.ckpt'))
3 config = projector.ProjectorConfig()
4     embedding = config.embeddings.add()
5     embedding.tensor_name = images.name
6     embedding.sprite.image_path = os.path.join(logs_path, 'sprite.png')
7     embedding.metadata_path = metadata
8     embedding.sprite.single_image_dim.extend([28,28])
9
10     projector.visualize_embeddings(tf.summary.FileWriter(logs_path), config)
```

Hier werden sowohl die vorhin erstellten Metadaten zu den Embeddings hinzugefügt, als auch ein Sprite Image angegeben, welches die einzelnen Bilder der MNIST Daten als Thumbnails in einem .png File vereint. Zum Schluss noch das Visualisieren der Embeddings laut den gerade erstellten Settings (Sprite Image, Metadata) aktiviert. Durch Ausführen des Scripts und das Aktivieren von TensorBoard kann nun im Tab Projektor und unter T-SNE als Dimensions-Reduktions Algorithmus die dimensionale Trennung der einzelnen Datensätze begutachtet werden.

Diese Ansicht ist analog zu jener einfachen Klassifizierung zu sehen, nur dass es sich bei MNIST Bildern nicht um 2 sondern um 784 auf 3 Dimensionen reduzierte Daten handelt.

Chapter 8

Verwenden des gelernten Models: Realtime Klassifizierer

Das Modell, welches gerade angelernt wurde, wollen wir natürlich jetzt auch in einer wirklichen Anwendung verwenden. Dazu werden wir einen Realtime Klassifizierer programmieren, welcher ausgehend von der Webcam eines Notebooks oder PCs Ziffern einlesen und erkennen soll.

8.1 Implementation

Zuerst müssen wieder alle Abhängigkeiten importiert werden:

```
1 import tensorflow as tf
2 import cv2
3 import numpy as np
4 import matplotlib as matplotlib
5 import matplotlib.pyplot as plt
6 from PIL import Image
7 from resizeimage import resizeimage
8 import time
```

Dann muss wieder eine Sessionobjekt erzeugt werden.

```
1 sess=tf.Session()
```

Dann starten wir die Aufnahme via Webcam, wodurch sich dann das Webcamlämpchen einschalten wird.

```
1 cap = cv2.VideoCapture(0)
```

Wie vorher müssen auch wieder die MNIST Daten geladen werden. Dies wird getan, damit man die Labels extrahieren kann.

```
1 from tensorflow.examples.tutorials.mnist import input_data
2 mnist = input_data.read_data_sets('MNIST_data', one_hot=True)
```

Dann muss das Model wiederhergestellt werden. Dazu wird das Model von vorher geladen, also *mnist_workshop_model-1.meta*. Daraus kann dann das ganze Model wiederhergestellt werden.

```
1 saver = tf.train.import_meta_graph('mnist_workshop_model-1.meta')
2 saver.restore(sess,tf.train.latest_checkpoint('./'))
```

Als nächstes müssen die Tensoren vom wiederhergestellten Graphen extrahiert werden, also die Placeholder und die Prediction um wieder auf das Modell zugreifen zu können.

```
1 graph = tf.get_default_graph()
2 y = graph.get_tensor_by_name("model:0")
3 x = graph.get_tensor_by_name("InputData:0")
4 y_ = graph.get_tensor_by_name("LabelData:0")
5 pred = tf.argmax(y,1)
```

Der nächste Schritt ist das Aufnehmen eines Bildes mit der Webcam. Da dies immer wieder geschieht, machen wir das in einer Endlosschleife. Dies bedeutet aber auch, dass das Skript nur mit *STRG+C* oder einer Abbruchbedingung beendet werden kann.

```
1 while True:
2     time.sleep(0)
```

Mit den folgenden Befehlen wird ein Bild aufgenommen, ein Bereich des Bildes ausgewählt, dieser Bereich transformiert auf eine 28x28 Matrix und von RGB in Graustufen umgewandelt.

```
1 ret, image_np = cap.read()
2 image_np = image_np[100:660, 100:660]
3 image_np = cv2.resize(image_np, (28, 28))
4 image_np = cv2.cvtColor(image_np, cv2.COLOR_RGB2GRAY)
```

Als nächstes werden um das Bild zu vereinfachen, alle Pixel mit einem Wert über einem gewissen Threshold (in unserem Fall 50) auf den Maximalwert(255) gesetzt.

```
1 for i in range(0,28):
2     for a in range(0,28):
3         if image_np[i,a] > 50:
4             image_np[i,a] = 255
```

Nun wandeln wir der Einfachheit halber das Array in ein numpy Array um. Numpy ist ein Python package welches sich auf mathematische Operationen spezialisiert hat und dies deshalb sehr effizient realisiert. Da die MNIST Daten mit welchem das Model gelernt wurden nur einen Scope von 0 bis 1 und nicht 0 bis 255 hat, muss dieser Wert mit 255 dividiert werden. Da 255 weiß ist aber bei den MNIST Daten 0 weiß ist, wird vom Quotienten noch 1 abgezogen und der Absolutwert genommen. So kommen wir auf den selben Scope wie die MNIST Daten also 0(weiß) und 1(schwarz).

```
1 newarr = np.array(image_np)
2 newarr = abs((newarr / 255) - 1)
3 newshape = newarr.shape
```

Dieses Array muss dann noch in eine eindimensionale Matrix umgewandelt werden um dem Scope des Eingangs des Modells gerecht zu werden. Dazu wird das Array zuerst in ein eindimensionales Array umgewandelt und dann in vom eindimensionalen Array in eine eindimensionale Matrix.

```
1 new_flat_arr = newarr.ravel()
2 newvector = np.matrix(new_flat_arr)
```

Diese eindimensionale Matrix muss jetzt nur noch durch den Predictor geschickt werden und man erhält ein Ergebniss. Der Predictor benötigt als Y-Value noch die Testlabels welche wir ganz am Anfang ausgelesen haben.

```

1   erg = sess.run(pred, feed_dict={x: newvector, y_: mnist.test.labels})
2   print(erg)

```

Um das Bild und das Ergebnis in Realtime anzeigen zu können müssen wir noch folgenden Code einfügen. Der Code beinhaltet die Positionierung des Ergebnisses auf dem Plot, das Anzeigen des Bildes und das Anzeigen des Ergebnisses. Genauer wird auf den Code nicht eingegangen, da er relativ selbstverständlich ist.

```

1   #ergebnisse zeigen im realtime window
2   font = cv2.FONT_HERSHEY_SIMPLEX
3   pts = np.array([[120,100],[420,100],[420,420],[120,420]], np.int32)
4   pts = pts.reshape((-1,1,2))
5   cross = np.array([[300,0],[300,560],[0,560],[0,280],[560,280],[560,0]], np.int32)
6
7   image_np = cv2.resize(image_np, (560,560))
8
9   font = cv2.FONT_HERSHEY_SIMPLEX
10  bottomLeftCornerOfText = (10,500)
11  fontScale = 1
12  fontColor = (0,0,0)
13  lineType = 2
14
15  cv2.putText(image_np, str(erg[0]),
16              bottomLeftCornerOfText,
17              font,
18              fontScale,
19              fontColor,
20              lineType)
21  cv2.polylines(image_np,[cross],True,(0,0,0))

```

Um das Programm jederzeit abbrechen zu können fehlt jetzt noch einen Abbruchbedingung

```

1   cv2.imshow('object detection', image_np)
2   if cv2.waitKey(25) & 0xFF == ord('q'):
3       cv2.destroyAllWindows()
4       break

```

Und schon funktioniert unser selbst programmierter Realtime Klassifizierer.