Facial Keypoint Detection with Tensorflow

Inhaltsverzeichnis

Aufgabenstellung

Im Rahmen der Projektarbeit sollten die Machine Learning Framework Tensorflow und Caffe hinsichtlich ihrer Benutzerfreundlichkeit, ihrer Möglichkeiten sowie ihrer Performanz miteinander verglichen werden.

Für diese Aufgabe wurden 3 Netzwerkarchitekturen vorgegeben:

- Eine Baseline, welche aus einer sogenannten versteckten Schicht (Hidden Layer) mit 500 Neuronen besteht.
- Eine an der "lenet" angelehnte Architektur, bestehend aus 5 aufeinanderfolgende Faltungsschichten (Convolutional Layer), gefolgt von 5 voll verbundenen Schichten (Fully Connected- oder auch Dense Layer)
- Eine am vgg16 angelehnte Architektur.

Die Netzarchitekturen werden im Abschnitt Netzarchitekturen näher vorgestellt.

Für Datenset diente das in diesem [Kaggle Wettberb] bereitgestellte Datenset, welches 96 x 96 Pixel Bilder von zentrierten Gesichtern, sowie dazugehörige 15 Facial Keypoint Annotationen beinhaltet.

Datenset

Das in dieser Projektarbeit verwendete Datenset stammt aus diesem Kaggle Wettbewerb. Es beinhaltet 7049 annotierte Grauwert Bilder der Größe 96x96.

Lediglich ca. 2000 der Bilder waren vollständig annotiert, das bedeutet nur bei diesen waren alle 15 Keypoints vorhanden. In der Gruppe entschied man früh, dass auch nur diese 2000 Bilder für das Training verwendet werden sollen.

Kaggle stellte hier drei verschiedene .csv Dateien zu Verfügung:

- training.csv
- test.csv
- SampleSubmission.csv

Für das Training wurde die "training.csv" in ein Trainings- und ein Testdatenset aufgesplittet. Als Verhältnis wurde hier aufgrund der wenigen Daten 9:1 gewählt.

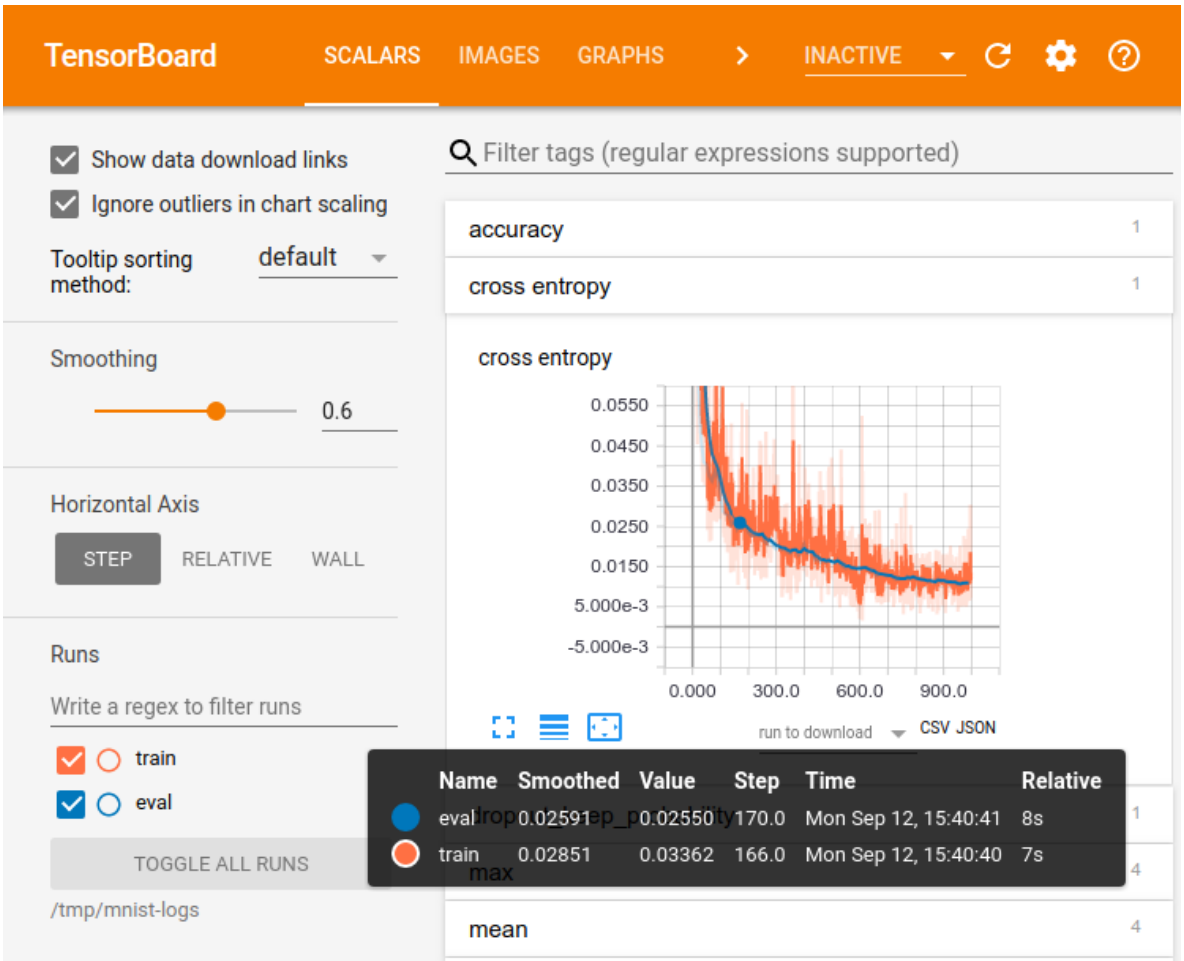
Tensorflow

Tensorflow ist ein von Google entwickeltes Open Source Framework, das in Python und C++ geschrieben ist. Es richtet sich hauptsächlich an Entwickler im Bereich des maschinellen Lernens. Google verspricht sich aufgrund der Verwendung von sogenannten Tensoren eine besonders effizienten Berechnung. Laut Google zeichnet sich Tensorflow dadurch aus, dass sowohl auf einem "High-" als auch auf einem "Low-Level" Anwendungen programmiert werden können. Dies liegt unter anderem am großen Umfang von Tensorflow API's. Während beispielsweise die Verwendung der Session API es dem Entwickler ermöglicht, grundlegende mathematische Funktionen selbst zu definieren, bietet zugleich die KERAS API einen Ansatz, der bereits mit vorgefertigten Schichtenimplementierungen daherkommt.

In Tensorflow müssen Netze entweder in Python oder C++ implementiert werden. Hier liegt ein grundlegender Unterschied zum Caffe Framework, welches eine Netzdefinition mithilfe einer Konfigurationsdatei ermöglicht.

Tensorboard

Tensorboard ist eine zusätzlich Bibliothek die mit der Installation von Tensorflow mitgeliefert wird. Das Tensorboard kann dazu genutzt werden, um verschiedene Variablen und Trainingsverläufe zu visualisieren.



Im Folgenden wird anhand eines Code Snippets gezeigt, welche Schritt notwendig sind, um eine Variable in das Tensorboard zu schreiben. Der Code ist der Python Datei *lenet.py* entnommen.

```
import tensorflow as tf

with tf.name_scope("loss"):
    loss = tf.reduce_mean(tf.losses.mean_squared_error(labels=self.y, predictions=prediction))
    tf.summary.scalar("mse", loss)
summ = tf.summary.merge_all()
.
.
.
sess.run(tf.global_variables_initializer())
writer = tf.summary.FileWriter('../tmp/facial_keypoint/le_net/{}epochs_{}bs_Adam_lr{}_{}'.format(epochs,
batch_size, learning_rate, time))
writer.add_graph(sess.graph)
```

```
.
.
.
s = sess.run(summ, feed_dict={self.x: batch_x, self.y: batch_y})
writer.add_summary(s, epoch)
```

- Schritt 1: Tensorflow importieren
- Schritt 2: Einen sogenannten "Scope" `tf.name_scope("loss")` benennen, in dessen Rahmen die Variable im Tensorboard angelegt werden soll. Dadurch wird das Board übersichtlicher und die Variablen können schneller gefunden werden. Dies ist insbesondere beim Visualisieren eines Graphen besonders hilfreich.
- Schritt 3: Eine Tensorflow Variable definieren. Im Fallse des obigen Beispiels ein Loss. Diese kann per `tf.summary.scalar("mse", loss)` als Skalar unter dem Namen *mse* geloggt werden.
- Schritt 4: Per `tf.summary.merge_all()` werden alle bisher definierten Tensorboard Logs "zusammengefasst". Dies ermöglicht nun das Ausführen des Loggings.
- Schritt 5: Mit `tf.summary.FileWriter()` wird ein sogenannter "Writer" angelegt. Mithilfe von diesem können die logs in einer logging Datei persistiert werden.
- Schritt 6: Mit `sess.run()` und der zuvor zusammengelegten `summ` als Parameter wird ein Inferenzschritt ausgeführt und die Ergebnisse weggeschrieben. Mit `add_summary` werden diese dann an das Tensorfboard übergeben und sind dort sichtbar.

Installation

Für den Versuch war ein Jetson TX 2 Evalboard vorgesehen. Auf diesem kann die entsprechende Firmware "Jetpack" nach dieser Anleitung installiert werden.

Für die Verwendung von Tensorflow wird eine speziell für die Tegra Architektur gebaute Version beötigt. Diese kann mithilfe von pip installiert werden.

```
pip install --extra-index-url https://developer.download.nvidia.com/compute/redist/jp33 tensorflow-gpu
```

Für nähere Informationen zu Tensorflow auf dem Jetson TX2 empfiehlt sich die Dokumentation von NVIDIA

Weitee im Rahmen der Projektarbeit verwendeten Pakete können der im Projekt enthaltenen *requirements.txt* entnommen werden. Dabei ist darauf zu achten, dass sich ggf. nicht alle der Pakte über den Paketmanager pip installieren lassen.

Projektdateien

Datei	**Beschreibung**
create_pb_from_ckpt.py	Wandelt die durch Training erstellten Checkpoints in protobuf Dateien um-
data_loader.py	Zuständig für das Laden, Konvertieren und Abspeichern der Annotationen
image_extraction.py	Extrahieren der Bilder aus der csv-Datei
lenet.py	Implementierung der LeNet-Architektur
one_layer.py	Implementierung der Baseline
vgg14.py	Implementierung der VGG-Architektur
test_inference.py	Ermöglicht das Testen der eingefrorenen Gewichte

Netzarchitekturen

Arch_Baseline			**Arch_LeNet**			**Arch_VGG**		
				conv2d: 3x3, pool: 2x2, stride 1				
Layer	Name	Size	Layer	Name	Size	Layer	Name	Size
0	input	1x96x96	0	input	1x96x96	0	input	1x96x96

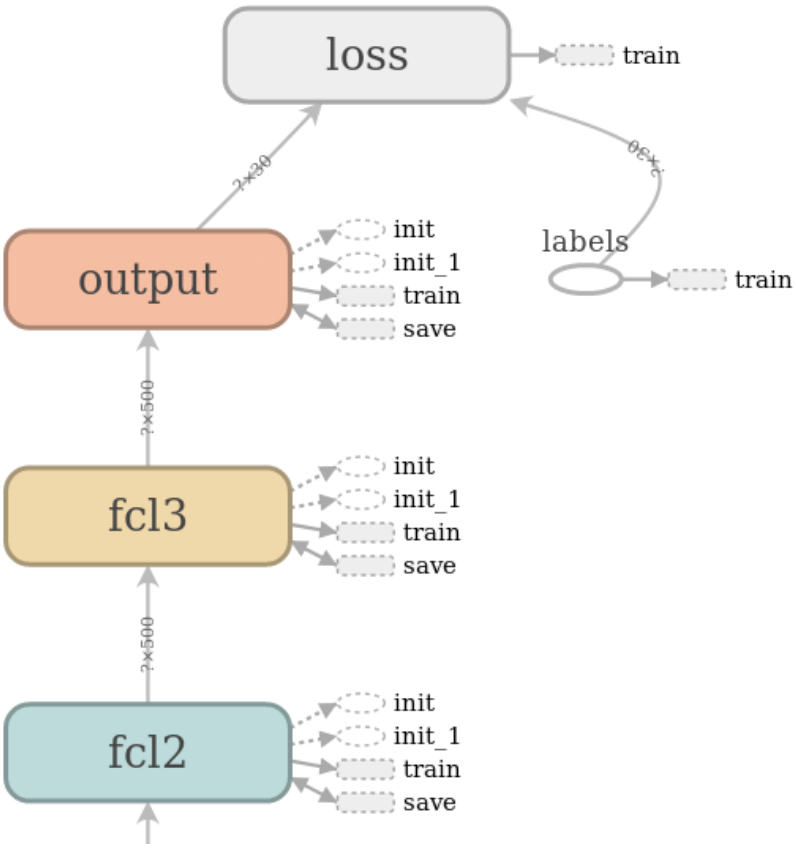
Arch_Baseline			Arch_LeNet			Arch_VGG		
1	hidden	500	1	conv2d	16x94x94	1	conv2d	64x94x94
2	output	30	2	maxpool2d	16x47x47	2	conv2d	64x92x92
			3	conv2d	32x45x45	3	maxpool2d	64x46x46
			4	maxpool2d	32x22x22	4	conv2d	128x44x44
			5	conv2d	64x20x20	5	conv2d	128x42x42
			6	maxpool2d	64x10x10	6	maxpool2d	128x21x21
			7	conv2d	128x8x8	7	conv2d	256x19x19
			8	maxpool2d	128x4x4	8	conv2d	256x17x17
			9	conv2d	256x2x2	9	maxpool2d	256x8x8
			10	maxpool2d	256x1x1	10	Dense	512
			11	Dense	500	11	dropout	512
			12	Dense	500	12	Dense	512
			13	Dense	500	13	dropout	512
			14	Dense	500	14	output	30
			15	output	30			

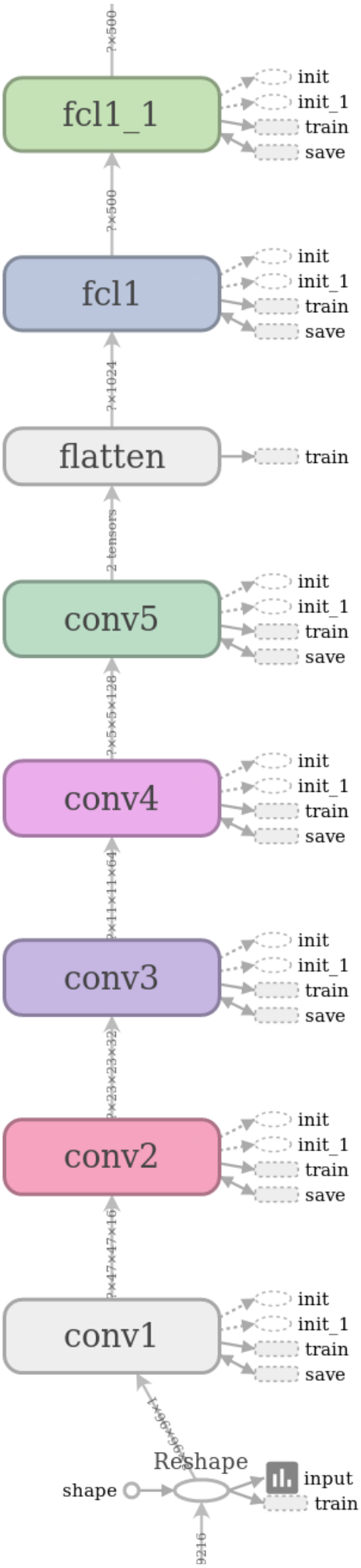
Baseline

Die Baseline besteht aus einer Eingangsschicht, gefolgt von einer mit 500 Neuronen bestückten versteckten Schicht und einer Ausgabeschicht.

LeNet-Architektur

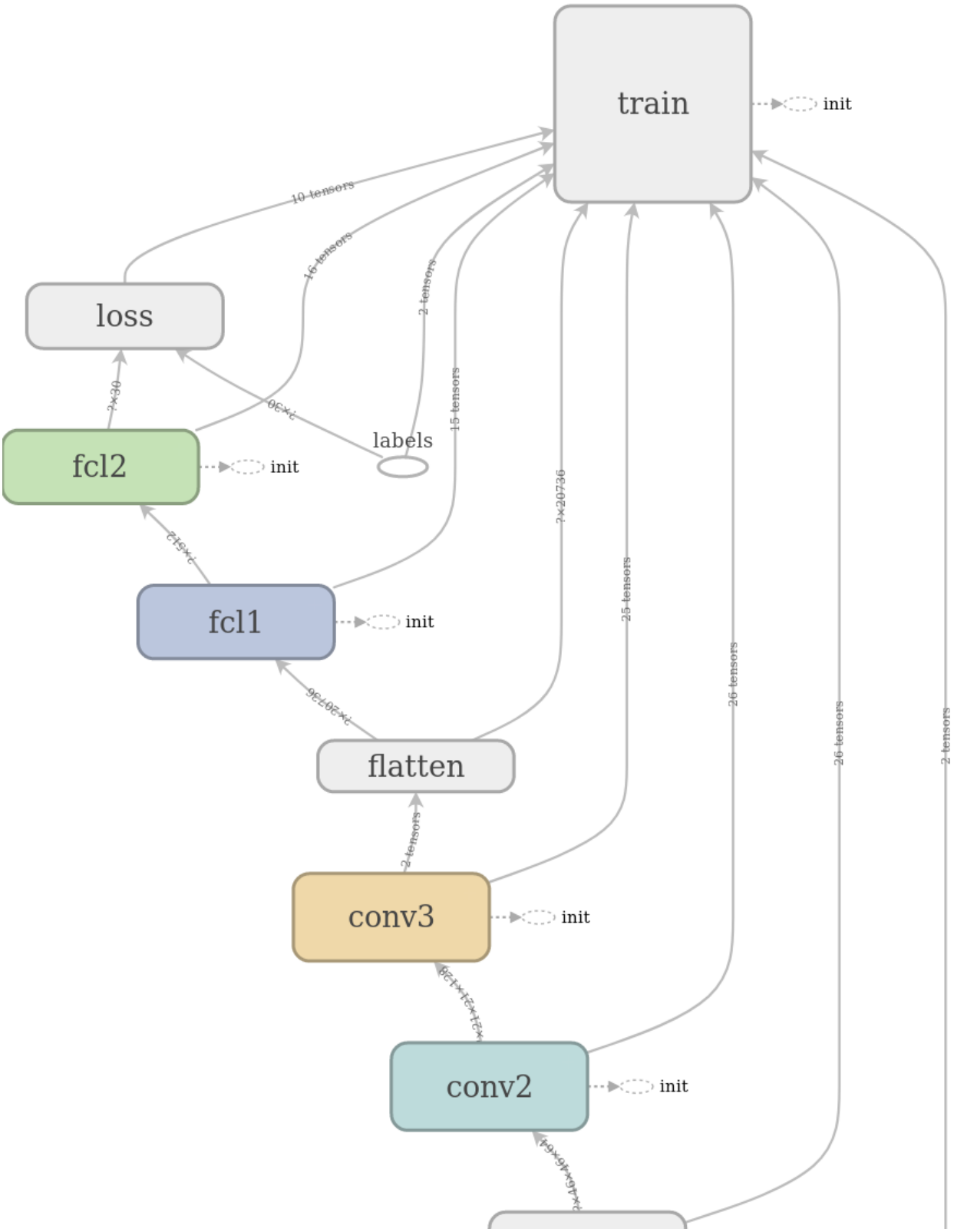
Die LeNet Architektur besteht aus 5 mal aufeinanderfolgenden Faltungs- und Max Pooling- Schichten. Anschließend folgen 5 *Dichte Schichten* (Dense Layer) bestehend auf 500 Neuronen. Die Ausgabeschicht liefert auch hier 15 x,y Punkte (30 Werte) stellvertretend für die Keypoints.

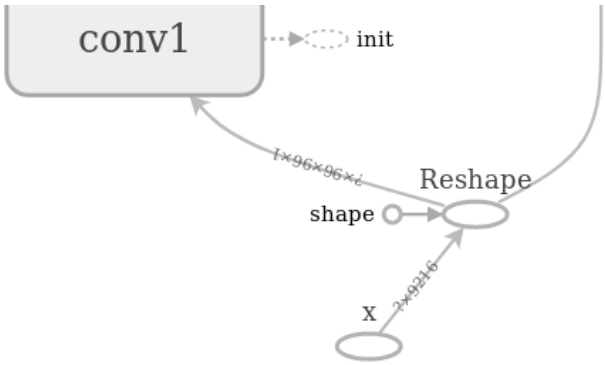



?×500
fcl1_1
init
init_1
train
save
?×500
fcl1
init
init_1
train
save
?×1024
flatten
train
2 tensors
conv5
init
init_1
train
save
?×5×5×128
conv4
init
init_1
train
save
?×11×11×64
conv3
init
init_1
train
save
?×23×23×32
conv2
init
init_1
train
save
?×47×47×16
conv1
init
init_1
train
save
1×96×96×1
Reshape
shape
input
train
9216

VGG-Architektur

Bei der VGG-Architektur wiederholt sich das Pattern von 2 aufeinanderfolgenden Faltungsschichten und einer Max Pooling-Schicht drei mal. Anschließend folgen 2 "Fully Connected" Schichten mit einer Dropout-Eigenschaft. Das bedeutet, ein Teil der Ergebnisse aus den vorherigen Werte wird verworfen.





Die Dopout Schichten wirken regularisierend auf die Trainingsprozedur, wodurch das Netz besser verallgemeinern kann.

Exemplarische Implementierung eines Mehrschichtenmodell in Tensorflow

Für die exemplarische Darstellung eines Mehrschichtenmodells wird im folgenden ein Codeausschnitt aus der *lenet.py* -Datei vorgestellt.

Bei Tensorflow unterscheidet man zwichen 2 grundlegenden Datentypen: **Variables** `tf.Variable` und **Placeholder** `tf.placeholder`. Variables sind jene Variablen, die Tensorflow selbst während eines Trainings verändern kann. Im folgenden Code Beispiel sind dies die Gewichte oder der Bias. Placeholder hingegen sind Variablen, die der Nutzer "Nutzer" verändert. Sie werden genutzt um Ein- oder Ausgabedaten zu verwenden. Placeholder können von Variabler Größe definiert werden. Dies geschieht, indem man `None` als Größe übergibt:

```
self.input = tf.placeholder(tf.float32, [None, 9216], name="x")
self.output = tf.placeholder(tf.float32, [None, 30], name="labels")
```

Tensorflow ermöglicht auch die Definition eigener Schichten. Um dies zu veranschaulichen und den Programmieraufwand zu minimieren wurde im Folgenden ein eigenes "Convolutional Layer" definiert. Dazu werden mithilfe von `tf.Variable` `weights` und `bias` definiert. Die Gewichte (weights) werden dann an die conv2d-Schicht, die in Tensorflow implementierte zweidimensionale Convolution übergeben, sodass ein Optimierer diese später anpassen kann. Als Aktivierungsfunktion kommt eine ReLu (Rectifier Linear Unit) zum Einsatz. Sie bekommt die Ausgabe der Faltung sowie einen Bias übergeben. In Tensorflow wird eine ReLu mithilfe von `tf.nn-relu` erzeugt.

```
def conv_layer(self, input, size_in, size_out, name="conv"):
    with tf.name_scope(name):
         weights = tf.Variable(tf.truncated_normal([3, 3, size_in, size_out], stddev=0.1), name="weights")
         biases = tf.Variable(tf.constant(0.1, shape=[size_out]), name="biases")
         conv = tf.nn.conv2d(input, weights, strides=[1, 1, 1, 1], padding="VALID")
         act = tf.nn.relu(conv + biases)
         tf.summary.histogram("weights", weights)
         tf.summary.histogram("biases", biases)
         tf.summary.histogram("activations", act)
    return tf.nn.max_pool(act, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding="SAME")
```

Die vorher definierte Schicht kann nur verwendet werden, um ein Mehrschichtenmodell zu erzeugen. Der mögliche Aufbau ist im folgenden Codeausschnitt zu sehen:

```
def le_net_model(self, x):
    conv1 = self.conv_layer(x, 1, 16, "conv1")
    conv2 = self.conv_layer(conv1, 16, 32, "conv2")
    conv3 = self.conv_layer(conv2, 32, 64, "conv3")
    conv4 = self.conv_layer(conv3, 64, 128, "conv4")
    conv5 = self.conv_layer(conv4, 128, 256, "conv5")

    # Flatten the array to make it processable for fc layers
    flattened = tf.layers.Flatten()(conv5)  #
    fc1 = self.fc_layer(flattened, int(flattened.shape[1]), 500, "fc1")
    fc2 = self.fc_layer(fc1, 500, 500, "fc1")
```

```
        fcl3 = self.fc_layer(fcl2, 500, 500, "fcl2")
        fcl4 = self.fc_layer(fcl3, 500, 500, "fcl3")
        output = self.fc_layer(fcl4, 500, 30, "output")

        return output
```

Performanz

Ergebnisse Jetson TX2

	Average Train Loss	**Average Test Loss**	**Average Time**
Baseline	1.28 * 10^7	4,191 * 10^7	~16:26 min
LeNet	4.47	10.93	~93:00 min
VGG	-	-	-

Aufgrund mehrer Probleme beim Training, konnte auf dem Jetson lediglich vereinzelt Werte für die Baseline sowie die LeNet Architektur gemessen werden. Um trotzdem Werte zu liefern, wurde das Training aller drei Architekturen auf einer GTX1070 erneut ausgeführt.

Ergebnisse GTX 1070

	Average Train Loss	**Average Test Loss**	**Average Time**
Baseline	1.77 * 10^7	5.2 * 10^7	~2:38 min
LeNet	4.51	10.156	~7:50 min
VGG	827.7494	866.4116	~40:32min

Probleme

Während der Projektarbeit ergaben sich mehrere Probleme, die teilweise behoben werden konnten. Eine detaillierte Beschreibung folgt:

Jetson TX2

Der Jetson TX2 ist eine erste Lösung, um Machine Learning auf Embedded Systemen zu verwenden. Ein Training in Verbindung mit Tensorflow gestaltet sich jedoch äußerst Schwierig. Zum einen, bringt der Jetson für das Training tieferer nmeuronaler Netze nicht ausreichend Leistung mit. Dies sorgte dafür, dass das Training der VGG Architektur nach kürzester Zeit mit CUDA Errors abgebrochen wurde. Auch das Training der beiden anderen Architekturen führte zu Problemen, sodass nicht das komplette Training durchgeführt wurden konnte. Kennzahlen hinsichtlich Loss und Geschwindigkeit der korrekt durchgeführten Trainingsdurchläufe können in der Sektion Performanz gefunden werden.

Session-API vs. Keras

Die Session API ist die LowLevel-Api von Tensorflow. Sie ermöglich zwar sehr viele Einstellungsmöglichkeiten, erfordert jedoch eine sehr genau Kenntnis dieser. Dies machte es ungemein schwer einfache Netze zu implementieren und vorallem eine komplette ToolChain auf die Beine zu stellen. So verursachgte das Abspeichern der Gewichte und das Wiederverwenden für die Inferenz einige Probleme, bei denen zu wenig oder falsche Gewichtswerte eingefroren wurden. Um die beiden Frameworks (Caffe und TF) auf Augenhöhe zuv ergleichen, sollte man ggf. Keras hinzuziehen. Dies ermöglicht das Konzentrieren auf den für die Projektarbeit relevanten Teil, nämlich der Vergleich der beiden Frameworks.

Sehr unterschiedliche Loss-Werte

Die drei unterschiedlichen Architekturen liefern sehr unterschiedliche Loss-Werte. Dies ist speziell der Caffe nicht der Fall. Ob dies nun ein Implementierungsfehler in der Tensorflow Version der Netze ist konnte noch nicht herausgefunden werden.

Dropout Schichten sehr rechenaufwändig

Bei Verwendung einer einzelnen Dropout Schicht konnte festgestellt werden, dass diese die benötigte Zeit für den Forwardpass um 25% erhöht. Bei zwei Dropout Schichten verschlechtert sich die benötigte Zeit weiter deutlich. Auch andere Nutzer konnten dies festellen und machten die zusätzlichen Matrizen, die benötigt werden dafür verantwortlich https://stats.stackexchange.com/a/377126 .

Unterschiedliche Loss Funktionen als Standard

Die Projektteilnehmer hatten sich auf die Summe der Quadratfehler als Kostenfunktion geeinigt. Im Caffe Framework heißt diese "Euclid" in Tensorflow kann diese mithilfe von `tf.reduce_mean(tf.losses.mean_squared_error())` errechnet werden. Zu spät wurde dann allerdings festgestellt, dass diese Implementierungen nicht identisch sind. Bei Caffe wird stattdessen nach dieser Formel das Loss berechnet:

$$\frac{1}{2N} \sum_{i=1}^{N} \|x_i^1 - x_i^2\|_2^2.$$

Fazit

Tensorflow (TF) bietet mit unterschiedlichen API's unterschiedliche Möglichkeiten und Komplexitätsgrade. Für den Einsteiger beziehungsweise für Anwender, die den Schwerpunkt auf Architekturen legen bietet TF die Keras API. Sie erlaubt das Bauen eigener Netze mit Schichten aus einem Baukausten. Doch auch dieser wächst immer weiter, weswegen man nicht mehr von einer Einschränkung sprechen kann.

Die in dieser Arbeit verwendete Session API setzt an unterster Stelle (Low Level) an und erlaubt es jegliche Schichten und Architekturen selbst zu entwerfen. Sie richtet sich an erfahrene Anwender.