Concolutional Neural Network for classification of handwritten digits

Zusammenfassung—In diesem Projekt haben wir ein *convolutional neural network* zur Klassifikation von handgeschriebenen Ziffern implementiert. Wir haben zwei *convolutional* Schichten, gefolgt von zwei *fully-connected* Schichten in unserer Architektur umgesetzt. Nach jeder Faltung addieren wir ein *bias* und wenden dann als Aktivierung die *ReLU* an. Dann wird jeweils *max-pooling* dahinter geschaltet um die Anzahl der Parameter zu reduzieren, ohne den Verlust der wesentlichen Merkmale. Auch bei den *fully-connected* Schichten kommt *ReLU* zum Einsatz und zum Schluss benutzen wir *softmax* um auf die zehn Klassen ein *scope* auszugeben. Trainiert und getestet wurde mit dem MNIST Datensatz, dabei diente TensorFlow als Framework zur Modellierung des Netzes der in Python implementiert wird. Wir haben uns auf einige wesentliche Fragestellungen bezüglich der Architektur konzentriert, um am Ende der Analyse dieser Fragen, eine möglichst gute CNN Architektur zur Klassifikation von handgeschriebenen Ziffern zu erhalten.

Index Terms—Machine Learning, Deep Learning, CNN, Neural Network, MNIST, handwritten digits

1 Einleitung

Im Rahmen des ersten praktischen Projekts der Vorlesung „Machine Learning II - Deep Learning“ haben wir ein Convolutional Neural Network, kurz CNN, selbst programmiert und umfangreich getestet. Als Trainings- und Testdatensatz nutzen wir dazu den bereits vom verwendeten Framework „Tensorflow“ mitgelieferten MNIST Datensatz „Handwritten Digits“Durch Einsatz verschiedener Kombinationen von convolutional layers und pooling layers konnten wir die darin enthaltenen handschriftlich notierten Zahlen klassifizieren. Die Fehlerrate konnten wir durch den Einsatz von zwei convolutional layers und zwei pooling layers mit einer learning rate von 0.01 bzw. einer adaptive Rate bis auf 2% beschränken.

2 Die Architektur

Unsere Architektur besteht insgesamt aus sechs Layern. Dabei ist unsere Architektur in einen Teil von Convolutional und Pooling Layern und einen Teil Fully Connected Layer unterteilt. Unsere beiden Convulutional Layer haben jeweils ReLU als Aktivierungsfunktion und einen nachgestellten Pooling Layer zur Reduzierung der Datenmenge. Auf dieser Layer folgen zwei Fully Connected Layer, wobei zwischen den Layern noch Dropout verwendet wird.

Nun zu der Entstehung unserer Architektur. Wir haben unser neuronales Netz ausgehend von der vorherigen Übung aufgebaut, dabei haben wir schrittweise mehr Layer hinzugefügt und die Funktionalität unseres Netzes nach jedem Schritt überprüft. Zuerst haben wir einen Convulutional Layer samt Aktivierungsfunktion eingefügt. Daraufhin haben wir, wie es in der Vorlesung schon beschrieben wurde, einen Pooling Layer hinter die Convulution geschaltet. Dies haben wir wiederholt und einen zweiten Convulutional Layer samt Aktivierung und Pooling eingefügt. Diesmal mit einer anderen Filtergröße. Um unsere Ergebnisse weiter zu verbessern, haben wir einen zweiten Fully Connected Layer vor den Ausgabelayer geschaltet. Außerdem kam als letztes noch Dropout zwischen den beiden Fully Connected Layern als Funktion dazu.

Unsere Convulutional Layer nutzten in unseren Versuchen Filter in den Größen zwischen 3×3 und 7×7, wobei wir in unserer finalen Konfiguration nicht über eine Filtergröße hinausgehen. Außerdem haben wir in den Layern mit der Anzahl der Filter zwischen 16 und 64 experimentiert, wobei bereits mit 16 Filtern eine Genauigkeit von über 90% erzielt wurde.

Als Pooling nutzen wir 2×2-max-Pooling mit 2er Strides zur Reduzierung der Auflösung der Inputdaten. Damit brechen wir die Daten insgesamt von 28×28 Pixel auf 7×7-Pixel herunter. Dies wirkt sich enorm positiv auf die Laufzeit unseres Netzes aus. Eine weitere Reduzierung der Auflösung war uns jedoch nicht möglich, da 7 prim ist.

Bei unserem extra Fully Connected Layer haben wir mit der Anzahl der Neuronen bei 50 begonnen und sind bis 128 hochgegangen.

Die *keep probability* bei unserem *Dropout* haben wir zwischen 50% und 90% schwanken lassen, ohne dass sich Auswirkungen auf das Ergebnis gezeigt hätten. Generell war unsere Architektur maßgeblich durch den verfügbaren Arbeitsspeicher der Hardware begrenzt. Bei Versuchen mit mehr Convulution Filtern oder einer höheren Anzahl an Neuronen in den Fully Connected Layern traten sehr schnell Speicherfehler auf, welche zum Absturz unseres Netzes führten.

Nun noch einmal unsere Architektur in Kurzform:

C(32, 5, 1)-ReLU-P(2, 2)-C(64, 3, 1)-ReLU-P(2, 2)-F(128)-D(.9)-F(10)

- $C(X, Y, Z)$ Convulution mit X Filtern, Y quadratischer Filtergröße, Z Stride
- ReLU Rectified Linear Unit
- $P(X, Y)$ Pooling mit X quadratischem Filter und Y Stride
- $F(X)$ Fully Connected mit X Neuronen
- $D(X)$ Dropout mit keep probability X

3 Das Training

In einem Versuch haben wir die beiden von uns verwendeten Max-Pooling-Layer entfernt. Da wir in unseren Convolutional Layern stets Stride 1 verwenden, führt das Entfernen der Pooling-Layer natürlich dazu, dass das gesamte Netzwerk zunächst die Inputgröße beibehält.

Dadurch vergrößert sich die Trainingszeit bedeutend. In unserem Fall haben wir im Schnitt einen zeitlichen Mehraufwand von Faktor 2 bis 3 gemessen. Die Accuracy veränderte sich dabei nicht bedeutend. In den meisten Fällen war sogar eine Verschlechterung der Accuracy zu verzeichnen.

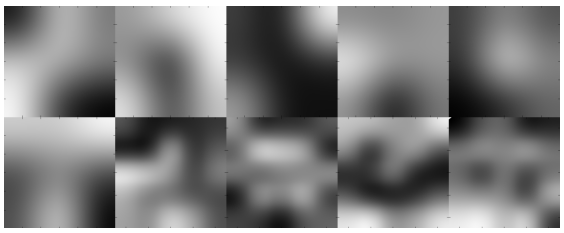

Abbildung 1. Gelernte Filter aus der ersten und zweiten Faltungs-Schicht.

Im folgenden werden unsere Ergebnisse mit unterschiedlichen Learning Rates beschrieben, dabei wurde der AdamOptimizer genutzt. Die zufällige Initialisierung unseres Netzes erreichte in der ersten Runde des Trainings immer eine ungefähre Trefferrate von 10%. Die schlechtesten Ergebnisse erhielten wir mit einer Learning Rate von 1. Dabei hat das Netz durch Training keinerlei Verbesserung erfahren und erzielte weiterhin Klassifizierungsraten von circa 10%.

Daraufhin haben wir die Learning Rate im nächsten Schritt auf 0.5 verringert. Auf die Qualität des Netzes hatte dies keinen Einfluss - die Ergebnisse waren genauso schlecht wie vorher. Auch bei einer Learning Rate von 0.1 erzielten wir eine Klassifikationsrate von 0.1. Die besten Ergebnisse lieferte uns jedoch eine Learning Rate 0.01 bzw. eine nicht-eingestellte Learning Rate, die vom Netz selbst angepasst wird. Dabei erreichten wir eine Genauigkeit von 98% auf den Testdaten.

4 Test und Auswertung

Wir haben beobachtet, dass die Test-Accuracy am Anfang relativ schnell auf über 90% steigt und dann gegen einen Wert, nahe bei 100% strebt. Besonders mit einer festen Learningrate stellt man Schwankungen in der Training-Accuracy fest, dies wird bei adaptiver Leraning-Rate und bei beobachtung der Test-Accuracy deutlich stabiler.

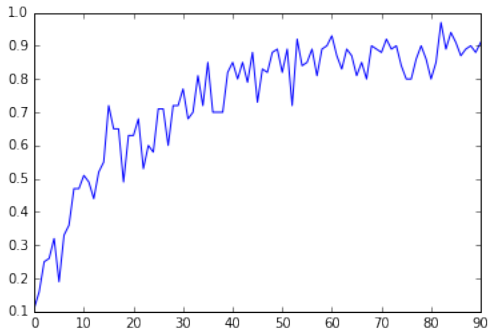

Abbildung 2. Training *accuracy* mit konstanter Lernrate in 90 iterationen, Batch-Größe 100.

Um eine Test-Accuracy von über 98% zu erreichen, benötigt ein Rechner (z.B. aktueller Laptop) eine Rechenzeit von einigen wenigen Minuten. Die Trainingsdaten werden gut generalisiert, sodass keine zusätzliche *overfitting*-Prevention nötig ist. Bei gleicher Konfiguration bekommt man dennoch leicht unterschiedliche Ergebnisse, weil die Initialisierung zufällig ist. Dennoch kann man insgesamt ein konvergentes Verhalten beobachten.

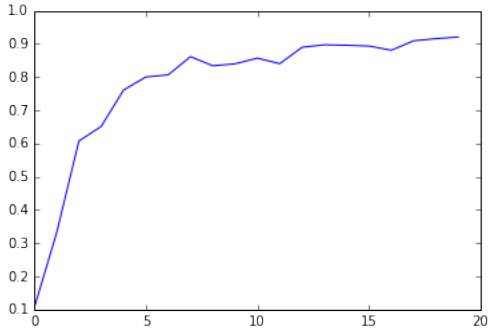

Abbildung 3. Test *accuracy* bei adaptiver Lernrate.

Anhang A
Code-Schnipsel

```python
def weight_variable(shape):
    """Creates a tf Variable of shape 'shape' with
    random elements"""
    initial = tf.truncated_normal(shape, stddev=0.1)
    return tf.Variable(initial)
```

Listing 1. Erzeugt ein Gewichts-Array mit zufällig initialisierten Werten. Die Dimensionen werden als Parameter entegengenommen.

```python
def bias_variable(shape):
    """Create a small bias of shape 'shape' with
    contants values of 0.1"""
    initial = tf.constant(0.1, shape=shape)
    return tf.Variable(initial)
```

Listing 2. Erzeugt ein bias-Array mit zufällig initialisierten Werten. Die Dimensionen werden als Parameter entegengenommen.

```python
def conv2d(x, W):
    """Use a conv layer with weights W on zero
    padded input x and stride 1"""
    return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1],
    padding='SAME')
```

Listing 3. Eine zweidimensionale Faltung mit Eingabe x und Filter W.

```python
def max_pool_2x2(x):
    """Create a 2x2 pooling layer with strides 2,
    reducing the resolution"""
    return tf.nn.max_pool(x, ksize=[1, 2, 2, 1],
    strides=[1, 2, 2, 1], padding='SAME')
```

Listing 4. Hier wird 2x2 max pooling auf die Eingabe x angewendet.

```python
# x is 1x28*28 ( x = [34,43,5,6,0,0,7,5,...] )
# W_conv1 is 5x5
# b_conv1 is 1x32
W_conv1 = weight_variable([5,5,1,32])
b_conv1 = bias_variable([32])
# input data is embedded in 4d tensor
x_image = tf.reshape(x, [-1, 28, 28, 1])
# h_conv1 is result of applied conv filter W to x
h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) +
    b_conv1)
h_pool1 = max_pool_2x2(h_conv1)

# Apply second convolutional layer
W_conv2 = weight_variable([3, 3, 32, 64])
b_conv2 = bias_variable([64])
h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2) +
    b_conv2)

h_pool2 = max_pool_2x2(h_conv2)

#Fully connected layer
W_fc1 = weight_variable([7 * 7 * 64, 128])
b_fc1 = bias_variable([128])
h_pool2_flat = tf.reshape(h_pool2, [-1, 7*7*64])
h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) +
    b_fc1)

keep_prob = tf.placeholder(tf.float32)
h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)

W_fc2 = weight_variable([128, MNIST_CLASSES])
b_fc2 = bias_variable([MNIST_CLASSES])

y_conv = tf.matmul(h_fc1_drop, W_fc2) + b_fc2
```

Listing 5. Modellierung des Berechnungsgraphen.

LITERATUR

[1] TODO: Namen einfügen, *Vorlesung: Machine Learning II - Deep Learning*, WS 2016/17