IS Praktikum 4: Neuronale Netze

Neuronale Netze zählen zu den spannendsten und vielfältigsten Algorithmen aus dem Machine Learning. Aktuell erfahren neuronale Netze in der Forschung und Industrie eine enorme Aufmerksamkeit, da sich durch immer schnellere Hardware viele komplexe Anwednungsfälle auf Basis von Deep-Learning-Methoden lösen lassen.

In diesem Praktikum werden aktuelle Methoden zum Training von neuronalen Netzen an einem einfachen Beispiel vermittelt. Folgen Sie dazu bitten den aufeinander aufbauenden Aufgaben 1, 2 und 3.

1. Vorbereitung

Im Folgenden werden Informationen zum verwendeten Framework, dem Datensatz und dem zu erstellenden Protokoll aufgeführt.

Keras

Keras ist ein von Francios Chollet (@fchollet) entwickeltes Python-Framework zur einfachen Implementierung von neuronalen Netzen. Dafür bietet Keras eine high-level API als Abstraktionsschicht über TensorFlow oder Theano. Keras ist besonders beliebt für schnelle Experimente, da die API viele Best-Practices beinhalten und damit einen sehr übersichtlichen Code ermöglicht.

```
import keras
```

Keras bietet grundsätzlich eine funktionale und eine sequenzielle API. Für die Versuche in diesem Praktikum wird die sequenzielle API von Keras verwendet.

```
from keras.models import Sequential
```

Datensatz

In diesem Praktikum wird der MNIST-Datensatz zur Erkennung von handschriftlichen Ziffern verwendet. Der Datensatz enthällt dabei 70.000 Bilder die jeweils aus einer 28×28 Pixel-Matrix bestehen.

Keras bietet eine einfache Funktion zum Herunterladen und Bereitstellen des Datensatzes:

```
from keras.datasets import mnist

# Laden des Datensatzes und Aufteilung in 60.000 Trainings- und 10.000 Testdaten
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

Laden Sie den Datensatz und schauen Sie sich einige Exemplare mithilfe des folgenden Plots an:

```
import matplotlib.pyplot as plt

# Plot eines ausgewählten Beispieles in den Trainingsdaten
example_idx = 0

plt.title("Label: {}".format(y_train[example_idx]))
plt.imshow(x_train[example_idx], cmap='gray')
plt.show()
```

Eine interaktive Visualisierung finden Sie online im Embedding-Projector von TensorFlow.

Hinweise zum Protokoll

Das Protokoll soll jeweils eine knappe Beschreibung der Teilaufgabe und die dazugehörigen Plots enthalten. Auch sollen in dem Protokoll die Beobachtungen zu den Aufgaben dokumentiert werden. Dazu können Sie jeweils die Genauigkeit der Modelle festhalten und den Effekt der Anpassung am Modell auf das Ergebnis beschreiben. Wie üblich, soll das Protokoll spätestens eine Woche nach dem Praktikum, also am selben Wochentag, 23:59 Uhr, eingereicht werden. Die Abgabe von Code ist in diesem Fall nicht notwendig.

2. Feedforward Neural Network

Zunächst soll ein Klassifikator auf Basis eines einfachen neuronalen Netzes realisiert werden. Dabei muss beachtet werden, dass die Bilder des Datensatzes Form einer Pixel-Matrix vorliegen.

Für die folgenden Versuche müssen zunächst ein paar Parameter festgelegt werden. Dazu zählt die Anzahl der parallel zu verarbeitenden Datensätze (`BATCH_SIZE`), die Anzahl der Klassen in den Daten (`NUM_CLASSES`) und die Anzahl der Training-Iterationen über die vorhandenen Trainingsdaten (`EPOCHS`).

```
BATCH_SIZE = 128
NUM_CLASSES = 10
EPOCHS = 200
```

Der Parameter `BATCH_SIZE` beläuft sich somit auf die Anzahl der Datensätze pro Mini-Batch, also den Trainingsdaten, die zur Berechnung eines Update-Schrittes im Gradientenverfahren verwendet werden.

Das nun zu implementierende neuronale Netz kann lediglich mit einem Input-Vektor pro Datensatz in der Mini-Batch umgehen. Aus diesem Grund muss die Pixel-Matrix der Bilder in den Trainingsdaten jeweils in einem Vektor repräsentiert werden. Dazu können die Zeilen der Pixel-Matrix einfach aneinandergereiht weren.

```
# Transformation der 28x28 Bilder als Vektoren der Länge 28*28
x_train = x_train.reshape(60000, 784)
x_test = x_test.reshape(10000, 784)
```

Damit das Gradientenverfahren möglichst schnell konvergiert, können die Daten normalisiert werden.

```
# Normalisierung der Daten zwischen 0 und 1
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255
```

Für die Vorhersage der Wahrscheinlichkeitsverteilung der Klassen müssen zudem die als Integer vorliegenden Klassen (zwsichen 0, 9) in einem binären **One-Hot**-Vektor dargestellt werden. Beispielsweise:

$$[5] \rightarrow [0, 0, 0, 0, 0, 1, 0, 0, 0, 0]$$

$$[2] \rightarrow [0, 0, 1, 0, 0, 0, 0, 0, 0, 0]$$

```
# Encodieren der Integer-Klassen (0-9) in One-Hot Vektoren
y_train = keras.utils.to_categorical(y_train, NUM_CLASSES)
y_test = keras.utils.to_categorical(y_test, NUM_CLASSES)
```

2.1 Einfaches Netz mit Softmax-Funktion

Für den ersten Versuch soll ein Netz konzipiert werden, welches direkt aus der Eingabe-Schicht die Wahrscheinlichkeiten zur Vorhersage der Klassen berechnet. Dies kann in Form einer Softmax-Schicht erfolgen:

$$Y = softmax(X * W + b)$$

Dabei ist Y eine $m \times c$ Matrix, wobei m die Anzahl der Datensätze in der Mini-Batch ist und c die Anzahl der möglichen Klassen beschreibt. X ist die gewohnte $m \times n$ Matrix, wobei n nun die Anzahl der Pixel in den zu Klassifizierenden Daten ist. W ist dann die $n \times c$ Gewichts-Matrix und b der Bias-Vektor der Länge c.

Die Softmax-Funktion sorgt nun dafür, dass jedes der c Elemente in einem Zeilenvektor von Y die Wahrscheinlichkeit für eine der c Klassen vorhersagt und die Summen aller Elemente der Zeilenvektoren 1 ergeben. Somit ergibt jede Zeile von Y eine Wahrscheinlichkeitsverteilung zur Vorhersage der Klasse des jeweiligen Bildes in der Mini-Batch.

$$softmax(z)_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

In Keras muss zunächst ein Modell anhand der sequenziellen API definiert werden. Dem Modell können danach Schichten hinzugefügt werden. Dazu wird `model.add()` mit einem Keras-Layer aufgerufen. Hier wird ein `Dense()`-Layer verwendet, welches eine Schicht in einem forwährts-gerichteten neuronalen Netz beschreibt. Die Anzahl der Neuronen in der jeweils letzten Schicht wird dabei durch die Anzahl der Klassen bestimmt.

```
from keras.layers import Dense

# Softmax-Layer zur Klassifikation
model = Sequential()
model.add(Dense(10, activation='softmax', input_shape=(784,)))
```

Das erzeugte neuronale Netz lässt sich mittels `model.summary()` untersuchen. Nun kann das Netz für das Training vorbereitet werden. Dazu muss die **Cost**-Funktion (*hier* **Loss**) und das Optimierungsverfahren ausgewählt werden. Für diese Aufgabe kann die Loss-Funktion `categorical_crossentropy` gewählt werden. Die **Cross-Entropy**-Funktion misst dabei die Abweichung zwischen zwei Wahrscheinlichkeitsverteilungen p und q. In diesem Fall ist q gegeben durch die Vorhersage des Modells und p durch die tatsächliche Verteilung der Klassen.

$$H(p, q) = -\sum_x p(x) * \log q(x)$$

In Keras lassen sich die Loss-Funktion und die Optimierungsfunktion bequem über `model.compile()` auswählen. Hier kann als Optimierungs-Funktion das bereits bekannt **SGD**-Verfahren (Stochastic Gradient Descent) verwendet werden, welches das einfachste in Keras implementierte Verfahren zu diesem Zweck ist. Der Parameter `lr` entspricht dabei der Lernrate α.

```
# Vorbereitung des Modells für das Training
model.compile(loss='categorical_crossentropy',
     optimizer=keras.optimizers.SGD(lr=0.01),
     metrics=['accuracy'])
```

Nun kann das Netz trainiert werden. Die Funktion `model.fit()` liefert dabei Informationen über den Verlauf des Trainings, die zur Evaluation des Modells verwendet werden können. Dazu wird die Funktion mit den Trainingsdaten `x_train, y_train`, der Größe der Mini-Batches `BATCH_SIZE` und der Anzahl der Iterationen über die gesamten Trainingsdaten `EPOCHS` aufgerufen. Zusätzlich werden die Daten zur Validierung des Modells `x_text, y_test` übergeben, mit denen die Fähigkeit zur Generalisierung des Modells getestet werden kann.

```
# Training des Modells
history = model.fit(x_train, y_train,
    batch_size=BATCH_SIZE,
    epochs=EPOCHS,
    validation_data=(x_test, y_test))
```

In der Consolen-Ausgabe zeigt Keras nun den Verlauf des Trainings. Nach abgeschlossenem Training, kann die folgende Funktion zum Plot des Verlaufes der `accuracy` und des `cross-entropy loss` verwendet werden:

```
def plot_training(training_loss, validation_loss, training_accuracy, validation_accuracy):
    """ Plot von Accuracy und Loss der Trainings- und Validierungs-Daten
    """

    plt.figure(1)
    plt.plot(training_accuracy)
    plt.plot(validation_accuracy)
    plt.title('model accuracy')
    plt.ylabel('accuracy')
    plt.xlabel('epoch')
    plt.legend(['train', 'test'], loc='upper left')

    plt.figure(2)
    plt.plot(training_loss)
    plt.plot(validation_loss)
    plt.title('cross-entropy loss')
    plt.ylabel('loss')
    plt.xlabel('epoch')
    plt.legend(['train', 'test'], loc='upper left')

    plt.show()

# Plot von Accuracy und Loss über das Keras History-Object
plot_training(training_loss=history.history['loss'],
    validation_loss=history.history['val_loss'],
    training_accuracy=history.history['acc'],
    validation_accuracy=history.history['val_acc'])
```

Ein trainiertes Modell kann in Keras nun zur Vorhersage von neuen Daten verwendet werden. Demnach können nun auch die Test-Daten zur Bestimmung der Genauigkeit des Modells verwendet werden.

```
# Genauigkeit des trainierten Modells
score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

Nehmen Sie die Trainings- und Test-Genauigkeit des Modells nach beendetem Training und die Plots in das Protokoll auf.

2.2 Mehrschichtiges Netz

Bauen Sie nun ein mehrschichtiges neuronales Netz, indem Sie nach dem vorangehenden Beispiel mehrmals die Function `model.add(Dense(...))` aufrufen.

$$sigmoid(x) = \frac{1}{1 + e^{-x}}$$

Das Netz soll aus aufeinanderfolgenden Schichten mit 200, 100, 60 und 30 Neuronen bestehen, welche jeweils die **Sigmoid**-Funktion zur Aktivierung verwenden. Darauf folgend kann wie zuvor eine Schicht mit 10 Neuronen und der **Softmax**-Funktion zur Vorhersage der Klassen verwendet werden.

Hinweis: Den Parameter `input_shape` benötigen Sie jeweils nur in der ersten Schicht eines neuronalen Netzes in Keras.

Führen Sie nun erneut das Training durch. Verändern Sie ggf. die Anzahl der Iterationen über die Traingsdaten `EPOCHS`, sodass die Loss-Funktion konvergiert. Nehmen Sie dann die Plots und die Genauigkeit des Modells in das Protokoll auf.

Was können Sie beobachten, in Anbetracht der deutlich komplexeren Architektur des neuronalen Netzes im Vergleich zum vorangehenden Netz? Was für Veränderungen erkennen Sie in dem Plot der Loss-Funktion?

2.3 ReLU als Aktivierungsfunktion

Im Gegensatz zur **Sigmoid**-Funktion wird in der Praxis häufig **ReLU** (Rectified Linear Unit) als Aktivierungsfunktion verwendet.

$$relu(x) = \max(0, x)$$

Ersetzen Sie nun die **Sigmoid**- durch die **ReLU**-Funktion zur Aktivierung. Führen Sie erneut das Training durch und dokumentieren Sie ihre Beobachtungen. Welcher Effekt lässt sich nun in den Plots beobachten, bei `EPOCHS=200` Iterationen? Wie genau ist das Modell nun im Vergleich zu den vorangehenden Modellen?

2.4 Adam-Optimizer und adaptive Learnrate

Anstatt des **SGD**-Verfahrens werden in der Praxis oftmals erweiterte Verfahren eingesetzt. Ein sehr beliebtes Verfahren dazu ist der **Adam**-Optimizer. Verwenden Sie nun **Adam** zum Training des Netzes.

```
# Verwenden von Adam als Optimizer
model.compile(loss='categorical_crossentropy',
    optimizer=keras.optimizers.Adam(lr=0.001, decay=0.0),
    metrics=['accuracy'])
```

Der Parameter `decay` steuert dabei die Verringerung der Lernrate nach jedem Lernschritt. Führen Sie nun für `EPOCHS=100` Iterationen zwei Versuche durch, wobei Sie zunächst `decay=0.0` verwenden und im Anschluss `decay=0.01` einstellen. Was beobachten Sie nun im Plot der Loss-Funktion?

2.5 Dropout

Ein beliebtes Verfahren zum Verhindern eines **Overfittings** im Zuge des Trainings von komplexen Netzen ist **Dropout**. Dabei werden während des Trainings nach einer eingestellten Wahrscheinlichkeit einzelne Gewichte eines neuronalen Netzes einmalig gleich 0 gesetzt und damit effektiv "ausgeschaltet". Mit diesem Verfahren wird das Netz zum Generalisieren gezwungen.

Fügen Sie nun dem zuvor verwendeten Modell Dropout hinzu, indem Sie nach jeder Schicht (außgenommen die Softmax-Schicht) die folgende Zeile hinzufügen:

```
from keras.layers import Dropout

model.add(Dense(...))
model.add(Dropout(0.1))
```

Nun können Sie verschiedene Werte für Dropout ausprobieren und den Effekt auf das Training des Netzes beobachten. Auch können Sie verschiedene Architekturen ausprobieren. Gute Ergebnisse sollten Sie z.B. mit der folgenden Architektur erzielen:

```
model = Sequential()
model.add(Dense(512, activation='relu', input_shape=(784,)))
model.add(Dropout(0.2))
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(10, activation='softmax'))
```

Nehmen Sie einen Plot in das Protokoll auf, bei dem man gut den Effekt von Dropout erkennen kann.

3. Convolutional neural Network (CNN)

In der Bildverarbeitung sind CNNs zur Erkennung und Klassifizierung von Objekten weit verbreitet. Das lässt sich vor allem auf die Fähigkeit zurückführen, mehrdimensionale Strukturen als Eingangsdaten verarbeiten zu können. CNNs verwenden dazu verschiedene **Faltungs-Kernel**, um aus den Eingangsbildern Feature-Maps zu extrahieren. Die Faltungs-Kernel werden über das Eingangsbild geschoben und erkennen somit bestimmte lokale Muster unabhängig von ihrer Position im Bild.

Anschließend sorgt eine **Pooling**-Funktion dafür, dass semantisch ähnliche Features aus den Feature-Maps vereinigt werden. Dadurch wird eine Invarianz bzgl. geringfügiger Verschiebung oder Verzerrung der Muster erlangt. Schließlich kann das Netz um herkömmliche, voll-verknüpfte Schichten (Dense-Layers) erweitert werden, um die Klasse vorherzusagen.

Verändern Sie zunächst die Vorverarbeitung der Trainings- und Testdaten. Die zuvor verwendete Aneinanderreihung von Spalten der Pixel-Matrizen wird nun nicht mehr benötigt. Stattdessen muss eine Dimension hinzugefügt werden, damit die Daten mit den auszuführenden TensorFlow-Funktionen kompatibel sind.

```
# Dimension der Daten erweitern um Kanal
x_train = x_train.reshape(x_train.shape[0], 28, 28, 1)
x_test = x_test.reshape(x_test.shape[0], 28, 28, 1)
```

Behalten Sie die restlichen Funktionen bei und ändern Sie nun das Modell:

```
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D

# Convolutional Neural Network
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=(28,28,1)))
```

```
model.add(Conv2D(64, kernel_size=(3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(NUM_CLASSES, activation='softmax'))
```

Trainieren Sie das Modell für `EPOCHS=15` Iterationen. Dabei sollten Sie ein sehr gutes Ergebnis feststellen. Sie können ebenfalls eine größere Anzahl `EPOCHS` ausprobieren und beobachten, wann das Training konvergiert. Des Weiteren können anstatt von `Adam` auch die Optimizer `Adadelta` oder `RMSprop` ausprobieren. Plotten Sie die Ergebnisse und nehmen Sie die Plots in das Protokoll auf.