

Neuronales Netzwerk – MNIST 2

Die nachfolgenden Aufgabenstellungen basieren auf den Ergebnissen von „Neuronales Netzwerk – MNIST 1“.

Die `fit`-Methode¹ trainiert ein Modell für eine vorgegebene Anzahl von Epochen². Mit dem Argument `batch_size` legt man fest, nach wie vielen Beispielen die Gewichte aktualisiert werden (Default: 32). Beachtenswert ist der Return-Wert von `fit`. Dieser beinhaltet historische Information bezüglich des Trainings- und, wenn gewünscht, Validierungsvorganges.

```
model_history = model.fit(train_images, train_labels, epochs=15,
batch_size=128)
type(model_history.history)
```

Output:

```
dict
```

Wie dem vorangegangenen Codeauszug zu entnehmen ist, steht die „Trainings-History“ in Form des *Dictionary* `model_history` bereit. Der Zugriff auf die historischen Trainingsdaten erfolgt durch die vordefinierten Key. Hierbei handelt es sich um Metriken³, wie z.B. *accuracy* o. *loss*. Die praktische Umsetzung gestaltet sich wie folgt:

```
acc = model_history.history["accuracy"]
acc
```

Output:

```
[0.9012667,
 0.9533833,
 0.9663333,
 0.97276664,
 0.97748333,
 0.9798167,
 0.98255,
 0.98441666,
 0.9856333,
 0.98675,
 0.9878,
 0.98935,
 0.98995,
 0.9910833,
 0.9915]
```

Nachdem das Training in 15 Epochen erfolgt ist, liefert die Accuracy-History – welche Überraschung – 15 Werte. Jetzt stellt sich nur noch die Frage: Wozu das Ganze? Um schöne Grafiken zu erstellen!

¹ https://keras.io/api/models/model_training_apis/#fit-method

² Als Epoche versteht man eine Iteration über das gesamte Dataset!

³ <https://keras.io/api/metrics/>

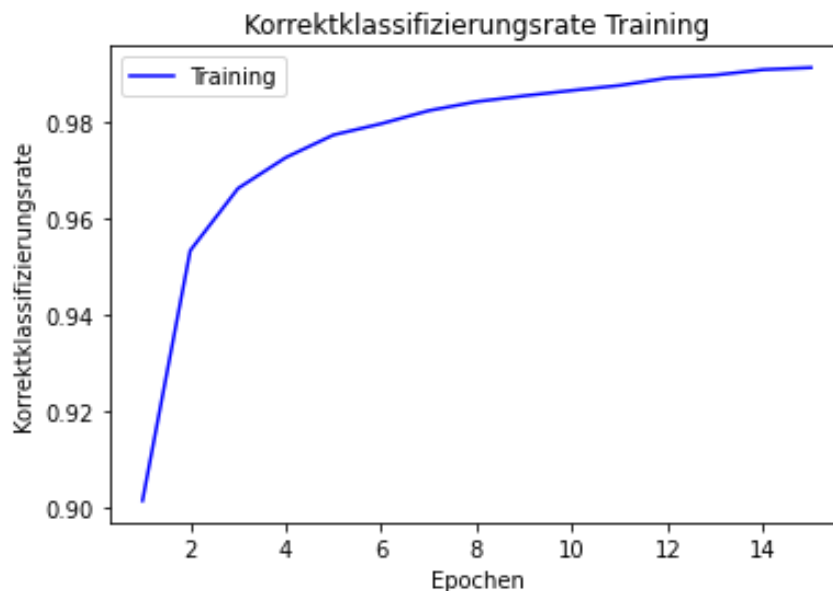




Abbildung 1: Visualisierung der Korrektklassifizierungsrate des Trainings (vgl. *Accuracy*)

Code-Auszug für den Plot in Abbildung 1:

```
plt.plot(?, ?, 'b', label="Training")
plt.title("Korrektklassifizierungsrate Training")
plt.xlabel("Epochen")
plt.ylabel("Korrektklassifizierungsrate")
plt.legend()
plt.show()
```

 18.2.1 Erstellen Sie basierend auf der „fit-History“ einen Plot (vgl. Abbildung 1) der *Accuracy*.

Die `fit`-Methode leistet aber noch mehr: Mit dem Argument `validation_data` kann man auch gleich die Validierungsdaten bereitstellen. Somit erfolgen Training und Validierung in einem Arbeitsschritt; genau genommen wird nach jeder Epoche validiert. Das Epochen-basierte Validierungsergebnis ist ebenfalls in der *History* verfügbar.

 18.2.2 Finden Sie mithilfe der `fit`-API heraus, wie die Validierungsdaten zu übergeben sind. Was deren Aufbereitung (Stichwort One-Hot-Encoding) angeht: siehe Trainingsdaten!

Trainieren Sie das Modell unter Berücksichtigung der Validierungsdaten und generieren Sie jenen Plot, der Abbildung 2 entspricht.

Hinweis: Um herauszufinden, unter welchem Key die Liste mit den „Validierungs-Accuracies“ verfügbar ist, ist `keys()` überaus hilfreich!

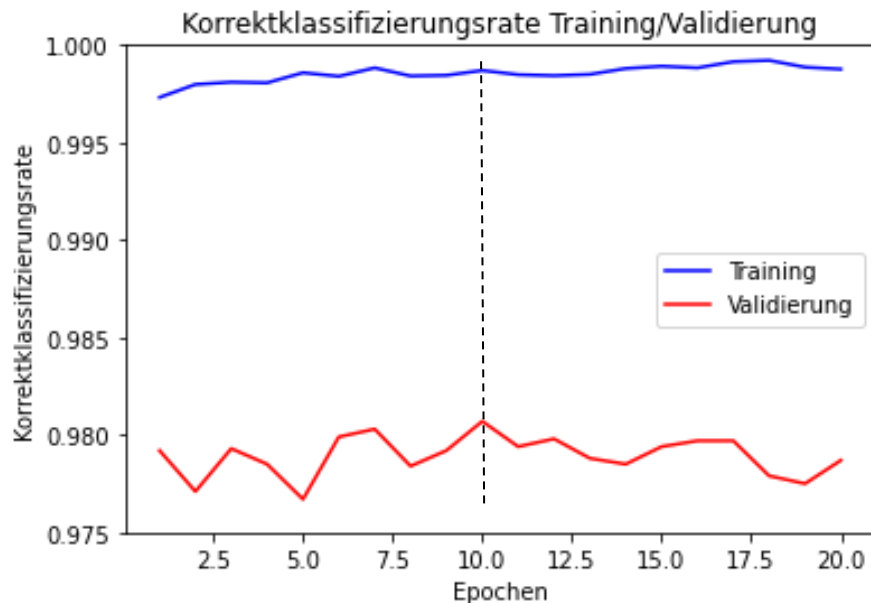


Abbildung 2: Visualisierung der Korrektklassifizierungsrate für das Training und die Validierung

Der Abbildung 2 ist zu entnehmen (siehe strichlierte Linie), dass man das Training nach 10 Epochen abbrechen kann⁴, danach findet anscheinend eine Überanpassung an die Trainingsdaten statt.

Hyperparameter

`epochs` und `batch_size` sind zwei wichtige Hyperparameter⁵ bei einem Neuronalen Netz. Um hier die optimalen Werte zu finden, braucht es viel Erfahrung oder wiederum `Grid`⁶ bzw. `RandomSerach`⁷ (vgl. `RandomForest`).

```
model.fit(
    x_train,
    y_train,
    epochs=10,
    batch_size=64,
    validation_data=(x_test, y_test))
```

Mit `epochs` wird abgegeben, wie oft der ganze Datensatz durchlaufen wird. Nachdem es schwierig ist, ein ganzes Dataset auf einmal zu verarbeiten – man bedenke den Rechenaufwand bei der Forward- und Backpropagation – erfolgt die Unterteilung in kleinere Einheiten, die Batches. Ein Batch (siehe `batch_size`) ist im Beispielsfall 64 Beispiele umfassend. Das heißt: Nach 64 Beispielen erfolgt jetzt bereits die Anpassung der Gewichte (Backpropagation). Der Arbeitsspeicher (RAM) ist sicherlich dankbar!

⁴ Zumindest im Beispielsfall. Das ist aber von Fall zu Fall verschieden.

⁵ Ein weiterer Hyperparameter wäre bspw. die Neuronen-Anzahl eines Layers.

⁶ https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html

⁷ https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.RandomizedSearchCV.html

Dividiert man die Anzahl der Beispiele durch die Batch-Size, erhält man die Anzahl der sog. Iterationen (je Epoche).


Bsp. MNIST mit 60.000 Trainingsbeispielen:


$60.000/64 = 937,5 \Rightarrow$ also **938** Iterationen in einer Epoche!

Dieser Prozess wiederholt sich nun 10-mal – also 10 Epochen!

model.fit()-Output:

```
Epoch 1/10
938/938 [=====] - 5s 5ms/step - loss: 0.3822 - accuracy: 0.9736 - val_loss: 0.6543 - val_accuracy: 0.9687
Epoch 2/10
938/938 [=====] - 5s 5ms/step - loss: 0.3635 - accuracy: 0.9776 - val_loss: 0.8607 - val_accuracy: 0.9644
Epoch 3/10
938/938 [=====] - 6s 6ms/step - loss: 0.3315 - accuracy: 0.9800 - val_loss: 0.7232 - val_accuracy: 0.9722
Epoch 4/10
938/938 [=====] - 6s 7ms/step - loss: 0.2850 - accuracy: 0.9827 - val_loss: 0.7470 - val_accuracy: 0.9708
Epoch 5/10
938/938 [=====] - 6s 6ms/step - loss: 0.2839 - accuracy: 0.9839 - val_loss: 0.7804 - val_accuracy: 0.9734
Epoch 6/10
938/938 [=====] - 6s 6ms/step - loss: 0.2436 - accuracy: 0.9859 - val_loss: 0.9319 - val_accuracy: 0.9679
Epoch 7/10
938/938 [=====] - 6s 7ms/step - loss: 0.2276 - accuracy: 0.9870 - val_loss: 0.8591 - val_accuracy: 0.9733
Epoch 8/10
938/938 [=====] - 7s 7ms/step - loss: 0.2458 - accuracy: 0.9870 - val_loss: 0.8482 - val_accuracy: 0.9769
Epoch 9/10
938/938 [=====] - 6s 6ms/step - loss: 0.2213 - accuracy: 0.9884 - val_loss: 0.9891 - val_accuracy: 0.9738
Epoch 10/10
938/938 [=====] - 6s 6ms/step - loss: 0.2219 - accuracy: 0.9885 - val_loss: 0.8172 - val_accuracy: 0.9771
```

 **18.2.3** Erstellen Sie einen Plot für die Werte der Verlustfunktion (siehe *loss* und *val_loss*). Experimentieren Sie mit unterschiedlichen Werten für die Epochen bzw. Batch Size. Ziel ist, dass die Kurven so gut wie möglich konvergieren.

 **18.2.4** Das Modell kann mit *save* gespeichert werden. Das Laden in ein Notebook lässt sich mit *load_model* realisieren. Details sind dieser Quelle zu entnehmen⁸. Speichern Sie ihr Modell und laden Sie dieses in ein neues Notebook.

⁸https://www.tensorflow.org/guide/keras/save_and_serialize