Applications of Neural Networks

Tensorflow basics
with MNIST

Table of Contents

- Recap: Neural Network – The Basics
- Tensorflow/Keras
- Example: MNIST
  - Step by Step in Keras
  - Evaluation metrics
  - Training
- Outlook

Neural Networks

Recap of the basics

Recap Neural Networks

- A Neural Network in general is a function f applied to „input data“ x:

$$x \mapsto y = f(x)$$

- The output y can be a number, a vector (classification) or a matrix (e.g. an image)

- f is parametrized by weights θ (all weights):

$$f = f_\theta(x)$$

Recap Optimization

- Optimization problem: Given are data $D = \{(x_i, t_i)\}$. Find θ, so that the **Loss-Function** $L(f_\theta, D)$ is (t Target)
- L here is a function, which maps the data D to a real number l (**Loss**)
  - e.g. **Euclidean Loss, Mean-Squared-Error MSE**:

$$\ell_2 = \frac{1}{2N} \sum_i (f_\theta(x_i) - t_i)^2$$

  - **Negative-Log-Likelihood, Cross-Entropy:**

$$NLL = -\frac{1}{|D|} \sum_i \log \left[f_\theta(x_i) \Big|_{t_i} \right]$$

Recap Optimization

- Find $\min_{\theta} L(f_\theta, D)$ with Gradient Descent (step size/learning rate η), since in general not analytically solvable:

$$\boxed{\theta \leftarrow \theta - \eta \nabla_\theta L(f_\theta, D)}$$

- Gradient Descent:

$$\theta \leftarrow \theta - \eta \nabla_\theta \frac{1}{2N} \sum_i (f_\theta(x_i) - t_i)^2$$

Optimizing the Neural Network

In practise (with ℓ_2-loss as an example) often not **Gradient Descent** (sums over all data points)

$$\theta \leftarrow \theta - \eta \nabla_\theta \frac{1}{2N} \sum_i (f_\theta(x_i) - t_i)^2$$

but either:

- **Stochastic Gradient Descent** (one data point per iteration step):

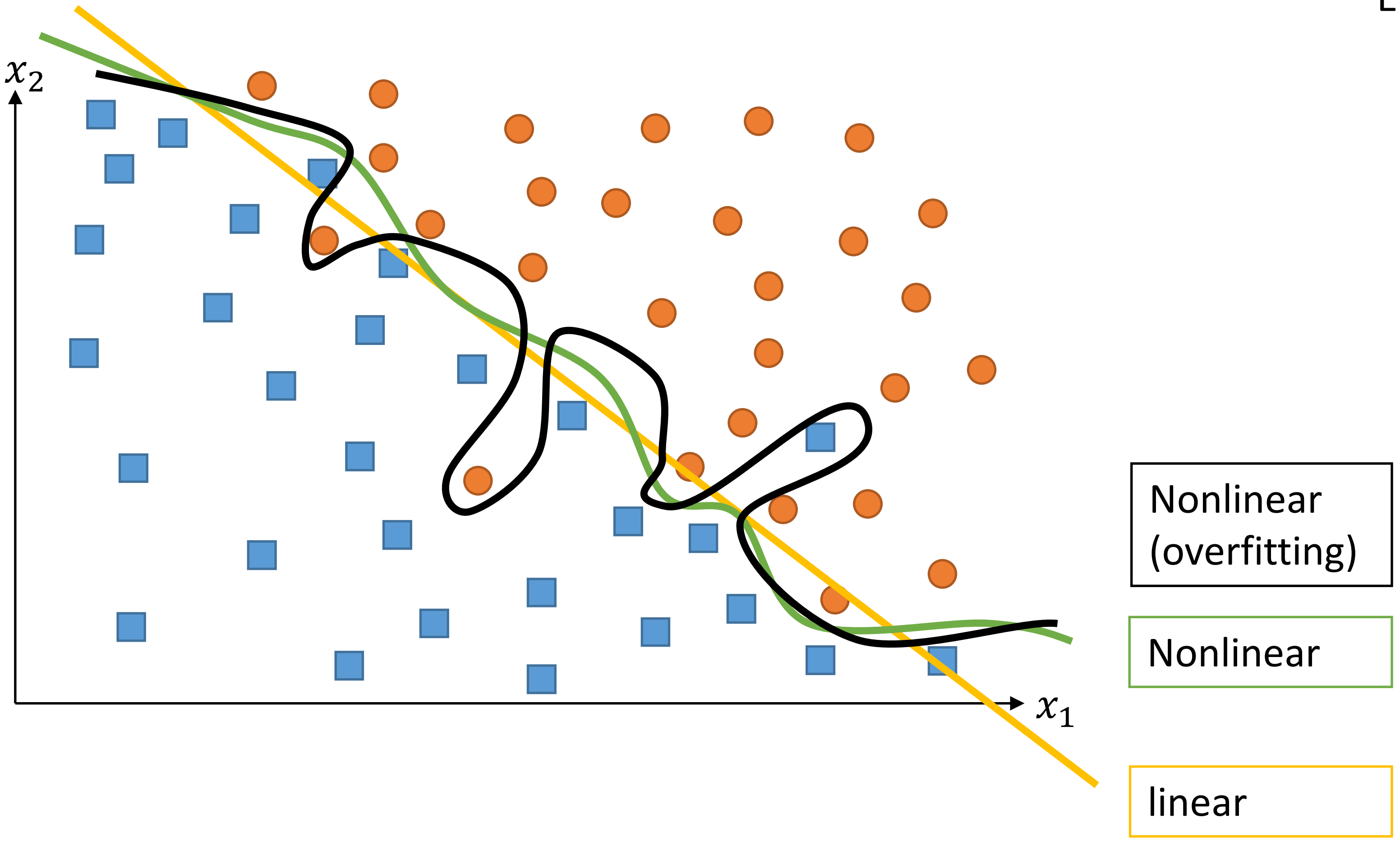
$$\theta \leftarrow \theta - \eta \nabla_\theta \frac{1}{2} (f_\theta(x_i) - t_i)^2$$

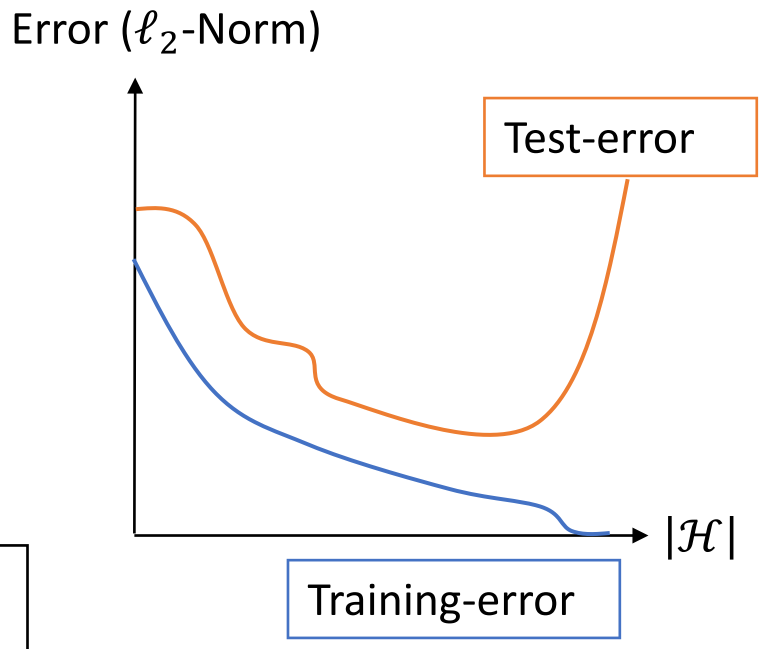
- **Batch Gradient Descent** (a batch B of data per iteration step)

$$\theta \leftarrow \theta - \eta \nabla_\theta \frac{1}{2|B|} \sum_{i \in B} (f_\theta(x_i) - t_i)^2$$

This method approximates thetotal loss with one or a small amount of samples.

Outlook: Overfitting





Optimizing the Neural Network

Iterative-Training:

- Split the data into Train/Validation/Test (e.g. 60/20/20)
  - Train: Training the Network
  - Validation: Optimizing/finding the hyperparameter
  - Test: Independent evaluation of the network

Training-Loop:

- **Epoch**: Loop through all data once
- **Iteration**: Process one batch
- **Iterations per epoch**: Number of data / Batch size

Optimizing the Neural Network

Typical process:

- Split the data
- For (epoch) in (Number of epochs):
  - Randomize oder of data within the epoch
  - For (batch) in (generate batches):
    - Compute weight updates (Backpropagation)
    - Update weights
- Evaluate model on validation data (optional to improve hyperparameters)
- Evaluate model on test data (at the very very very end, do not change hyperparameter at this point anymore)

Example: Optimizing on MNIST

- You know this already from your framework development
- Simulating the real case:
  - Split data into Train/Val/Test
  - Build one or more models
  - Choose the loss
  - Choose the optimizer

→ Implement into a stronger framework

→ Tensorflow

Tensorflow

Introduction, functions, Keras

Why Tensorflow?

```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense, Conv2D

model = Sequential()
```

- Lecture 2:

```
model.add(Dense(784, activation='relu'))
```

- Lecture 3:

```
model.add(Conv2D(64, kernel_size=3, activation='relu‘, input_shape=28, 28, 1))
```

+ Initialize, Regularization, …

→ A lot more functions and flexibility

What is Tensorflow?

- End-to-end open source machine learning library from Google
- Release 2015
- Backend: C++ and CUDA
- APIs in several languages
  - Python (most popular)
  - JavaScript
  - C++
  - Java
  - Go
  - Swift (Early Release)

Tensorflow 1.x

- Uses Graphs and Sessions

1. First build a „Computation Graph“
   - Define Variables/Placeholders for input
   - Sequentialize the operations (Layer)
   - Loss, Training, etc.
2. Run operations in a „Session“
   - Initialise variables, run training steps etc.
   - Computations during run time are like a black box

Tensorflow 1.x

```
import tensorflow as tf
x = tf.placeholder(tf.float32, [None, 784])
y_ = tf.placeholder(tf.float32, [None, 10])

V = tf.get_variable("V", [784, 500], initializer=xavier_initializer())
c = tf.get_variable("c", [500])

W = tf.get_variable("W", [500, 10], initializer=xavier_initializer())
b = tf.get_variable("b", [10])

y = tf.matmul(tf.nn.relu(tf.matmul(x, V) + c), W) + b
loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(labels=y_,logits=y))
train_step = tf.train.GradientDescentOptimizer(0.1).minimize(loss)

with tf.Session() as session:
    loss, _ = sess.run([loss, train_step])
```

Tensorflow 2

- Released in 2019
- Many deprecated and redundant APIs removed or merged
- Eager Execution per default
  → No graphs anymore
- Keras as official High-level API
  - keras → tf.keras
- Important: No separate Tensorflow GPU library necesssary since 2.0

GPU Support

- Summary: https://www.tensorflow.org/install/gpu
- Only on NVIDIA GPUs
- Most recent driver software
- CUDA (note the information on the tensorflow page)
  - CUDA 11.2 für aktuelles Tensorflow (2.5)
- cuDNN: CUDA Deep Learning Library

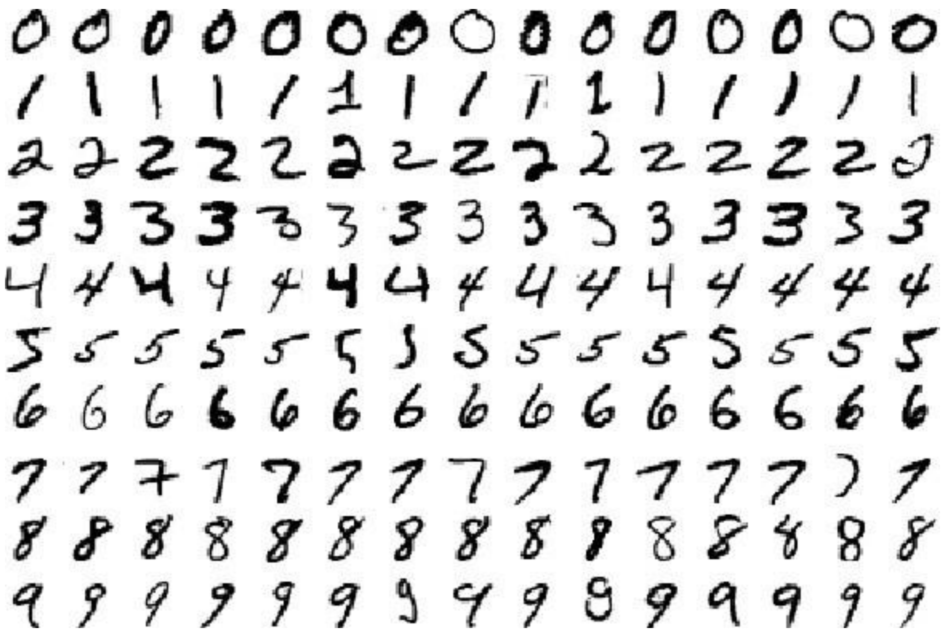
→ Watch installation instructions

MNIST with Tensorflow

Deep Learning in practise

MNIST: Hello World of Deep Learning

- Dataset of hand written numbers
- Each number is a 28×28 px large image (actually white on black)
- Task: Recognize the class of each image



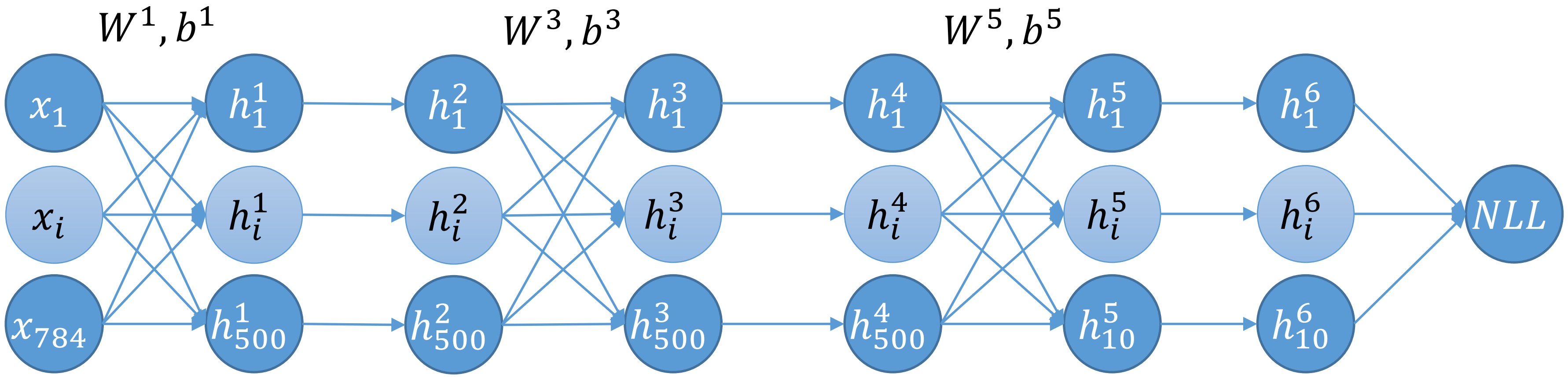
MNIST: Hello World of Deep Learning

- Training a Fully Connected Network
- Data
  - Each node has a value from 0 to 1 (normalized grayscale)
  - Input: depending on the model 2D [28,28] oder 1D [784]
- Output is a vector:
  - 10 nodes, one per class
  - Softmax for probability distribution across the 10 classes
- Categorical Cross-Entropy (NLL) as Loss

Roadmap

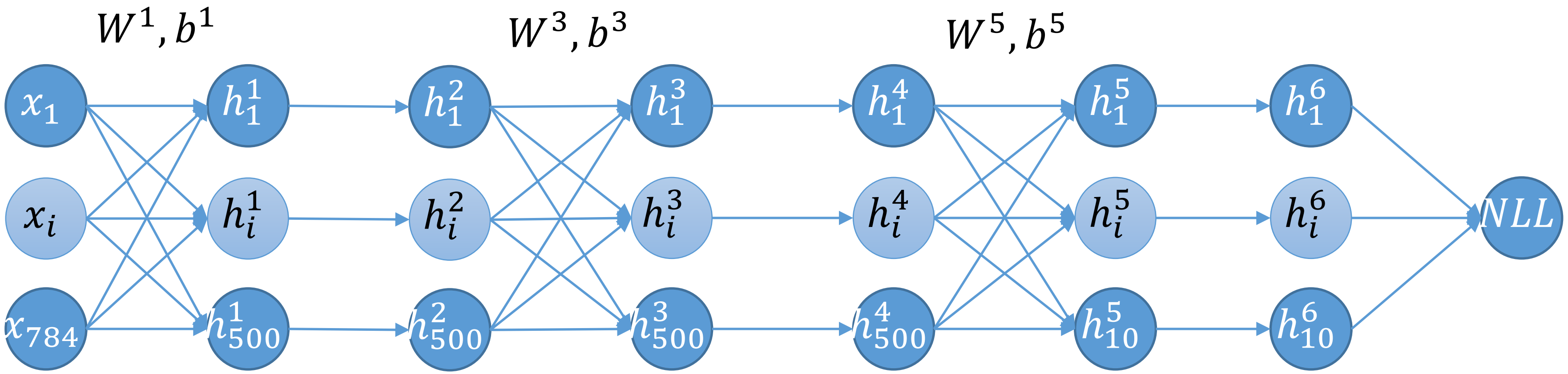
- Model architecture
  - First only Fully Connected
  - Parameter, functions etc.
- Dataset
  - Preprocessing
  - Reading the data
- Evaluation
  - Explaining the metrics
- Training the model

Fully Connected Architecture



	Input	Hidden Layer 1	Activation 1 $\sigma(x)$	Hidden Layer 2	Activation 2 $\sigma(x)$	Output-Layer	Softmax	Loss-Function
Layer dimensions	784	500	500	500	500	10	10	1
Weight dimensions	-	784 x 500	-	500 x 500	-	500 x 10	-	-

MNIST: Fully Connected



Forward-Variables:

- $h^1 = x \cdot W^1 + b^1$
- $h^2 = \sigma(h^1)$
- $h^3 = h^2 \cdot W^3 + b^3$
- $h^4 = \sigma(h^3)$
- $h^5 = h^3 \cdot W^5 + b^5$
- $h^6 = \text{softmax } h^5$
- $NLL = -\log h_t^6$

Backward-Variables:

- $\delta x = \delta h^1 \cdot {W^1}^T$
- $\delta h^1 = [\sigma(x) \cdot (1 - \sigma(x))] \odot \delta h^2$
- $\delta h^2 = \delta h^3 \cdot {W^3}^T$
- $\delta h^3 = [\sigma(x) \cdot (1 - \sigma(x))] \odot \delta h^4$
- $\delta h^4 = \delta h^5 \cdot {W^5}^T$
- $\delta h^5 = \delta h^6 \odot \text{softmax}'(h^5)$
- $\delta h^6 = -1/h_t^6$

Weight-Updates:

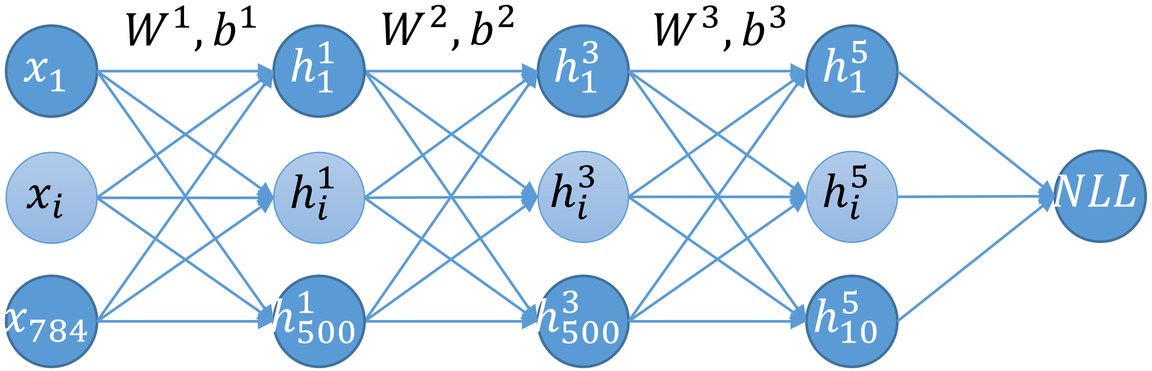
- $\delta W^1 = \delta h^1 \cdot x$
- $\delta b^1 = \delta h^1$
- $\delta W^3 = {h^2}^T \cdot \delta h^3$
- $\delta b^3 = \delta h^3$
- $\delta W^5 = {h^4}^T \cdot \delta h^5$
- $\delta b^5 = \delta h^5$

Fully Connected in Keras

```
from tensorflow import keras

fcn = keras.Sequential([
    keras.layers.Input(shape=(28,28)),
    keras.layers.Flatten(),
    keras.layers.Dense(units=500, activation='sigmoid'),
    keras.layers.Dense(units=500, activation='sigmoid'),
    keras.layers.Dense(units=10, activation='softmax')
    ])
```

Fully Connected Architecture

→ Layers can be summarized



	Input	Hidden Layer 1	Hidden Layer 2	Output-Layer	Loss-Function
Layer Dimensions	784	500	500	10	1
Activation function	-	Sigmoid	Sigmoid	Softmax	-

Weight dimensions implicit, since Fully Connected

Loss, Optimizer, Metrics

```
optimizer = tf.keras.optimizers.SGD(learning_rate=0.1)
# optimizer = tf.keras.optimizers.Adam()
loss = tf.keras.losses.categorical_crossentropy
metrics = ['accuracy']

fcn.compile(loss=loss,
            optimizer=optimizer,
            metrics=metrics)
```

- Choose from several built-in functions
  - Choose by string
  - Custom functions possible
- Compile model at the end
  - You can recompile the model as necessary (e.g. when loading an existing model)

MNIST: Data preparation

- Tensorflow provides MNIST, but other sources are also valid
- Official Training- and Test-Split

```
In [4]: mnist = tf.keras.datasets.mnist
        (x_train, y_train), (x_test, y_test) = mnist.load_data()

In [5]: y_train

Out[5]: array([5, 0, 4, ..., 5, 6, 8], dtype=uint8)
```

→ Labels are saved as „raw“ / „sparse“

One-Hot-Encoding

- Classifikation: Output of the network is a vectorof probabilities, e.g. $[0.97, 0.01, 0.00, \dots, 0.02]^T$
- Labels are easier to save in direct context
  - e.g. for MNIST directly save labels as „5" or „9"

→ Labels have to be encoded for the network when loading

Label		One-Hot encoded
5	➡	$[0, 0, 0, 0, 0, 1, 0, 0, 0, 0]^T$

5th Vector element

MNIST: Date preparation

```
In [5]: y_train

Out[5]: array([5, 0, 4, ..., 5, 6, 8], dtype=uint8)
```

- Two possibilities
  - Manual One-Hot-Encoding

```
In [6]: tf.keras.utils.to_categorical(y_train)

Out[6]: array([[0., 0., 0., ..., 0., 0., 0.],
               [1., 0., 0., ..., 0., 0., 0.],
               [0., 0., 0., ..., 0., 0., 0.],
               ...,
               [0., 0., 0., ..., 0., 0., 0.],
               [0., 0., 0., ..., 0., 0., 0.],
               [0., 0., 0., ..., 0., 1., 0.]], dtype=float32)
```

  - Loss: Sparse Categorical Cross Entropy

  → loss = tf.keras.losses.sparse_categorical_crossentropy

MNIST: Data preparation

Use part of the training data as validation data

```
mnist = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_val, y_val = x_train[0:5000], y_train[0:5000]
x_train, y_train = x_train[5000:], y_train[5000:]
```

In practise usually shuffle the data

→ Avoid correlation between the data sets

MNIST: Data preparation

- MNIST can theoretically be used directly for training

```
x_train, x_val x_test = x_train / 255.0, x_val / 255.0, x_test / 255.0

fcn.fit(x_train, y_train, epochs=5,
        validation_data=(x_val, y_val)) # batch_size default = 32
fcn.evaluate(x_test, y_test)
```

- But: Data is not always fully readable
  - Especially problematic for big datasets
- Image manipulation must be performed manually or as a pre-processing step → Real-time?

→ Keras ImageDataGenerator

MNIST Data preparation

```
datagen = tf.keras.preprocessing.image.ImageDataGenerator(
                              rescale=1./255.)
                              # target_size
                              # featurewise_center=True,
                              # featurewise_std_normalization=True)
# optional: datagen.fit(x_train)

batch_size = 128
train_generator = datagen.flow(x_train, y_train,
                                        batch_size=batch_size,
                                        shuffle=True)
                              # flow_from_dataframe
                              # flow_from_directory
fcn.fit(train_generator,
      steps_per_epoch=len(x_train) / batch_size,
      epochs=5,
      validation_data=datagen.flow(x_val, y_val))
```

- ImageDataGenerator comes with many functions for image manipulation and processing
- Custom generators also possible
- Allows dynamic loading and manipulation of images
- Important here: Extend image with channel dimension

Evaluation of MNIST

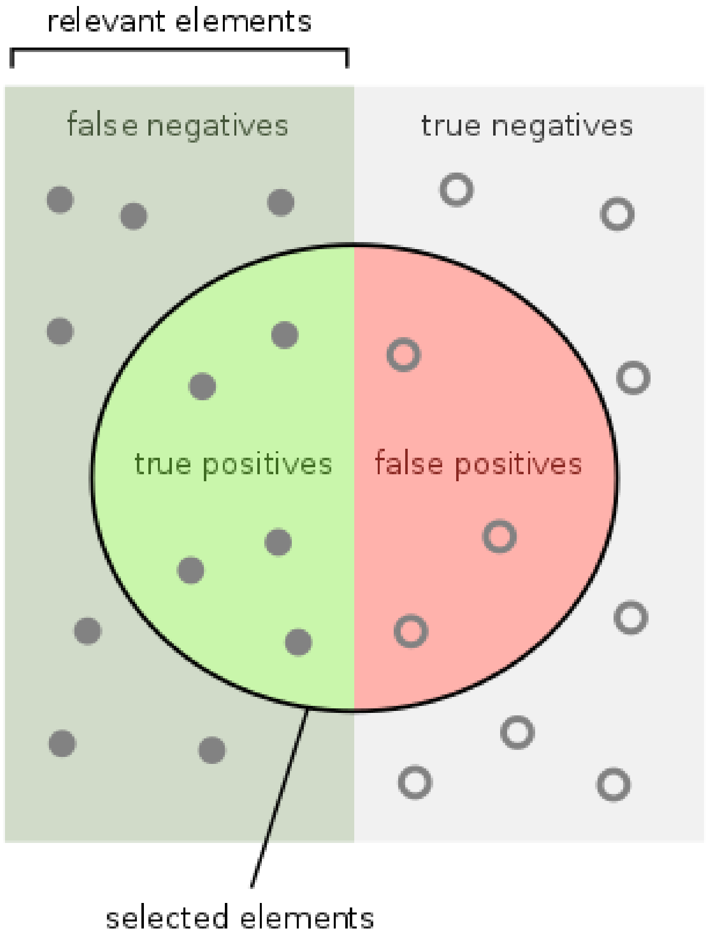
Several questions are relevant for evaluating the MNIST-Dataset:

- What is the total accuracy (number of correctly classified numbers)?
- What is the specific accuracy for each number class (e.g. is 7 worse due to different way of writing it)?
- Which numbers are more commonly mistaken for each other (e. g. how often is 5 mistaken for 9 and vice versa)? → confusion matrix
- Which loss function do we need, e.g. what needs to be optimized?

Evaluation metrics

For each class k:

- Prediction p, Ground Truth g:
  - True Positives (TP): $p = k \wedge g = k$
  - True Negatives (TN): $p \neq k \wedge g \neq k$
  - False Positives (FP): $p = k \wedge g \neq k$
  - False Negatives (FN): $p \neq k \wedge g = k$



Evaluation metrics

Total Accuracy

- Compute the percentage of correct prediction, i.e. Prediction = Ground Truth of the total data N

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{N} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{FP} + \text{TN} + \text{FN}}$$

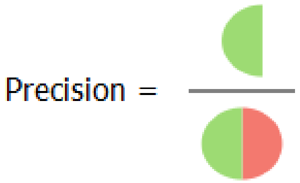
- Averages across all data: You cannot tell, whether one specific number class is classified good or bad

Evaluation metrics

More metrics:

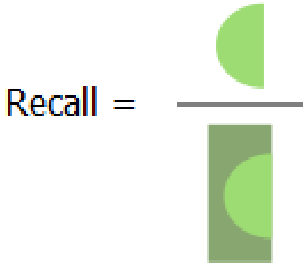
- Precision (If k is predicted, how often is the model correct?):

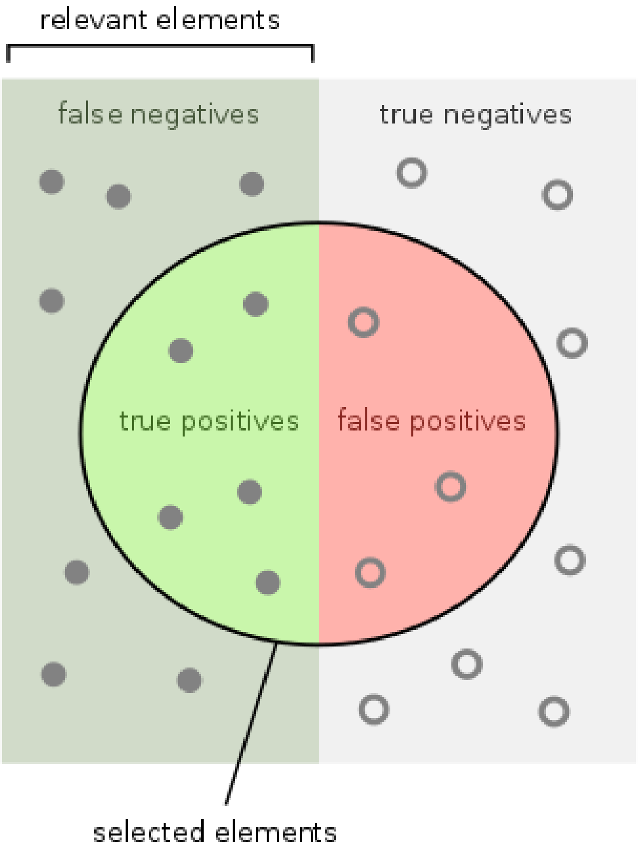
$$\frac{\mathrm{TP}}{\mathrm{TP}+\mathrm{FP}}$$



- Recall (What part of all GT data with class k is detected?):

$$\frac{\mathrm{TP}}{\mathrm{TP}+\mathrm{FN}}$$





Evaluation metrics

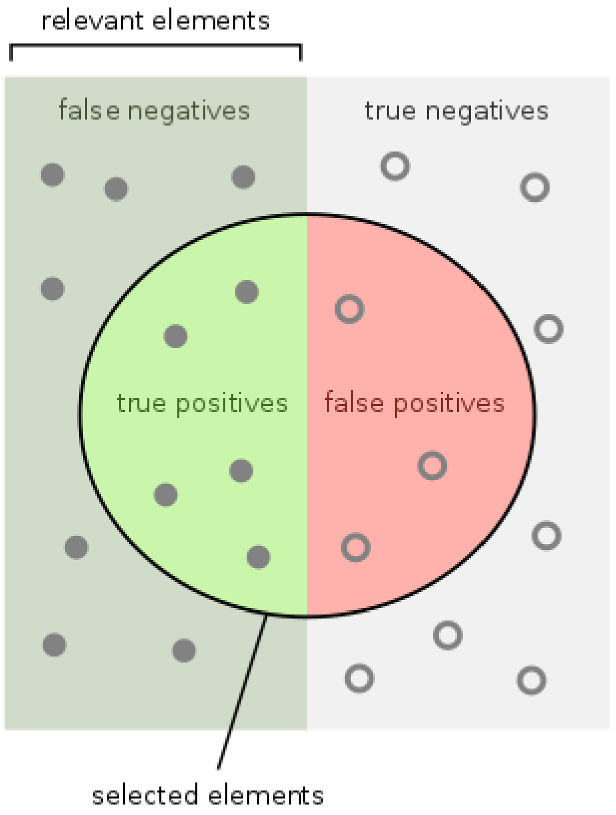
More metrics:

- F_1 (Harmonic average of Precision and Recall):

$$F_1 = 2\frac{P \cdot R}{P + R} = \frac{\mathrm{TP}}{\mathrm{TP} + \frac{\mathrm{FP}}{2} + \frac{\mathrm{FN}}{2}}$$

- Overlap (or Intersection over Union, used especially in object detection):

$$\mathrm{IoU} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP} + \mathrm{FN}}$$



Evaluation metrics Overview

		True condition				
	Total population	Condition positive	Condition negative	Prevalence = $\frac{\Sigma \text{ Condition positive}}{\Sigma \text{ Total population}}$	Accuracy (ACC) = $\frac{\Sigma \text{ True positive} + \Sigma \text{ True negative}}{\Sigma \text{ Total population}}$	
Predicted condition	Predicted condition positive	**True positive**	**False positive**, Type I error	Positive predictive value (PPV), Precision = $\frac{\Sigma \text{ True positive}}{\Sigma \text{ Predicted condition positive}}$	False discovery rate (FDR) = $\frac{\Sigma \text{ False positive}}{\Sigma \text{ Predicted condition positive}}$	
	Predicted condition negative	**False negative**, Type II error	**True negative**	False omission rate (FOR) = $\frac{\Sigma \text{ False negative}}{\Sigma \text{ Predicted condition negative}}$	Negative predictive value (NPV) = $\frac{\Sigma \text{ True negative}}{\Sigma \text{ Predicted condition negative}}$	
		True positive rate (TPR), Recall, Sensitivity, probability of detection, Power $= \frac{\Sigma \text{ True positive}}{\Sigma \text{ Condition positive}}$	False positive rate (FPR), Fall-out, probability of false alarm $= \frac{\Sigma \text{ False positive}}{\Sigma \text{ Condition negative}}$	Positive likelihood ratio (LR+) $= \frac{\text{TPR}}{\text{FPR}}$	Diagnostic odds ratio (DOR) $= \frac{\text{LR+}}{\text{LR−}}$	F_1 score = $2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$
		False negative rate (FNR), Miss rate $= \frac{\Sigma \text{ False negative}}{\Sigma \text{ Condition positive}}$	Specificity (SPC), Selectivity, True negative rate (TNR) $= \frac{\Sigma \text{ True negative}}{\Sigma \text{ Condition negative}}$	Negative likelihood ratio (LR−) $= \frac{\text{FNR}}{\text{TNR}}$		

More: Wikipedia entry for „Confusion Matrix“

Micro vs Macro

Micro-averaged

- First count all TP, FP, TN, FN over all classes
- Compute metric (e.g. F1) from this count
- Weight each class by number of samples within that class

→ Underrepresented classes get smaller weights

Macro-averaged

- Count TP, FP, TN, FN for each class separately
- Compute metric (e.g. F1) for each class
- Mean across all classes

→ Same weighting for all classes

Evaluation metrics

	0	1	2	3	4	5	6	7	8	9
0										
1										
2										
3										
4										
5										
6										
7										
8										
9										

Confusion matrix:

- Shows, how many numbers were falsely classified with another
- The diagonal represents all correctly classified cases
- Can be used to understand network errors

Why not accuracy as loss?

Loss (NLL)

$$-\log y_t$$

- Maximizes total probability
- Softmax can be interpreted as confidence:
  - Small for unsure predictions
  - High for certain predictions
- Differentiable function

Accuracy/Error

$$1-\delta(\operatorname{argmax} y, t) = \begin{cases} 0 & \operatorname{argmax} y = t \\ 1 & \text{else} \end{cases}$$

- Maximizes accuracy/minimizes the error
- Non differentiable, but still possible analogous to Hinge-Loss (error is 1 oder 0)
- Unstable training

MNIST: All in all

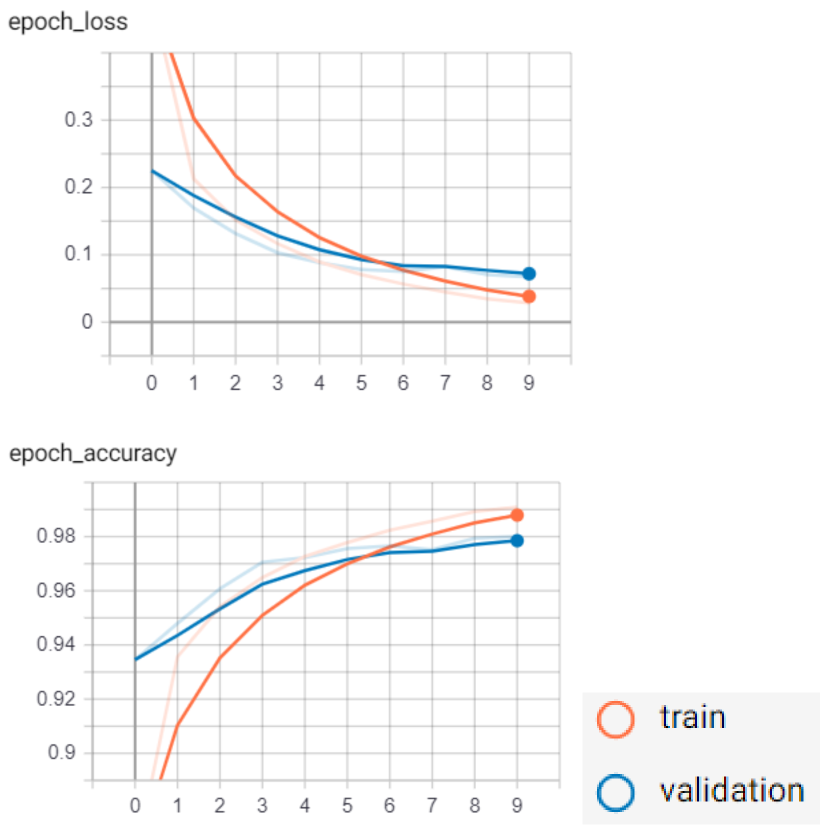
```
import tensorflow as tf

fcn = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(28,28)),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(units=500, activation='sigmoid'),
    tf.keras.layers.Dense(units=500, activation='sigmoid'),
    tf.keras.layers.Dense(units=10, activation='softmax')
])
fcn.compile(loss='sparse_categorical_crossentropy', optimizer='adam', metrics=['accuracy'])

mnist = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_val, y_val = x_train[0:5000], y_train[0:5000]
x_train, y_train = x_train[5000:], y_train[5000:]
x_train, x_val, x_test = x_train/255., x_val/255., x_test/255.

fcn.fit(x_train, y_train, epochs=5,
        validation_data=(x_val, y_val)
fcn.evaluate(x_test, y_test)
```

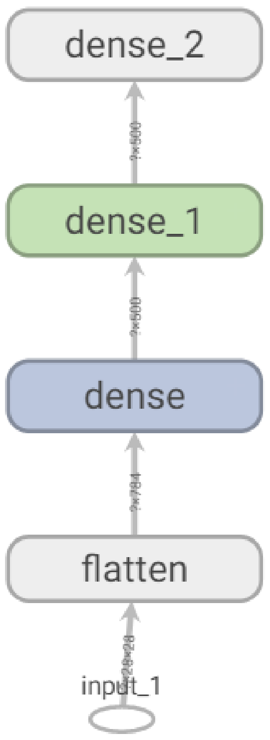
Callbacks

- Functions called during training
- Built-in and custom functions
    - Saving after each epoch
    - Adjusting the learning rate
    - Early Stopping
    - Logs during training

```
callbacks = [tf.keras.callbacks.ReduceLROnPlateau(monitor="val_loss",
             factor=0.1, patience=10),
             tf.keras.callbacks.ModelCheckpoint("model.h5", monitor="val_loss"),
             tf.keras.callbacks.EarlyStopping(monitor="val_loss", patience=10),
             tf.keras.callbacks.TensorBoard(log_dir="./log", histogram_freq=1)]
fcn.fit(x_train, y_train, epochs=5,
        validation_data=(x_val, y_val),
        callbacks=callbacks)
```

Tensorboard





- Nice tool to get an overview across different runs
- Usable in Jupyter
- Live Monitoring available

Training example

- 3 epoch training
- Accuracy
  - Training: 0.95
  - Validation: 0.96
  - Test: 0.96

	0	1	2	3	4	5	6	7	8	9
0	970	0	0	3	0	2	2	1	1	1
1	0	1126	2	2	0	1	3	0	1	0
2	7	2	986	7	7	1	5	6	9	2
3	1	1	7	972	1	3	0	7	6	12
4	1	0	8	0	942	0	3	1	1	26
5	7	4	1	27	6	818	8	4	9	8
6	14	3	0	2	8	6	919	0	6	0
7	1	12	13	4	5	1	0	962	0	30
8	3	6	8	18	12	5	9	4	897	12
9	6	6	1	12	16	1	0	5	0	962

Outlook

Exercise and next lecture

Outlook: Exercise

1. Data preparation
   - Loading MNIST data
   - Splitting into train and test (no validation necessary)
2. Neural Network
   - Implementing a neural network in Keras
   - Training the NN
3. Evaluation
   - Computing different evaluation metrics
   - Presenting the results (e.g. plotting the confusion matrix)

Outlook: Next Lecture

CNN for leaf detection (TF)

- Problems:
  - Little data per class
  - Big input images, not normalized
- Solutions:
  - Regularization
  - Data augmentation
  - Pretraining/Transferlearning
  - Batch Normalization
  - Sampling/Hyperparameter tuning