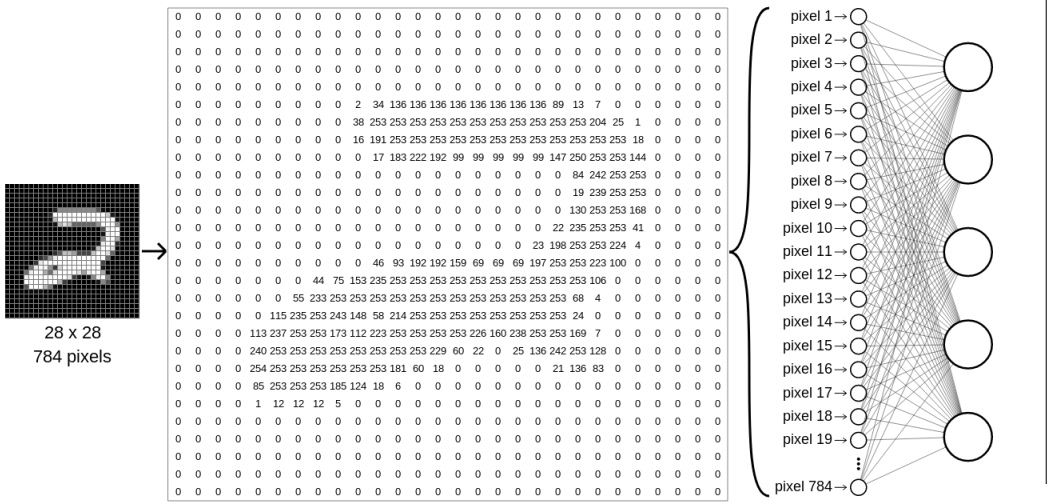
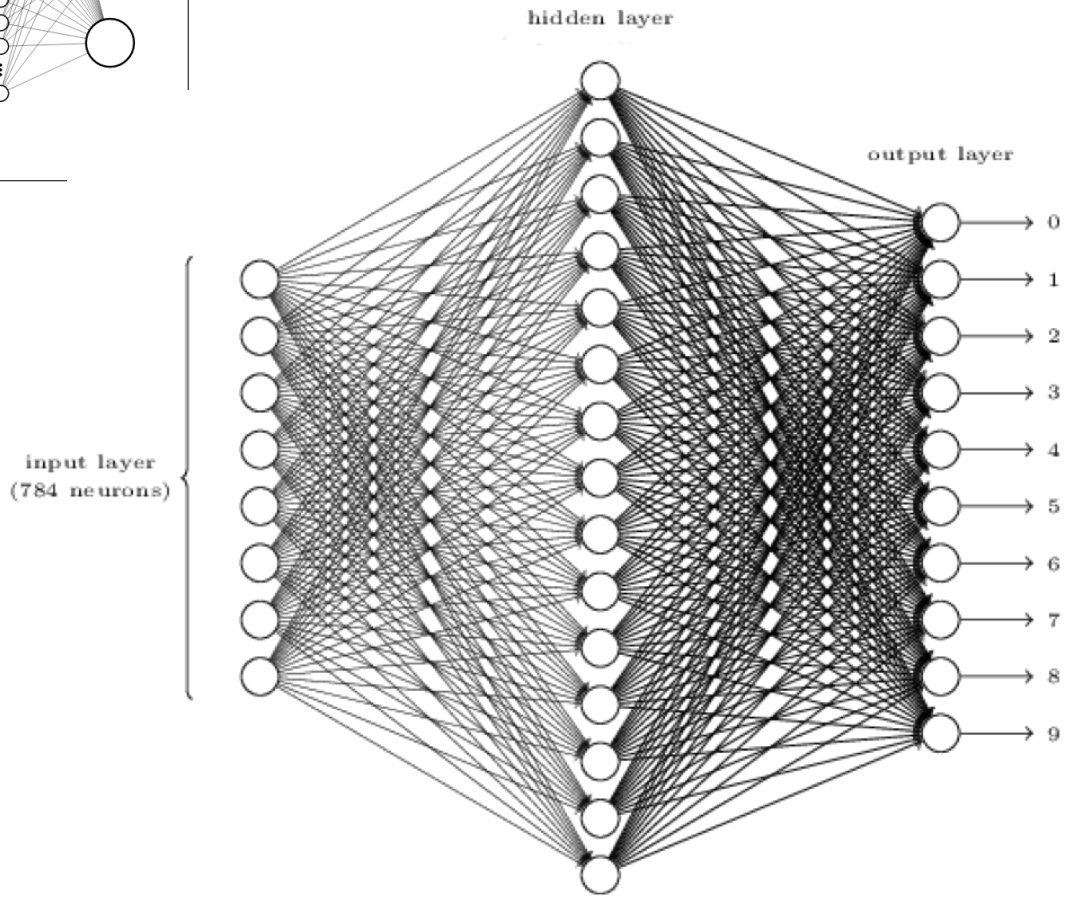
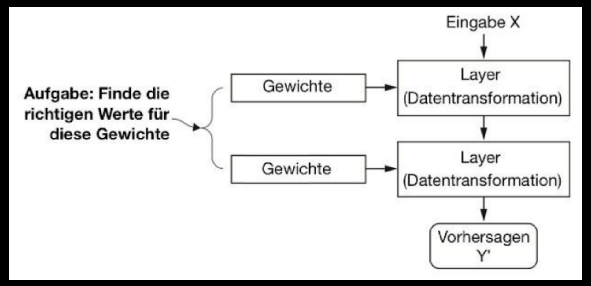
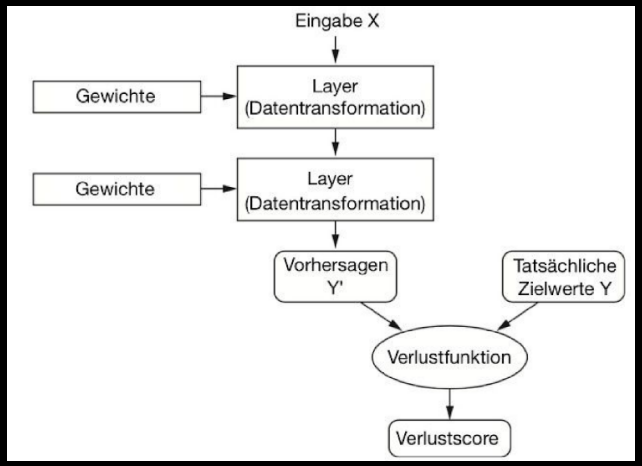

28 x 28
784 pixels
pixel 1→
pixel 2→
pixel 3→
pixel 4→
pixel 5→
pixel 6→
pixel 7→
pixel 8→
pixel 9→
pixel 10→
pixel 11→
pixel 12→
pixel 13→
pixel 14→
pixel 15→
pixel 16→
pixel 17→
pixel 18→
pixel 19→
pixel 784→


hidden layer
output layer
input layer
(784 neurons)
0
1
2
3
4
5
6
7
8
9



Lernaufgabe: ***Die „richtigen“ Werte für die Gewichtungen (Parameter des Layers) finden***

d.h. 1. zufallsgewichtungen ins Netz schicken



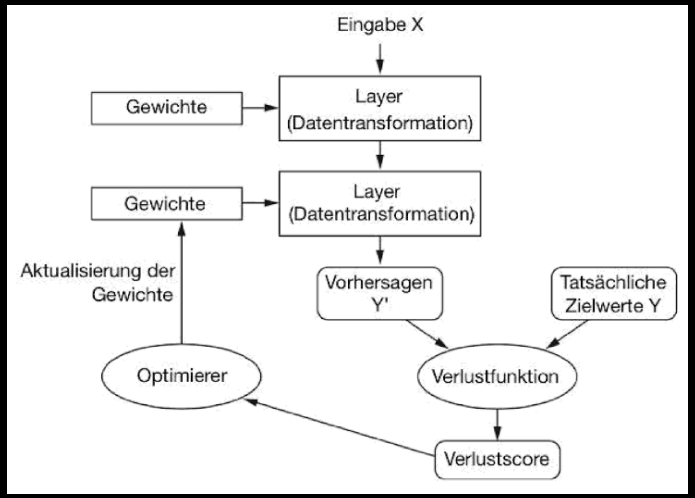
2. Messen wie stark der Istwert vom Sollwert abweicht

Diese Aufgabe erfüllt die *Verlustfunktion* `loss`, sie berechnet die Differenz [E=S-I]

Und bildet hierdurch den *Verlustscore*

Dieser *Score* dient im Traingsverlauf als Feedback-Signal zur Feinabstimmung der Gewichtungen.

```
network.compile(optimizer='rmsprop',
                loss='categorical_crossentropy',
                metrics=['accuracy'])
```



3. Gewichtungen so ändern (anpassen), damit sich im nächsten Trainingsurchlauf der Verlustscore verringert.

Diese Anpassung ist die Aufgabe des *Optimierers* `optimizer`, der einen sogenannten Backpropagations-Algorithmus implementiert.

Zu Beginn werden den Gewichtungen zufällige Werte zugewiesen.

Das heißt, der anfängliche Verslustscore ist Anfangs dementsprechend hoch.

Bei einem „trainierten neuronalen Netz liegen die Istwerte so nah wie möglich an den Sollwerten

Mehrfachklassifizierung mit


```python
#tensorflow
import tensorflow as tf

#keras
import keras
from keras.datasets import mnist
from keras import models
from keras import layers
from keras.utils import to_categorical

#numpy
import numpy as np

#matplotlib
%matplotlib inline
import matplotlib.pyplot as plt

#time
import time
```

```python
#importieren der gesamten MNIST Datenbank
from keras.datasets import mnist
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
```

```python
#convert labels into one hot representation
train_labels_one_hot = to_categorical(train_labels)
test_labels_one_hot = to_categorical(test_labels)
```

```python
#Modell initialisieren bzw. definieren
network = models.Sequential()
network.add(layers.Reshape((28 * 28,), input_shape=(28,28)))
network.add(layers.Dense(512, activation='sigmoid'))
network.add(layers.Dense(324, activation='sigmoid'))
network.add(layers.Dense(36, activation='sigmoid'))
network.add(layers.Dense(10, activation='softmax'))
```

```python
#Modell kompilieren
network.compile(optimizer='rmsprop',
                loss='categorical_crossentropy',
                metrics=['accuracy'])
```

```python
# Modell trainieren (epochen, batch-größe...)
network.fit(train_images, train_labels_one_hot, epochs=20, batch_size=512)
```

```python
# Validierung des Trainingsdurchlaufs
acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']

## Hierzu 2 Diagramme erstellen mit Matplotlib
epochs = range(1, len(acc) + 1)
plt.plot(epochs, acc, 'bo', label='Trainings_acc')
plt.plot(epochs, val_acc, 'b', label='Validierungs_acc')
plt.title('Trainings- und Validierungsgenauigkeit (acc)')
plt.legend()
plt.show()
print("Anstieg der Genauigkeit über die Epochen")
plt.plot(epochs, loss, 'bo', label='Trainings_loss')
plt.plot(epochs, val_loss, 'b', label='Validierungs_loss')
plt.title('Trainings- und Validierungsverlustrate (loss)')
plt.legend()
plt.show()
print("Abnahme des Fehlers über die Epochen")
```

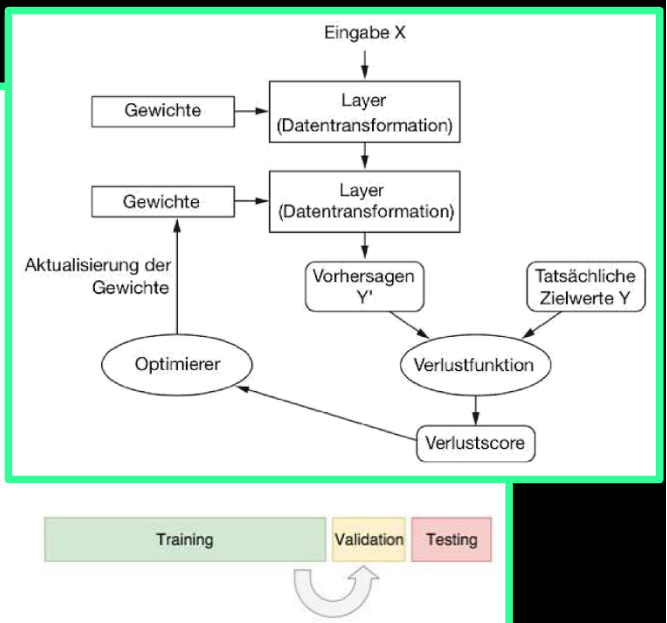
```python
# Evaluierung (Genauigkeit am Testset)
test_loss, test_acc = network.evaluate(test_images, test_labels_one_hot)
```

```python
# Testen
image_index = 4442
predictions = network.predict(test_images)

## Vorhergesagtes Label anzeigen
print("Vorhergesagtes Label aus dem Index", image_index, "lautet: ", np.argmax(predictions[image_index]), "\n")

## Grafische Darstellung des ausgewählten Datensatzes aus der Testdatenbank
test_im = test_images[image_index]
plt.imshow(test_im.reshape(28,28), cmap='Greys')
```

- Hyperparameter definieren die Architektur Neuronales Netzes
  - Definition der Layer, Neuronen und der Aktivierungsfunktion
  - Definition einer Loss-Funktion
  - Definition der Performance-Metrik
  - Optional: Definition von Regularization-Parameter (L1/L2, drop-out-rate)
  - Definition der Batch-Größe/Anzahl Epochen/Validation Anteil
  - Definition des Optimizers
- Model wird compiliert
- Trainieren der Training-Daten
  - Performanceverifikation am Validation-Satz
  - Optimieren der Gewichtungen
- Abschließend: **einmalige** Performanceverifikation



Fehlerminimierung:

Diese Suche nach der idealen Aktivierung bedeutet mathematisch das Finden von Minimalwerten im Fehlerraum. Gibt es nur zwei Parameter, so kann man eine Fehlerfunktion wunderschön visualisieren. Dabei können verschiedene mathematische Suchen implementiert werden. Der Klassiker ist sicherlich die Stochastic Gradient Descent (SGD) Methode, welche numerisch einen Anstieg der Funktion bestimmt und in die abfallende Richtung optimiert. Es gibt andere Verfahren wie Nesterov Momentum, Adagrad und Adadelta oder Rmsprop. Alle haben Vor- und Nachteile sowie Parameter, welche einzustellen sind.

```
#Gewichtsaktualisierungsformel:
#
##delta_wjk = alpha * Ek * OK (1-OK) * OJT
##
##delta_wjk = Gewichtsänderung der Gewichte zwischen versteckter (j) und Ausgabeschicht (k)
##alpha     = Lernrate
##EK        = Fehlerwerte der Ausgabeschicht << output_errors
##OK        = Ausgabewerte der Ausgabeschicht << final_outputs
##OjT       = Ausgabewerte der versteckten Schicht << hidden_outputs
#
#Gewichtsaktualisierungscode für zwischen versteckter Schicht und Ausgabeschicht:
self.who += self.lr * numpy.dot((output_errors * final_outputs * (1.0 - final_outputs)),
                                numpy.transpose(hidden_outputs))

#Gewichtsaktualisierungscode für zwischen Eingabeschicht und versteckter Schicht:
self.wih += self.lr * numpy.dot((hidden_errors * hidden_outputs * (1.0 - hidden_outputs)),
                                numpy.transpose(inputs))
```

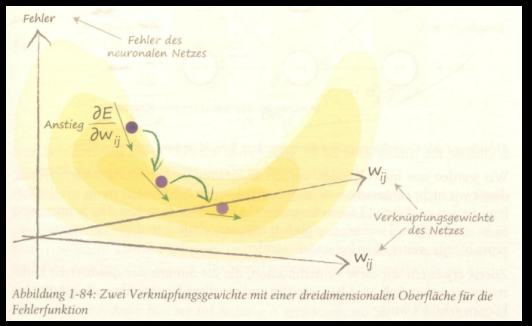


Abbildung 1-84: Zwei Verknüpfungsgewichte mit einer dreidimensionalen Oberfläche für die Fehlerfunktion

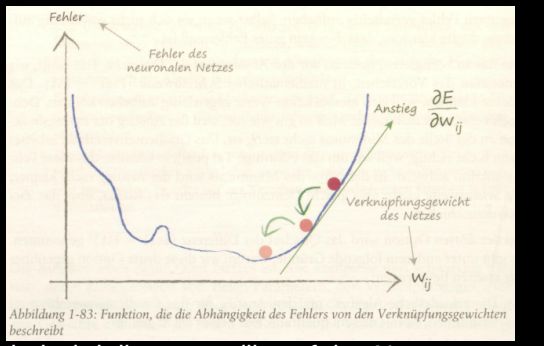


Abbildung 1-83: Funktion, die die Abhängigkeit des Fehlers von den Verknüpfungsgewichten beschreibt

Backpropagation:

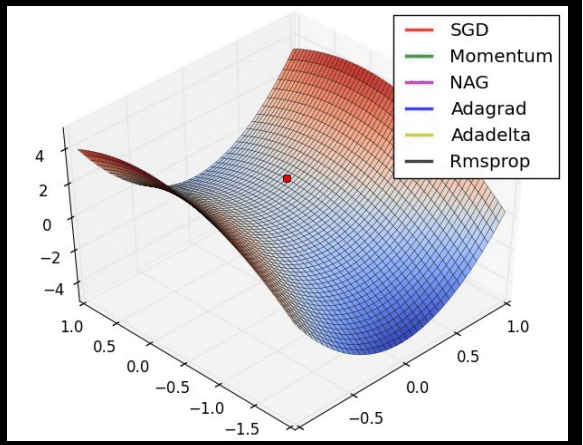
Der Fehler \(e\), den das Netzwerk bei einer Schätzung des Ausgangswertes macht, wird zurück an das Netzwerk gegeben. Dabei wird dieser anteilig auf das Neuron verteilt, welches maßgeblich am Fehler \(e\) beteiligt war. Das mathematische Verfahren ist die Backpropagation.

```
network.compile(optimizer='rmsprop',
                loss='categorical_crossentropy',
                metrics=['accuracy'])
```

$$\frac{\partial E}{\partial w_{jk}}$$

Dieser Ausdruck gibt an, wie sich der Fehler E ändert, wenn das Gewicht w_{jk} geändert wird. Das ist der Anstieg der Fehlerfunktion, die wir zum Minimum hinabsteigen wollen.

TensorFlow Optimizers
GradientDescentOptimizer
AdadeltaOptimizer
AdagradOptimizer
MomentumOptimizer
AdamOptimizer
FtrlOptimizer
RMSPropOptimizer



Aktivierungsfunktion / Activation Function:

Spricht man davon, dass ein Neuron ‘aktiviert’ wird (d.h. das Netzwerk lernt), so muss man eine mathematische Funktion \(f\) hinterlegen, die diese Aktivierung modelliert. Typisch und oft genutzt ist die Sigmoid Funktion…:

Sie gibt für einen Input je nach Aktivierung einen Output. Der funktionelle Zusammenhang ist nichtlinear. Je nachdem wie stark das Neuron aktiviert ist, gibt es ein Signal unterschiedlich stark weiter.

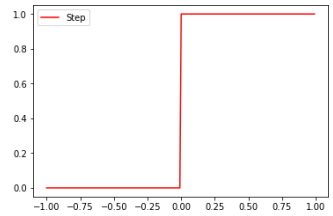
Glossar

```python
# Aktivierungsfunktion als Schwellenwert << das Feuern...
if u < 0:
    output=0
else:
    output=1
print("\nAusgabe der Schwellenwertfunktion: ", output, " = true\n")

print("Ergebnis des Vergleichsoperators »bool« = ", bool(u))
```

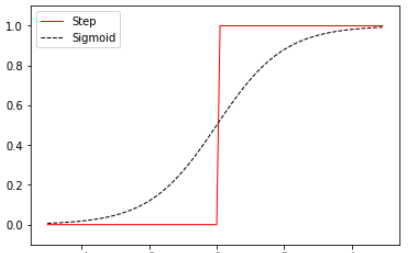
```
Ausgabe der Schwellenwertfunktion:  1  = true

Ergebnis des Vergleichsoperators »bool« =  True
```




```python
# Aktivierungsfunktion als Sigmoidfunktion
sigmoid = 1 / (1+ np.exp(-u))
print("Ausgabe der sigmoidfunktion: ", sigmoid)
```

```
Ausgabe der sigmoidfunktion:  0.5938731029341427
```



TensorFlow

https://keras.io/activations/

Activation functions

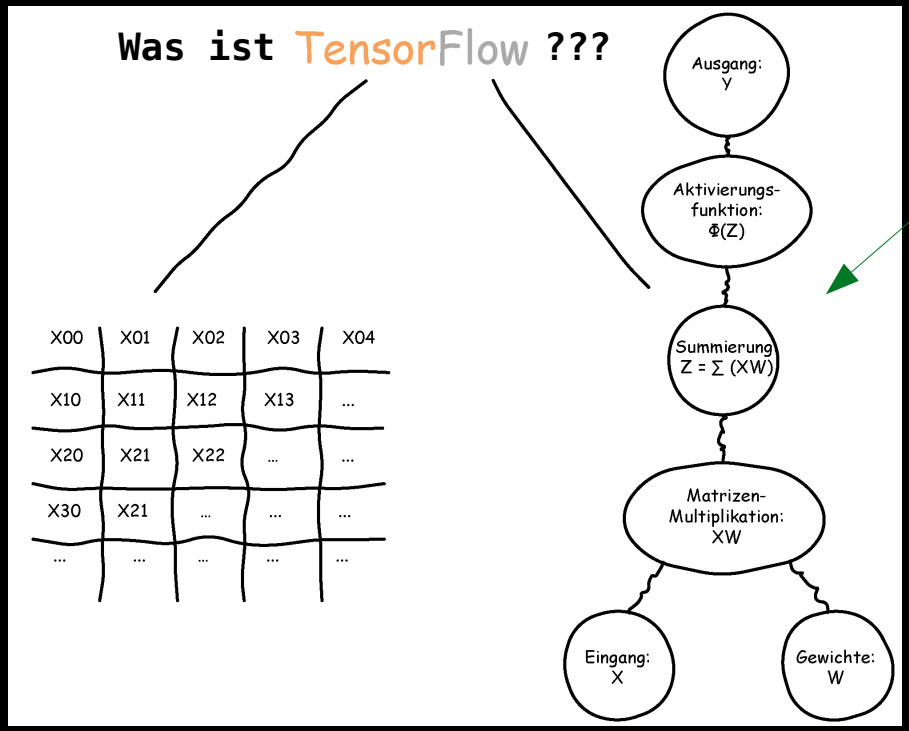
tf.nn?
relu
relu6
elu
softplus
softsign
dropout
bias_add
sigmoid
tanh
sigmoid_cross_entropy_with_logits
softmax
log_softmax
softmax_cross_entropy_with_logits
sparse_softmax_cross_entropy_with_logits
weighted_cross_entropy_with_logits
etc.

```python
network = models.Sequential()

network.add(layers.Dense(512, activation='sigmoid', input_shape=(28 * 28,)))

network.add(layers.Dense(36, activation='sigmoid'))

network.add(layers.Dense(10 , activation='softmax'))
```



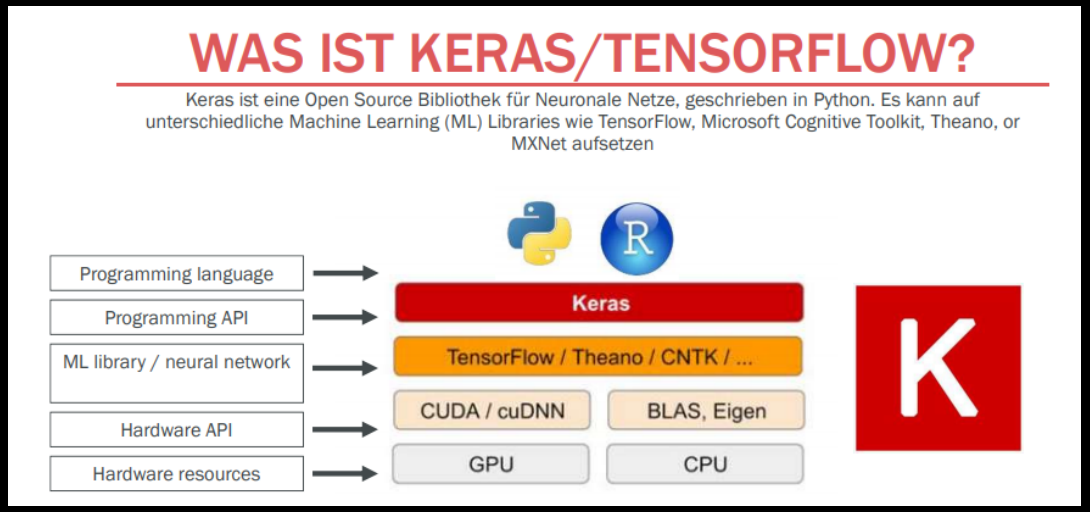
Allgemein betrachtet ist TensorFlow **ein Software-Framework zur numerischen Berechnung von Datenflussgraphen** mit dem Fokus maschinelle Lernalgorithmen zu beschreiben.

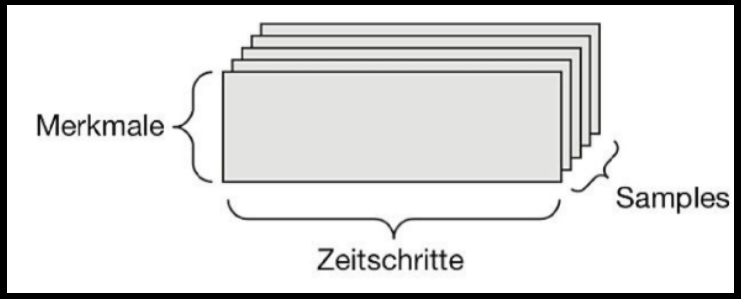
Die grundlegende Funktionsweise von TensorFlow basiert auf einem sog. Graphen.

Dieser bezeichnet eine abstrakte Darstellung des zugrunde liegenden mathematischen Problems in Form eines gerichteten Diagramms.

Das Diagramm besteht aus Knoten und Kanten die miteinander verbunden sind.

Die Knoten des Graphen repräsentieren in TensorFlow Daten und mathematische Operationen.





In Deep-Learning-Netzen können Eingangssignale, Gewichte etc. verschiedene Erscheinungsformen haben;

von Skalaren, zweidimensionalen Tabellen bis hin zu mehrdimensionalen Matrizen kann alles dabei sein.

Diese Erscheinungsformen werden in Tensorflow allgemein als *Tensoren* bezeichnet, welche durch ein Datenflussgraph ‘fließen’.

2-D-Tensoren (Datenmenge für Textdokumente. z.B. 5 Files mit jeweils 3000 zeichen)
Vektordaten mit der Shape:
Achse < Samples
Achse < Merkmale

3-D-Tensoren (Datenmenge aus z.b. Börsenkurs, z.B. jede minute der aktuelle börsenkurs, sowie der höchste und niedrigste kurs)
Zeitreihen oder sequenzielle Daten mit der Shape:
Achse < Samples
Achse < Zeitschritte
Achse < Merkmale

4-D-Tensoren (Bilder)
Bilder mit der Shape:
Achse < Samples (60000 bilder in der trainingsdatenbank)
Achse < Höhe (28px)
Achse < Breite (28px)
Farbtiefe (bei color 3, bei graustufen 1)

5-D-Tensoren (Videofiles)
Videos mit der Shape:
Achse < Samples (60000 bilder in der trainingsdatenbank)
Achse < Frames
Achse < Höhe (28px)
Achse < Breite (28px)
Farbtiefe (bei color 3, bei graustufen 1)

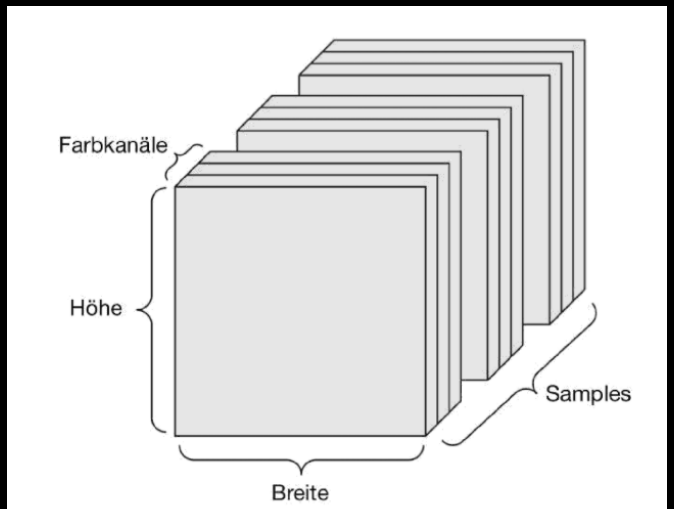


Abb. 2.4: Ein 4-D-Tensor zum Speichern von Bilddaten (gemäß der Konvention, die Farbkanäle zuerst anzugeben)

Python For Data Science *Cheat Sheet*

Keras

Learn Python for data science Interactively at www.DataCamp.com

Keras

Keras is a powerful and easy-to-use deep learning library for Theano and TensorFlow that provides a high-level neural networks API to develop and evaluate deep learning models.

A Basic Example

```
>>> import numpy as np
>>> from keras.models import Sequential
>>> from keras.layers import Dense
>>> data = np.random.random((1000,100))
>>> labels = np.random.randint(2,size=(1000,1))
>>> model = Sequential()
>>> model.add(Dense(32,
                    activation='relu',
                    input_dim=100))
>>> model.add(Dense(1, activation='sigmoid'))
>>> model.compile(optimizer='rmsprop',
                  loss='binary_crossentropy',
                  metrics=['accuracy'])
>>> model.fit(data,labels,epochs=10,batch_size=32)
>>> predictions = model.predict(data)
```

Data

Also see NumPy, Pandas & Scikit-Learn

Your data needs to be stored as NumPy arrays or as a list of NumPy arrays. Ideally, you split the data in training and test sets, for which you can also resort to the `train_test_split` module of `sklearn.cross_validation`.

Keras Data Sets

```
>>> from keras.datasets import boston_housing,
                               mnist,
                               cifar10,
                               imdb
>>> (x_train,y_train),(x_test,y_test) = mnist.load_data()
>>> (x_train2,y_train2),(x_test2,y_test2) = boston_housing.load_data()
>>> (x_train3,y_train3),(x_test3,y_test3) = cifar10.load_data()
>>> (x_train4,y_train4),(x_test4,y_test4) = imdb.load_data(num_words=20000)
>>> num_classes = 10
```

Other

```
>>> from urllib.request import urlopen
>>> data = np.loadtxt(urlopen("http://archive.ics.uci.edu/
ml/machine-learning-databases/pima-indians-diabetes/
pima-indians-diabetes.data"),delimiter=",")
>>> X = data[:,0:8]
>>> y = data [:,8]
```

Model Architecture

Sequential Model

```
>>> from keras.models import Sequential
>>> model = Sequential()
>>> model2 = Sequential()
>>> model3 = Sequential()
```

Multilayer Perceptron (MLP)

Binary Classification

```
>>> from keras.layers import Dense
>>> model.add(Dense(12,
                    input_dim=8,
                    kernel_initializer='uniform',
                    activation='relu'))
>>> model.add(Dense(8,kernel_initializer='uniform',activation='relu'))
>>> model.add(Dense(1,kernel_initializer='uniform',activation='sigmoid'))
```

Multi-Class Classification

```
>>> from keras.layers import Dropout
>>> 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'))
```

Regression

```
>>> model.add(Dense(64,activation='relu',input_dim=train_data.shape[1]))
>>> model.add(Dense(1))
```

Convolutional Neural Network (CNN)

```
>>> from keras.layers import Activation,Conv2D,MaxPooling2D,Flatten
>>> model2.add(Conv2D(32,(3,3),padding='same',input_shape=x_train.shape[1:]))
>>> model2.add(Activation('relu'))
>>> model2.add(Conv2D(32,(3,3)))
>>> model2.add(Activation('relu'))
>>> model2.add(MaxPooling2D(pool_size=(2,2)))
>>> model2.add(Dropout(0.25))
>>> model2.add(Conv2D(64,(3,3), padding='same'))
>>> model2.add(Activation('relu'))
>>> model2.add(Conv2D(64,(3, 3)))
>>> model2.add(Activation('relu'))
>>> model2.add(MaxPooling2D(pool_size=(2,2)))
>>> model2.add(Dropout(0.25))
>>> model2.add(Flatten())
>>> model2.add(Dense(512))
>>> model2.add(Activation('relu'))
>>> model2.add(Dropout(0.5))
>>> model2.add(Dense(num_classes))
>>> model2.add(Activation('softmax'))
```

Recurrent Neural Network (RNN)

```
>>> from keras.klayers import Embedding,LSTM
>>> model3.add(Embedding(20000,128))
>>> model3.add(LSTM(128,dropout=0.2,recurrent_dropout=0.2))
>>> model3.add(Dense(1,activation='sigmoid'))
```

Inspect Model

```
>>> model.output_shape
>>> model.summary()
>>> model.get_config()
>>> model.get_weights()
```

Model output shape
Model summary representation
Model configuration
List all weight tensors in the model

Compile Model

MLP: Binary Classification

```
>>> model.compile(optimizer='adam',
                  loss='binary_crossentropy',
                  metrics=['accuracy'])
```

MLP: Multi-Class Classification

```
>>> model.compile(optimizer='rmsprop',
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])
```

MLP: Regression

```
>>> model.compile(optimizer='rmsprop',
                  loss='mse',
                  metrics=['mae'])
```

Recurrent Neural Network

```
>>> model3.compile(loss='binary_crossentropy',
                   optimizer='adam',
                   metrics=['accuracy'])
```

Model Training

```
>>> model3.fit(x_train4,
               y_train4,
               batch_size=32,
               epochs=15,
               verbose=1,
               validation_data=(x_test4,y_test4))
```

Evaluate Your Model's Performance

```
>>> score = model3.evaluate(x_test,
                            y_test,
                            batch_size=32)
```

Prediction

```
>>> model3.predict(x_test4, batch_size=32)
>>> model3.predict_classes(x_test4,batch_size=32)
```

Save/ Reload Models

```
>>> from keras.models import load_model
>>> model3.save('model_file.h5')
>>> my_model = load_model('my_model.h5')
```

Model Fine-tuning

Optimization Parameters

```
>>> from keras.optimizers import RMSprop
>>> opt = RMSprop(lr=0.0001, decay=1e-6)
>>> model2.compile(loss='categorical_crossentropy',
                   optimizer=opt,
                   metrics=['accuracy'])
```

Early Stopping

```
>>> from keras.callbacks import EarlyStopping
>>> early_stopping_monitor = EarlyStopping(patience=2)
>>> model3.fit(x_train4,
               y_train4,
               batch_size=32,
               epochs=15,
               validation_data=(x_test4,y_test4),
               callbacks=[early_stopping_monitor])
```

Preprocessing

Also see NumPy & Scikit-Learn

Sequence Padding

```
>>> from keras.preprocessing import sequence
>>> x_train4 = sequence.pad_sequences(x_train4,maxlen=80)
>>> x_test4 = sequence.pad_sequences(x_test4,maxlen=80)
```

One-Hot Encoding

```
>>> from keras.utils import to_categorical
>>> Y_train = to_categorical(y_train, num_classes)
>>> Y_test = to_categorical(y_test, num_classes)
>>> Y_train3 = to_categorical(y_train3, num_classes)
>>> Y_test3 = to_categorical(y_test3, num_classes)
```

Train and Test Sets

```
>>> from sklearn.model_selection import train_test_split
>>> X_train5,X_test5,y_train5,y_test5 = train_test_split(X,
                                                         y,
                                                         test_size=0.33,
                                                         random_state=42)
```

Standardization/Normalization

```
>>> from sklearn.preprocessing import StandardScaler
>>> scaler = StandardScaler().fit(x_train2)
>>> standardized_X = scaler.transform(x_train2)
>>> standardized_X_test = scaler.transform(x_test2)
```

SAMPLE:

Einen “Datensatz” oder ein Beispiel nennt man ein Sample. In unserem Fall ist das eine handschriftlich geschriebene Ziffer.

Glossar

Label:

Das erwartete Ergebnis, das ein gut trainiertes DNN liefern soll. In unserem Fall schicken wir beispielsweise das Bild einer “3” durch das Netz und erwarten, dass das Netz auch eine Drei und nicht eine Neun erkennt.

Feature:

Eine einzelne Eigenschaft oder ein einzelner Attributwert eines Samples. In unserem Fall ist das der Grauwert eines einzelnen Bildpunktes. Unsere Bilder (Samples) sind aus 28 x 28 Bildpunkten zusammengesetzt und jeder Bildpunkt hat einen Grauwert von 0 für Weiß bis 255 für Schwarz. Im Deutschen passt auch das Wort Kriterium recht gut: “Was sind denn die Kriterien, die dein Beispiel unterlegen?”