



# Mathematisches Praktikum

## Grundlagen TensorFlow

**M.Sc. Johannes Michael**

Universität Rostock



- TensorFlow
- Tensoren
- Berechnungsgraph
- TensorFlow 1.x vs 2.x
- TensorFlow 2.0 Toolkits Hierarchie
- Tensor-Basics
- Tensor-Transformationen
- Tutorial: Lineare Regression

## TensorFlow

- Plattformunabhängige Open-Source-Programmbibliothek für maschinelles Lernen von Google
- Kanonische import-Anweisung in Python:  
`import tensorflow as tf`
- Unterstützt numerische Berechnungen unter Verwendung von Datenflussgraphen
- Kernelement für Daten in TensorFlow: **Tensor** ( $n$ -dim. array)



## Tensoren

- Tensoren sind immutable (Ausnahme: `tf.Variable`).
- **Rang** (rank) und **Form** (shape) eines Tensors analog zu NumPy arrays

Tensor	Rank	Shape	Bemerkung
2	0	[]	Skalar
[1, 2]	1	[2]	Vektor
[[1, 2], [3, 4]]	2	[2, 2]	Matrix
[[[1, 2], [3, 4]], [[5, 6], [7, 8]]]	3	[2, 2, 2]	3-dim. Array

Scalar      Vector      Matrix      Tensor

Source: "TensorFlow 2.0 Tutorial for Beginners" ([youtube.com/watch?v=QPDsEtUK\\_D4](https://youtube.com/watch?v=QPDsEtUK_D4))



## Berechnungsgraph I

- Klassische TensorFlow Programme bestehen aus 2 Phasen:
  - 1) Erzeugung des Berechnungsgraphen.
  - 2) Ausführung des Berechnungsgraphen.
- Der **Berechnungsgraph** (computational graph) bildet ein Netz von Knoten, die über Kanten verbunden werden und repräsentiert Berechnungen als Abhängigkeiten zwischen individuellen Operationen.
  - Knoten: TensorFlow Operationen (*ops*) mit  $n \geq 0$  Eingabetensoren und 1 Ausgabetensor.
  - Kanten: Datenfluss über Tensoren.

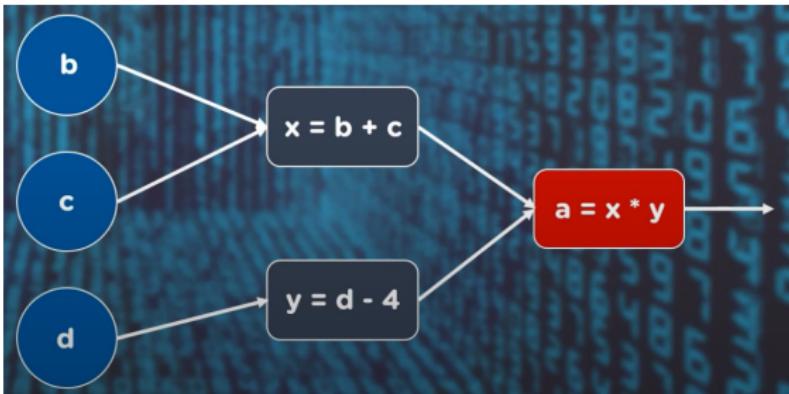


## Berechnungsgraph II

- Berechne folgende Funktion:

$$a(b, c, d) = (b + c) * (d - 4)$$

- Zugehöriger Datenflussgraph:



Source: "TensorFlow 2.0 Tutorial for Beginners" ([youtube.com/watch?v=QPDsEtUK\\_D4](https://www.youtube.com/watch?v=QPDsEtUK_D4))

## Berechnungsgraph III

- Warum Datenflussgraphen?
  - **Parallelität:** Kanten im Graph repräsentieren Abhängigkeiten zwischen Operationen → Das System kann Operationen identifizieren die parallel ausgeführt werden können.
  - **Verteilte Ausführung:** TensorFlow kann das Programm über mehrere Geräte (CPUs, GPUs) verteilen und die nötige Koordination selbstständig hinzufügen.
  - **Kompilieren:** TensorFlows XLA compiler kann den Datenflussgraph optimieren (z.B. Operationen verschmelzen).
  - **Portierbarkeit:** Datenflussgraph ist sprachen-unabhängige Modelrepräsentation → In Python gebauter Graph kann gespeichert und in Java geladen werden.



## TensorFlow 1.0 vs 2.0

### TensorFlow 1.x

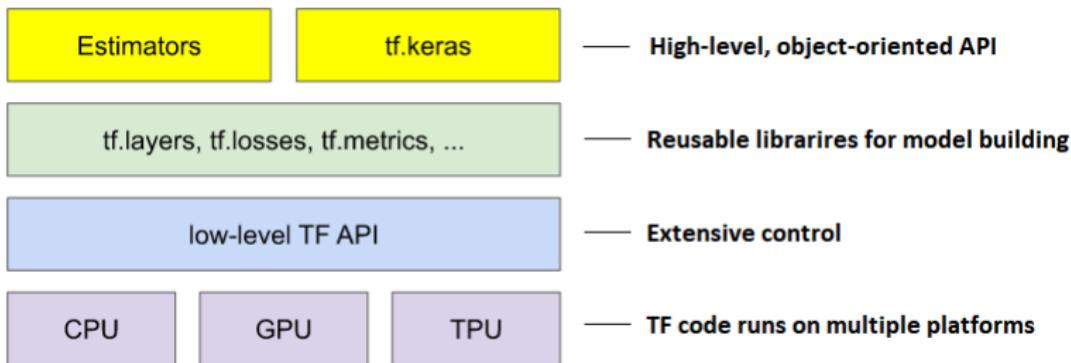
- Arbeitet *nur* mit Datenflussgraphen
  - Eingaben über `tf.placeholder()`
  - Auswertung über `tf.Session()`
- outputs = `session.run(f(placeholder), feed_dict={placeholder:input})`
- Eingeschränkte Python Funktionen im Graphmodus

### TensorFlow 2.x

- Unterstützt standardmäßig *eager execution*  
`outputs = f(input)`
- `tf.function()` decorator lässt Python Funktionen im Graphmodus laufen
  - *AutoGraph* feature wandelt Python Code automatisch in TensorFlow ops um



## TensorFlow 2 Toolkits Hierarchie



Source: [developers.google.com/machine-learning/crash-course/first-steps-with-tensorflow/toolkit](https://developers.google.com/machine-learning/crash-course/first-steps-with-tensorflow/toolkit)

- **tf.keras** ist die offizielle high-level API von TensorFlow 2.0
- Stellt mehrere APIs zum Bauen von Modellen zur Verfügung (*Sequential, Functional, Subclassing*)



## Tensor-Basics I

- `tf.constant(value, dtype=None, shape=None)`  
(intern gespeicherter, unveränderlicher Tensor)
- Standardops `+, -, *, /` in TensorFlow überladen, unterstützen Broadcasting, erwarten passende dtypes  
(`tf.add`, `tf.subtract`, `tf.multiply`, `tf.divide`)

```
c1 = tf.constant(3.0) # Skalar
c2 = tf.constant(5.0) # Skalar
c3 = c1 + c2 # oder tf.add(c1, c2)
c4 = c1 * c2 # oder tf.multiply(c1, c2)
print(c1, c2, c3, c4, sep="\n")
# tf.Tensor(3.0, shape=(), dtype=float32)
# tf.Tensor(5.0, shape=(), dtype=float32)
# tf.Tensor(8.0, shape=(), dtype=float32)
# tf.Tensor(15.0, shape=(), dtype=float32)
```



## Tensor-Basics II

- Datentyp von Tensoren kann verändert werden

```
x = tf.constant([[8, 8], [6, 6]]) # int32  
x = tf.cast(x, dtype=tf.float32) # float32
```

- Diverse Tensor-Initialisierungen analog zu NumPy

```
tf.constant(list(range(1, 9)), shape=[4, 2]) # Shape-Umformung  
tf.ones(shape) # Einsen  
tf.zeros(shape) # Nullen  
tf.range(start, limit, delta) # gleichverteilte Werte aus [start, limit), analog zu range()  
tf.linspace(start, stop, num) # 'num' viele gleichverteilte Werte aus [start, stop]  
tf.random.uniform(shape, minval, maxval) # Zufallswerte aus Gleichverteilung  
tf.random.normal(shape, mean, stddev) # Zufallswerte aus Normalverteilung
```

... und viele weitere (siehe [tensorflow.org/api\\_docs/python/tf](https://tensorflow.org/api_docs/python/tf))



## Tensor-Basics III

- **Variablen** repräsentieren Tensoren die zur Laufzeit veränderlich sind

- `tf.Variable(initial_value, dtype=None, shape=None)`

```
x = tf.Variable([[1, 2, 3]], name="x")
print(x) # <tf.Variable 'x:0' shape=(1, 3) dtype=int32, numpy=array([[1, 2, 3]])>
x.assign(x + tf.ones_like(x)) # Variable neuen Wert zuweisen
print(x) # <tf.Variable 'x:0' shape=(1, 3) dtype=int32, numpy=array([[2, 3, 4]])>
```

- TensorFlow Operationen verarbeiten Tensoren und Variablen gleichermaßen

```
y = tf.ones([3, 3], dtype=tf.int32)
z = tf.matmul(x, y) # Matrixmultiplikation
print(z) # tf.Tensor([[9 9 9]], shape=(1, 3), dtype=int32)
```



## Tensor-Basics IV

- Kompatibilität zu NumPy arrays:
  - TensorFlow Operationen konvertieren NumPy arrays zu Tensoren
  - NumPy Operationen konvertieren Tensoren zu NumPy arrays
  - Tensoren können explizit zu NumPy arrays umgewandelt werden

```
ndarray = np.ones([2, 3])
tensor = tf.multiply(ndarray, 5)
array = np.add(tensor, -3)
print(tensor) # tf.Tensor([[5. 5. 5.]], shape=(1, 3), dtype=float64)
print(array, type(array)) # [[2. 2. 2.]] <class 'numpy.ndarray'>
print(tensor.numpy(), type(tensor.numpy())) # [[5. 5. 5.]] <class 'numpy.ndarray'>
```

- Indizieren und Slicen analog zu NumPy arrays
- [start:stop:step] für jede Dimension des Tensors

## Tensor-Transformationen I

- `tf.reshape(tensor, shape)`:  
Umformung der Dimensionen

```
tensor = tf.constant(2.0, shape=[28, 28])
t1 = tf.reshape(tensor, [14, 56])
t2 = tf.reshape(tensor, [-1, 2]) # -1 entspricht "restlicher" shape
t3 = tf.reshape(tensor, [28, 14, 2])
t4 = tf.reshape(tensor, [28, 1, 28])
```

- `tf.expand_dims(tensor, dim)`:  
Hinzufügen einer 1-Dimension

```
tensor = tf.constant(2.0, shape=[28, 28])
t_expand = tf.expand_dims(tensor, axis=1) # (28, 1, 28)
```

- `tf.squeeze(tensor, dim)`:  
Löschen einer (aller) 1-Dimension(en)



## Tensor-Transformationen II

```
tensor = tf.constant(2.0, shape=[28, 1, 1, 28, 1])
t_squeeze = tf.squeeze(tensor, 1) # (28, 1, 28, 1)
t_squeeze_2 = tf.squeeze(t_squeeze) # (28, 28)
```

- `tf.transpose(tensor, perm):`  
Transponieren (Permutieren der Dimensionen)

```
tensor = tf.constant(list(range(12)), shape=[3, 4])
t_transpose = tf.transpose(tensor) # perm=None entspricht [n-1, n-2, ..., 0]
tensor = tf.constant(list(range(24)), shape=[2, 3, 4])
t_transpose = tf.transpose(tensor, [1, 2, 0]) # (3, 4, 2)
```

- `tf.concat(tensors, axis):`  
Liste von Tensoren entlang einer Dimension konkatenieren
- `tf.unstack(tensor, axis):`  
Tensor entlang einer Dimensionen in Liste von Sub-Tensoren entpacken

## Tensor-Transformationen III

- `tf.stack(tensors, axis):`

Liste von Tensoren entlang einer Dimension in einen Tensor packen

```
t1 = tf.constant(list(range(12)), shape=[3, 4])
t2 = tf.constant(list(range(8)), shape=[2, 4])
t_concat = tf.concat([t1, t2], axis=0) # (5, 4)
t_unstack = tf.unstack(t_concat, axis=1) # list of 4 vectors with shape (5,)
t_stack = tf.stack(t_unstack, axis=0) # (4, 5)
```

## Tutorial: Lineare Regression I

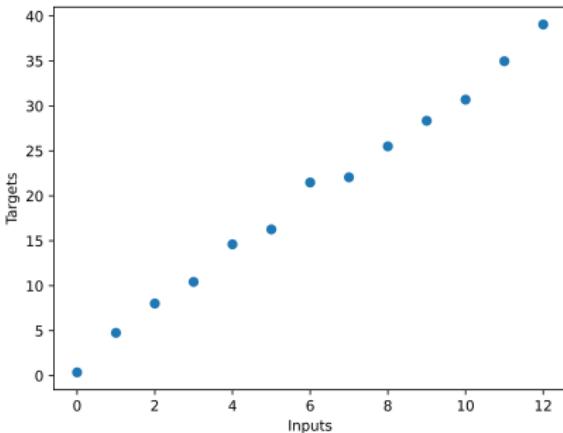
### Überblick

- Daten einer linearen Funktion verrauschen
- Modell in TensorFlow anlegen um Funktion zu approximieren
  - low-level: TensorFlow Variablen und Operationen
  - high-level: `tf.keras` models (*Sequential, Functional*)
- Modell-Performance evaluieren → Fehlerfunktion (loss function)
- Modell mit Daten trainieren → Gradientenabstiegsverfahren
  - low-level: `tf.GradientTape`
  - high-level: `model.fit`
- Trainiertes Modell testen



## Tutorial: Lineare Regression II

- Verrauschte Daten der Funktion  $f(x) = 3x + 2$ :



```
data = np.arange(0, 13, dtype=np.float32)
labels = data * 3 + 2 + np.random.normal(0, 1, size=data.shape)
labels = labels.astype(np.float32)
```



## Tutorial: Lineare Regression III

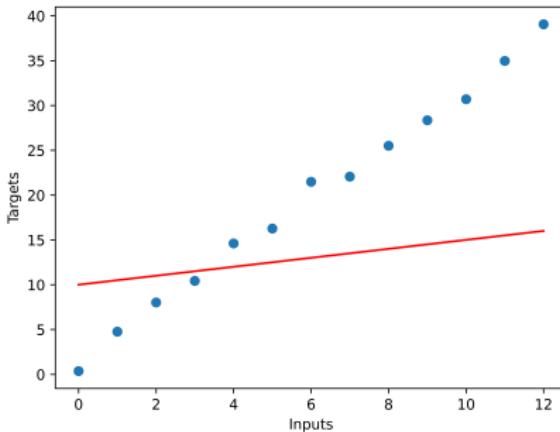
### Modell (low-level)

- Modell-Parameter:

```
w = tf.Variable([0.5])  
b = tf.Variable([10.0])
```

- Lineares Modell:

```
def linear_model(inputs):  
    return w * inputs + b
```



## Tutorial: Lineare Regression IV

### Modell-Performance

- **Fehlerfunktion** (loss function) misst den Fehler zwischen den Vorhersagen des Modells und den gewünschten Ausgaben
- Standardzielfunktion für lineare Regression: Mittelwert der Fehlerquadrate:  $\mathcal{L} = \frac{1}{n} \sum_{i=1}^n (y_i - f(x_i))^2$

```
def loss_fn(inputs, targets):
    return tf.reduce_mean(tf.square(targets - inputs))
```

- Diverse Lossfunktionen im maschinellen Lernen  
→ TensorFlow keras.losses  
([tensorflow.org/api\\_docs/python/tf/keras/losses](https://tensorflow.org/api_docs/python/tf/keras/losses))  
`tf.keras.losses.mean_squared_error(targets, linear_model(inputs))`

## Tutorial: Lineare Regression V

### Modell-Optimierung

- Diverse Optimierungsverfahren im maschinellen Lernen
  - TensorFlow keras.optimizers  
([tensorflow.org/api\\_docs/python/tf/keras/optimizers](https://tensorflow.org/api_docs/python/tf/keras/optimizers))
- Optimierer versuchen die Lossfunktion zu minimieren, indem die zugrundeliegenden Modellvariablen angepasst werden
- Grundidee: **Gradientenabstieg** zum Finden lokaler/globaler Minima
  - Variablen werden in Richtung des negativen Gradienten der Fehlerfunktion verschoben (hier:  $-\frac{\partial \mathcal{L}}{\partial w}$  bzw.  $-\frac{\partial \mathcal{L}}{\partial b}$ ).

```
optimizer = tf.keras.optimizers.SGD(learning_rate=0.01)
```

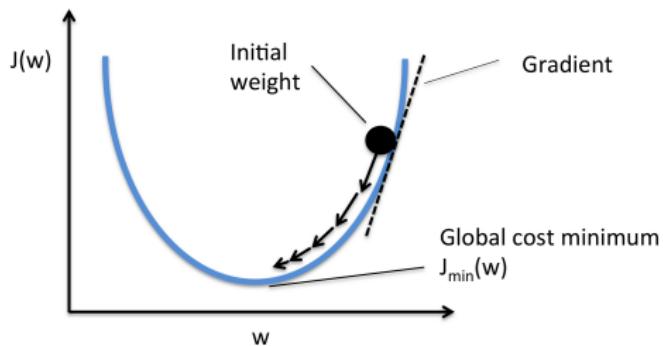


## Tutorial: Lineare Regression VI

### Gradientenabstieg

- Gradientenabstieg für Funktion  $J: \mathbb{R}^n \rightarrow \mathbb{R}$

$$\mathbf{w}^{(i+1)} = \mathbf{w}^{(i)} - \alpha \nabla f(\mathbf{w}^{(i)})$$



[rasbt.github.io/mlxtend/user\\_guide/general\\_concepts/gradient-optimization](https://rasbt.github.io/mlxtend/user_guide/general_concepts/gradient-optimization)



## Tutorial: Lineare Regression VII

### Berechnung der Gradienten

- TensorFlow unterstützt automatische Differentiation
- `tf.GradientTape()` erzeugt eine Umgebung in der TensorFlow Operationen zur automatischen Differentiation aufgezeichnet werden
- `tape.gradient(target, sources)` berechnet die Gradienten der *target* Tensoren bzgl. der *sources* Variablen

```
@tf.function
def gradients(inputs, targets):
    with tf.GradientTape() as tape:
        outputs = linear_model(inputs)
        loss = loss_fn(outputs, targets)
        grads = tape.gradient(loss, [w, b])
    return loss, grads
```

## Tutorial: Lineare Regression VIII

### Modell-Training

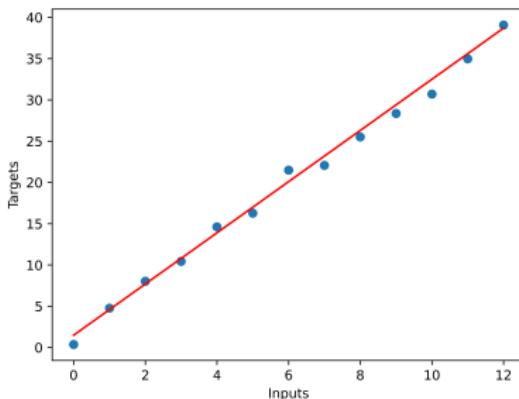
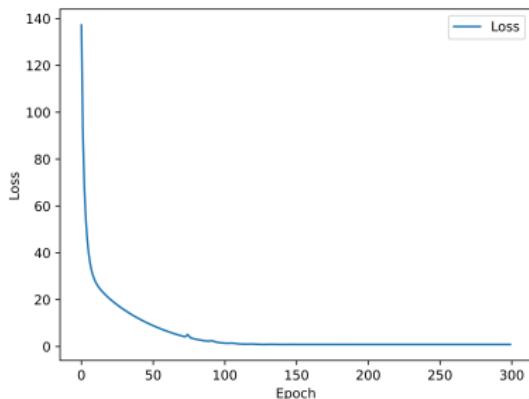
- Mehrfach über den Datensatz iterieren (*Epochen*)
- Gradienten berechnen
- Optimierungsschritt durchführen
  - `optimizer.apply_gradients(grads_and_vars)`, wobei `grads_and_vars` eine Liste von Tupeln (*gradient*, *variable*) ist

```
num_epochs = 300
for epoch in range(num_epochs):
    loss, grads = gradients(data, labels)
    optimizer.apply_gradients(zip(grads, [w, b]))
```



## Tutorial: Lineare Regression IX

- Modell-Evaluation



- Trainierte Parameter:  $w = [3.1010873]$ ,  $b = [1.4811424]$

## Tutorial: Lineare Regression X

### tf.keras

- Erlaubt es komplexere Modelle aus vordefinierten Schichten (`keras.layers`) zusammenzubauen
  - *Sequential API* (`keras.models.Sequential`)
  - *Functional API* (`keras.Model(inputs, outputs)`)
  - *Subclassing API* (eigene layers, Models)  
→ Bequemlichkeit vs. Flexibilität
- Ein layer kapselt seine Variablen (Gewichte, weights) und eine Input-Output-Transformation  
([tensorflow.org/api\\_docs/python/tf/keras/layers](https://tensorflow.org/api_docs/python/tf/keras/layers))
- `model.fit` übernimmt den gesamten Trainingsprozess eines keras Modells und liefert eine `history` über den Verlauf des Trainings

## Tutorial: Lineare Regression XI

### Sequential Modell

- `keras.model.Sequential` erhält eine Liste von `keras.layers`, welche sequentiell ausgeführt werden
- `model.compile` erhält einen Optimizer, sowie eine Loss- und Metrikfunktion ([tensorflow.org/api\\_docs/python/tf/keras/metrics](https://tensorflow.org/api_docs/python/tf/keras/metrics))

```
def sequential_model(learning_rate):  
    model = keras.models.Sequential(  
        [keras.layers.Input(shape=(1,)), # shape of a single input  
         keras.layers.Dense(units=1)]) # fully-connected layer  
    model.compile(optimizer=keras.optimizers.RMSprop(learning_rate=learning_rate),  
                  loss=keras.losses.mean_squared_error,  
                  metrics=[keras.metrics.mean_squared_error])  
    return model
```



## Tutorial: Lineare Regression XII

### Modell-Training

- `model.fit` erhält
  - die Eingabedaten und zugehörigen Targets
  - die `batch_size` (Größe der Blöcke/Batches in der die Daten verarbeitet werden)
  - die Anzahl der Epochen zum Trainieren
- und liefert eine `history` mit Informationen über die Epochen, den Loss und die Metrik

```
history = model.fit(x=inputs,  
                     y=targets,  
                     batch_size=batch_size,  
                     epochs=epochs)  
  
epochs = history.epochs  
losses = history.history['loss']  
errors = history.history['mean_squared_error']
```

## Sonstiges

- Modellausgaben erzeugen mit `model.predict`
- In keras aufbereitete Datensätze:
  - `keras.datasets.mnist.load_data()`
    - Multi-Klassifizierungsproblem (Ziffern 0-9)
  - `keras.datasets.fashion_mnist.load_data()`
    - Multi-Klassifizierungsproblem (Kleidungsstücke)
  - weitere siehe <https://keras.io/api/datasets/>