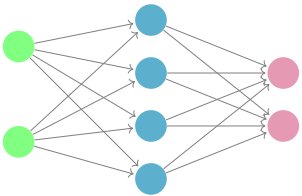
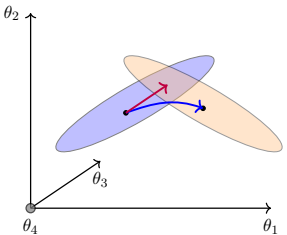
Continual Learning in Neuronalen Netzen

Prof. Dr. Jörg Frochte

Maschinelles Lernen





Warum Continual Learning?


1


1

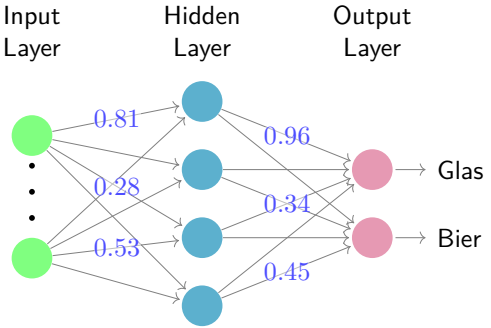
0

0

Aufgabe A

Warum Continual Learning?

Aufgabe A

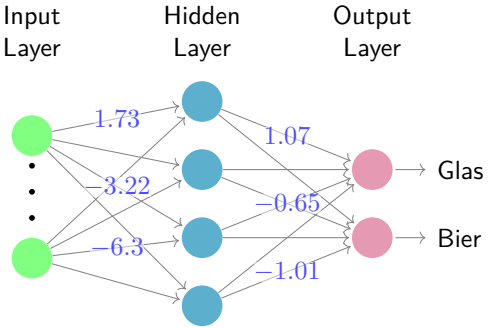


Epoch: 1

Training auf Aufgabe A...

Warum Continual Learning?

Aufgabe A

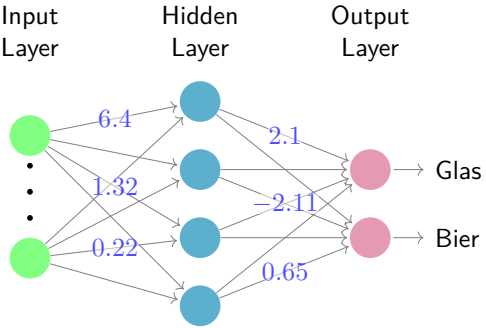


Epoch: 2

Training auf Aufgabe A...

Warum Continual Learning?

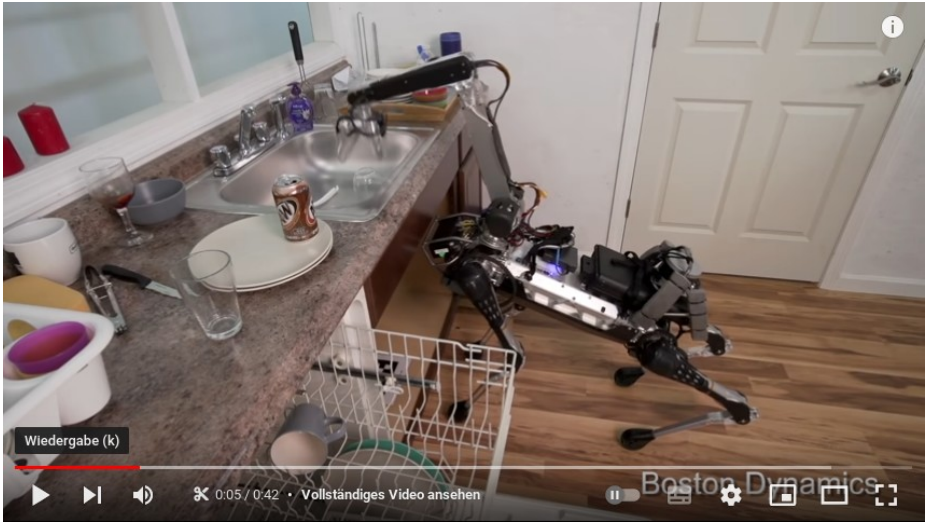
Aufgabe A



Epoch: 3

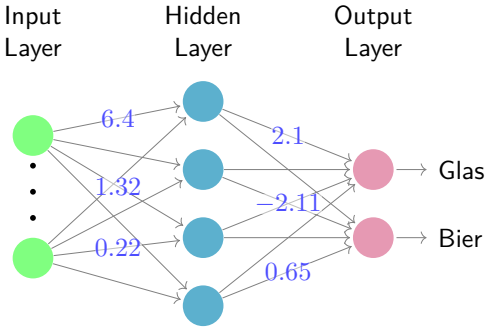
Fertig trainiertes Netz

Warum Continual Learning?



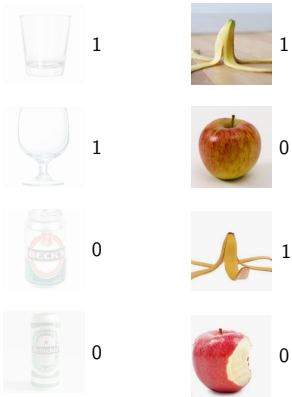
Das Problem des katastrophalen Vergessens

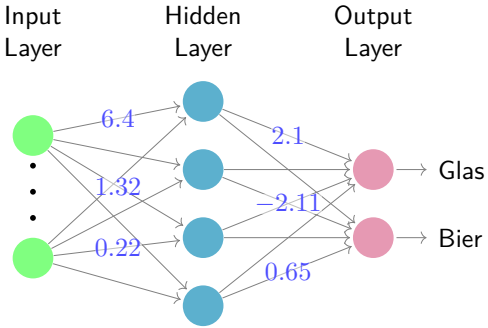
Aufgabe A



Auf Aufgabe A fertig trainiertes Netz

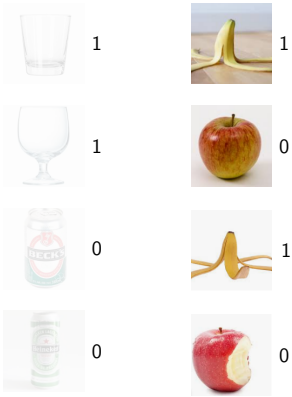
Das Problem des katastrophalen Vergessens

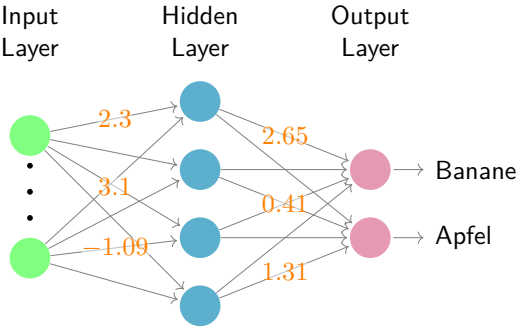




Auf Aufgabe A fertig trainiertes Netz

Das Problem des katastrophalen Vergessens

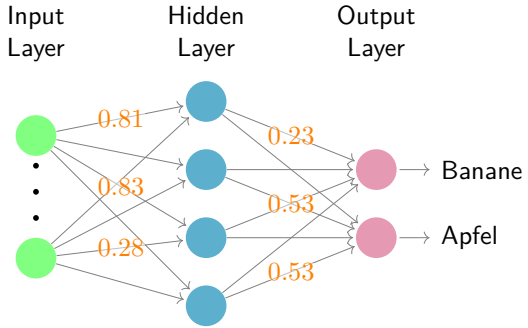




Epoch: 1

Training auf Aufgabe B...

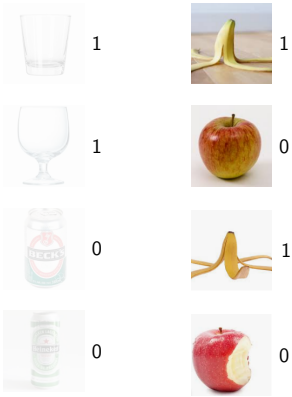
Das Problem des katastrophalen Vergessens

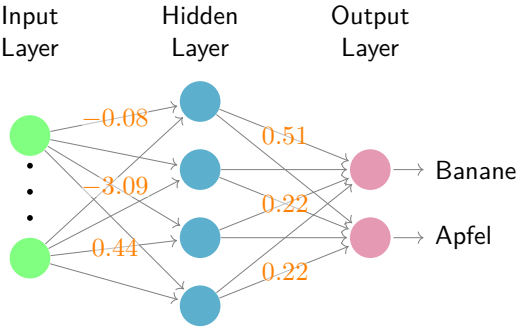


Epoch: 2

Training auf Aufgabe B...

Das Problem des katastrophalen Vergessens



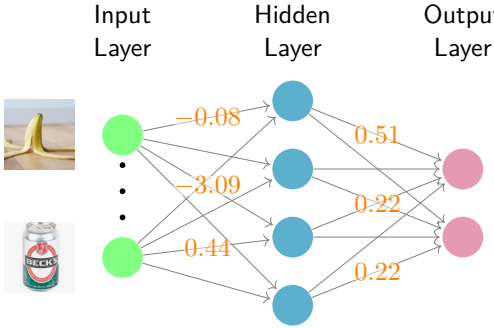


Epoch: 3

Auf Aufgabe B fertig trainiertes Netz

Das Problem des katastrophalen Vergessens

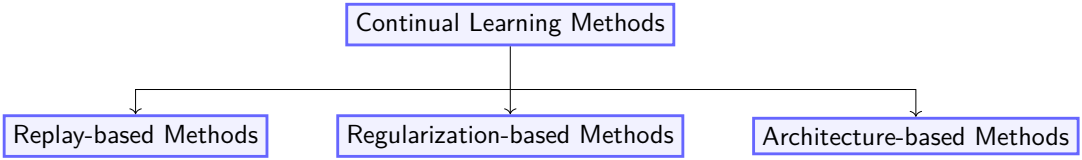
- Das Netz kann Aufgabe B hervorragend lösen, vergisst dabei aber völlig Aufgabe A.
- Das ist das sogenannte "Catastrophic Forgetting" Problem.
- **Continual Learning** zielt darauf ab, das katastrophale Vergessen zu überwinden und dabei die Modellkapazität, die Rechenkosten und den Speicherbedarf zu begrenzen.



Accuracy auf B: 96%

Accuracy auf A: 14%

Lösungsansätze zum katastrophalen Vergessen



- Seit der Mitte des letzten Jahrzehnts nimmt das Thema immer mehr Fahrt auf.
- Das Thema betrifft nicht exklusiv neuronale Netze, aber wir legen heute auf diesen Bereich den Fokus.
- Dabei haben sich drei größere Methoden-Familien herausgebildet.
- Die Einteilung ist nicht exklusiv, sondern es gibt einige Mischformen als Ansätze und weiche Übergänge.

Regularisierungsbasierte Methoden

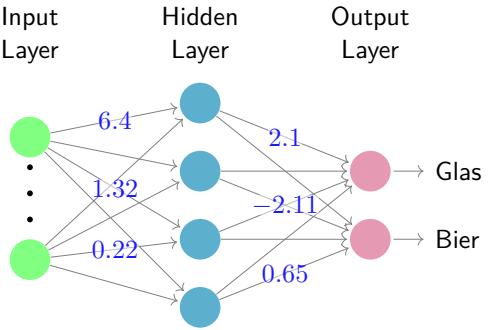

1


1

0

0

Aufgabe A



Fertig trainiertes Netz

$$\theta_A^* = \{6.4, 1.32, 0.22, 2.1, -2.11, 0.65\}$$

Regularisierungsbasierte Methoden

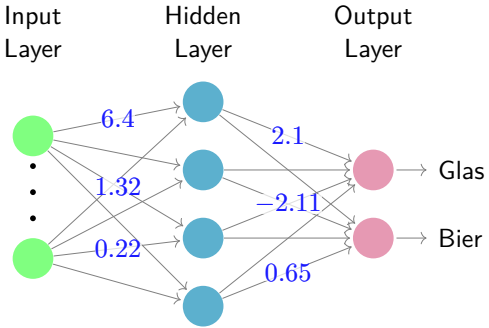

1


1

0

0

Aufgabe A



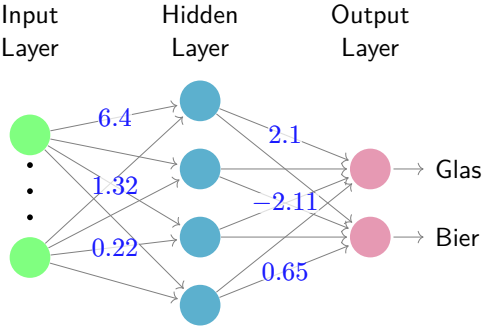
Fertig trainiertes Netz

$$\theta_A^* = \{6.4, 1.32, 0.22, 2.1, -2.11, 0.65\}$$

Ist diese Lösung eindeutig?

Regularisierungsbasierte Methoden

Aufgabe A



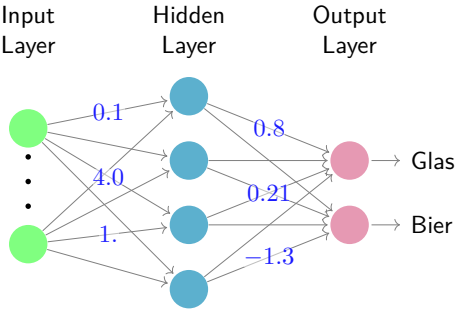
Fertig trainiertes Netz

$$\theta_A^* = \{6.4, 1.32, 0.22, 2.1, -2.11, 0.65\}$$

Ist diese Lösung eindeutig? **Nein**

Regularisierungsbasierte Methoden

- Regularization-based Methods nutzen die Tatsache, dass neuronale Netze **überparametrisiert** sind.
- Dies bedeutet, dass es viele Konfigurationen von θ gibt, die zur gleichen Leistung führen.
- Im Falle von 2 Aufgaben A und B, macht die Überparametrisierung es wahrscheinlich, dass es eine Lösung θ_B^* für Aufgabe B gibt, die **nahe an der zuvor gefundenen Lösung** θ_A^* für Aufgabe A liegt.



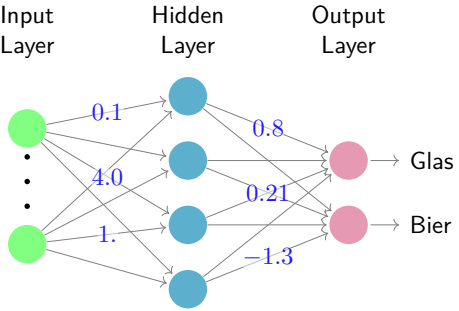
Fertig trainiertes Netz

$$\theta_A^* = \{6.4, 1.32, 0.22, 2.1, -2.11, 0.65\}$$

$$\theta_A^* = \{0.1, 4.0, 1., 0.8, 0.21, -1.3\}$$

Regularisierungsbasierte Methoden

- Regularization-based Methods nutzen die Tatsache, dass neuronale Netze **überparametrisiert** sind.
- Dies bedeutet, dass es viele Konfigurationen von θ gibt, die zur gleichen Leistung führen.
- Im Falle von 2 Aufgaben A und B, macht die Überparametrisierung es wahrscheinlich, dass es eine Lösung θ_B^* für Aufgabe B gibt, die **nahe an der zuvor gefundenen Lösung** θ_A^* für Aufgabe A liegt.



Fertig trainiertes Netz

$$\theta_A^* = \{6.4, 1.32, 0.22, 2.1, -2.11, 0.65\}$$

$$\theta_A^* = \{0.1, 4.0, 1., 0.8, 0.21, -1.3\}$$

Regularisierungsbasierte Methoden

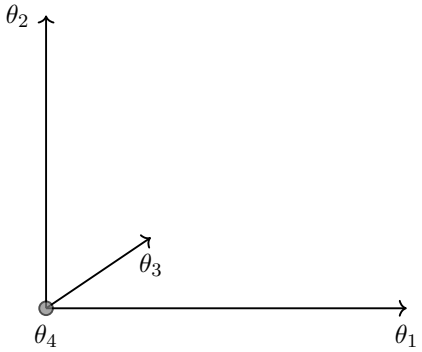
1

1

0

0

Aufgabe A



Regularisierungsbasierte Methoden

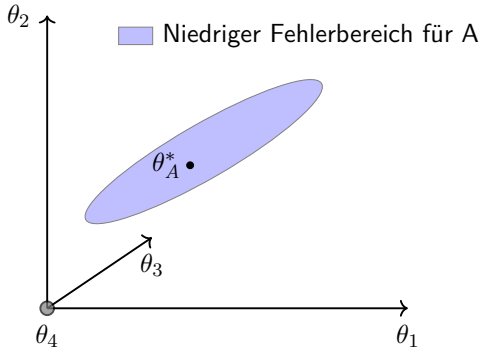

1


1


0


0

Aufgabe A



Regularisierungsbasierte Methoden

1


1

1


0

0

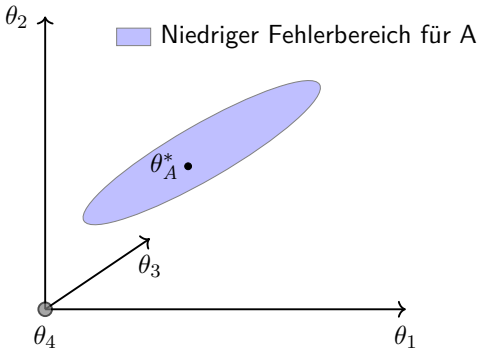

1

0

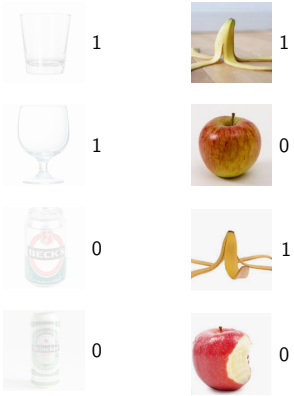

0

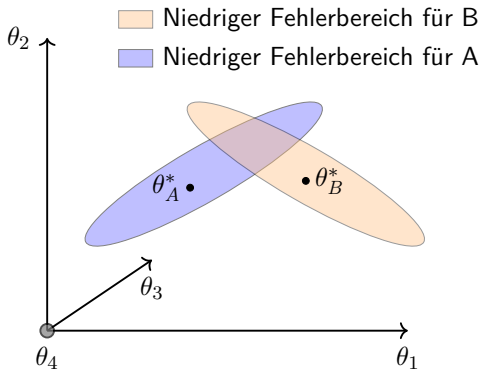
Aufgabe A

Aufgabe B

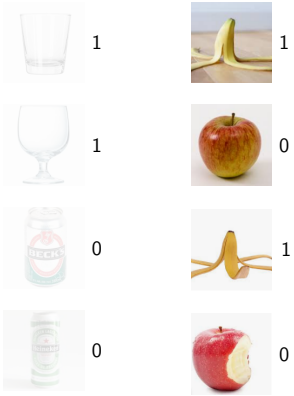


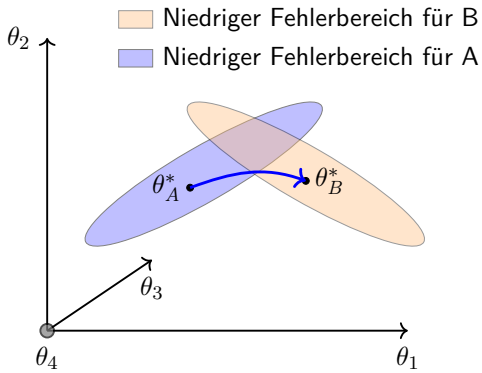
Regularisierungsbasierte Methoden



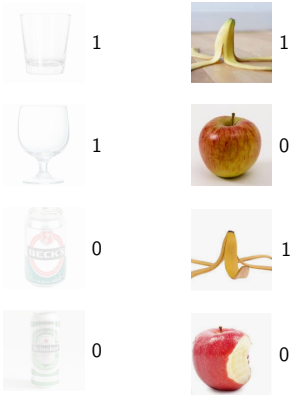


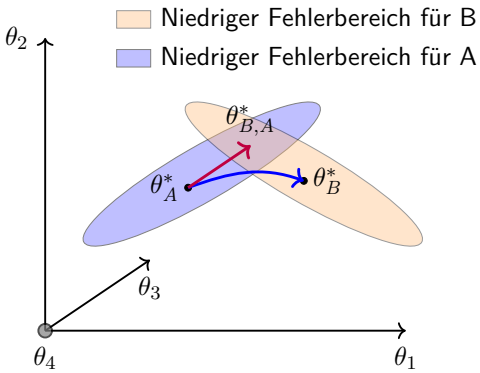
Regularisierungsbasierte Methoden





Regularisierungsbasierte Methoden



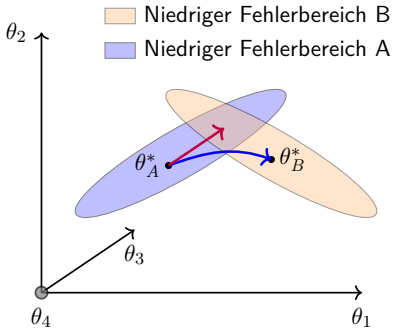


Elastic Weight Consolidation (EWC)

- Einer der bekanntesten Methoden ist Elastic Weight Consolidation (**EWC**) aus dem Jahr 2016.
- Hierfür verändert man das Fehlerfunktional $\mathcal{L}$ wie folgt:

$$\mathcal{L}(\theta) = \mathcal{L}_B(\theta) + \frac{\lambda}{2} \sum_{i=1}^{4} I_i \cdot (\theta_i - \theta^*_{A,i})^2$$

- I ist ein Maß wie wichtig Gewichte für die Aufgabe A waren, Bsp: $I = [1, 0, 0, 1]$.

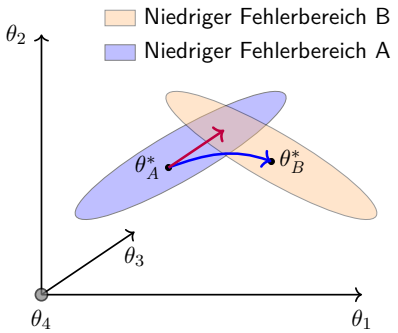


Elastic Weight Consolidation (EWC)

- Einer der bekanntesten Methoden ist Elastic Weight Consolidation (**EWC**) aus dem Jahr 2016.
- Hierfür verändert man das Fehlerfunktional $\mathcal{L}$ wie folgt:

$$\mathcal{L}(\theta) = \mathcal{L}_B(\theta) + \frac{\lambda}{2} \sum_{i=1}^{4} I_i \cdot (\theta_i - \theta^*_{A,i})^2$$

- I ist ein Maß wie wichtig Gewichte für die Aufgabe A waren, Bsp: $I = [1, 0, 0, 1]$.

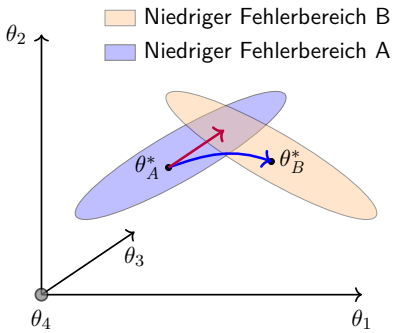


Elastic Weight Consolidation (EWC)

- Einer der bekanntesten Methoden ist Elastic Weight Consolidation (**EWC**) aus dem Jahr 2016.
- Hierfür verändert man das Fehlerfunktional $\mathcal{L}$ wie folgt:

$$\mathcal{L}(\theta) = \mathcal{L}_B(\theta) + \frac{\lambda}{2} \sum_{i=1}^{4} I_i \cdot (\theta_i - \theta^*_{A,i})^2$$

- I ist ein Maß wie wichtig Gewichte für die Aufgabe A waren, Bsp: $I = [1, 0, 0, 1]$.



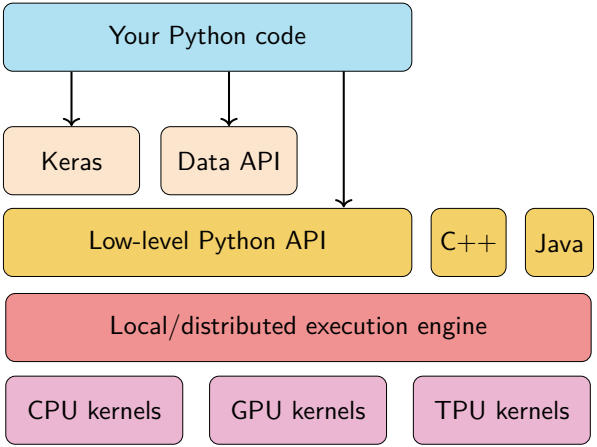
Einführung in TensorFlow

- TensorFlow ist eine mächtige Bibliothek für numerische Berechnungen, die für maschinelles Lernen im großen Stil optimiert ist.
- TensorFlow wurde vom Google Brain entwickelt und ist das Herzstück vieler Google-Dienste (Google Search, Google Translate, Google Photos, YouTube, Gmail, Google Maps, ...).
- TensorFlow wurde im November 2015 als Open Source veröffentlicht und ist heute die am weitesten verbreitete Deep-Learning-Bibliothek in der Industrie (Snapchat, Airbnb, Twitter, PayPal, eBay, ...).
- Der TensorFlow-Kern ist NumPy sehr ähnlich, bietet aber zusätzlich GPU-Unterstützung.

Einführung in TensorFlow

- TensorFlow ist eine mächtige Bibliothek für numerische Berechnungen, die für maschinelles Lernen im großen Stil optimiert ist.
- TensorFlow wurde vom Google Brain entwickelt und ist das Herzstück vieler Google-Dienste (Google Search, Google Translate, Google Photos, YouTube, Gmail, Google Maps, ...).
- TensorFlow wurde im November 2015 als Open Source veröffentlicht und ist heute die am weitesten verbreitete Deep-Learning-Bibliothek in der Industrie (Snapchat, Airbnb, Twitter, PayPal, eBay, ...).
- Der TensorFlow-Kern ist NumPy sehr ähnlich, bietet aber zusätzlich GPU-Unterstützung.

TensorFlow's Architektur



TensorFlow - Tensors

- TensorFlow's API basiert auf Tensoren. Ein Tensor ist einem NumPy ndarray sehr ähnlich: Es ist normalerweise ein mehrdimensionales Array, kann aber auch einen Skalar enthalten.

```
# create a 2x3 matrix of constant floats
t = tf.constant([[1., 2., 3.], [4., 5., 6.]])
>> t
# tf.Tensor: shape=(2, 3), dtype=float32, numpy=
# array([[1., 2., 3.],
#        [4., 5., 6.]], dtype=float32)>
```

- Genau wie ein ndarray hat ein tf.Tensor eine shape und einen Datentyp (dtype).

```
>> t.shape
# TensorShape([2, 3])
>> t.dtype
# tf.float32
```

TensorFlow - Indexierung

- Die Indexierung funktioniert ähnlich wie in NumPy:

```
15 #  array ([[1.,   2.,   3.],
16 #          [4.,   5.,   6.]],  dtype=float32)>
17 >> t[:, 1:]
18 #    <tf.Tensor: shape=(2, 2), dtype=float32, numpy=
19 #    array ([[2.,   3.],
20 #            [5.,   6.]],  dtype=float32)>
21
22 >> t[:, 1, tf.newaxis]
23 #    <tf.Tensor: shape=(2, 1), dtype=float32, numpy=
24 #      array ([[2.],
25 #              [5.]],  dtype=float32)>
```

Operationen auf Tensors

- Es können auch Operationen mit Tensors durchführt werden.

```
>> t + 10    #same as tf.add(t, 10)
#    <tf.Tensor: shape=(2, 3), dtype=float32, numpy=
#    array([[11., 12., 13.],
#           [14., 15., 16.]], dtype=float32)>
>> tf.square(t)
#    <tf.Tensor: shape=(2, 3), dtype=float32, numpy=
#    array([[ 1., 4., 9.],
#           [16., 25., 36.]], dtype=float32)>
>> t @ tf.transpose(t)    # Matrix multiplication (seit python 3.5)
#    <tf.Tensor: shape=(2, 2), dtype=float32, numpy=
#    array([[14., 32.],
#           [32., 77.]], dtype=float32)>
```

Operationen auf Tensors

- TensorFlow bietet die grundlegenden mathematischen Operationen.
- Die meisten davon haben den gleichen Namen wie in Numpy: tf.add(), tf.multiply(), tf.square(), tf.exp(), tf.sqrt(), etc.
- Einige Funktionen haben einen anderen Namen als in NumPy; zum Beispiel sind tf.reduce_mean(), tf.reduce_sum(), tf.reduce_max() die Äquivalente von np.mean() np.sum(), np.max().
- Wenn der Name abweicht, gibt es in der Regel einen guten Grund dafür. Die Operation tf.reduce_sum() heißt zum Beispiel so, weil ihr GPU-Kernel (d. h. die GPU-Implementierung) einen Reduktionsalgorithmus verwendet, um die Berechnung zu beschleunigen.

Operationen auf Tensors

- TensorFlow bietet die grundlegenden mathematischen Operationen.
- Die meisten davon haben den gleichen Namen wie in Numpy: tf.add(), tf.multiply(), tf.square(), tf.exp(), tf.sqrt(), etc.
- Einige Funktionen haben einen anderen Namen als in NumPy; zum Beispiel sind tf.reduce_mean(), tf.reduce_sum(), tf.reduce_max() die Äquivalente von np.mean() np.sum(), np.max().
- Wenn der Name abweicht, gibt es in der Regel einen guten Grund dafür. Die Operation tf.reduce_sum() heißt zum Beispiel so, weil ihr GPU-Kernel (d. h. die GPU-Implementierung) einen Reduktionsalgorithmus verwendet, um die Berechnung zu beschleunigen.

Tensors und Numpy

- Tensoren harmonieren gut mit NumPy: ein Tensor kann aus einem NumPy-Array erzeugt werden und umgekehrt.
- Es können sogar TensorFlow-Operationen auf NumPy-Arrays und NumPy-Operationen auf Tensoren angewendet werden.

```
6  a = np.array([2., 4., 5.]) #numpy array
7  tf.constant(a), tf.square(a)
8  t = tf.constant([[1., 2., 3.], [4., 5., 6.]]) # tf array
9  t.numpy() # or np.array(t)
10 np.square(t)
```

- NumPy verwendet 64-Bit Präzision, während TensorFlow 32-Bit verwendet. Das liegt daran, dass 32-Bit-Präzision in der Regel mehr als ausreichend für neuronale Netze ist, außerdem läuft es schneller und verbraucht weniger RAM. Wenn also ein Tensor aus einem NumPy-Array erstellt wird, muss dtype=tf.float32 gesetzt sein.

Tensors und Numpy

- Tensoren harmonieren gut mit NumPy: ein Tensor kann aus einem NumPy-Array erzeugt werden und umgekehrt.
- Es können sogar TensorFlow-Operationen auf NumPy-Arrays und NumPy-Operationen auf Tensoren angewendet werden.

```
6  a = np.array([2., 4., 5.]) #numpy array
7  tf.constant(a), tf.square(a)
8  t = tf.constant([[1., 2., 3.], [4., 5., 6.]]) # tf array
9  t.numpy() # or np.array(t)
10 np.square(t)
```

- NumPy verwendet 64-Bit Präzision, während TensorFlow 32-Bit verwendet. Das liegt daran, dass 32-Bit-Präzision in der Regel mehr als ausreichend für neuronale Netze ist, außerdem läuft es schneller und verbraucht weniger RAM. Wenn also ein Tensor aus einem NumPy-Array erstellt wird, muss dtype=tf.float32 gesetzt sein.

Typ-Umwandlungen in TensorFlow

- In Bibliotheken (z. B. Numpy) werden Typ-Uwandlungen häufig automatisch durchgeführt. Dies kann jedoch die Leistung erheblich beeinträchtigen und unbemerkt bleiben.
- Um dies zu vermeiden, führt TensorFlow keine automatischen Typ-Umwandlungen durch, sondern wirft eine "exception", wenn versucht wird, eine Operation auf Tensoren mit inkompatiblen Typen auszuführen.
- Zum Beispiel können keine Float- und Integer-Tensoren addiert werden, und auch keine 32-Bit-Float- und 64-Bit-Float-Tensoren.
- Es kann tf.cast() verwendet werden, wenn Typen unbedingt konvertiert werden müssen.

```
16 tf.constant(2.) + tf.constant(40)   #float and int, ERROR!
17 t2 = tf.constant(40., dtype=tf.float64)
18 tf.constant(2.) + t2   #float32 and float64, ERROR!
19 tf.constant(2.0) + tf.cast(t2, tf.float32)
```

Typ-Umwandlungen in TensorFlow

- In Bibliotheken (z. B. Numpy) werden Typ-Uwandlungen häufig automatisch durchgeführt. Dies kann jedoch die Leistung erheblich beeinträchtigen und unbemerkt bleiben.
- Um dies zu vermeiden, führt TensorFlow keine automatischen Typ-Umwandlungen durch, sondern wirft eine "exception", wenn versucht wird, eine Operation auf Tensoren mit inkompatiblen Typen auszuführen.
- Zum Beispiel können keine Float- und Integer-Tensoren addiert werden, und auch keine 32-Bit-Float- und 64-Bit-Float-Tensoren.
- Es kann tf.cast() verwendet werden, wenn Typen unbedingt konvertiert werden müssen.

```
16 tf.constant(2.) + tf.constant(40)   #float and int, ERROR!
17 t2 = tf.constant(40., dtype=tf.float64)
18 tf.constant(2.) + t2   #float32 and float64, ERROR!
19 tf.constant(2.0) + tf.cast(t2, tf.float32)
```

TensorFlow - Variable

- Die Tensoren, die wir bisher gesehen haben (tf.constant()), sind unveränderlich.
- Das bedeutet, dass sie z. B. nicht praktisch sind, um Gewichte in einem neuronalen Netz zu implementieren.
- Für veränderliche Tensoren bietet TensorFlow den Variablen Tensor.

```
2 v = tf.Variable([[1., 2., 3.], [4., 5., 6.]])
3 v.assign(2 * v) # v now equals [[2., 4., 6.], [8., 10., 12.]]
4 v[1] = [7., 8., 9.] # Direct assignment will not work! ERROR
5 v[1].assign([7., 8., 9.]) # v now equals [[2., 4., 6.], [7., 8., 9.]]
```

TensorFlow - Variable

- Die Tensoren, die wir bisher gesehen haben (tf.constant()), sind unveränderlich.
- Das bedeutet, dass sie z. B. nicht praktisch sind, um Gewichte in einem neuronalen Netz zu implementieren.
- Für veränderliche Tensoren bietet TensorFlow den Variablen Tensor.

```
2 v = tf.Variable([[1., 2., 3.], [4., 5., 6.]])
3 v.assign(2 * v) # v now equals [[2., 4., 6.], [8., 10., 12.]]
4 v[1] = [7., 8., 9.] # Direct assignment will not work! ERROR
5 v[1].assign([7., 8., 9.]) # v now equals [[2., 4., 6.], [7., 8., 9.]]
```

Implementierung einer Benutzerdefinierten Verlustfunktion

- Die Implementierung von EWC in Keras/TensorFlow resultiert in der Umsetzung seines Fehlerfunktionals:

$$\mathcal{L}(\theta) = \mathcal{L}_B(\theta) + \frac{\lambda}{2} \sum_i \cdot I_i \cdot (\theta_i - \theta^*_{A,i})^2$$

- Dazu müssen wir verstehen, wie sich benutzerdefinierte Fehlerfunktionale in Keras/TensorFlow umsetzen lassen.
- Nehmen wir an, wir wollen unseren eigenen mittleren quadratischen Fehler (mean square error – MSE) implementieren.

$$\text{MSE} = \frac{1}{m} \sum_{i=1}^{m} (y_i - \hat{y}_i)^2$$

Implementierung einer Benutzerdefinierten Verlustfunktion

- Die Implementierung von EWC in Keras/TensorFlow resultiert in der Umsetzung seines Fehlerfunktionals:

$$\mathcal{L}(\theta) = \mathcal{L}_B(\theta) + \frac{\lambda}{2} \sum_i \cdot I_i \cdot (\theta_i - \theta^*_{A,i})^2$$

- Dazu müssen wir verstehen, wie sich benutzerdefinierte Fehlerfunktionale in Keras/TensorFlow umsetzen lassen.
- Nehmen wir an, wir wollen unseren eigenen mittleren quadratischen Fehler (mean square error – MSE) implementieren.

$$\text{MSE} = \frac{1}{m} \sum_{i=1}^{m} (y_i - \hat{y}_i)^2$$

Implementierung einer Benutzerdefinierten Verlustfunktion

- Dies geschieht wie folgt $\left(\frac{1}{m}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2\right)$:

```
34 # Benutzerdefiniertes Fehlerfunktional
35 def mse_fn(y_true, y_pred):
36     m = y_true.shape[0]
37     error = y_true - y_pred
38     squared_loss = tf.square(error)
39     return tf.sum(squared_loss) / m
```

- Dann kann dieses MSE-Fehlerfunktional verwendet werden, wenn das Keras-Modell kompiliert wird. Anschließend wird das Modell wie üblich trainiert.

```
40 model.compile(loss=mse_fn, optimizer="adam", metrics=["mae"])
41 model.fit(X_train, y_train, epochs=5)
```

Implementierung einer Benutzerdefinierten Verlustfunktion

- Dies geschieht wie folgt $\left(\frac{1}{m}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2\right)$:

```
34 # Benutzerdefiniertes Fehlerfunktional
35 def mse_fn(y_true, y_pred):
36     m = y_true.shape[0]
37     error = y_true - y_pred
38     squared_loss = tf.square(error)
39     return tf.sum(squared_loss) / m
```

- Dann kann dieses MSE-Fehlerfunktional verwendet werden, wenn das Keras-Modell kompiliert wird. Anschließend wird das Modell wie üblich trainiert.

```
40 model.compile(loss=mse_fn, optimizer="adam", metrics=["mae"])
41 model.fit(X_train, y_train, epochs=5)
```

Implementierung einer Benutzerdefinierten Verlustfunktion

- Das Fehlerfunktional, das wir zuvor definiert haben, basiert auf den Labels und den Vorhersagen.
- Wenn wir Fehlerfunktionale definieren möchten, die auf anderen Teilen des Modells basieren, z. B. auf den Gewichten (wie in EWC loss) oder Aktivierungen der Hidden-Layers, funktioniert diese Methode nicht.
- Um ein Fehlerfunktional basierend auf modellinternen Elementen zu definieren, braucht man mehr Kontrolle über die Trainingsschleife. D.h., model.fit() muss ersetzt werden.

```
9  for epoch in n_epochs:
10
11     # Forward propagation
12             ...
13
14     # Fehler berechnen
15             ...
16
17     # Backward propagation
18             ...
19
20     # Gewichte aktualisieren
21             ...
```

Gradienten in TensorFlow berechnen

- Während der Backpropagation berechnet man den Gradienten des Fehlerfunktionals in Bezug auf alle trainierbaren Gewichte des Netzes. (Daraus geht hervor, wie die Gewichte verändert werden sollen, um die Fehler zu reduzieren).
- Wie funktioniert dies in TensorFlow ?
- Betrachten wir folgendes Fehlerfunktional:

$$f(w_1, w_2) = 3w_1^2 + 2w_1 w_2$$

- Analytisch lässt sich feststellen, dass der Gradientenvektor wie folgt lautet:

$$\nabla f(w_1, w_2) = \left(\frac{\partial f}{\partial w_1} , \frac{\partial f}{\partial w_2} \right) = (6w_1 + 2w_2 \ , 2w_1)$$

- Zum Beispiel ergibt sich für den Punkt $(w_1, w_2) = (5, 3)$ der Gradientenvektor als $(36, 10)$.

Gradienten in TensorFlow berechnen

- Während der Backpropagation berechnet man den Gradienten des Fehlerfunktionals in Bezug auf alle trainierbaren Gewichte des Netzes. (Daraus geht hervor, wie die Gewichte verändert werden sollen, um die Fehler zu reduzieren).
- Wie funktioniert dies in TensorFlow ?
- Betrachten wir folgendes Fehlerfunktional:

$$f(w_1, w_2) = 3w_1^2 + 2w_1 w_2$$

- Analytisch lässt sich feststellen, dass der Gradientenvektor wie folgt lautet:

$$\nabla f(w_1, w_2) = \left(\frac{\partial f}{\partial w_1} , \frac{\partial f}{\partial w_2} \right) = (6w_1 + 2w_2 , 2w_1)$$

- Zum Beispiel ergibt sich für den Punkt $(w_1, w_2) = (5, 3)$ der Gradientenvektor als $(36, 10)$.

Gradienten in TensorFlow berechnen

- Um solche Gradienten zu berechnen, verwendet TensorFlow eine Technik, die als Reverse-Mode-Autodiff bekannt ist.
- Dafür nutzt man die Methode tf.GradientTape(), die in seinem Scope alle Berechnungen aufzeichnet, die Variablen involvieren.
- Danach kann man die Gradienten mithilfe der Methode tf.GradientTape().gradient() berechnen, die die aufgezeichneten Berechnungen in umgekehrter Reihenfolge durchläuft.

```
3 def f(w1, w2):
4     return 3 * w1 ** 2 + 2 * w1 * w2
```

```
6 w1, w2 = tf.Variable(5.), tf.Variable(3.) # define two variables w1 and w2
7 with tf.GradientTape() as tape:
8     z = f(w1, w2)
9 gradients = tape.gradient(z, [w1, w2]) #returns tensor 36.0, 10.0
```

Benutzerdefinierte Trainingsschleife

- Möchte man z. B. das vorherige MSE Fehlerfunktional mit L2 Regularisierung erweitern, gelingt dies wie folgt:

$$\mathcal{L}(\theta) = MSE + \frac{\lambda}{2m} \sum_i \theta_i^2$$

```
41 def my_l2_reg_mse_loss(y_true, y_pred, weights, lambd=0.2):
42     mse = my_mse(y_true, y_pred)
43     l2_reg = 0
44     for w in weights:
45         #if len(w.shape) != 2:   continue    #do not  regularize   bias
46         l2_reg += tf.reduce_sum(tf.square(w))
47     m = y_true.shape[0]
48     l2_reg *= (lambd/(2*m))
49     return mse + l2_reg
```

Benutzerdefinierte Trainingsschleife

- Die benutzerdefinierte Trainingsschleife ergibt sich somit folgendermaßen:

```
for epoch in range(n_epochs): # loop over epochs
    for batch in range(n_batch): # loop over batches
        # sample a random batch from the training set
        X_batch, y_batch = random_batch(X_train, y_train)
        # automatically record every operation that involves a variable
        with tf.GradientTape() as tape:
            y_pred = model(X_batch)  # forward propagation
            loss = my_l2_reg_mse_loss(y_batch,y_pred,model.weights)#loss
        # backpropagation
        gradients = tape.gradient(loss, model.trainable_variables)
        # weights update
        optimizer.apply_gradients(zip(gradients,
                                      model.trainable_variables))
```