jupyter eigenes_word2vec-model Last Checkpoint: Last Thursday at 11:02 AM (unsaved changes)

Logout

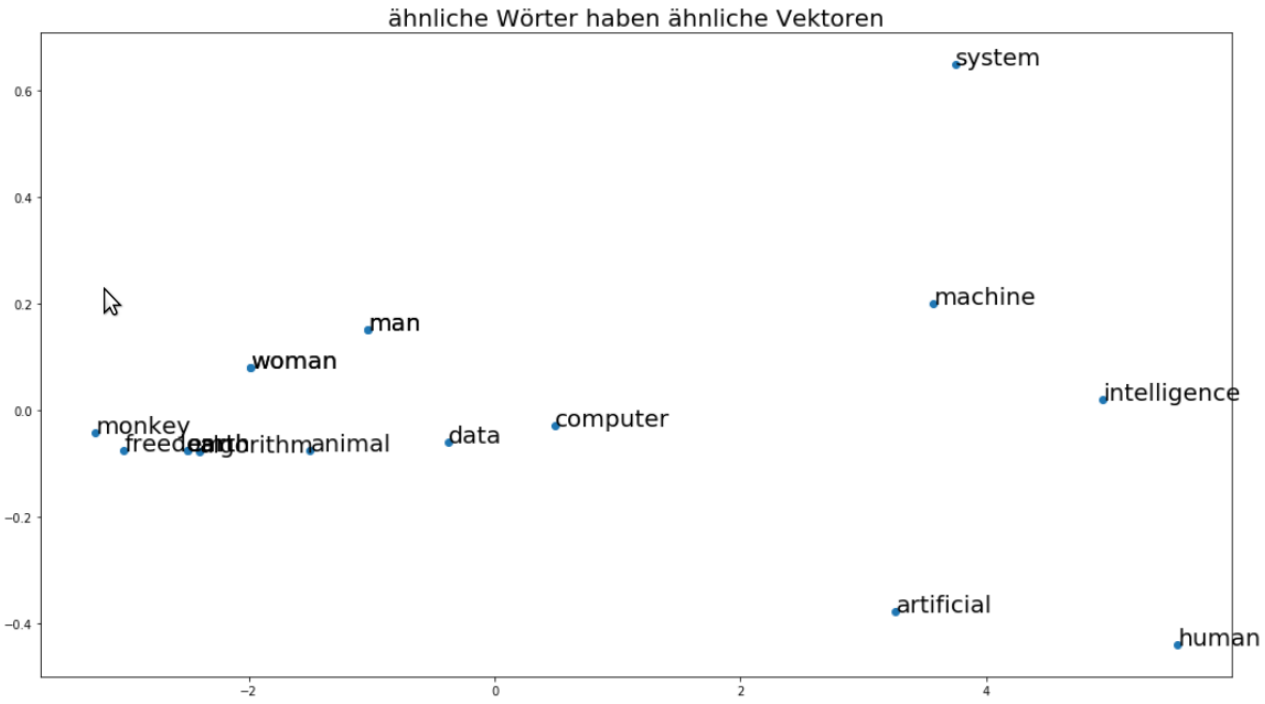
File Edit View Insert Cell Kernel Widgets Help

Not Trusted | Python 3

Run Markdown

```
21
22 plt.title('ähnliche Wörter haben ähnliche Vektoren', fontsize=20)
23
24 plot_words('earth', 'intelligence', 'human', 'computer', 'algorithm', 'animal',
25            'monkey', 'data', 'man', 'woman', 'man', 'woman', 'artificial',
26            'system', 'earth', 'freedom', 'machine')
27
28 #plot_words('earth', 'intelligence', 'human', 'mountain', 'computer', 'algorithm', 'animal',
29 #           'monkey', 'data', 'man', 'woman', 'man', 'woman', 'artificial',
30 #           'system', 'earth', 'freedom', lines=True)
```

Out[50]:
```
PCA(copy=True, iterated_power='auto', n_components=2, random_state=None,
    svd_solver='auto', tol=0.0, whiten=False)
```



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.

$$\frac{\partial E}{\partial w_{jk}} = -(t_k - o_k) \cdot sigmoid(\Sigma_j w_{jk} \cdot o_j)(1 - sigmoid(\Sigma_j w_{jk} \cdot o_j)) \cdot o_j$$

Abbildung 1-93: Der Fehleranstieg in ausgeschriebener Form

Führen wir das nun Stück für Stück durch:

- Der erste Term $(t_k - o_k)$ ist der Fehler $e_1 = 0{,}8$, wie wir ihn zuvor gesehen haben.
- Die Summe in den Sigmoidfunktionen $\Sigma_j w_{jk} o_j$ ist $(2{,}0 \times 0{,}4) + (3{,}0 \times 0{,}5) = 2{,}3$.
- Die Sigmoidfunktion $1/(1 + e^{-2{,}3})$ liefert dann 0,909. Dieser mittlere Ausdruck ergibt $0{,}909 \times (1 - 0{,}909) = 0{,}083$.
- Das letzte Element ist einfach o_j, was $o_{j=1}$ entspricht, weil wir am Gewicht w_{11} interessiert sind, bei dem $j = 1$ ist. Hier hat es einfach den Wert 0,4.

Multipliziert man alle diese drei Terme (das Minuszeichen am Anfang nicht vergessen), ergibt sich –0,0265.

Wenn wir eine Lernrate von 0,1 haben, ergibt das eine Änderung von $-(0{,}1 \times -0{,}0265) = +0{,}00265$. Das neue Gewicht w_{11} ist somit das ursprüngliche Gewicht $2{,}0 + 0{,}00265 = 2{,}00265$.

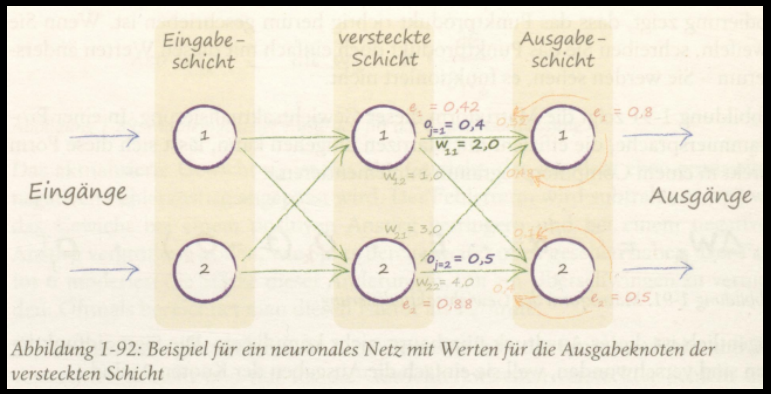
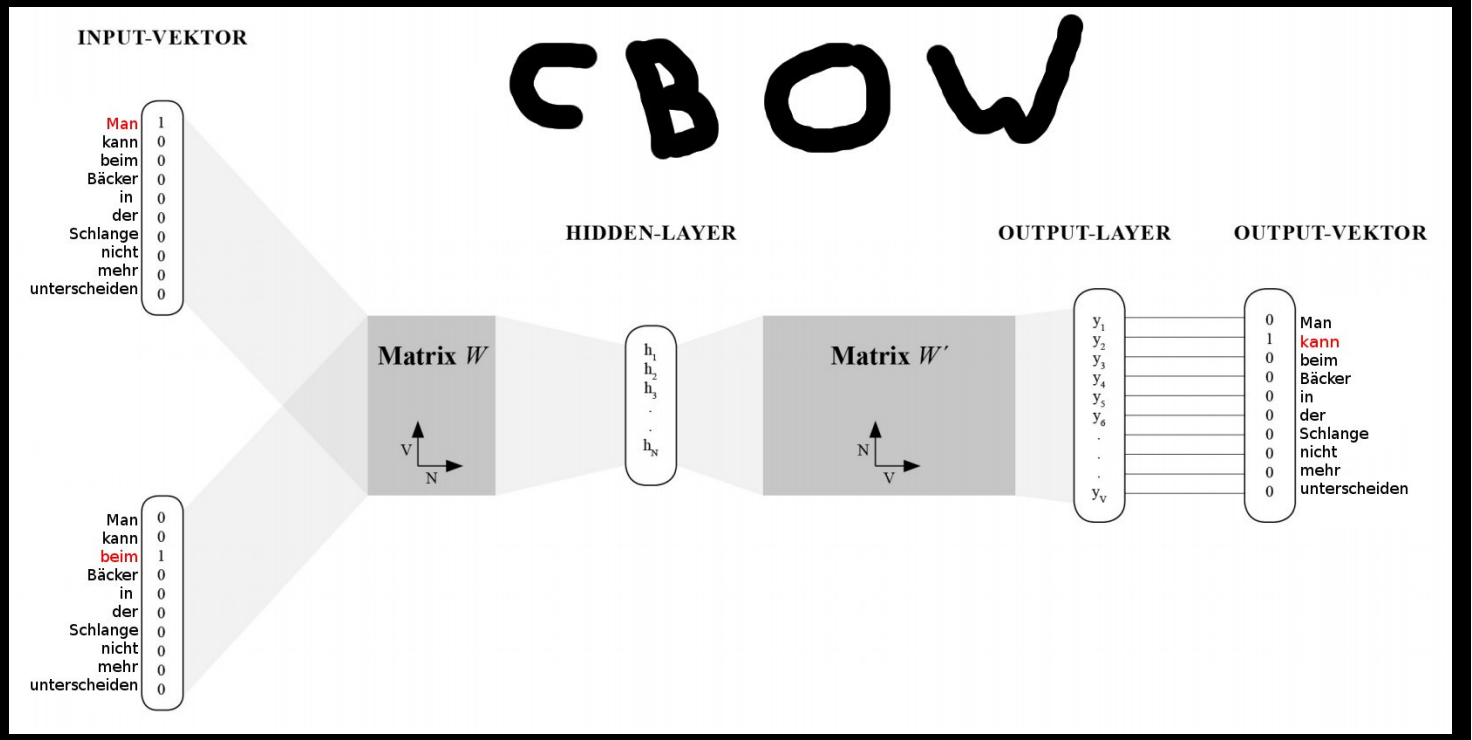
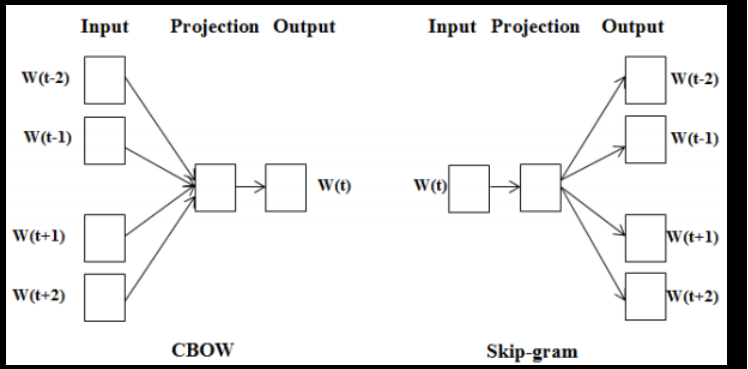


Abbildung 1-92: Beispiel für ein neuronales Netz mit Werten für die Ausgabeknoten der versteckten Schicht

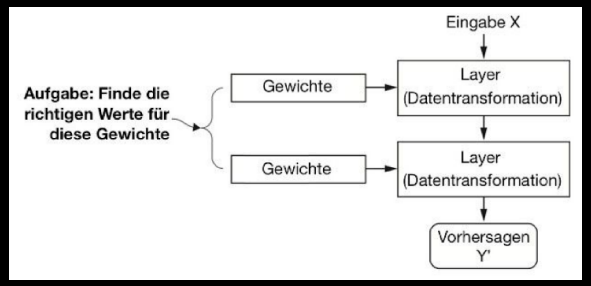




Lernen:

Die Neuronen sind im ersten Schritt initialisiert mit zufälligen Aktivierungen. Das Netzwerk berechnet die Ausgabegröße und wird von der Zielfunktion (Loss) gnadenlos bestraft. Der Fehler den es gemacht hat, wird über so genannte Backpropagation auf die jeweiligen Neuronen zurück verteilt, die ihn verursacht haben.

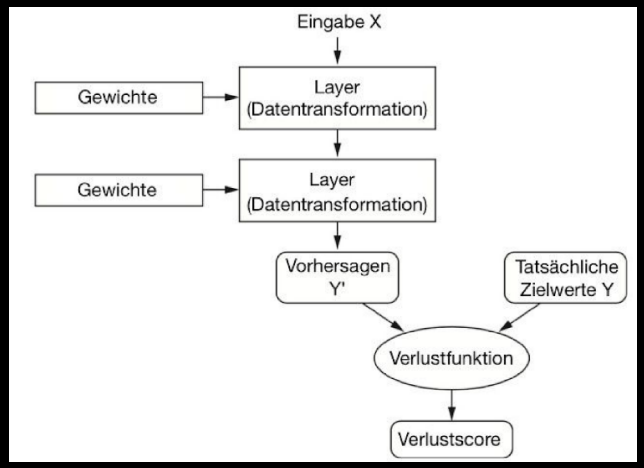
Man schaut, ob der Fehler größer oder kleiner wird, wenn man die Aktivierung erhöht.
Mathematisch gesehen wird die Steigung der Fehlerfunktion bestimmt.
Idealer Weise gibt es eine Richtung in die man optimieren kann, sodass die Fehlerfunktion minimiert wird.



...die „richtigen" Werte für die Gewichtungen (Parameter des Layers) finden

d.h. 1. zufallsgewichtungen ins Netz schicken

Deep Learning



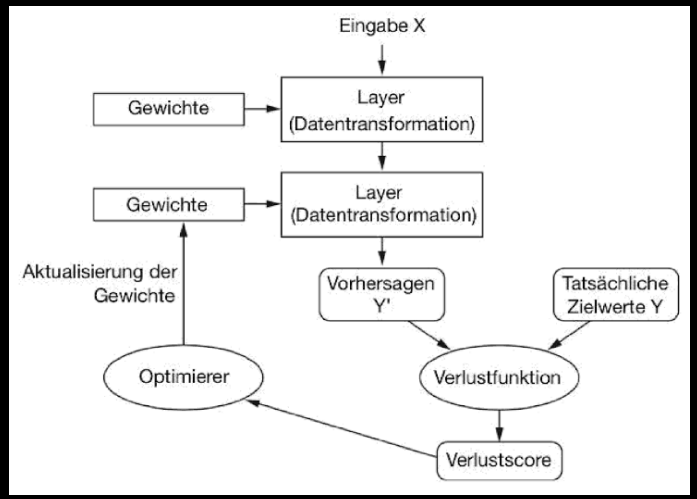
2. Messen wie stark der Istwert vom Sollwert abweicht

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.

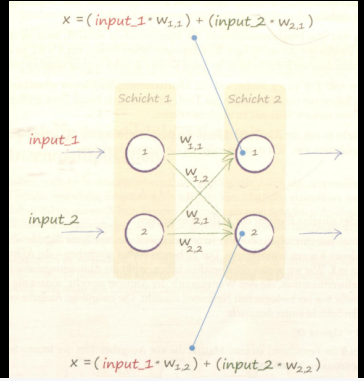
Diese Anpassung ist die Aufgabe des *Optimierers* `optimizer`, der einen sogenannten Backpropagations-Algorithmus implementiert.

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

```python
#trainieren des Netzes << train

    #Teil 1, Ausgabe erzeugen (siehe query)
    def train(self, inputs_list, targets_list):
        # umwandeln der Inputs von 3D in 2D Array
        inputs = numpy.array(inputs_list, ndmin=2).T
        targets = numpy.array(targets_list, ndmin=2).T

        #Berechnung der Eingangssignale in hidden_layer (siehe query)
        hidden_inputs = numpy.dot(self.wih, inputs)
        #Berechnung der ausgehenden Signale aus hidden_layer
        hidden_outputs = self.activation_function(hidden_inputs)

        #Berechnung der Eingangssignale in output_layer (siehe query)
        final_inputs = numpy.dot(self.who, hidden_outputs)
        #Berechnung der ausgehenden Signale aus output_layer
        final_outputs = self.activation_function(final_inputs)
```




```python
# Aktivierungs(sigmoid-)funktion mit lambda definieren:
self.activation_function = lambda x: scipy.special.expit(x)
```

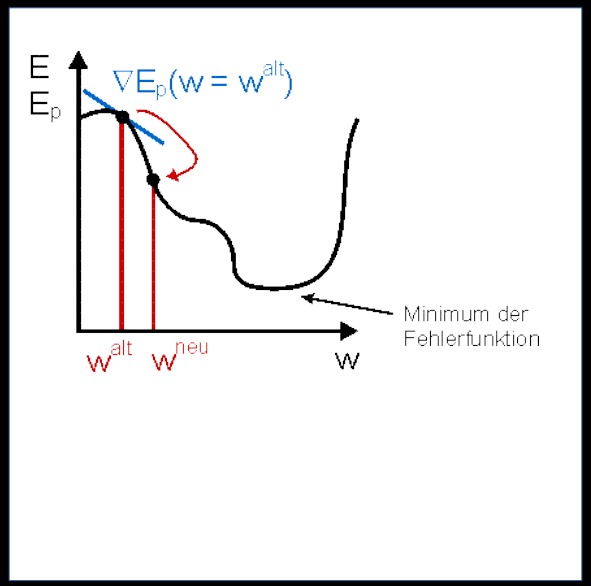
```python
#Teil 2, Fehlerbackpropagierung
    ##          E = S - K
    ##     Fehler = Sollwert(targets z.48) - aktuellem Istwert(final_outputs z.58)
    #
    output_errors = targets - final_outputs
    #
    ##               (  w11          w12  )
    ##               |-------     ------- |
    ##               |w11+w21     w12+w22 |   ( e1 )          ( w11   w12 )   ( e1 )
    ##error_hidden = |                    | * |    |     =    |           | * |    |
    ##               |  w21          w22  |   ( e2 )          (w21    w22 )   ( e2 )
    ##               |-------     ------- |
    ##               (w21+w11     w22+w12 )
    #
    # hidden_errors << Gewichte zwischen versteckter- und Ausgabeschicht
    # output_errors << Gewichte zwischen Eingabe- und versteckter Schicht
    hidden_errors = numpy.dot(self.who.T, output_errors)
    #
    #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))

    pass
```

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

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.

Glossar



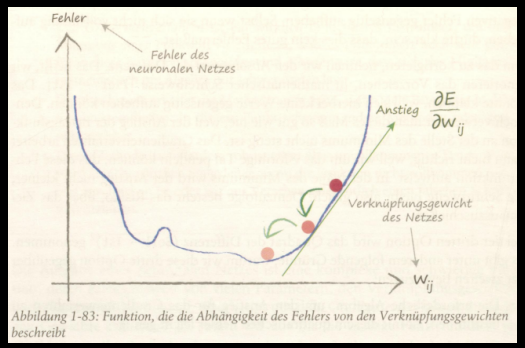
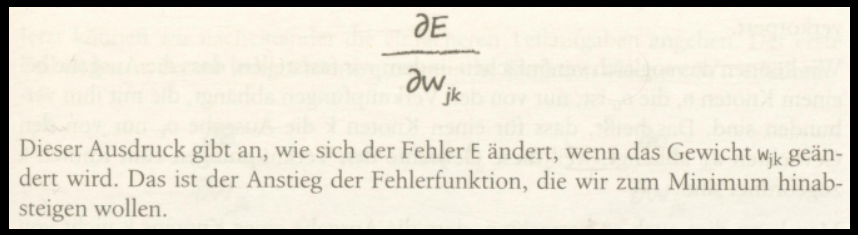


Abbildung 1-83: Funktion, die die Abhängigkeit des Fehlers von den Verknüpfungsgewichten beschreibt

```
#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))
```

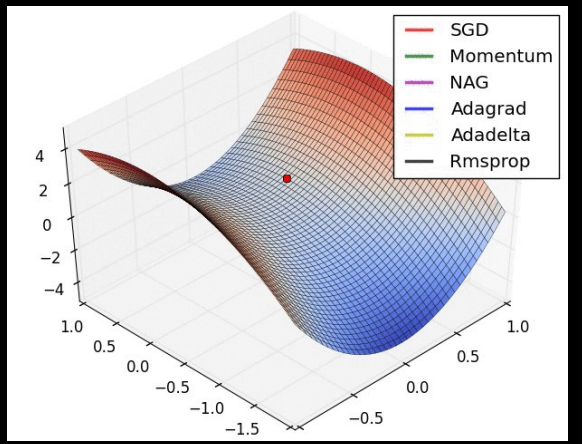
```
network.compile(optimizer='rmsprop',
                loss='categorical_crossentropy',
                metrics=['accuracy'])
```


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

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

```
# Aktivierungs(sigmoid-)funktion mit lambda definieren:
self.activation_function = lambda x: scipy.special.expit(x)
```

NumPy

python™

TensorFlow

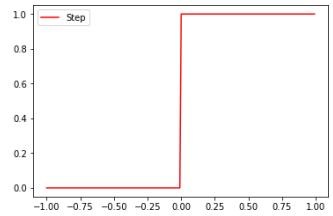
https://keras.io/activations/

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.

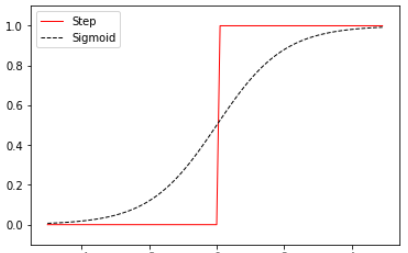
```
# Aktivierungsfunktion als Schwellenwert << das Feuern...
if u < 0:
    output=0
else:
    output=1
print("\nAusgabe der Schwellenwertfunktion: ", output, " = true\n")

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

```
Ausgabe der Schwellenwertfunktion:  1  = true

Ergebnis des Vergleichsoperators »bool« =  True
```



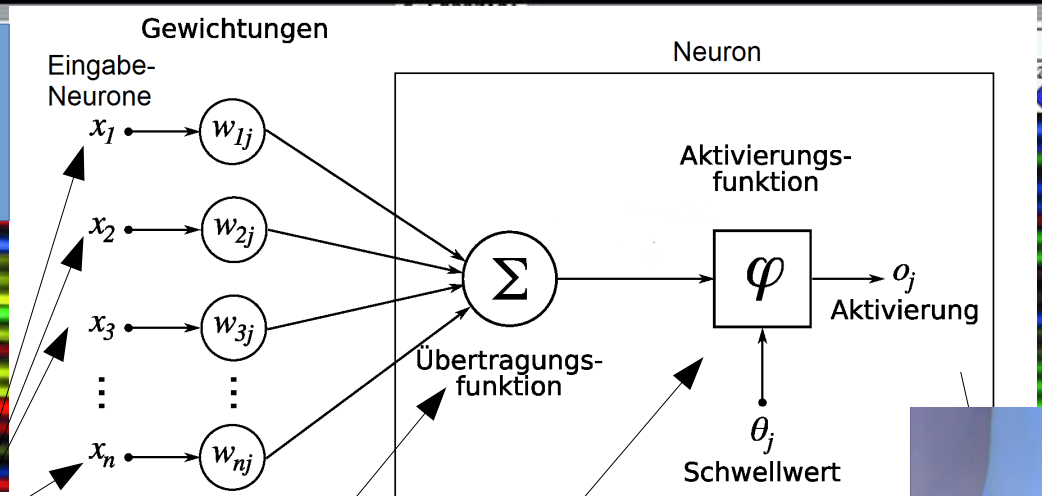

```
# Aktivierungsfunktion als Sigmoidfunktion
sigmoid = 1 / (1+ np.exp(-u))
print("Ausgabe der sigmoidfunktion: ", sigmoid)
```

```
Ausgabe der sigmoidfunktion:  0.5938731029341427
```

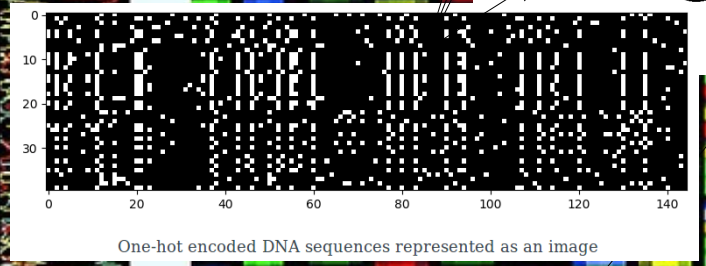



```
network = models.Sequential()

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'))
```

Das Perzeptron



Der Linearer Klassifizierer



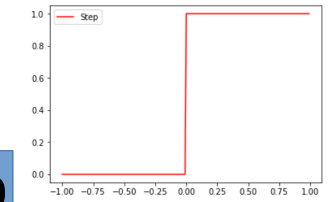
One-hot encoded DNA sequences represented as an image

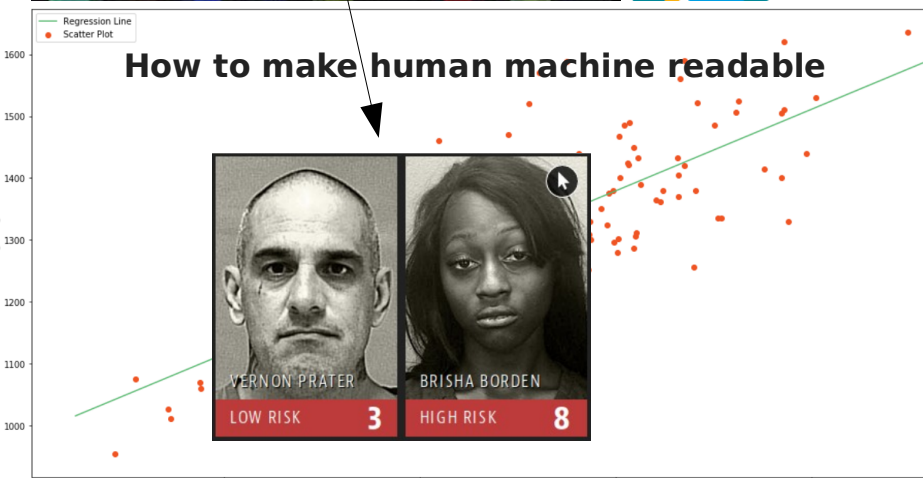
```
# Übertragungsfunktion anwenden
u = x1*w1+x2*w2+x3*w3
print("summierter und gewichteter wert: ", u)
```

```
summierter und gewichteter wert:  0.38
```

```
# 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
```





Classify yourself @ open lab

Face	Position	Classifiers and measurements
	Score: 13.39 X: 146.62 Y: 66.58 Width: 83.45 Height: 83.45 Angle: 3.69	age : 43 (31%), beard : no (0%), gender : male (61%), glasses : no (71%), mustache : no (20%), race : white (7%), smile : no (50%), chin size : extra large, color background : 080f2e (29%), color beard : 6d3d21 (65%), color clothes middle : edebf5 (4 0%), color clothes sides : 1e1e25 (47%), color eyes : 53342d (47%), color hair : 4e4641 (74%), color skin : c1936b, eyebrows corners : average, eyebrows position : average, eyebrows size : average, eyes corners : extra low, eyes distance : extra close, eyes position : extra high, eyes shape : round, glasses rim : no, hair beard : thick, hair color type : brown light (74%), hair forehead : no, hair length : short, hair mustache : none, hair sides : thin, hair top : short, head shape : extra heart, head width : extra narrow, mouth corners : extra low, mouth height : average, mouth width : extra small, nose shape : extra triangle, nose width : narrow, teeth visible : no [collapse]

George Boole

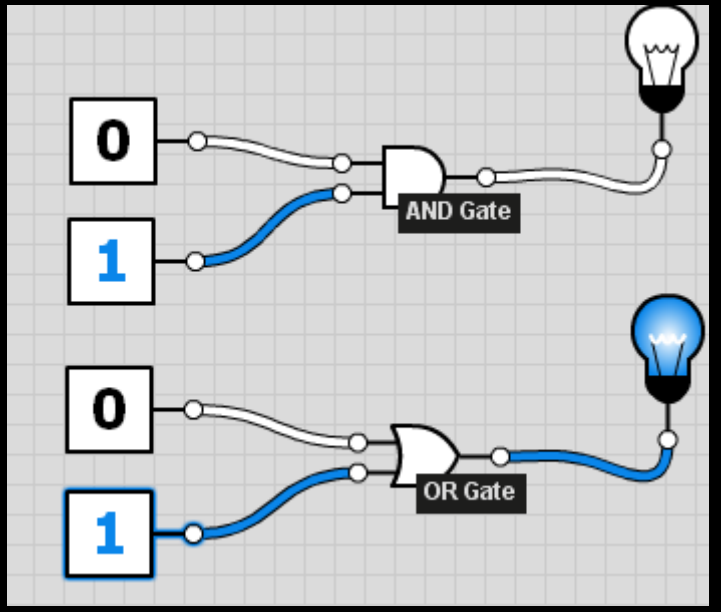
»An investigation into the Laws of Thought« (Eine Untersuchung der Gesetze des Denkens), 1854
...eine mathematisch-formale Beschreibung der klassischen Logik. Verstandestätigkeiten ließen sich Erstmals in symbolische Operationen beschreiben.

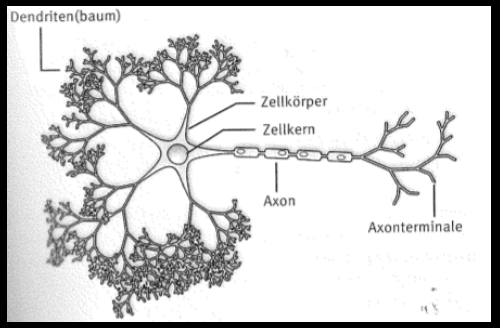
Warren McCulloch & Walter Pitts

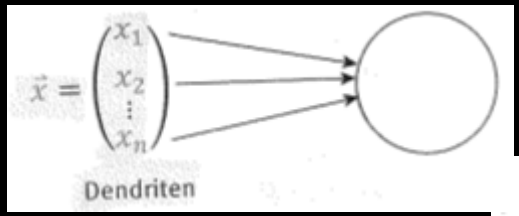
»A logical Calculus of the Ideas immanent in Nervous Activity«, 1943
...McCulloch und Pitts weisen nach, dass die Nerventätigkeit des Gehirns den streng logischen Booleschen Besetzen folgt.

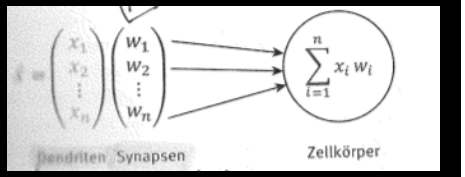
Claude Shannon

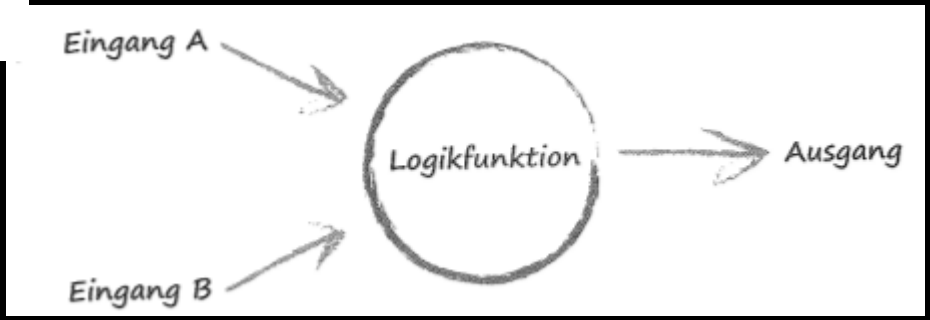
»A Symbolic Analysis of Relay an Switching Circuits«, 1937
...Shannon beschreibt den Zusammenhang zwischen logischen Operationen und elektrischen Schaltkreisen











Steht da ein Entwickler Künstlicher Intelligenz (EKI) vor mir in des Bäckers Schlange?					
männlich	**Geldscheinklammer**	NOT	**AND**	OR	XOR
False	**False**	True	**False**	False	False
False	**True**	True	**False**	True	True
True	**False**	False	**False**	True	True
True	**True**	False	**True**	True	False