Kapitel 1

Grundlagen Neuronaler Netze

In diesem Kapitel wird die Funktionsweise eines aus vollständig verknüpften Schichten bestehenden Neuronalen Netzes erläutert. Hierbei werden zunächst die wesentlichen Elemente eines einfachen Neurons betrachtet. Begleitend zu den theoretischen Grundlagen wird anschließend die Implementierung eines Neuronalen Netzes mittels Python gezeigt. Die so entwickelte Netzstruktur wird abschließend zur Klassifikation von Bilddaten genutzt.

1.1 Das Neuron

Dem Konzept künstlicher Neuronaler Netze (kNN) liegt die Funktionsweise biologischer Neuronaler Netze zugrunde, die aus miteinander verschalteten Nervenzellen (Neuronen) bestehen und im weiteren erläutert werden.

1.1.1 Das biologische Neuron

Bei einem biologischen Neuron handelt sich um eine Zelle, welche darauf spezialisiert ist, Informationen von anderen Neuronen zu empfangen und weiterzuleiten. Ein vereinfachtes Modell eines biologischen Neurons ist in Abbildung 1.1 dargestellt.

An den dendritischen Verästelungen werden über synaptische Verbindungen einkommende Reize erfasst. Der Dendrit leitet den Reiz über den Zellkörper bis zum Axonhügel weiter, an dem sich die einzelnen Reize summieren. Übersteigt die Summe der Erregungen einen Schwellwert, depolarisiert die Zelle in Form eines Aktionspotentials, welches über das Axon gerichtet weitergeleitet wird. Die Axonterminalen wiederum stehen über Synapsen mit Dendriten weiterer Neuronen in Verbindung, so dass Reize an weitere Neuronen weitergeleitet werden. Dabei kann eine einzige Nervenzelle mit mehreren tausend Nervenzellen in Verbindung stehen [1].

Inwieweit die die ***Dendriten*** erreichende Reize weitergeleitet werden, hängt von den synaptischen Kontaktstellen ab. Hier existieren zwei verschiedenartige synaptische Verbindungstpyen. Sogenannte erregende Synapsen sorgen für die Weiterleitung eines Reizes, während hemmende ***Synapsen*** die Signalweiterleitung unterdrücken. Je nachdem in welchem Verhältnis die Synapsen zueinander stehen, kommt es zu einer Depolarisation des Neurons und damit dem Ausbilden eines Aktionspotentials. Die Stärke der synaptischen Übertragung bzw. Hemmung ist aktivitätsabhängig und somit eine veränderliche Größe.

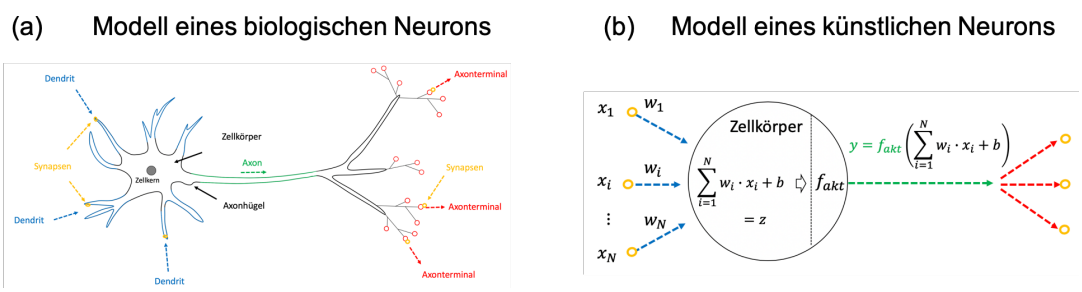


Abbildung 1.1: Beziehung zwischen (a) biologischen und (b) künstlichen Neuron.

1.1.2 Das künstliche Neuron

In Analogie zum biologischen Neuron zeigt Abbildung 1.1 das Modell eines künstlichen Neurons. Die einfallenden Reize, welche das Neuron über die Dendriten erreichen, werden durch den Inputvektor x_i repräsentiert. Die Modellierung der veränderlichen synaptische Erregung bzw. Hemmung erfolgt über die Gewichtung w_i der Reize, während die Summe z der Superposition der Reize am Axonhügel entspricht und als Aktivierung z des Neurons bezeichnet wird. Bei dem Term b handelt es sich in dem Modell um einen Offset, der die Aktivierung des Neurons zusätzlich verändert. Das über das Axon weitergeleitete Outputsignal y resultiert aus einer nichtlinearen Aktivierungsfunktion $y = f_{akt}(z)$ und kann mit der Eingangsseite beliebig vieler weiterer Neuronen verbunden werden. Durch Änderung der Gewichte w_i lässt sich das Verhalten des Neurons, bzw. des gesamten Netzes neuronaler Verbindungen, gezielt verändern.

In dem gezeigten Modell prozessiert ein künstliches Neuron einen Eingabevektor (Input: x) und gibt entsprechend der definierten Gewichte w_i, b sowie der Aktivierungsfunktion $f_{akt}(z)$ einen Output y als Ergebnis zurück. Im Folgenden wird die Realisierung der Funktionalität eines derartigen Neurons durch eine Implementierung der Klasse ***Neuron*** mittels Python verdeutlicht.

```python
#--------------------------------------------------
# Importieren der notwendigen Bibliotheken
# HIER: Numpy
#--------------------------------------------------
import numpy as np

#--------------------------------------------------
# Definition der Klasse Neuron
# Diese Klasse repräsentiert ein einfaches künstliches Neuron,
# welches einen Inputvektor und eine Aktivierungsfunktion
# übergeben bekommt und die resultierende Aktivierung
# als Ergebnis zurückgibt
#--------------------------------------------------
class Neuron:
    """
    ------------------------------------------------------------------------
    Argumente:   dim_input (int): Dimension des Inputvektors x
                 act_func:        Aktivierungsfunktion des Neurons
    ---------
    Attribute:   w (ndarray):     Werte, mit denen der Input x gewichtet wird
                 b (float):       Wert des Bias (Offset)
    ------------------------------------------------------------------------
    """
    #--------------------------------------------------
    # Konstruktor
    #--------------------------------------------------
    def __init__(self, dim_input, act_func):

        #--------------------------------------------------
        # Zufällige Initialisierung der Gewichte (w und b)
        #--------------------------------------------------
        self.w = np.random.standard_normal(dim_input)
        self.b = np.random.standard_normal(1)

        #--------------------------------------------------
        # Aktivierungsfunktion
```

```
        #-----------------------------------------------------
        self.act_func = act_func


    #-----------------------------------------------------
    # Die Funktion "forward" prozessiert den Input "0027x"0027
    # durch das Neuron unter Verwendung der aktuellen
    # Gewichte sowie Anwendung der Aktivierungsfunktion
    #-----------------------------------------------------
    def forward(self, x):

        """
        ------------------------------------------------------------------------
        Argumente:       x (ndarray)   -->  Größe: (1, num_inputs)
        ---------------
        Rückgabewerte:   y (ndarray)  -->  Größe:  (1, layer_size)
        ------------------------------------------------------------------------
        """

        #-----------------------------------------------------
        # Durchführung der Gewichtung und Addition des Bias
        #-----------------------------------------------------
        z = np.dot(x, self.w) + self.b

        #-----------------------------------------------------
        # Der gewichtete Input (net) wird der
        # Aktivierungsfunktion übergeben und als
        # Ergebnis zurückgegeben.
        #-----------------------------------------------------
        y = self.act_func(z)

        return y
```

Code 1: Pythonimplementierung der Klasse Neuron.

1.1.3 Aktivierungsfunktionen

Die Ausgabe eines Neurons hängt wie oben dargestellt von den einfallenden Reizen und der Aktivierungsunktion ab, die im Allgemeinen ein nichtlineares Verhalten aufweist. Das Verhalten eines künstlichen Neurons lässt sich über die Aktivierungsfunktion steuern, die dem Konstruktor der Klasse Neuron übergeben wird. Im Folgenden werden drei häufig verwendete Aktivierungsfunktionen definiert:

1.) Bei der ***Sprung- bzw. Stufenfunktion*** leitet das Neuron eine konstante Information am Ausgang weiter, sobald die Aktivierung z im positiven Definitionsbereich liegt. Dies auch als Heaviside-Funktion bezeichnete Funktion ist wie folgt definiert:

$$f_{Sprung}(z) := \begin{cases} 0: & z < 0 \\ 1: & z \geq 0 \end{cases} \tag{1.1}$$

2.) Die Sprungfunktion $f_{Sprung}(z)$ weißt an der Stelle $z = 0$ eine Unstetigkeit auf und ist daher nicht stetig differenzierbar. Eine weitere Eigenschaft besteht darain, dass die Ableitungen der Sprungfunktion für $z \neq$ verschwinden, da die Sprungfunktion für den negativen und positiven Definitionsbereichen einen konstanten Verlauf aufzeigt. Bei der im Folgenden definierten ***Sigmoid-Aktivierungsfunktion*** handelt es sich um eine stetig differenzierbare Variante der Sprungfunktion, welche diese beiden Eigenschaften nicht aufweißt und wie folgt definiert ist:

$$f_{Sig}(z) := \frac{1}{1 + e^{-z}} \tag{1.2}$$

3.) Bei der sogenannten ***Rectified Linear Unit (RELU)*** Aktivierungsfunktion entspricht der Ausgang des Neurons der Höhe der Aktivierung, wenn diese im positiven Wertebereich liegt. Der Ausgang eines Neurons kann somit mit zunehmender Aktivierung beliebig ansteigen. Für Aktivierungen $z < 0$ bleibt das Neuron dekativiert. Somit ist die RELU-Funktion wie folgt definiert:

$$f_{RELU}(z) := \begin{cases} z: & z > 0 \\ 0: & z \leq 0 \end{cases} \tag{1.3}$$

Der folgende Code zeigt die Python-Realisierung der obigen drei Aktivierungsfunktionen.

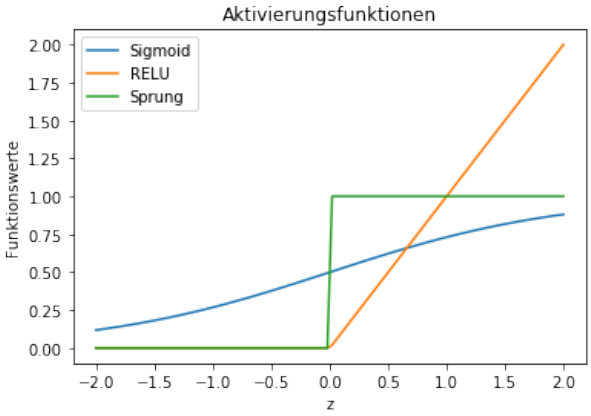
```
#-------------------------------------------------
# Dem Neuron zu übergebende Aktivierungsfunktionen
#-------------------------------------------------
#-------------------------------------------------
# Aktivierungsfunktion 1: Sprungfunktion
#-------------------------------------------------
def step_func(z):
    out = np.copy(z)
    out[out > 0] = 1.0
    out[out <= 0] = 0.0
    return out
#-------------------------------------------------
# Aktivierungsfunktion 2: Sigmoidfunktion
#-------------------------------------------------
def sigmoid_func(z):
    out = 1 / (1 + np.exp(-z))
    return out
#-------------------------------------------------
# Aktivierungsfunktion 3: Rectified Linear Unit (RELU)
#-------------------------------------------------
def relu_func(z):
    return np.maximum(z, 0)
```

Code 2: Implementierung dreier Aktivierungsfunktionen.

Im nachfolgenden Code werden die Verläufe der 3 oben definierten Aktivierungsfunktionen graphisch ausgegeben.

```
import matplotlib.pyplot as plt
import numpy as np

#-------------------------------------------------
# Definitionsbereich festlegen
#-------------------------------------------------
z = np.linspace(-2.0, 2.0, num=100,endpoint=True)

#-------------------------------------------------
# Funktionswerte berechnen
#-------------------------------------------------
f_step = step_func(z)
f_sig  = sigmoid_func(z)
f_relu = relu_func(z)

#-------------------------------------------------
# Graphische Ausgabe
#-------------------------------------------------

sig, = plt.plot(z,f_sig,label="0027Sigmoid"0027)
relu, = plt.plot(z,f_relu,label="0027RELU"0027)
step, = plt.plot(z,f_step,label="0027Sprung"0027)

plt.xlabel("0027z"0027)
plt.ylabel("0027Funktionswerte"0027)
plt.title("Aktivierungsfunktionen")

plt.legend(handles=[sig, relu, step])
plt.show()
```

Code 3: Graphische Ausgabe ausgewählter Aktivierungsfunktionen.



1.1.4 Verwendung der Klasse Neuron

Im Folgenden wird die oben definiert ***Klasse Neuron*** exemplarisch dazu genutzt, einen zufällig generierten Inputvektor zu verarbeiten. Unter Verwendung einer der obigen Aktivierungsfunktion generiert somit ein so instanziiertes künstliches Neuronen in Abhängigketi des Inputvektors und der zufälligen initialisierten Gewichte einen Ausgabewert.

```
#-----------------------------------------------------
# Zufallsgenerator fixieren, um bei jeden Durchlauf
# reproduzierbare Ergebnisse zu erlangen
#-----------------------------------------------------
np.random.seed(0)

#-----------------------------------------------------
# Erzeugen eines Inputvektors x der Größe "size_input"
#-----------------------------------------------------
size_input   = 5
x            = np.random.rand(size_input).reshape(1, size_input)

print("Belegung des Inputsvektors "0027x"0027:\n\n{}".format(x))

#-----------------------------------------------------
# Instanziieren Neuron
# Hier: Eine der obigen Aktivierungsfunktionen wählen
#-----------------------------------------------------
first_neuron = Neuron(dim_input=size_input, act_func=step_func)

print("\nInitiale Gewichte "0027w"0027 des Neurons:\n\n{}".format(first_neuron.w))

print("\nInitiales Biasgewicht "0027b"0027: \n\n {}".format(first_neuron.b))

#-----------------------------------------------------
# Inputvektor dem Neuron übergeben und
# vorwärts prozessieren
#-----------------------------------------------------
y_neuron = first_neuron.forward(x)

#-----------------------------------------------------
# Ergebnis ausgeben
#-----------------------------------------------------
print("\nAktivierung "0027y"0027 des Neurons: {}".format(y_neuron))
```

Code 4: Verarbeitung eines Inputsignals durch ein künstliches Neuron.

1.2 Neuronale Netze

Wie in dem vorangehenden Kapitel beschrieben, können die Axonterminale eines Neurons über ***Synapsen*** in Kontakt zu Dendriten anderer Neuronen stehen. An einem derartigen synaptischen Übergang

erfolgt eine Signalweiterleitung über chemische bzw. elektrische Prozesse von einem Neuron zum nächsten. Verbinden sich wie in Abbildung 1.2 gezeigt mehrerer Neuronen in dieser Art miteinander, bilden die neuronalen Verbindungen ein sogenanntes ***Neuronales Netz (NN)*** aus.

1.2.1 Modellierung vollständig verknüpfter Neuronaler Netze

Die Modellieurg dieser neuronalen Verknüpfungen zu einem ***künstlichen Neuronalen Netz (kNN)*** ist in Abbildung 1.2 dargestellt. Die Anordnung der künstlichen Neuronen erfolgt hier in einzelnen Schichten (layer). Sind alle Ausgänge (rot) der Neuronen einer Schicht mit den Eingängen (blau) der Neuronen einer darauf folgenden Schicht verbunden, werden diese Schichten als miteinander ***vollständig verknüpft*** bezeichnet.

Wie bei einem einzelnen Neuron wird der Input einer jeden Schicht s mit einem Gewicht $w_{j,k}$ belegt. Werden zwei Schichten mit j bzw. k Gewichten vollständig miteinander verknüpft, resultiert dies in $j \cdot k$ Gewichten, welche sich in einer Gewichtsmatrix $\mathbf{W}$ zusammenfassen lassen. Weiterhin ist für jedes Neuron einer Schicht ein entsprechendes Biasgewicht $b_{s,j}$ definiert. Die Aktivierungsfunktion der Neuronen, wie sie in Abbildung 1.1 mit angeführt ist, wird in dieser Graphik aus Übersichtsgründen nicht explizit dargestellt.

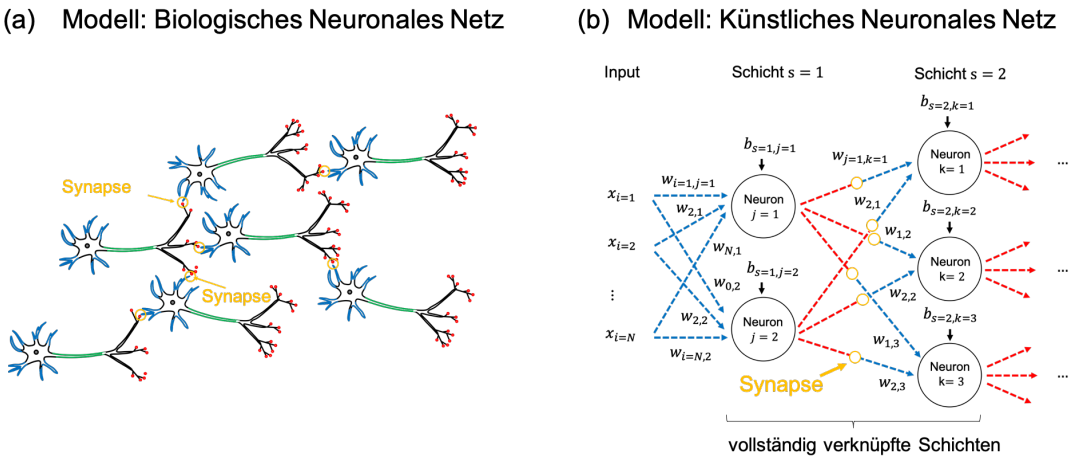


Abbildung 1.2: Beziehung zwischen (a) biologischen und (b) künstlichen Neuronalen Netz.

1.2.2 Implementierung einer vollständig verknüpften Schicht

Im Folgenden wird exemplarisch eine Python-Implementierung einer vollständig verknüpften Schicht gezeigt, bei der ein Inputvektor x mit allen Neuronen einer Schicht verbunden ist. Dies erfolgt innerhalb der Klasse ***FullyConnectedLayer***. Im Vergleich zur Implementierung eines einzelnen Neurons bestehen die Anpassung hautsächlich darin, dass der Gewichtungsvektor w (Klasse Neuron: Codezeile 28) in die Gewichtungsmatrix $\mathbf{W}$ übergeht (Klasse FullyConnectedLayer: Codezeile 25). Dementsprechend wird aus dem skalaren Bias des Neurons ein Biasvektor, welcher alle Biaswerte der Schicht zusammenfasst. Die Funktion ***forward*** prozessiert den Input x durch die vollständig verknüpfte Schicht.

```
class FullyConnectedLayer:
    """
    ---------------------------------------------------------------------
    Argumente:   dim_input (int):  Dimension des Inputvektors
                 size_layer (int): Anzahl der Neuronen in der Schicht
                 act_func:         Aktivierungsfunktion der Schicht
    ---------
    Attribute:   W (ndarray):      Gewichtungsmatrix
                 b (ndarray):      Bias-Vektor
                 x (ndarray):      Input-Vektor
                 y (ndarray):      Output-Vektor
    ---------------------------------------------------------------------
    """
```

```
    #-------------------------------------------------
    # Konstruktor
    #-------------------------------------------------
    def __init__(self, dim_input, size_layer, act_func):

        self.size     = size_layer
        self.act_func = act_func

        #-------------------------------------------------
        # Gewichtungsmatrix W der Größe "0027dim_input x size_layer"0027
        #-------------------------------------------------
        self.W = np.random.standard_normal((dim_input, size_layer))

        #-------------------------------------------------
        # Biasvektor der Länge (size_layer)
        #-------------------------------------------------
        self.b = np.random.standard_normal(size_layer)

        #-------------------------------------------------
        # Input
        #-------------------------------------------------
        self.x = None

        #-------------------------------------------------
        # Outputs
        #-------------------------------------------------
        self.y = None

    #-------------------------------------------------
    # Prozessieren eines Inputs durch die Schicht
    #-------------------------------------------------
    def forward(self, x):
        """
        ------------------------------------------------------------------------
        Argumente:       input_neuron (ndarray)   -->  Shape: (batch_size, num_inputs)
        ---------------
        Rückgabewerte:   output_neuron (ndarray)  -->  Shape: (batch_size, layer_size)
        ------------------------------------------------------------------------
        """
        #-------------------------------------------------
        # Input für späteren Gebrauch speichern
        #-------------------------------------------------
        self.x = x

        #-------------------------------------------------
        # Gewichtung
        #-------------------------------------------------
        z = np.dot(self.x, self.W) + self.b

        #-------------------------------------------------
        # Output für späteren Gebrauch speichern
        #-------------------------------------------------
        self.y = self.act_func(z)

        return self.y
```

Code 4: Pythonimplementierung der Klasse FullyConnectedLayer.

1.2.3 Verwendung der Klasse FullyConnectedLayer

Im Folgenden wird die oben definiert Klasse ***FullyConnectedLayer*** exemplarisch dazu genutzt, einen Stapel zufällig generierter Inputvektoren x zu prozessieren. Der vollständig verknüpften Schicht können dabei die Inputvektoren einzeln nacheinander oder als kompletter Stapel auf einmal (Batch) übergeben werden.

```
#-------------------------------------------------
# Zufallsgenerator fixieren, um bei jeden Durchlauf
# reproduzierbare Ergebnisse zu erlangen
#-------------------------------------------------
np.random.seed(0)

#-------------------------------------------------
```

```
# Größe des Inputs definieren
#-------------------------------------------------
input_size    = 4

#-------------------------------------------------
# Anzahl der Neuronen der Schicht definieren
#-------------------------------------------------
num_neurons   = 3

#-------------------------------------------------
# Vollständig verknüpfte Schicht instanziieren
# Wahl einer Aktivierungsfunktion (s.o.)
#-------------------------------------------------
layer = FullyConnectedLayer(dim_input=input_size, size_layer=num_neurons, act_func=relu_func)

#-------------------------------------------------
# Zwei Inputs per Zufallsgenerator generieren
#-------------------------------------------------
x1 = np.random.uniform(-1, 1, input_size).reshape(1, input_size)
x2 = np.random.uniform(-1, 1, input_size).reshape(1, input_size)

#-------------------------------------------------
# Inputs zu einem Batch verbinden
#-------------------------------------------------
x_batch = np.concatenate((x1, x2))
#-------------------------------------------------
# Batch durch die Schicht propagieren
# (oder jeden Input einzeln)
#-------------------------------------------------
y12 = layer.forward(x_batch)
#-------------------------------------------------
# Ausgabe
# Für jedes Neuron der Schicht wird für jeden
# Datensatz des Batches ein Output erzeugt
#-------------------------------------------------
print("Ausgabe der vollständig verknüpften Schicht für alle Inputdaten (Batch):\n\n{}".format(y12))
```

Code 5: Verwendung der Klasse FullyConnectedLayer.

1.3 Zusammenfassung

In diesen Teil der Lerneinheit wurde gezeigt, wie ein biologisches Neuron modelliert und mittels Python implementiert werden kann. Aufbauend darauf wurde das Prinzip neuronaler Verknüpfunge erläutert. Exemplarisch wurde weiterhin die Realisierung einer vollständig verknüpften Schicht mittels Python vorgestellt. Im nachfolgendem Notebook wird erläutert, wie die Gewichte eines aus vollständig verknüpften Schichten bestehenden Neuronalen Netzes so verändert werden können, dass sich dem Netz ein gewünschtes Verhalten antrainiert lässt. Die Umsetzung des Trainingsverfahrens wird ebenfalls in Python gezeigt.