Einführung in die Neuroinformatik SoSe 2019

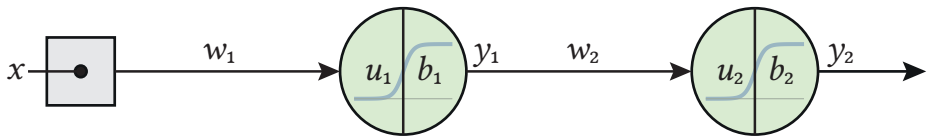
Institut für Neuroinformatik
PD Dr. F. Schwenker

7. Aufgabenblatt (Abgabe bis 25. Juni 2019 zur Vorlesung)

Aufgabe 1 (4 Punkte): *Learning Slowdown* [Pen and Paper]

Wenn wir ein neuronales Netzwerk auf ein Problem anwenden, hätten wir gerne, dass der Lernprozess möglichst schnell vonstattengeht. Beim Gradientenverfahren, welches wir für diesen Zweck verwenden, kann der Lernvorgang allerdings sehr langsam sein. In dieser Aufgabe wollen wir uns mit diesem Problem befassen. In späteren Aufgaben werden wir Lösungsansätzen begegnen.

Als Beispiel soll uns dabei das folgende Netzwerk dienen, das aus einem Eingabe-, einem Neuron in der Zwischenschicht und einem Ausgabeneuron besteht.



Bei den gezeigten Transferfunktionen handelt es sich um die Sigmoidfunktionen (Abbildung 1)

$$f(x) = \frac{1}{1 + e^{-x}} \quad \text{mit der Ableitung} \quad f'(x) = f(x) \cdot (1 - f(x)). \tag{1}$$

Die dendritischen Potenziale ergeben sich zu

$$u_1 = w_1 \cdot x + b_1$$
$$u_2 = w_2 \cdot y_1 + b_2$$

und die dazugehörigen axonalen Potenziale als $y_i = f(u_i)$. Wir arbeiten mit der quadratischen Fehlerfunktion, wobei nicht über mehrere Trainingssamples gemittelt wird

$$E = (T - y_2)^2 = (T - y_2 \left[u_2 \left(y_1 \left[u_1 \left(b_1\right)\right], b_2\right)\right])^2. \tag{2}$$

Betrachten wir nun, welche Probleme hier auftreten können.

1. Nehmen wir zuerst an, wir würden nur das 2. Neuron lernen lassen. Des Weiteren wollen wir uns nur den Bias b_2 ansehen, für das Gewicht w_2 sind die Probleme jedoch ähnlich. Die Ableitung von Gleichung 2 nach b_2 ergibt sich durch Anwendung der Kettenregel als

$$\frac{\partial E}{\partial b_2} = \frac{\partial E}{\partial y_2} \frac{\partial y_2}{\partial u_2} \frac{\partial u_2}{\partial b_2} = -2 \cdot (T - y_2) \cdot f'(u_2). \tag{3}$$

   Gehen wir davon aus, dass wir ein Lehrersignal von $T = 0$ erreichen wollen und uns das 1. Neuron mit einer konstanten Ausgabe von $y_1 = 1$ versorgt.

a) Wir setzen das initiale Gewicht zuerst auf $w_2 = 0.6$ und den Bias auf $b_2 = 0.9$. Berechne den Gradienten mit diesen Variablen. Führe zudem die dazu passende Animation[1] aus dem Buch *Neural Networks and Deep Learning* von Michael A. Nielsen aus, welche ein einzelnes Neuron mit diesen Parametern trainiert.

b) Jetzt setzen wir das Gewicht und den Bias auf $w_2 = b_2 = 2$. Berechne auch hier wieder den Gradienten und führe die entsprechende Animation[2] aus.

c) Welches Problem haben wir im letzen Fall und welcher Faktor aus Gleichung 3 ist dafür hauptsächlich verantwortlich (gemäß der eben durchgeführten Rechnungen)?

2. Lassen wir nun beide Neuronen lernen.

a) Berechne die Ableitung der Fehlerfunktion nach dem Bias b_1

$$\frac{\partial E}{\partial b_1} = \dots$$

und argumentiere, ob das eben kennengelernte Problem für das 1. Neuron verstärkt oder abgeschwächt wird.

b) Wie würde sich das Problem bei noch mehr Zwischenschichten weiterentwickeln (keine Rechnung notwendig)?

c) Warum kann sich dadurch die Suche nach einem lokalen Minimum in die Länge ziehen?

3. Es gibt noch ein weiteres Problem, welches zwar seltener ist, allerdings ebenfalls auftreten kann. Sei auch hier wieder ein Lehrersignal von $T = 0$ gegeben und der Input des Netzwerks als konstant $x = 1$ angenommen. Die Parameter des 1. Neurons sind zudem auf $w_1 = 100, b_1 = -100$ und die des 2. Neurons auf $w_2 = 100, b_2 = -50$ gesetzt.

a) Berechne für beide Neuronen die Gradienten

$$\frac{\partial E}{\partial b_2} = \dots$$
$$\frac{\partial E}{\partial b_1} = \dots$$

b) Nehmen wir an, wir hätten noch mehr Zwischenschichten in unserem Netzwerk und würden alle Gewichte weiterhin auf $w_i = 100$ setzen sowie jeden Bias b_i so wählen, dass das dendritische Potenzial $u_i = 0$ verschwindet. Wie entwickelt sich dann die eben begonnene Folge weiter? Es ist dabei nicht notwendig, weitere Ableitungen zu berechnen. Es genügt, aus den bestehenden Ergebnissen eine Fortsetzung abzuschätzen.

[1] http://neuralnetworksanddeeplearning.com/chap3.html#saturation1
[2] http://neuralnetworksanddeeplearning.com/chap3.html#saturation2

c) Was für ein Problem können wir nun beobachten? Warum kann dies die Suche nach einem lokalen Minimum ebenfalls erschweren?

4. Tabelle 1 vergleicht die Lernregeln für Neuronen der Ausgabe- und Zwischenschicht einmal unter Verwendung der quadratischen Fehlerfunktion und einmal unter Verwendung der *cross entropy*-Funktion als Kostenfunktion. Inwieweit löst die *cross entropy*-Funktion zumindest zum Teil das hier beschriebene Problem?

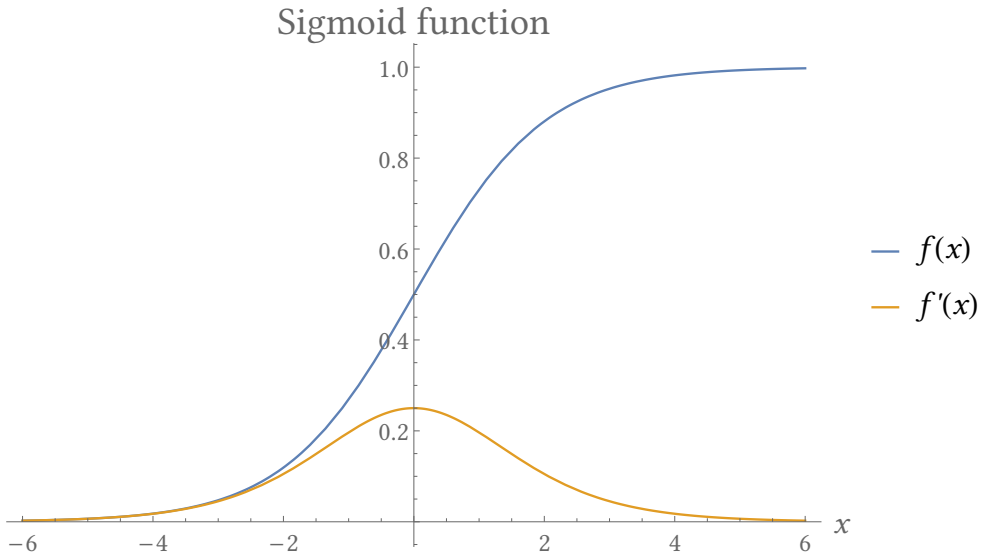


Abbildung 1: Sigmoidfunktion und deren Ableitung (Gleichung 1).

Tabelle 1: Vergleich der Ableitungen für die Ausgabe- und Zwischenschicht unter Verwendung der quadratischen Fehlerfunktion $E_q = \frac{1}{2}\sum_{j=1}^{n}\left(T_j - y_j^{(2)}\right)^2$ bzw. der *cross entropy*-Funktion $E_c = -\sum_{j=1}^{n} t_j \cdot \ln\left(y_j^{(2)}\right)$. $k = 1, \ldots, m$ iteriert über die m Eingabeneuronen, $i = 1, \ldots, h$ über die h Neuronen in der Zwischen- und $j = 1, \ldots, n$ über die n Neuronen in der Ausgabeschicht. Jeweils zur Vereinfachung keine Betrachtung von einzelnen Trainingssamples.

Ausgabeschicht	Zwischenschicht
$\frac{\partial E_q}{\partial w_{ij}^{(2)}} = \left(y_j^{(2)} - T_j\right) \cdot f'\left(u_j^{(2)}\right) \cdot y_i^{(1)}$	$\frac{\partial E_q}{\partial w_{ki}^{(1)}} = \sum_{j=1}^{n}\left(y_j^{(2)} - T_j\right) \cdot f'(u_j^{(2)}) \cdot w_{ij}^{(2)} \cdot f'(u_i^{(1)}) \cdot x_k$
$\frac{\partial E_c}{\partial w_{ij}^{(2)}} = \left(y_j^{(2)} - t_j\right) \cdot y_i^{(1)}$	$\frac{\partial E_c}{\partial w_{ki}^{(1)}} = \sum_{j=1}^{n}\left(y_j^{(2)} - t_j\right) \cdot w_{ij}^{(2)} \cdot f'\left(u_i^{(1)}\right) \cdot x_k$

Aufgabe 2 (6 Punkte): *Flat vs. Deep Networks*

Wir haben bis jetzt nur mit neuronalen Netzwerken gearbeitet, die maximal eine Zwischenschicht verwenden. Auch wenn wir theoretisch mit einer einzigen Zwischenschicht bereits jede stetige Funktion approximieren können, so haben Netze mit mehreren Zwischenschichten dennoch praktische Vorteile. Ein Problem bei nur einer Zwischenschicht kann beispielsweise sein, dass sehr viele Neuronen benötigt werden, um die Zielfunktion gut zu approximieren. Dies führt wiederum zu vielen lernbaren Gewichten sowie Schwellwerten und damit zu einem erhöhten Lernaufwand. In dieser Aufgabe wollen wir uns dieses Problem ansehen und in einem Experiment simulieren.

Als Beispiel arbeiten wir dazu mit binären Variablen $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$ und $\boldsymbol{y} = (y_1, y_2, \ldots, y_n)$ sowie dem Skalarprodukt Modulo 2

$$\langle \boldsymbol{x}, \boldsymbol{y} \rangle_n = \left(\sum_{i=1}^{n} x_i \cdot y_i \right) \mod 2. \tag{4}$$

Der Vorteil dieser Funktion ist, dass eine theoretische Fundierung darüber besteht, wie viele Neuronen mit einer und wie viele mit mehreren Zwischenschichten benötigt werden. Konkret reichen für den letzteren Fall sogar bereits zwei mittlere Schichten. Wir wollen das Netzwerk mit einer Zwischenschicht net_f (für *flat*) und das Netzwerk mit zwei mittleren Schichten net_d (für *deep*) bezeichnen.

Wie viele Neuronen $h(net)$ benötigt werden, hängt maßgeblich von der Dimension n der Variablen $\boldsymbol{x}$ und $\boldsymbol{y}$ ab. Es ist zu erwarten, dass bei höherem n mehr Neuronen benötigt werden, allerdings ist die Wachstumsrate in beiden Fällen unterschiedlich:

$$\begin{aligned} h(net_f) &= \mathcal{O}(2^n) \\ h(net_d) &= 2n + 1 \end{aligned} \tag{5}$$

Wir haben es also bei net_f mit exponentiellem Wachstum zu tun, wohingegen net_d mit $2n + 1$ Neuronen auskommt (n in jeder Schicht und 1 Ausgabeneuron).

1. [Pen and Paper] Bevor wir die Ergebnisse aus Gleichung 5 in einer Simulation überprüfen, wollen wir uns erst mit der Lösung net_d vertraut machen. Abbildung 2 zeigt dazu das Netzwerk net_d für den Fall $n = 2$.

   a) Berechne für die Eingabe $\boldsymbol{x} = (0, 0)$ und $\boldsymbol{y} = (1, 1)$ die Netzwerkausgabe $f(\boldsymbol{x}, \boldsymbol{y})$ und überprüfe das Ergebnis auch anhand von Gleichung 4.

   b) Erweitere das Netzwerk für $n = 3$.

   c) Welche Rolle kommt den Neuronen in der 1. und 2. Zwischenschicht jeweils zu? Welche Art Zwischenergebnis berechnen sie? Hinweis: mit den Neuronen in der ersten Zwischenschicht hatten wir bereits in einem früheren Aufgabenblatt zu tun.

2. [Python] Wir wollen nun mehrere Netzwerke trainieren, um Gleichung 5 zu überprüfen. Dabei geben wir nur die Hyperparameter des Netzwerkes vor (wie z. B. die Anzahl der Neuronen) und lassen die restlichen Parameter anhand von Trainingsdaten, die aus Gleichung 4 generiert werden, lernen. Zur Beurteilung der Netzwerke soll uns dabei die quadratische Fehlerfunktion dienen. Wir nehmen die Implementierung der neuronalen Netzwerke nicht selbst vor, sondern setzen hierbei auf Keras und das TensorFlow-Backend.

   a) Einer der Hyperparameter ist die Aktivierungsfunktion der einzelnen Neuronen. In Abbildung 2 wurden dazu Schwellwertneuronen verwendet. Erkläre, warum wir diese Funktion in unseren Netzwerken nicht verwenden können (Hinweis: Lernregeln für die quadratische Fehlerfunktion, siehe Tabelle 1). Stattdessen verwenden wir $\tanh(x)$ als Aktivierungsfunktion sowohl in den Zwischen- als auch in der Ausgabeschicht.

   b) Zuerst müssen die Eingabedaten generiert werden. Bestimme dazu für ein vorgegebenes n alle möglichen Binärkombinationen von $\boldsymbol{x}$ und $\boldsymbol{y}$ und ermittele die zugehörigen Lehrersignale (Gleichung 4).

   c) Es empfiehlt sich, ein neuronales Netzwerk mit positiven und negativen Werten arbeiten zu lassen (besonders bei Verwendung von $\tanh(x)$ als Transferfunktion). Verwende daher -1 anstatt 0 sowohl in den Eingabedaten als auch im Lehrersignal.

   d) Kümmern wir uns nun um die Netzwerke. Implementiere dazu eine Funktion, welche für eine gegebene Konfiguration der Zwischenschicht (Anzahl Schichten und Anzahl Neuronen in den Schichten) ein neuronales Netzwerk trainiert und dann den geringsten Fehler zurückgibt. Verwende dabei den folgenden Codeausschnitt als Basis und beachte auch die unten aufgeführten Punkte.

```
import numpy as np
import tensorflow as tf
import tensorflow.keras.backend as K

def train_network(n_hidden):
    """
    Trains a neural network and returns the lowest error.

    :param n_hidden: Number of hidden neurons to use per
        layer (as vector to indicate when multiple hidden
        layers should be used). For example, [2] uses one
        hidden layer with two neurons and [2, 2] uses two
        hidden layers each with two neurons.
    :return: The lowest error (MSE) occurred over all
        training epochs.
    """
```

```
# Start fresh and at least try to get reproducible
   results
tf.reset_default_graph()
K.clear_session()
tf.set_random_seed(42)
np.random.seed(42)

# ...
```

- Erstelle ein `Sequential`-Modell[3] in Keras.
  - Die Eingabeschicht wird automatisch erstellt.
  - Für die Zwischen- und Ausgabeschicht können `Dense`-Layer verwendet werden.
  - Verwende für die Zwischenschichten eine andere Initialisierungsmethode. Es sollen Daten aus einer Gleichverteilung im Bereich $[-0.5; 0.5]$ generiert werden. Hinweis: hierfür gibt es in Keras bereits ein vorgefertigtes Objekt.
- Verwende die folgenden Einstellungen für den Optimierer[4]:

  ```
  optimizers.SGD(lr=0.2, decay=0.0001, momentum=0.9,
     nesterov=True)
  ```

- Verwende die quadratische Fehlerfunktion (MSE).
- Trainiere jeweils über 300 Epochen und speichere den kleinsten Fehler über alle Epochen.

e) Beginnen wir mit der eigentlichen Simulation. Um zu sehen, ob wir Gleichung 5 bestätigen können, lassen wir mehrere Netzwerke trainieren und erhöhen dabei schrittweise die Anzahl der Neuronen in der Zwischenschicht (bzw. Zwischenschichten im Falle von net_d) bis leicht über das theoretische Optimum von Gleichung 5 hinaus. Wir sollten dann bemerken, dass sich ab diesem Optimum keine größeren Verbesserungen mehr ergeben. Aus Performance-Gründen beschränken wir hier die Berechnung auf $n = 3$.

   i. Trainiere für net_f Netzwerke mit $\{1, 2, \ldots, 2^n + 4\}$ Neuronen in der Zwischenschicht und speichere jeweils den besten Fehler.

   ii. Trainiere für net_d Netzwerke mit $\{(1, 1), (2, 2), \ldots, (n + 4, n + 4)\}$ Neuronen in den beiden Zwischenschichten. Speichere auch hier jeweils den besten Fehler.

[3] https://keras.io/getting-started/sequential-model-guide/
[4] Mit Optimierern beschäftigen wir uns noch ausführlicher im nächsten Aufgabenblatt.

f) Zeige die Ergebnisse aus der vorherigen Simulation in einer Grafik an. Gib dazu den Fehler als Funktion der verwendeten Gesamtanzahl von Neuronen (Zwischen- und Ausgabeschicht) an (analog zu den Grafiken in Abbildung 3).

3. [Pen and Paper] Abbildung 3 zeigt Simulationsergebnisse auch für höhere Eingabedimensionen n. Interpretiere die Ergebnisse insbesondere im Hinblick auf Gleichung 5.

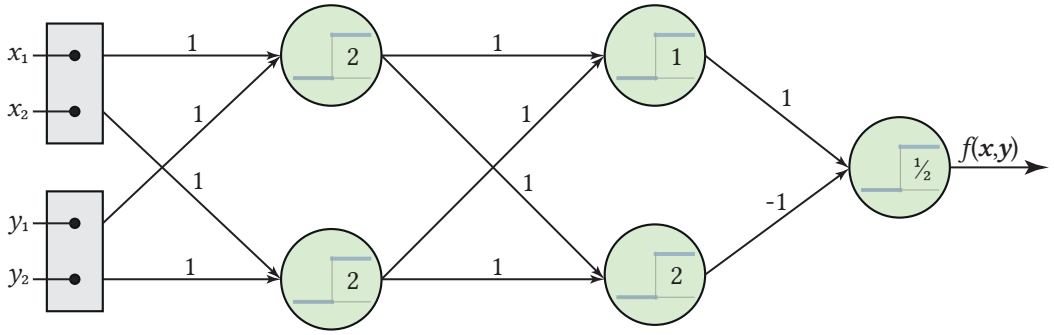
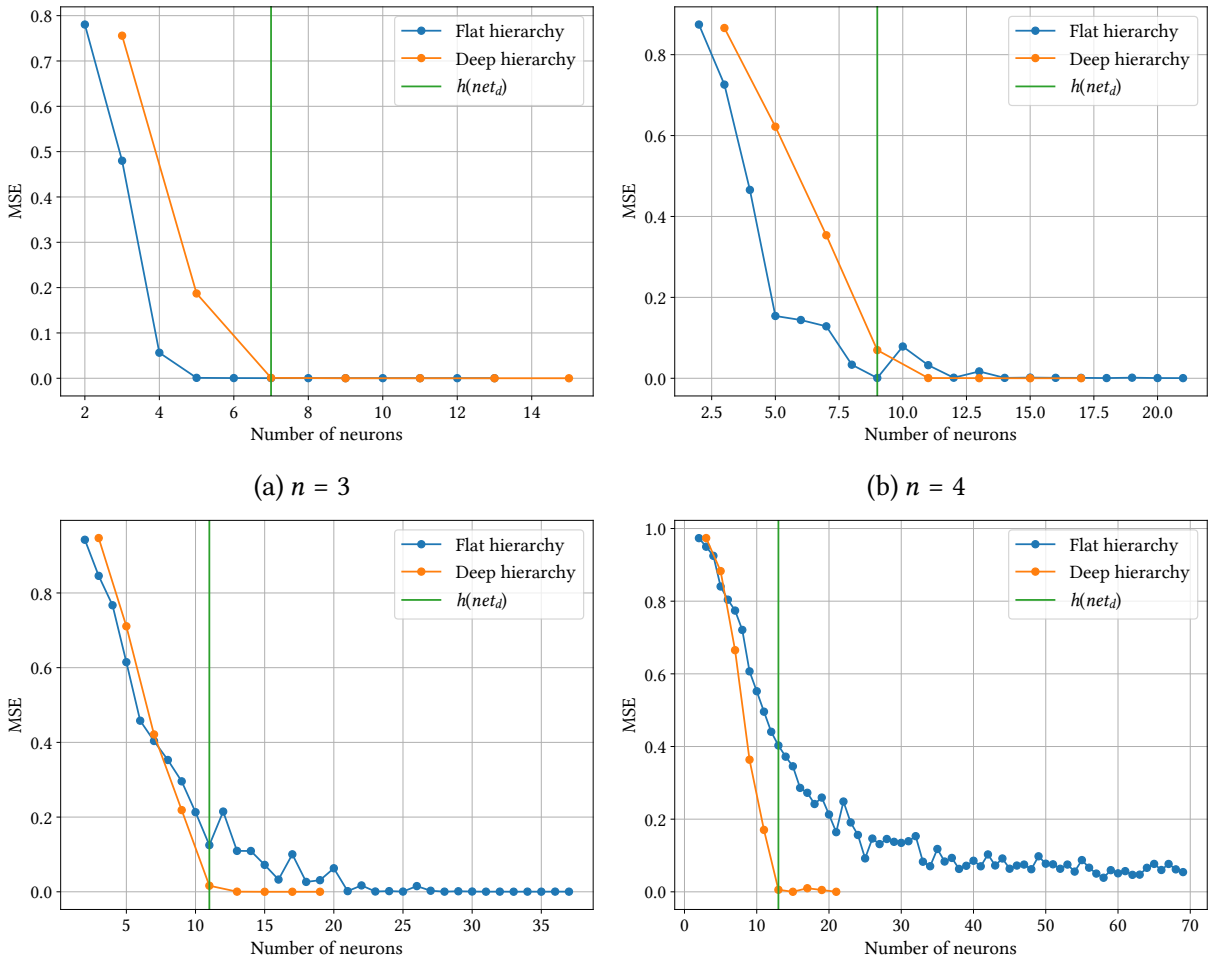


Abbildung 2: Netzwerk, welches Gleichung 4 für $n = 2$ berechnet. Es gibt vier Eingabeneuronen (zwei für $\boldsymbol{x}$ und zwei für $\boldsymbol{y}$), jeweils zwei Neuronen in den Zwischenschichten und ein Ausgabeneuron. Als Aktivierungsfunktion wird die Schwellwertfunktion mit dem im Neuron angegebenen Schwellwert θ verwendet. Fehlende Verbindungen von der Eingabe- zur ersten Zwischenschicht sind auf 0 gesetzt.



(a) $n = 3$

(b) $n = 4$

(c) $n = 5$

(d) $n = 6$

Abbildung 3: Simulationsergebnisse zum Erlernen von Gleichung 4 für verschiedene Eingabedimensionen n. Hinweis: im Vergleich zu der in Aufgabe 2 beschriebenen Vorgehensweise wurden für diese Simulationen insgesamt 20 Netzwerke pro Einstellung trainiert, um den kleinsten Fehler unter verschiedenen Gewichtsinitialisierungen zu ermitteln.