Gruppe TIB: Blank, Karsten; Ray, Sebastian; Groß, Fabian
23.12.2022

Laborversuch 3

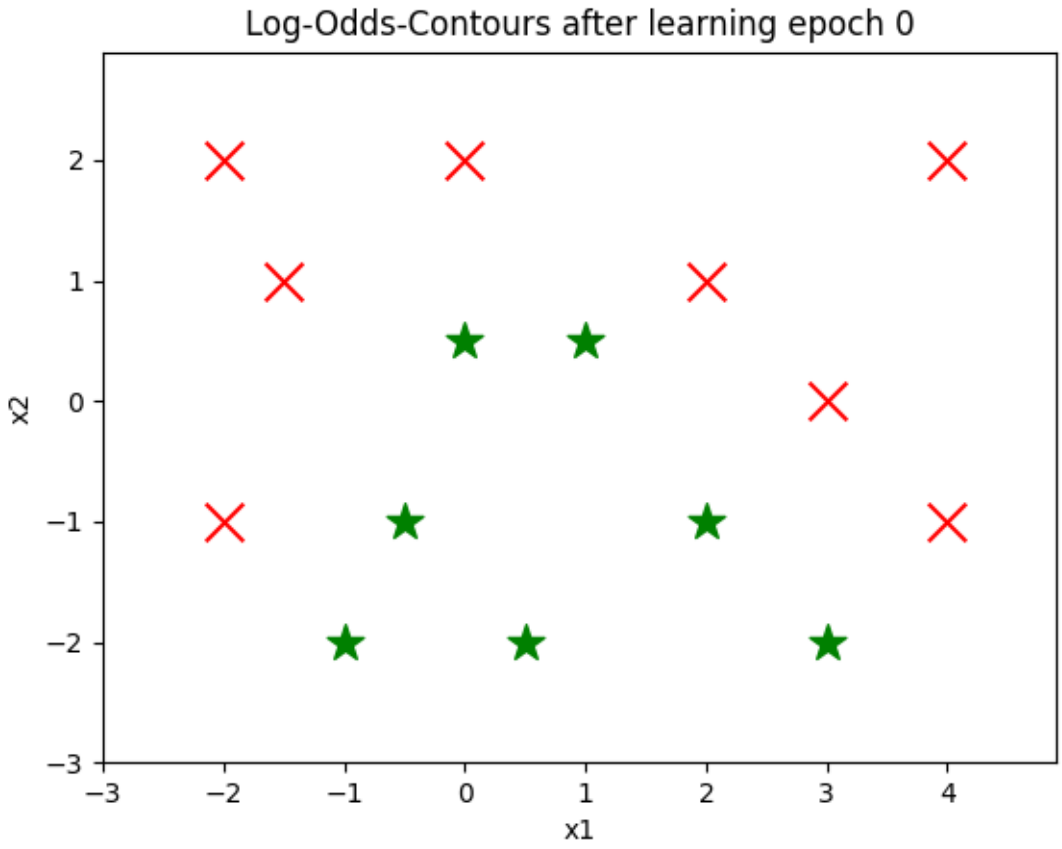
Versuch Neuronale Netze und Backpropagation

Fach Intelligente Lernende Systeme

Semester WS 2022/23

Aufgabe 1:

a) siehe Skizze:



b) Linear separierbar: Es kann eine Gerade / Linie durch die Klassen gelegt werden, welche beide Klassen eindeutig trennt. Im Beispiel nicht möglich, daher nicht linear separierbar

c) Die Datenmenge ist nicht fehlerfrei durch ein einschichtiges Perzeptron klassifizierbar, da dieses nur für linear separierbare Klassen einsetzbar ist.

d) Die Datenmenge ist durch ein MLP fehlerfrei klassifizierbar (für nicht linear sepb. Klassen geeignet)

Aufgabe 2:

a)

def softmax(): Berechnet die softmax-Funktion (normalisierte Exponentialfunktion) eines Vektors, d.h. transfomiert den Vektor in den Wertebereich (0,1) mit Komponentensumme 1. Dies ist hilfreich für die Klassifikation in der Output-Layer, da die Werte als Wahrscheinlichkeiten, dass der Vektor einer bestimmten Klasse angehört, interpretiert werden können („One-Hot-Kodierung“,hilfreich insb. bei K>2 Klassen)

def forwardPropagateActivity(): Ermittelt die neuronale Aktivität durch das Netz, beginnend vom Inputlayer in Vorwärtsrichtung zum Outputlayer. Berechnet die dendritischen Potentiale und daraus im Anschluss die Feuerraten(vektoren).

def backPropagateErrors(): Fehler-Rückverbreitung - Die Funktion intitialisiert Fehlersignale / Abweichungen im Outputlayer und berechnet das Fehlerpotential durch das Netz vom Outputlayer in Rückwärtsrichtung zum Inputlayer. Als Ergebnis erhält man die beiden Fehlersignalvektoren.

def doLearningStep(): Führt einen Backpropagation Lernschritt mit dem Backpropagation-Algorithmus und Lernschrittweite b aus. Ruft forwardPropagateActivity() auf und berechnet die Feuerraten. Ruft anschließend backPropagateErrors() auf und berechnet die Fehlerraten. Mit diesen wird dann das Gewichtsupdate berechnet.

def getError(): Berechnung der Gesamt-Kreuzentropie-Fehlerfunktion über den gesamten Datensatz für MLP mit Gewichtsmatrizen.

def plotDecisionSurface(): Stellt das „Ergebnis der Klassifikation“ (bzw. die Veränderung der Entscheidungsgrenze für eine spätere Klassifikation von neuen Datenvektoren) mittels MLP nach einer bestimmten Anzahl an Lernschritten dar (funktioniert nur für 2-dimensionale Vektoren und 2 Klassen). Die Kreuze und Sterne entsprechen den Datenpunkten, die Linien dem Log-Odds-Ratio und die Log-Odds-Ratio 0 entspricht somit der Klassengrenze (Klasse 1 und 2 sind an dieser Stelle genau gleich wahrscheinlich).

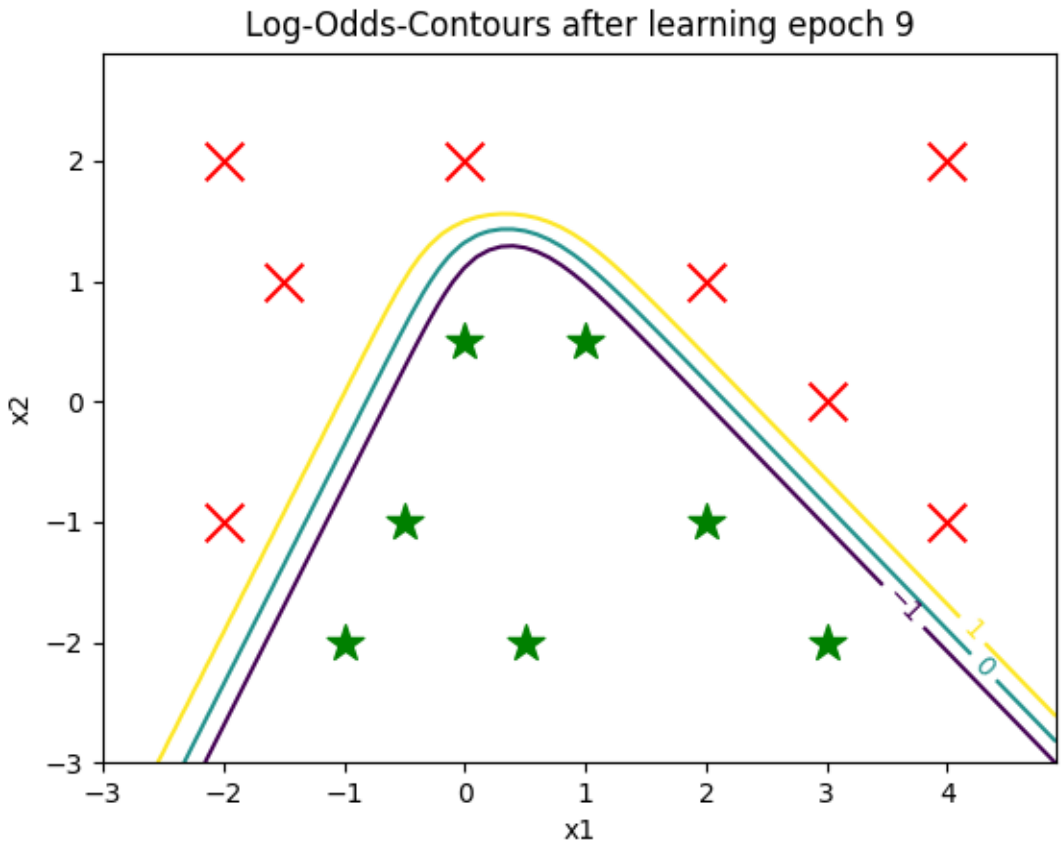
e)

Das Hauptprogramm generiert ein Trainingsdatenset in Arrays und trainiert das MLP in den angegebenen Durchläufen mit den gewünschten Parametern. Es gibt die gefundenen Klassifikationsfehler je Epoche auf der Konsole und das „Ergebnis der Klassifikation“ bzw. die berechnete Entscheidungsgrenze mit Hilfe der Funktion plotDecisionSurface (s.o.) als Schaubild aus.

MLP-Parameter:

- M: Größe des Hidden Layers = Anzahl der Neurone im Hidden Layer und der Pararmeter im Netzwerk.
- eta: Lernschrittweite
- nEpochs: Anzahl der durchzuführenden Trainings-Durchläufe / Gewichtupdates
- flagBiasUnit: Erweitert das hidden Layer um einen Bias.

Mit M=3 und eta=0,4 kann eine „korrekte Klassifizierung“ in 7-9 Lernepochen erreicht werden.




```
after epoch  5  error function E= 4.838071252345162  and classification errors =  3
after epoch  6  error function E= 3.2002141559195385  and classification errors =  1
after epoch  7  error function E= 2.1726872899804257  and classification errors =  0
after epoch  8  error function E= 1.149082069384411  and classification errors =  0
after epoch  9  error function E= 0.8129582202530147  and classification errors =  0
```

Aufgabe 3:

a)
MLPs sind sehr flexible Modelle/Algorithmen, die zur Klassifikation (auch bei nicht-linearen Klassifikations-Problemen) eingesetzt werden können. Daher macht es Sinn, sie zu den vorhandenen linearen Klassifikationsalgorithmen von Versuch 1 hinzuzufügen.

b)
def configure(): Dient der Konfiguration des MLP (falls Parameter von den Standardwerten abweichen sollen).

def printState(): Ausgabe des MLP (dessen Eigenschaften/Status) in der Konsole für Debugging bzw. Nachvollziehen für den Anwender.

def setTrainingData(): Definiert die Trainingsdaten als Arrays in One-Hot-Codierung, fügt ggf. Bias-Unit hinzu.

def setRandomWeights(): Generiert zwei Gewichtsmatrizen mit Zufallswerten zwischen -0,5 und 0,5.

def getError(): Gibt die Werte der Kreuzentropie mit Regularisierung zurück

def doLearningEpoch(): Führt einen Lernschritt mit festgelegter Lernschrittweite durch. Ein Lerndatensatz wird dabei vollständig durchlaufen.

def doLearningTrial(): Führt einen Lernversuch durch, d.h. führt Lernschritte so lange durch bis die maximale Epochen-Anzahl erreicht ist oder der Fehler kleiner ist als der vorgegebene „gewünschte/zulässige“ Fehler (eps).

def fit(): Trainiert den Klassifikator mit den Trainingsdatenarrays x und T. Konkret führt er eine bestimmte (als Parameter definierte) Anzahl an Lernversuchen durch. Wenn sich die Ergebnisse durch den Lernversuch verbessern (Fehler kleiner wird), werden die neuen Ergebnisse übernommen.

def predict(): Implementiert den Algorithmus zur Klassifikation, d.h. er verwendet die mit fit() ermittelten optimalen Gewichte und berechnet die Wahrscheinlichkeiten, dass der Vektor zu einer bestimmten Klasse gehört.

c)
Bei zu großer Lernrate wird unter Umständen das Minimum nie erreicht / divergiert. Kleine feste Lernraten benötigen unter Umständen (zu) viele Lernepochen bis ein gutes Modell erreicht wird.

Lösung:
Mit „adaptiver Kontrolle“ der Lernrate (=“annealing“, Lernrate wird immer kleiner). Die Lernrate wird dadurch kontinuierlich angepasst, der Gradient für den neuen Gewichtungsvektor erneut bestimmt und der Prozess wiederholt. Die Gewichte werden nur dann übernommen, wenn der Fehler sich verkleinert. So wird garantiert, dass in jeder Lernepoche das Netzwerk optimiert wird.

Die Parameter eta0=1.0 und eta_fade=1.0/5 bestimmen die Eingangslernrate und der Faktor zur Anpassung der Lernrate pro Lernschritt (der Faktor 1/5 bedeutet dabei, dass nach 5 Lern-Durchgängen die Lernrate halbiert wird).

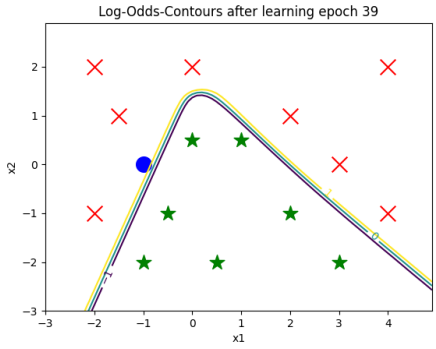
d)
Es werden 2 verschiedene Trainingsdatensets erstellt mit welchen anschließend nach Auswahl / Flag ein MLP mit unterschiedlichen Parametern trainiert und getestet werden kann. Werte werden auf der Konsole und Datensets als Plots ausgegeben.

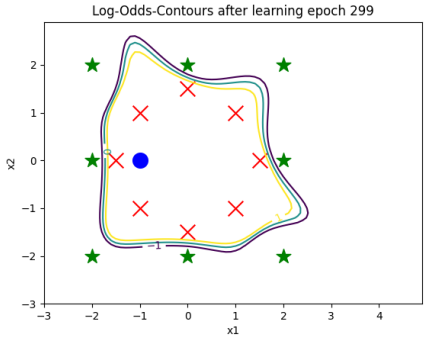
Erklären Sie kurz die Bedeutung der Parameter:

eta0:	Eingangslernrate. Initiale Lernrate mit der der erste Lernschritt durchgeführt wird.
eta_fade:	Faktor mit dem die Lernrate angepasst wird.
maxEpochs:	Maximale Lernschritte
nTrials:	Anzahl der Lernversuche
eps:	Grenzwert für Unterschreitung des Fehlers bei einer Lernepoche.

Optimierung der Parameter:

```
if flagDataset1>0:
    M=3                          # number of hidden units
    eta0=0.6                      # initial learning rate
    eta_fade=1.0/30                # fading factor for decreasing learning rate (e.g., 1/50
means after 50 epochs is learning rate half the initial value...)
    maxEpochs=75                 # number of learning epochs
else:
    M=8                           # number of hidden units
    eta0=0.6                      # initial learning rate
    eta_fade=1.0/30                # fading factor for decreasing learning rate (e.g., 1/50
means after 50 epochs is learning rate half the initial value...)
    maxEpochs=300                 # number of learning epochs
flagBiasUnits=1                   # bias units in input layer and hidden layers?
```





Aufgabe 4:

a)

Zunächst wird ein MLP-Objekt erzeugt, dann die Daten geladen und vorbereitet (ggf. standardisiert, s.u.) und anschließend mittels der Funktion crossvalidate auf den Daten trainiert sowie das Modell validiert (die Methode crossvalidate ruft hierzu die Funktionen fit und predict auf, siehe V1A2_Classifier.py).

b)

Der Data-Scaler dient zur Daten-Vorverarbeitung/ Standardisierung. MLP sind sensitiv zum Mittelwert und Varianz der Daten, was zu numerischer Instabilität führen kann. Das System auf Rohdaten lernen zu lassen kann darin resultieren, dass nicht immer die optimalen Gewichte gefunden werden, es zu schlechten Ergebnissen kommt. Um dem entgegenzuwirken, sollten die Trainingsdaten standardisiert werden.

c)

```
# (i) Define and construct MLP
print("\n(I) Define and construct MLP:")
M=4                     # number of hidden units
flagsBiasUnits=1        # bias units in input layer and hidden layer?
lmbda=1                 # regularization coefficient
eta0=0.6                # initial learning rate
eta_fade=1./10           # fading factor for decreasing learning rate (e.g., 1/50 means after 50
epochs is learning rate half the initial value...)
maxEpochs=300            # max. number of learning epochs
nTrials = 1             # number of learning trials
eps = 1e-4              # stop learning if (normalized) error function becomes smaller than eps
debug = 1               # if >0 then debug mode: 1 = print Error, mean weight; 2=additionally
check gradients; 3=additionally print weights
flagScaleData = 1       # if >0 then scale data vectors in X
mlp = MLP3Classifier(M,flagsBiasUnits,lmbda,eta0,eta_fade,maxEpochs,nTrials,eps,debug)
```

Ergebnis zu o.g. Hyper-Parametern:

```
S= 3  fold cross validation using the MLP yields the following results:
Classification error probability =  0.08795411089866156
Accuracy =  0.9120458891013384
Confusion Error Probabilities p(class i|class j) =
 [[0.91794872 0.10465116 0.02515723 0.02409639]
 [0.04102564 0.88372093 0.01257862 0.        ]
 [0.04102564 0.01162791 0.93710692 0.09638554]
 [0.         0.         0.02515723 0.87951807]]
Computing time =  20.156779766082764  sec
```