NNW-Übung 3

1 Neuron mit voreingestellten Gewichten

Ziel dieser Teilaufgabe ist zunächst ein ganz einfaches Neuron für die Unterscheidung von zwei Klassen zu implementieren – noch ohne es zu trainieren.

Als Daten soll wieder der Iris-Blüten Datensatz dienen und davon wieder nur die *ersten zwei Merkmale* je Blüte verwendet werden.

Das Neuron soll die Klasse 0 von den beiden anderen Klassen unterscheiden, Klasse 1 und 2 werden also *als eine Klasse* betrachtet (sozusagen die Klasse „nicht-Klasse 0“).

Für die Implementierung soll die in der Vorlesung verwendete Konvention verwendet werden:

- Die Merkmale $x_1, x_2, \ldots$ *eines* Objekts werden in *Spalten*vektoren $\vec{x}$ gespeichert. Die Merkmale stehen also in den *Zeilen*.
- Haben wir viele Objekte, z.B. N Trainingsbeispiele, so speichern wir diese Daten in einer Matrix X: *je Trainingsbeispiel eine Spalte*! Das ist nichts anderes, als das Nebeneinanderschreiben der Merkmalsvektoren (Spalten) der einzelnen Objekte!
- Die Zielwerte bei supervised learning sollen entsprechend in einer Matrix T (für target) stehen.

Die Datei `NNW_P3_A1_Vorlage.py` enthält ein Code-Gerüst als Ausgangspunkt.

a) Ergänzen Sie die Funktion `neuron(X)`, die die Berechnungen eines Neurons mit *zwei*, zunächst *fest vorgegebenen* Gewichten `W=[-0.3,1]` und *fester* Aktivierungsschwelle von 2 für *alle* Iris-Daten (=vermessene Iris-Blüten) in X durchführen soll.

   Implementieren Sie die Aktivierungsschwelle als Bias: Speichern Sie das Bias-Gewicht in einer Variablen `b` und nutzen Sie `b` im Sinne des Bias in der Berechnung.

   Der Rückgabewert soll ein Array mit boolschen Werten (`True` = Neuron sendet Signal, `False` = Neuron sendet kein Signal) sein, das genauso viele XXX wie X enthält, wobei XXX für „Spalten“ oder „Zeilen“ steht.

   Tipps/Anmerkungen:

   - `N=X.shape[XXX]` soll die Anzahl der Blüten enthalten (`XXX` steht wieder für Spalten oder Zeilen).
   - `net` ist ein zunächst mit Nullen gefülltes Array, das die Netzwerksummen für die insgesame `N` Blüten aufnehmen soll.
   - Definieren Sie `W=[-0.3,1]` und `b` einfach als lokale Variablen der Funktion.
   - Benutzen Sie `for`-Schleifen zur Berechnung der Netzwerksummen `net[n]`, wobei `n` für die Nummer der Blüte stehen soll.
   - Die Anwendung der Aktivierungsschwelle ist dann nur noch eine Zeile, siehe Aufgabenblatt 1 – oder Sie fragen mich. Auf *keinen Fall* eine Schleife hierfür programmieren!

b) Lassen Sie das Ergebnis der Klassifikations durch Ihre `neuron`-Funktion mit Hilfe von der Funktion `nnwplot.plotTwoFeatures` (siehe das vorige Aufgabenblatt) plotten.

c) Experimentieren Sie mit anderen Aktivierungsschwellen als 2, also zum Beispiel 1 und 3. Was ändert sich?

   Mit `plt.figure()` können Sie neue Plot-Fenster öffnen, damit Sie die verschiedenen Plots zur Beantwortung der folgenden Fragen besser vergleichen können. Wenn dann zu viele Fenster offen sind, können Sie sie mit `plt.close('all')` alle schließen.

d) Experimentieren Sie mit anderen Gewichten `W`, zum Beispiel `W=[-0.2,1]`,`W=[-0.1,1]`, `W=[0,1]`. Was ändert sich?

2 Arbeiten mit Python und NumPy – Teil 2

Probieren Sie den folgenden Python-Code zeilenweise auf der Console aus und vollziehen Sie ihn nach:

```
import numpy as np

x=np.array([1.,2,3]) # Der Punkt bewirkt Datentyp float.
W=np.array([[1.,2,3],[4,3,2]]) # Der Punkt bewirkt Datentyp float.

W   # auf der Console werden Variablen ohne Zuweisung ausgegeben
    # in Skripten müssen Sie print(W) für Ausgaben schreiben

x.shape
W.shape

W.T
x.T       # seltsam ...? Logik:  1D bleibt 1D. x ist *kein* Spalten-Vektor!
y=x[np.newaxis].T # erzeugt 2D Matrix 1x3 (=Zeilenvektor!) und transponiert sie.
y
y.shape

# numpy-Arrays können beliebig viele Dimensionen haben:
M=np.ones( (2,3,4,5) )
M.ndim
M.shape
M.T.shape # Transponieren bei >2 Dimensionen ... wie ändert sich der shape?

# Zu Klassen siehe https://docs.python.org/3/tutorial/classes.html
# und insbesondere https://docs.python.org/3/tutorial/classes.html#tut-private
class myclass:
    def __init__(self,W): # Konstruktor, self entspricht this in Java/C++
        self._W = W       # Konvention: private Variablen beginnen mit _
                          # self muss beim Zugriff angegeben werden, sonst
                          # ist es eine lokale Variable
    def f(self): # self muss immer explizit übergeben werden, sonst "static"
        return self._W*2

m=myclass(np.array([-1,1]))
m._W  # Zugriff ist auch auf private Variablen erlaubt
m.f() # m wird automatisch als "self" übergeben
```

Aus dem History-Tab von Spyder (zweites Tab im Fenster ganz rechts unten) können Sie den Code nachträglich in eine Datei im Editor-Fenster kopieren und abspeichern.

3 Klasse für ein einschichtiges Netz, Training mit Delta-Regel

Ziel dieser Teilaufgabe ist eine Klasse `SLN` (für single-layer network), die ein einschichtiges *Netz* realisiert – also mehrere Neuronen enthalten kann, die aber alle in derselben Schicht (Layer) sind und

damit letztlich unabhängig voneinander sind. Die Datei `SLN_Vorlage.py` enthält ein Code-Gerüst als Ausgangspunkt.

a) Der Konstruktor soll folgenden Parameter bekommen:

- `dIn`: Anzahl der Eingabe-Werte (in der Vorlesung mit d (bzw. d_k) bezeichnet)
- `cOut`: Anzahl der Neuronen/Ausgabe-Werte (in der Vorlesung mit c bezeichnet)

und der außerdem die folgenden Instanz-Variablen initialisiert:

- `_W`: *Matrix* der Gewichte (ohne Bias!). Die Dimension dieser Matrix ergibt sich aus den Parametern. Setzen Sie `_W` auf zufällige, normalverteilte Werte mit `numpy.random.randn` (diese Funktion kann direkt eine Matrix von Zufallszahlen erzeugen). Initialisieren Sie den Zufallsgenerator vorher durch den Aufruf `numpy.random.seed(42)`, um reproduzierbare Ergebnisse zu erhalten. Teilen Sie die generierten Zufallszahlen durch die Wurzel aus `dIn+1`.
- `_b`: (Spalten-)Vektor mit den Bias-Gewichten. Initialisieren Sie `_b`=$\vec{0}$ (mit `numpy.zeros`). Erzeugen Sie `_b` als Spaltenvektor.

b) Kopieren Sie Ihre Funktion `neuron(X)` von Aufgabe 1 in die Klasse. Ändern Sie die Funktion so ab, dass sie die Instanzvariable `_W` benutzt (und überhaupt benutzen kann), wobei Sie für diese Aufgabe weiter davon ausgehen können, dass nur ein Neuron (`cOut`=1) verwendet wird. Ergänzen Sie die Verwendung des Bias `_b`.

c) Testen Sie, ob Ihre neue Klasse bei den Experimenten der Aufgabe 1 dieselben Ergebnisse liefert. Setzen Sie dazu (entgegen der Konvention) die eigentlich privaten Variablen `_W` und `_b` in Ihrem Hauptprogramm entsprechend.

d) Die im Folgenden definierte globale Funktion `ErrorRate(Y, T)` soll die Fehlklassifikationsrate ausrechnen. Die Fehlklassifikationsrate ist die Anzahl der inkorrekt klassifizierten Trainingsmuster dividiert durch die Gesamtanzahl der Trainingsmuster. Vollziehen Sie den „if-Teil“ der Funktion nach – der „else-Teil“ ist erst für die Unterscheidung von mehr als zwei Klassen relevant, was noch nicht Teil dieses Aufgabenblatts ist.

```
def ErrorRate(Y,T):
    if Y.ndim==1 or Y.shape[0]==1:
        errors=Y!=T
        return errors.sum()/Y.size
    else: # für mehrere Ausgaben in one-hot Kodierung:
        # Dies brauchen Sie jetzt noch nicht nachzuvollziehen.
        errors=Y.argmax(0)!=T.argmax(0)
        return errors.sum()/Y.shape[1]
```

e) Ergänzen Sie die Implementierung der Delta-Regel in der Methode `SLN.DeltaTrain` der Klasse `SLN`, die die Parameter `X` und `T` für die Trainingsbeispiele, `eta` für die Lernrate η, `maxIter` für die maximale Anzahl Iterationen und `maxErrorRate` für die maximale Fehlklassifikationsrate hat.

Der Bias `b` soll mit trainiert werden, also genauso behandelt werden wie die übrigen Gewichte. Implementieren Sie das Verfahren als Batch-Learning.

Der Lernvorgang soll abgebrochen werden, wenn entweder die maximale Anzahl `maxIter` Iterationen (Lernzyklen, Epochen) erreicht oder die maximale Fehlklassifikationsrate `maxErrorRate` unterschritten wurde.

f) Trainieren Sie ein „Netz“ für das UND-Gatter-Problem.