Universität Münster
AG AIS – Prof. Dr. Malte Schilling
www.uni-muenster.de/AISystems Einsteinstr. 62 48149 Münster

Mini-Lab 2 zur Einführung Neuronale Netzwerke: Lineare Regression und Merkmalsanalyse

Janosch Bajorath
j.bajorath@uni-muenster.de
Simon Neumeyer
sneumeye@uni-muenster.de
Malte Schilling
malte.schilling@uni-muenster.de

Abgabe bis zum 18.1.2026 über das LearnWeb.

Ziele: In diesem Mini-Lab sollen Sie den Trainingsprozess mehrschichtiger neuronaler Netze von Grund auf verstehen. Dazu implementieren Sie die beiden Richtungen des *Learning-Cycle*: Die Vorwärtsberechnung / *Prediction* des Modells und den Lernprozess über *Back-propagation*. Dies setzen Sie um zuerst für ein minimales neuronales Modell aus Eingabe- und Ausgabeschicht in einer *Multi-Class*-Klassifikationsaufgabe. Dies wird dann erweitert zu einem *Multi-Layer Perceptron* (MLP). Dabei untersuchen Sie Gradientenberechnung, Aktivierungsfunktionen, Lernratenauswahl und Generalisierung. Anschliessend setzen Sie dies in PyTorch um.

Tasks (jeweils mit 25 % gewichtet; zum Bestehen werden 50 % der Gesamtpunkte benötigt):

1. **Analyse des Spiral-Datensatzes und einfacher neuronaler Klassifikator**
   Visualisieren und analysieren Sie den gegebenen Drei-Klassen-Spiral-Datensatz. Trainieren Sie zunächst ein einfaches neuronales Netz aus Eingabe- und Ausgabeschicht und untersuchen Sie dessen Grenzen.

2. **Einführung eines *Hidden Layers* und Backpropagation**
   Erweitern Sie Ihr Modell zu einem Multi-Layer Perceptron mit einem Hidden Layer. Leiten Sie die Backpropagation-Gleichungen für dieses Modell her und implementieren Sie das Training mittels Gradientenabstieg.

3. **Finden optimale Neuronale Netz-Architektur**
   Variieren Sie die Anzahl der Neuronen im Hidden Layer und analysieren Sie den Einfluss der Modellkapazität auf Trainingsverhalten, Generalisierung und Overfitting.

4. **PyTorch-Reimplementierung**
   Erstellen Sie ein äquivalentes MLP in PyTorch.

Datensatz: In diesem Aufgabenteil verwenden wir einen synthetischen Spiral-Datensatz aus drei Klassen. Jede Klasse bildet einen eigenen Spiralarm im zweidimensionalen Raum. Der Datensatz ist nicht linear separierbar und eignet sich daher gut, um die Grenzen linearer Klassifikatoren sowie die Leistungsfähigkeit von MLPs zu untersuchen.

Der Spiral-Datensatz wird als NumPy-Datei (.npz) bereitgestellt und enthält getrennte Trainings- und Validierungsdaten (dazu stellen wir einen weiteren Datensatz zur Verfügung, in dem die Datenpunkte mit einer größeren Streuung erzeugt wurden). Die Daten könne. so direkt geladen und verarbeitet werden (siehe in den vers. Tests und Beispiele in den Aufgabenteilen).

Aufgabe 1: Analyse Spiral-Datensatz und einfaches neuronales Modell

In dieser Aufgabe sollen Sie grundlegend ein einfaches neuronales Netz (ohne Hidden Layer) erstellen, trainieren und damit einen ersten Eindruck vom Datensatz bekommen. Dies wird als Baseline-Modell dienen und soll die Grenzen einfacher linearen Modelle aufzeigen.

Im Verlauf dieser Aufgabe sollen Sie:

- ein einfaches neuronales Modell zur Klassifikation (*one-vs-rest*) implementieren,
- das Modell trainieren,
- den Lernprozess in einer Lernkurve visualisieren,
- die Klassifikationseigenschaften untersuchen und darstellen über eine Visualisierung, die die linearen Entscheidungsregionen zeigt (farbige Flächen) sowie die Trainingspunkte überlagert.

Hinweise zur Implementierung:

- Verwenden Sie ein einfaches Interface für Ihren Klassifikator (angelehnt an scikit-learn, dies ist im git schon vorangelegt):

```
class SimpleNeuralNetwork:
    def fit(self, X, y):
        pass  # Training

    def predict(self, X):
        pass  # Klassenlabels
```

- **Sigmoid-Aktivierung:** Nutzen Sie eine sigmoide Aktivierungsfunktion. Im Forward Pass berechnet sich die Aktivierung dabei zuerst als

$$a(\vec{w}, \vec{x}^{(i)}) = \vec{w}^T \vec{x}^{(i)}$$

und dann nach Anwendung der Sigmoid-Funktion über

$$\hat{y} = \sigma(a) = \frac{1}{1 + \exp(-a)}.$$

Die Ableitung der Sigmoid-Funktion lautet

$$\frac{\partial \sigma}{\partial a} = (1 - \sigma(a))\sigma(a).$$

- **Fehlerfunktion:** Wir verwenden weiterhin den mittleren quadratischen Fehler (MSE) über alle Ausgabeneuronen (dargestellt für eine Outputdimension):

$$E(\vec{w}) = \frac{1}{2} \sum_{i=1}^{m} (\hat{y}^{(i)} - y^{(i)})^2.$$

Die Ableitung nach dem Modelloutput ergibt sich dann als

$$\frac{\partial E(\vec{w})}{\partial \hat{y}} = (\hat{y}^{(i)} - y^{(i)}).$$

- **Backward Pass:** Die Ableitung nach den Gewichten folgt aus der Kettenregel:

$$\frac{\partial E(\vec{w}, \vec{x}^{(i)})}{\partial \vec{w}} = \underbrace{\frac{\partial E(\vec{w})}{\partial \hat{y}}}_{(\hat{y}^{(i)} - y^{(i)})} \quad \underbrace{\frac{\partial \hat{y}}{\partial a}}_{(1-\sigma(a))\sigma(a)} \quad \underbrace{\frac{\partial a}{\partial \vec{w}}}_{\vec{x}^{(i)}}.$$

Falls ein expliziter Bias genutzt wird, gilt $a(\vec{w}, \vec{x}^{(i)}) = \vec{w}^T \vec{x}^{(i)} + b$,
und die Ableitung nach dem Bias ist

$$\frac{\partial E(\vec{w}, b, \vec{x}^{(i)})}{\partial b} = \underbrace{\frac{\partial E(\vec{w})}{\partial \hat{y}}}_{(\hat{y}^{(i)} - y^{(i)})} \quad \underbrace{\frac{\partial \hat{y}}{\partial a}}_{(1-\sigma(a))\sigma(a)} \quad \underbrace{\frac{\partial a}{\partial b}}_{1}.$$

Überprüfung / Test:

- Das einfache neuronale Netz erreicht auf dem Testdatensatz eine Klassifikationsgenauigkeit von mindestens 50 %.
- Eine Trainingskurve (Loss während des Trainings) wird als Grafik ausgegeben.
- Dazu zeigt eine Visualisierung den Spiral-Datensatz sowie die durch den trainierten Klassifikator erzeugten linearen Entscheidungsregionen.

Für diesen Aufgabenteil bearbeiten Sie die folgenden Dateien:
`src/simple_nn.py` — Klasse `SimpleNeuralNetwork`.
`src/visualization.py` — Funktionen `plot_training_curve()` und `plot_decision_regions()`

Praktische Hinweise:

- Speichern Sie Aktivierungen im *Forward-Pass* — sie werden im *Backward-Pass* benötigt.
- Eine zu große Lernrate führt schnell zur Divergenz; starten Sie mit kleinen Werten (um 0.01).
- Initialisieren Sie Gewichte zufällig und klein, aber nicht mit Null.

Aufgabe 2: Backpropagation für ein MLP mit einem Hidden Layer

In dieser Aufgabe erweitern Sie den Klassifikator zu einem Multi-Layer-Perceptron (MLP) mit genau einem Hidden Layer. Ziel ist es, die Backpropagation-Gleichungen für dieses Netzwerk herzuleiten, korrekt zu implementieren und das Lernverhalten auf dem nichtlinear separierbaren Datensatz zu untersuchen.

Im Verlauf dieser Aufgabe sollen Sie:

- die Backpropagation-Gleichungen für ein Netzwerk mit einem Hidden Layer und Sigmoid-Aktivierungen herleiten und implementieren,
- ein MLP mit genau einem Hidden Layer mit fünf Neuronen umsetzen,
- Ihr Modell auf dem Drei-Klassen-Spiral-Datensatz zu trainieren und evaluieren,
- überprüfen, ob das Netzwerk in der Lage ist, nichtlineare Entscheidungsgrenzen korrekt zu modellieren.

Hinweise zur Implementierung:

- **Einführung eines Hidden Layers (Forward Pass)**
  Erweitern Sie Ihr Modell um einen Hidden Layer mit fünf Neuronen und implementieren Sie zunächst ausschließlich die Vorwärtsrechnung. Zur Überprüfung Ihrer Implementierung können Sie die folgenden Gewichtsmatrizen (achten sie bei den Matrizen darauf, wie Sie die Werte in Spalten / Zeilen übertragen) verwenden, die eine Klassifikationsgenauigkeit von über 0.90 erreichen sollten:

$w_{ih}^{[1]}$	1	2	3	4	5
bias	13.955	-1.079	-1.420	-9.452	-6.745
1	5.290	24.748	9.823	-22.713	13.622
2	21.145	-28.266	0.001	1.745	8.618

Gewichte von den Eingabeeinheiten zum Hidden Layer.

$w_{ho}^{[2]}$	1	2	3
bias	14.265	-0.281	-25.521
1	-17.921	3.912	23.039
2	-10.631	-5.133	15.726
3	27.089	-17.227	-18.574
4	-0.646	-12.115	7.881
5	-24.915	32.302	-11.516

Gewichte vom Hidden Layer zur Ausgabeschicht.

- **Backpropagation** (Hidden ← Output)
  Beginnen Sie mit dem Lernen der Gewichte zwischen Ausgabeschicht und Hidden Layer, während die Gewichte der ersten Schicht zunächst fixiert bleiben. Für die Rückwärtsrechnung ergibt sich der Gradient als:

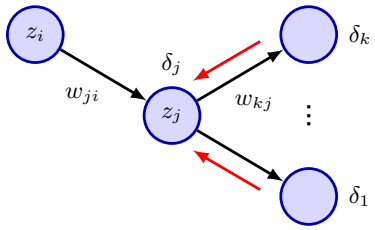
$$\frac{\partial E(\vec{w})}{\partial \vec{w}} = \underbrace{\frac{\partial E}{\partial \hat{y}}}_{(\hat{y}^{(i)} - y^{(i)})} \underbrace{\frac{\partial \hat{y}}{\partial a}}_{(1-\sigma(a))\sigma(a)} \underbrace{\frac{\partial a}{\partial \vec{w}}}_{\vec{z}^{[1]}}$$

  Nutzen sie hierbei die in der Vorlesung eingeführte δ-Notation (zusammengefasst im nächsten Abschnitt).

- **Vollständige Backpropagation** (Input ← Hidden)
  Erweitern Sie anschließend die Rückwärtsrechnung auf die erste Gewichtsmatrix, indem Sie den Fehler mithilfe der δ-Notation weiter zurückpropagieren und so alle Gewichte des Netzwerks lernen (Eingang ← Hidden Layer).

Die δ-Notation für Backpropagation

Für das Zurückpropagieren des Fehlers durch das Netzwerk ist es hilfreich, eine Kurzschreibweise für die Fehlerterme einzelner Neuronen zu verwenden. Die δ-Notation wurde in der Vorlesung eingeführt (angelehnt an Bishop and Bishop (2023)) und erlaubt eine kompakte Darstellung der Backpropagation-Gleichungen.



Dabei beschreibt δ stets die Ableitung der Fehlerfunktion nach der Voraktivierung eines Neurons.

1. **Fehlerterm der Ausgabeschicht (Output)**

   Für ein Ausgabeneuron k ergibt sich der Fehlerterm als Ableitung des Fehlers nach der Voraktivierung a_k:

   $$\delta_k = \frac{\partial E^{(n)}}{\partial a_k} = \frac{\partial E^{(n)}}{\partial \hat{y}_k} \frac{\partial \hat{y}_k}{\partial a_k}$$

   Bei Verwendung der Sigmoid-Aktivierungsfunktion gilt dabei:

   $$\frac{\partial \hat{y}_k}{\partial a_k} = (1 - \sigma(a_k))\sigma(a_k)$$

2. **Fehlerterm des Hidden Layers** (Hidden ← Output)

   Für ein Hidden-Neuron j ergibt sich der Fehlerterm aus der gewichteten Summe der Fehler der nachfolgenden Schicht, multipliziert mit der Ableitung der Aktivierungsfunktion:

   $$\delta_j = \frac{\partial E^{(n)}}{\partial a_j} = h'(a_j) \sum_k \delta_k w_{jk}$$

   Dabei ist $h(a) = \sigma(a)$ die Sigmoid-Aktivierungsfunktion des Hidden Layers.

   Gradienten der Gewichte

   Sobald die δ-Terme bekannt sind, ergibt sich der Gradient eines Gewichts als Produkt aus dem Fehlerterm des Zielneurons und der Aktivierung des Quellneurons:

   $$\frac{\partial E^{(n)}}{\partial w_{ji}} = \delta_j \, z_i$$

   wobei z_i die Aktivierung des Neurons der vorhergehenden Schicht bezeichnet (Eingabe oder Hidden Layer).

Überprüfung / Test:

- **Backpropagation-Schritt**
  Es wird überprüft, dass ein einzelner Gradienten-Schritt (berechnet über `backward(...)` und angewendet mit `gradient_step(...)`) die Fehlerfunktion reduziert.

- **Klassifikationsgenauigkeit**
  Nach vollständigem Training erreicht das implementierte MLP (Hidden Layer mit 5 Neuronen) auf dem Spiral-Datensatz eine Klassifikationsgenauigkeit von mindestens 90 %, bestimmt über den Aufruf von `predict(...)` auf dem Testdatensatz.

Tipps / Hinweise:

- **Aktivierungen speichern**
  Die Aktivierungswerte (Voraktivierungen a und Ausgaben nach der Sigmoid-Funktion) werden für die Rückwärtsrechnung benötigt und müssen während des Forward Passes gespeichert werden.

- **Lernrate**
  Bei Netzwerken mit Hidden Layern reagiert das Training deutlich sensibler auf die Wahl der Lernrate. Die vorgegebene Lernrate ist ein sinnvoller Startpunkt, sollte jedoch experimentell variiert werden.

- **Trainingsdauer**
  Das Training eines MLP benötigt in der Regel deutlich mehr Iterationen als ein lineares Modell. Mehrere hundert bis tausend Epochen sind für stabile Ergebnisse typisch.

- **Initialisierung der Gewichte**
  Gewichte sollten zufällig mit kleinen Werten initialisiert werden (aber ungleich Null).

Aufgabe 3: Modellkapazität und Generalisierung

In dieser Aufgabe untersuchen Sie den Einfluss der Modellkapazität auf das Lernverhalten und die Generalisierungsfähigkeit des implementierten MLPs. Ausgehend von Ihrem Netzwerk aus der Voraufgabe mit einem Hidden Layer, sollen Sie systematisch die Anzahl der Neuronen variieren und dabei die resultierende Performance analysieren.

Im Verlauf dieser Aufgabe sollen Sie:

- die Anzahl der Neuronen im Hidden Layer variieren,
- für jede Modellgröße das Netzwerk auf dem Spiral-Datensatz trainieren,
- Trainings- und Validierungsgenauigkeit vergleichen und dabei Overfitting beachten,
- eine geeignete Hidden-Layer-Größe finden, die gute Generalisierung zeigt.

Überprüfung / Test:

- **Einfluss der Modellgröße**
  Es wird eine Grafik erzeugt, die die Klassifikationsgenauigkeit in Abhängigkeit von der Hidden-Layer-Größe darstellt. Diese wird über die Funktion `plot_hidden_size_vs_accuracy(...)` erstellt und als Datei gespeichert.

Aufgabe 4: Umsetzung MLP-Klassifikation mit PyTorch

In dieser Aufgabe setzen Sie das zuvor selbst implementierte neuronale Netz in `PyTorch` als aktuellem Deep-Learning-Framework um. Ziel ist es, das MLP mit einem Hidden Layer zu trainieren und systematisch zu untersuchen, wie sich Optimierungsverfahren und dabei Batch-Größen auf das Lernverhalten auswirken.

Im Verlauf dieser Aufgabe sollen Sie:

- ein MLP mit einem Hidden Layer (dies können Sie gerne darüber hinaus noch variieren) in `PyTorch` anzuwenden,
- das Modell auf dem Drei-Klassen-Spiral-Datensatz trainieren,
- den Einfluss unterschiedlicher Batch-Größen auf das Lernverhalten untersuchen,
- verschiedene Optimierungsverfahren vergleichen (z. B. SGD vs. Adam),
- und die Lernkurven vergleichend visualisieren.

Hinweise zur Implementierung:

- **Modellarchitektur:** Implementieren Sie ein MLP mit einer Eingabeschicht (2 Dimensionen), einem Hidden Layer mit vorgegebener Neuronenzahl sowie einer Ausgabeschicht mit drei Neuronen. Verwenden Sie eine Sigmoid- oder ReLU-Aktivierung im Hidden Layer sowie eine geeignete Aktivierungsfunktion in der Ausgabeschicht.
- **Batch-Größen:** Untersuchen Sie mindestens zwei unterschiedliche Batch-Größen (z. B. `batch_size = 1` und `batch_size = 8`) und vergleichen Sie deren Einfluss auf Konvergenzgeschwindigkeit und Stabilität des Trainings.
- **Optimierungsverfahren:** Vergleichen Sie das klassische stochastische Gradientenverfahren (SGD) mit mindestens einem modernen Optimierer (z. B. Adam). Nutzen Sie hierfür die Implementierungen aus `torch.optim`. Hierfür wird ihnen bereits in `torch_train_mlp.py` entsprechender Code zur Verfügung gestellt, mit dem sie vers. Parameter anpassen können (aufgerufen wird dies von den Tests oder über die `main_task4_pytorch.py`).
- **Trainingsparameter:** Für die Anzahl der Trainings-Epochen nutzen sie einen gleichbleibenden Wert (z. B. 2500 Epochen). Beim SGD können sie mit einer Lernrate im Bereich von 0.05 arbeiten.
- **Visualisierung:** Berechnen Sie für jede Kombination aus Optimierer und Batch-Größe die mittlere Lernkurve über alle Durchläufe sowie die zugehörige Standardabweichung. Visualisieren Sie diese in einer gemeinsamen Grafik (Mittelwert mit Schattierung für die Streuung).

- **Training und Wiederholungen:** Schauen Sie sich auch einmal an, wie stark verschiedene Durchläufe des Trainings variieren und sich unterscheiden. Dies können Sie (optional – dies ist bereits in der Training-Loop angelegt und können Sie in dieser entsprechend einstellen) auch entsprechend visualisieren, in dem Sie das Training jeweils mehrfach mit unterschiedlichen Zufallsinitialisierungen durchführen, jeweils den Trainingsfehler pro Epoche speichern und darüber dann mittleren Trainingsfehler sowie Standardabweichung über Trainingszeit berechnen und darstellen.

Überprüfung / Test:

- **Lernkurven-Vergleich**
  Es wird eine Funktion genutzt, die ein MLP mit vorgegebener Batch-Größe und Wahl des Optimierers trainiert (inklusive mehrfacher Durchläufe) und die Lernkurven erzeugt und speichert (die entsprechende Trainings-Funktion, Laden und Beispielparameter sind schon bereits angelegt in `torch_train_mlp.py`; Sie müssen lediglich das entsprechende Modell und den Forward-Pass in `torch_mlp.py` implementieren).

- **Vergleich unterschiedlicher Batch-Größen**
  Dazu soll eine Lernkurve für mindestens zwei unterschiedliche Batch-Größen in einer gemeinsamen Visualisierung dargestellt werden (diese Visualisierung müssen Sie in `visualization.py` in der Funktion `plot_accuracy_comparison` umsetzen).

Für diesen Aufgabenteil bearbeiten Sie die folgenden Dateien:
`src/torch_mlp.py` (Modell und Training)
`src/visualization.py` (Visualisierung der Lernkurven)

Literaturverzeichnis

Christopher M Bishop and Hugh Bishop. *Deep learning: Foundations and concepts*. Springer Nature, 2023.