Fachbereich Mathematik und Statistik
PD Dr. Stefan Frei
Sommersemester 2025

Computergestützte Mathematik

Projekt 1 - Sentimentanalyse, Basisteil

Ausarbeitung von

Celil Durna, Dafina Kastrati

Ziel des Projekts

Im Rahmen des Projekts zur computergestützten Mathematik haben wir ein künstliches neuronales Netz entwickelt, das Filmkritiken automatisch mit einer Bewertung von 1 bis 10 klassifizieren soll. Grundlage dafür war ein realer Datensatz mit insgesamt 50.000 Kritiken, bei denen jeweils sowohl der Text als auch ein numerisches Rating vorgegeben waren. Unser Ziel war es, ein Modell zu bauen, das aus diesen Daten lernt und anschließend auch neue, unbekannte Kritiken sinnvoll einordnet.

Wir haben den Basisteil des Projekts bearbeitet. Dabei ging es vor allem um die Implementierung eines eigenen Feedforward-Netzes in Python: angefangen bei der Vorbereitung der Textdaten (z. B. mit Bag-of-Words-Vektorisierung und One-Hot-Encoding), über die Definition von Aktivierungsfunktionen, bis hin zum Training des Netzes mit Hilfe des stochastischen Gradientenverfahrens. Anschließend wurde der Lernfortschritt mithilfe von Kosten- und Erfolgsraten ausgewertet und visualisiert.

Alle Funktionen wurden innerhalb einer objektorientierten Klasse `nn_model` umgesetzt, wie es in der Aufgabenstellung vorgegeben war. Im Folgenden gehen wir auf unsere Implementierung für jede Teilaufgabe ein und beantworten auch die in den Aufgaben gestellten Fragen.

Implementierung und Antworten (auf 4, 6, 7)

Da unser Code bereits ausführlich kommentiert ist, gehen wir im Folgenden nur auf die wesentlichen Aspekte der Implementierung ein.

1) Datenvorverarbeitung

Zuerst haben wir mit `pandas` den Datensatz aus der Datei `movie_review_data.csv` geladen. Danach wurde mit der selbst implementierten Funktion `remove_stopwords()` eine Liste englischer Stoppwörter eingelesen (aus Textdatei), um diese aus der Spalte `text_clean` zu entfernen. Die gesäuberten Texte wurden als neue Spalte `text_wo_stopwords` gespeichert.

Im Anschluss haben wir mit `generate_data()` die Daten in Trainings-, Validierungs- und Testdaten aufgeteilt. Dafür wurden die Zeilen per `pandas.sample()` gemischt, um eine zufällige Verteilung zu bekommen. Daraufhin wurden nur die relevanten Spalten (`sentiment`, `rating`, `text_wo_stopwords`) beibehalten.

Aus den Trainingsdaten haben wir mit `bow_set()` die 250 häufigsten Wörter bestimmt. Diese wurden als Vokabular für die BoW-Vektorisierung genutzt. In der Funktion `bow_vectorization()` wurde jeder Text durch Zählen der Vorkommen dieser Wörter in einen 250-dimensionalen Vektor umgewandelt. Die Vektoren wurden als Liste in der neuen Spalte `bow_vectorized` gespeichert. Die bearbeiteten Datensätze wurden am Ende als CSV-Dateien exportiert.

2) Netzwerkinitialisierung

Das neuronale Netz wurde mit drei Schichten aufgebaut: 250 Eingabeneuronen (entsprechend der BoW-Vektoren), 100 Neuronen in der hidden Schicht und 10 Ausgabeneuronen (für die Ratingklassen 1-10). Die Initialisierung erfolgt in der Methode `nn_build()`. Die Gewichte wurden dabei mit `numpy.random.randn()` aus einer Normalverteilung mit Standardabweichung 0,12 erzeugt, Biases wurden mit Nullen initialisiert.

Die Eingabe- und Ausgabedaten aus dem DataFrame wurden mit `.tolist()` und `np.array()` in NumPy-Arrays konvertiert und dem Modell übergeben. Die Ratings wurden gleichzeitig per `nn_one_hot()` in One-Hot-Vektoren umgewandelt, ebenfalls als NumPy-Arrays.

3) One-Hot-Encoding

Für das One-Hot-Encoding wurde eine eigene Funktion geschrieben, die einen Integewert (Rating 1-10) in einen Vektor der Länge 10 umwandelt, wobei der passende Index auf 1 gesetzt wird. Zur Sicherheit wurde getestet, ob das Ergebnis der Funktion mit einer manuell erstellten Version übereinstimmt. Dies war der Fall.

4) Kostenfunktion

In dieser Aufgabe haben wir die Methode `nn_cost(Y, A_out)` implementiert, die die reduzierte Kreuzentropie-Kostenfunktion berechnet. Diese misst, wie stark die Vorhersagen des neuronalen Netzes von den tatsächlichen Klassen abweichen. Die Eingaben sind die Zielmatrix Y im One-Hot-Format sowie die Ausgabematrix A_{out} aus der letzten Schicht des Netzes. Die Kosten werden über alle Trainingsbeispiele gemittelt.

Um numerische Probleme wie log(0) zu vermeiden, haben wir ein kleines ε addiert. Der zentrale Ausdruck lautet:

```
cost = -np.sum(Y * np.log(A_out + epsilon)) / m
```

Zur Verifikation haben wir die Methode `nn_check()` verwendet, die die analytischen Gradienten mit numerisch berechneten vergleicht. Unsere Ergebnisse dabei waren:

```
pert=1.000e+00 | weights=1.449e-01 | biases=2.030e-01
pert=1.000e-01 | weights=1.791e-03 | biases=2.096e-03
pert=1.000e-02 | weights=1.803e-05 | biases=2.112e-05
pert=1.000e-03 | weights=5.022e-07 | biases=5.632e-07
pert=1.000e-04 | weights=4.060e-07 | biases=5.172e-07
```

Wie in Abschnitt 1.8 beschrieben, zeigt sich bei abnehmendem pert $\to 0$, dass der relative Fehler zwischen analytischen und numerischen Gradienten deutlich kleiner wird. Das bestätigt, dass die Ableitungen korrekt sind und die Kostenfunktion auch bei sehr kleinen Werten von A_{out} stabil bleibt. Der verwendete ε sorgt dafür, dass der Logarithmus definiert bleibt und keine NaNs oder Divergenzen entstehen.

5) Training

Das Training wurde mit Mini-Batch-Gradientenabstieg umgesetzt. Dabei haben wir pro Epoche die Trainingsdaten zufällig gemischt (`numpy.random.shuffle()`), in Batches der Größe 500 unterteilt und über 30 Epochen trainiert. Die Lernrate wurde dabei linear von $\eta_0 = 0{,}2$ auf $\eta_K = 0{,}01$ reduziert. Die Gewichte und Biases wurden in jeder Iteration mit den berechneten Gradienten aktualisiert.

6) Visualisierung

In Aufgabe 6 haben wir eine Methode `nn_successrate(Y, A_out)` implementiert, die berechnet, wie viele Beispiele vom Netzwerk korrekt klassifiziert wurden. Dafür wird jeweils die Position der höchsten Wahrscheinlichkeit in der Ausgabematrix `A_out` mit der tatsächlichen Klasse in `Y` verglichen. Die Erfolgsrate sowie die Kosten wurden zu Beginn des Trainings und nach jeder Epoche auf den Validierungsdaten berechnet und zur Auswertung in den Attributen `progress` und `costs` gespeichert. Mithilfe von Abbildung 1 lassen sich beide Metriken über die Epochen hinweg beobachten. Wir haben für die Visualisierung `matplotlib.pyplot` verwendet.

Beurteilung der Trainingserfolgs: Kosten vs. Erfolgsrate

Sowohl die Kostenfunktion als auch die Erfolgsrate geben uns wichtige Hinweise über den Zustand und Fortschritt des Modells. Die Erfolgsrate zeigt dabei klar, wie viele Beispiele das Modell richtig klassifiziert, also konkret, wie gut das Modell in der Praxis funktioniert. Sie ist intuitiv und leicht zu interpretieren: Eine Erfolgsrate von z. B. 85 % bedeutet einfach, dass auch 85 % der Bewertungen korrekt vorhergesagt wurden.

Die Kostenfunktion hingegen betrachtet auch die Wahrscheinlichkeit, mit der das Modell seine Vorhersagen trifft. Sie reagiert empfindlicher auf kleine Änderungen und ist somit ein gutes Maß für den Optimierungsprozess selbst. Wenn das Modell bei einer Vorhersage zwar falsch liegt, aber sehr unsicher war (z. B. 49 % für die richtige Klasse, 51 % für eine falsche), wird das von der Kostenfunktion berücksichtigt - bei der Erfolgsrate dagegen gilt das Beispiel einfach als falsch.

Ein Nachteil der Kostenfunktion ist allerdings, dass sie schwerer zu interpretieren ist. Ein niedrigerer Kostenwert heißt nicht automatisch, dass mehr Beispiele richtig klassifiziert wurden. Umgekehrt kann die Erfolgsrate lange gleich bleiben, auch wenn die Kostenfunktion bereits deutlich sinkt.

Für die Bewertung des tatsächlichen Trainingserfolgs, also wie gut das Modell in der Praxis funktioniert, empfinden wir deshalb die Erfolgsrate als aussagekräftiger. Die Kostenfunktion bleibt jedoch weiterhin ein sehr wichtiges Werkzeug, wenn es darum geht, den Trainingsprozess technisch zu steuern und die Optimierung voranzutreiben.

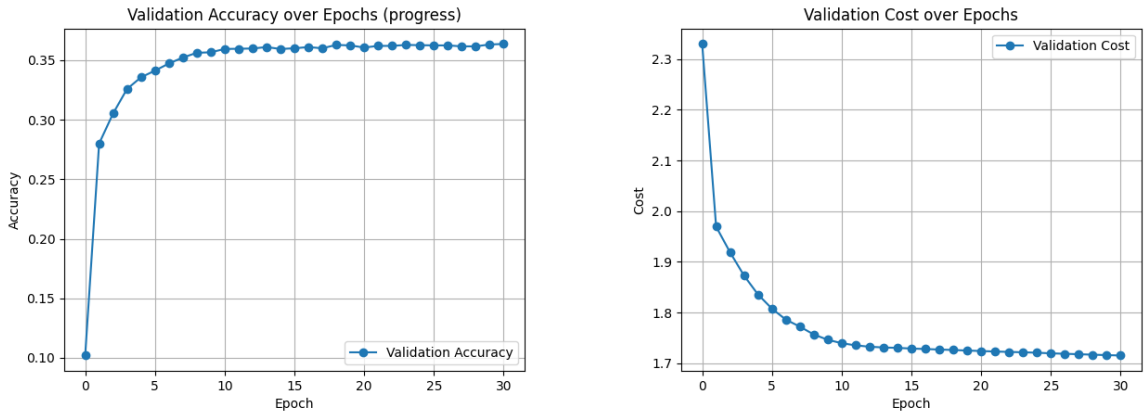

Abbildung 1: Links: Validierungsgenauigkeit (`progress`) über 30 Epochen.
Rechts: Validierungskosten (`costs`) über 30 Epochen.
Beide Metriken zeigen den Trainingsverlauf auf dem Validierungsdatensatz.

7) Vorhersage bzw Ergebnisse unserer Programme

Zum Schluss haben wir mit `nn_predict()` die Bewertungen der Validierungsdaten vorhergesagt. Die Methode führt einen Forward Pass durch und speichert sowohl die Softmax-Wahrscheinlichkeiten (Liste mit 10 Werten für jede Ratingklasse 1-10) als auch die gewählte Klasse (per `np.argmax()`) im DataFrame. Die Vorhersagen wurden dann als `val_predictions.csv` abgespeichert.
Im Folgenden werden wir diese Ausgaben analysieren.

Analyse des Ausgabelayers `A[-1]`:

Struktur der Ausgabewerte:

Beim Betrachten der Ausgaben ist uns aufgefallen, dass das Modell häufig nur die Ratingwerte 1 oder 10 vorhersagt. Diese beiden Klassen sind besonders dominant. Wahrscheinlich erkennt das Modell sehr klar, wenn eine Kritik besonders negativ oder besonders positiv ist. Dazwischen (also z.B Klassen wie 2 bis 9) wird seltener gewählt.

Die Ausgabewerte (die 10 Softmax-Wahrscheinlichkeiten) selbst sehen meist nicht wie perfekte Einheitsvektoren aus. In vielen Fällen ist die höchste Wahrscheinlichkeit zwar etwas höher als die anderen, aber nicht extrem dominant. Nur in wenigen Beispielen ist die höchste Komponente wirklich deutlich größer als der Rest.
Auffällig war aber auch, dass oft 3-4 Wahrscheinlichkeiten extrem klein sind im Vergleich zum Rest, was darauf hindeutet, dass das Modell diese Klassen für ein Beispiel komplett ausschließt.

Positive vs negative Beispiele:

Interessant war auch ein grober Vergleich zwischen positiven und negativen Kritiken: Bei positiven Beispielen wird häufig die Klasse 10 vorhergesagt oder zumindest eine Bewertung über 5. Bei negativen Beispielen kommt sehr oft die Klasse 1 vor, aber manchmal auch fälschlicherweise eine 10. Insgesamt wirkt das Modell bei negativen Beispielen etwas unsicherer.

Zusammenfassend lässt sich sagen, dass die Softmax-Ausgaben in `A[-1]` zwar teilweise in Richtung Einheitsvektoren gehen, aber bei weitem nicht perfekt eindeutig sind. Die Verteilungen (3-4 sehr kleine Wahrscheinlichkeiten und keine extrem dominante höchste Wahrscheinlichkeit) geben Hinweise auf die Unsicherheit des Modells.

Schwierigkeiten

Insgesamt sind wir beim Arbeiten am Projekt auf keine großen Schwierigkeiten gestoßen. Organisatorisch und zeitlich hat bei uns alles gut funktioniert. Wir haben nämlich ein gemeinsames Git-Repository benutzt und uns die Aufgaben (1-7) aufgeteilt: Einer/Eine hat jeweils eine Aufgabe bearbeitet, den Code gepusht und der/die andere hat den Teil dann überprüft und mit der nächsten Aufgabe weitergemacht. So konnten wir strukturiert arbeiten, ohne Zeitdruck.

Inhaltlich waren wir mit den meisten Themen gut vertraut, vor allem im Bereich neuronale Netze hatten wir schon Vorerfahrung. Allerdings haben wir anfangs Eingewöhnung bei der Arbeit mit Pandas und NumPy gebraucht, was ein paar Tage Zeit in Anspruch genommen hat.