MNIST-Projekt

Neuronales Netzwerk in Python

Lucien Deforge

2025

Inhaltsverzeichnis

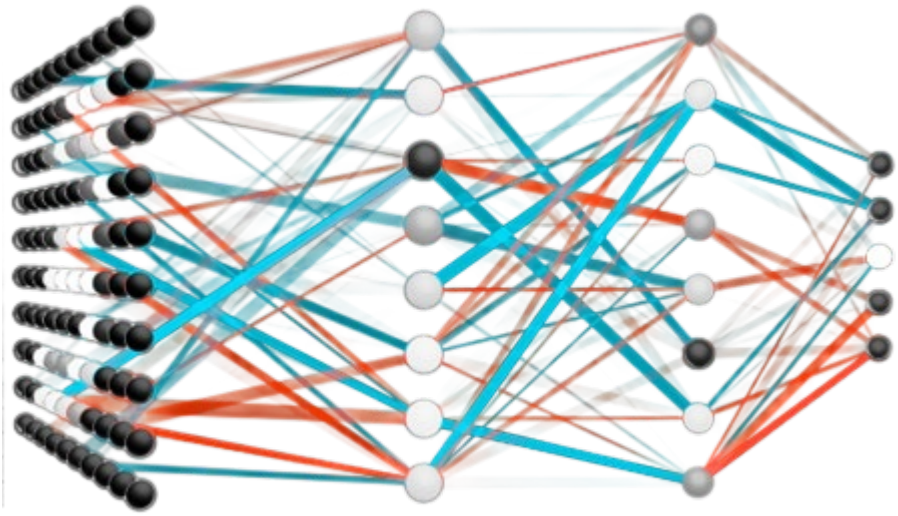
1 Einführung....3
2 Projektentwicklung....4
2.1 Planung....4
2.2 Setup....4
2.2.1 GitHub....4
2.2.2 Docker....4
3 Das Neuronale Netzwerk....5
3.1 Daten....5
3.2 Architektur und Aufbau....6
3.2.1 Layers....6
3.2.2 Gewichte (W) und Biasse (B)....7
3.3 Trainingsprozess....8
3.3.1 Forwardpass....8
3.3.2 Backwardpass....9
3.4 Programmstruktur....11
3.4.1 Daten....11
3.4.2 Netzwerk Aufbau....11
3.4.3 Training & Speicherung....11
3.4.4 Visualisierung & Prediction....12
4 Resultat und Evaluation....12
4.1 Erfolge....12
4.2 Misserfolge und Learnings....12
4.3 Nächste Features....13
5 Quellen....14

1 Einführung

Künstliche Intelligenz ist heute fast überall vorhanden, doch die wenigsten wissen, wie es funktioniert. Darum dachte ich mir, ich entwickle mein eigenes Neuronales Netzwerk. Dies ist eigentlich gut machbar, da es heute unzählige Libraries und Programme gibt, welche das Ganze für einen übernimmt. Aber dann wäre mein Ziel KI in der Tiefe zu verstehen nicht wirklich erreicht und darum ich wollte alles selbst programmieren und somit die Basis von künstlicher Intelligenz erlernen.

Nachdem ich meine Projektidee gefunden habe, musste ich erstmals viel neues Lernen. In meinen Recherchen kam immer wieder ein Name vor, nämlich MNIST. Es steht für Modified National Institute of Standards and Technology und ist ein weit verbreiteten Datensatz handgeschriebener Zahlen von Null bis Neun.

Der Zweck meines neuronalen Netzwerks war also diese Zahlen zu erkennen und dies so genau wie möglich. Auch wollte ich keine Libraries für Machine Learning oder Artificial Intelligence benutzen, welche die Arbeit für mich übernimmt.



Schöne Visualization eines Neuronales Netzwerk

2 Projektentwicklung

2.1 Planung

Bevor ich ans Programmieren ging, musst ich mich zuerst informieren. Da ich keine Ahnung hatte, wie neuronale Netzwerke funktionieren, habe ich einige YouTube Videos zu diesem Thema angeschaut. Die wichtigsten sind unten unter Quellen auffindbar. Danach habe ich mir genau überlegt, wie mein Projekt aussehen soll. Von Anfang an war mir klar, dass ich ein Modell von A bis Z entwickeln will, und dieses dann irgendwo benutzen. Es sollte etwas machen können, welches sonst normale Geräte nicht können. Schlussendlich sollte es interessant zu programmieren sein, sodass ich etwas lernen kann und Python besser beherrsche.

2.2 Setup

Danach musste ich mir überlegen, wie das ganze umsetzen sollte. Das ich Python benutzen wollte, war schon sehr schnell sicher, da es in der Entwicklung von künstlicher Intelligenz und Machine Learning sehr weitverbreitet ist. Aber ich habe keine Python Dateien benutzt, sondern es mit Jupyter Notebooks gemacht. Dies bringt den Vorteil Code Ausschnitte einzeln ausführen zu können. Ausserdem kann ich auch Markdown Zellen schreiben. Dies hilft, um alles zu organisieren und übersichtlicher zu machen. Als DIE habe ich VS-Code benutzt, da sie angenehm ist zum Benutzen ist und die einzelnen Extensions einfach zu verwalten sind. Ich hatte ich das Bedürfnis, auch zuhause daran zu arbeiten. Da ich einen leistungsstarken PC habe, wollte ich mein neuronales Netzwerk auch dort darauf trainieren. Darum habe ich eine Virtual Environment (venv) erstellt und darauf alle nötigen Libraries importierte. Somit habe ich die gleiche Entwicklungsumgebung auf all meinen Geräten.

2.2.1 GitHub

Das Ganze habe ich mit Hilfe von Git versioniert und auf GitHub publiziert. Dies bringt den Vorteil, dass ich von überall meinen aktuellen Stand benutzen kann und keine Versionsprobleme kriege. Das GitHub ist unten unter Quellen verlinkt.

2.2.2 Docker

Logo von Docker

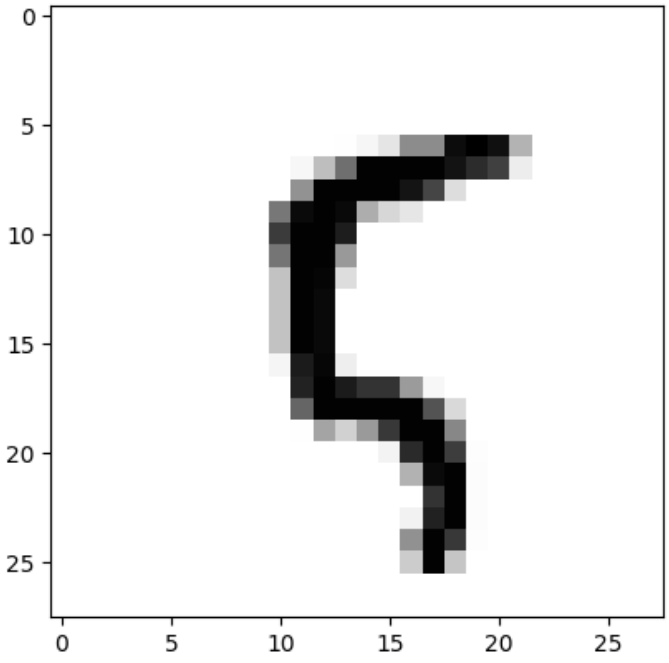
Zuerst habe ich ja alles mit meiner Venv und mein GitHub aufgesetzt. Doch die Venv's haben nach einer weile die Python Version automatisch aktualisiert, was einige Libraries nicht unterstützten und das ganze Programm nicht mehr funktionstüchtig machten. Deshalb entschied ich mich ein Docker Container zu schreiben und alles dort drauf laufen lassen. Doch die Docker Container waren mit total fremd. Ich musste einige male versuchen, bis es schlussendlich funktionierte. Dies ermöglichte mir die Venv's zu löschen und nur auf Docker Container umzusteigen.

3 Das Neuronale Netzwerk

3.1 Daten

Das Wichtigste bei einer künstlichen Intelligenz sind die Daten. Diese Daten dienen zum Training und ermöglichen, dass die künstliche Intelligenz lernen kann. Doch diese Daten können nicht einfach Fotos sein welche ich zum Beispiel bei einer Google Suche gefunden hätte. Hätte ich dies gemacht, wäre der Aufwand enorm. Das Problem ist eben, damit die künstliche Intelligenz lernen kann, muss sie auch wissen, ob sie falsch oder richtig ist. Dies werden normalerweise bei Datensätze als Label ausgewiesen. Aber dies manuell zu erarbeiten, würde Stunden dauern, insbesondere bei grosse Datensätzen. Darum habe ich einen schon vorgefertigten Datensatz genommen. Dieser hatte sechzigtausend (60 000) Trainingsbilder und zehntausend (10 000) Testbilder. Testbilder sind unteranderem wichtig, weil die künstliche Intelligenz nicht mit denen Trainiert worden ist und sie somit nicht kennt. Deshalb kann man recht genau Prognosen erstellen, wie genau ein Modell ist und wie es sich mit neuen Bildern verhält. Mein Datensatz stammt aus der MNIST-Datenbank. Diese wird vom National Institute of Standards and Technology betrieben.

Abbildung der Zahlen aus der MNIST Datenbank



So sieht das Netzwerk die Nummern

Diese Datenbank ist öffentlich zugänglich. Es sind alles Bilder von handgeschriebenen Ziffern. Alle Zahlen von 0-9 werden dort vertreten. Wobei jede Zahl als 28 x 28 Bild gespeichert ist. Ebenfalls wichtig ist, die Bilder sind alle als Graustufenbild vorhanden. Daher gibt es 256 verschiedene Graufarben. Doch nur mit Bildern kann mein Programm nicht viel anfangen, darum brauche ich die Datenbank als CSV. CSV steht für Comma Separated Value und ist ein Dateientyp, welcher die Bilder als einzelne Zeile speichert. Meine Daten sind jetzt in einer Datei mit sechzigtausend (60 000) Zeilen und 785 Nummer auf jeder Zeile. Die erste Nummer steht für das Label und ist daher die Zahl, welche das Bild repräsentieren soll. Danach kommen 784 (28 x 28) Zahlen und die stehen für jedes Pixel in einem Bild. Je höher die Zahl dann ist, je dunkler ist dann dieser Pixel. Doch das Konvertieren von einem Bild zu

dieses CSV ist zeitaufwendig und ich wollte eine schnellere Lösung. Ein GitHub Repository, hatte schon dieses CSV in dem Format, welches ich brauche. Es ist unter Quellen verlinkt. Mit diesem Schritt war dann die Recherche nach den Daten fertig.

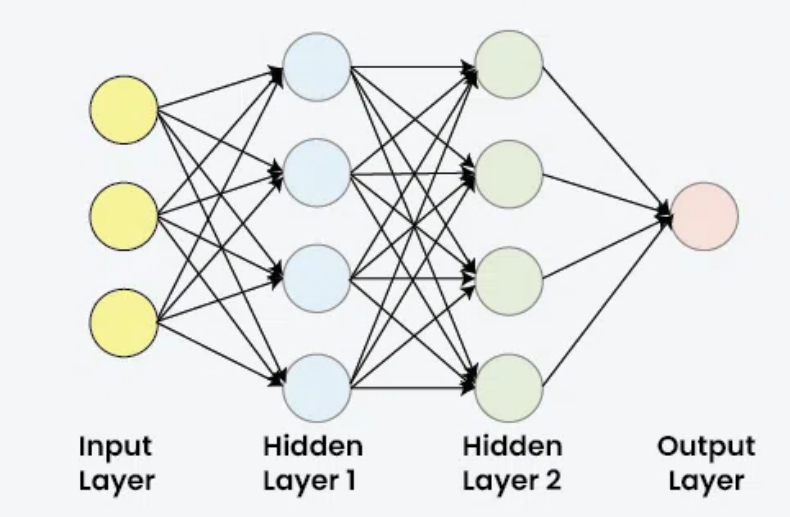
3.2 Architektur und Aufbau

3.2.1 Layers

Die Architektur eines künstlichen Neuronales Netzwerk ist dem menschlichen Gehirn nachempfunden und bestehen aus Schichten. Auf diesen Schichten sind dann viel Neuronen gestapelt. Diese Neuronen nehmen Informationen aus der vorhereigenen Schicht verarbeiten sie und stellen es der nächsten Schicht zur Verfügung. Ein Netzwerk besteht aus drei Schichtentypen:

```
1  sizes = [784, 128, 64, 10]
```

Die Anzahl Neuronen pro Schicht in meinem Neuronalem Netzwerk



Theoretische Abbildung einen «dense Neuronal Network» Grösse entspricht aber nicht die meines Netzwerkes. Dies wäre ein 3-4-4-1 Netzwerk

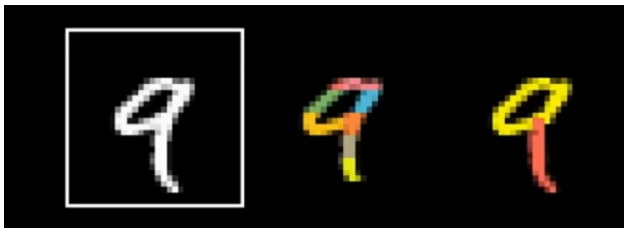
Quelle: GeeksForGeeks

Die Input Layer

Die Input Layer ist der erste Layer. Dort werden die Rohdaten reingeladen. Bei meinem Projekt sind es 784 Neuronen, welche jedes ein Pixel darstellen soll. Dort wird nichts gerechnet oder verarbeitet. Das heisst, dass nicht alle Daten reingeladen werden, sondern nur ein Bild pro Runde. Jeder dieser Neuronen speichert einen Wert, dieser wird als Aktivierung bezeichnet. Bevor die Daten ins Netzwerk kommen werden sie normalisiert, bei meinem Programm? auf einen Wert zwischen 0 und 1.

Die Hidden Layers

Die Hidden Layers liegen zwischen der Input Layer und der Output Layer. Dort passiert eigentlich das Ganze. Dort findet die Transformation der Inputs in neue Formen und Muster statt. Hidden Layers haben nicht, wie die Input Layer und die Output Layer, mehrere Schichten. Ich habe nur zwei Schichten, da mein Neuronales Netzwerk Einfach bleiben soll. Die erste Schicht hat 128 Neuronen und die zweite 64. Die Neuronen lernen dort Muster und Merkmale zu erkennen, welche für die endgültige Erkennung relevant sind. Frühere Schichten erfassen einfache Merkmale wie Striche oder Kurven. Spätere Schichten erfassen dann ganze Formen wie Kreise, Kreuze oder Vierecke. Zum Beispiel bei einer 9 können zuerst kleine Formen gebildet werden, welche nachher den Kreis und den Strich bilden und zusammen eine Neun sind. Doch der Kreis könnte auch zu einer Acht gehören und das Neuronale Netzwerk arbeitet sich so mit Formen durch. Doch man kann sich nicht sicher zu sein, wie das Netzwerk eigentlich wirklich seine Formen zieht oder ob er sie überhaupt macht. Es ist sehr schwierig es zu wissen und man soll es eher als «Black-Box» betrachten und sich es wie beschrieben vorstellen.



Dieses Beispiel aus diesem YT Video 7:38 zeigt gut wie man es sich vorstellen kann

Die Output Layer

Die Output Layer ist der letzte Layer und somit das Ergebnis des Netzwerks. Die Anzahl der Neuronen in dieser Schicht richtet sich nach der Anzahl möglichen Outputs. Also sind es bei der MNIST Datenbank, welche die Zahlen von Null bis Neun als Output haben können, Zehn Neuronen. Die Aktivierung dieser Neuronen beschreibt dann die Wahrscheinlichkeit, dass das Input Bild dieser Zahl entspricht.

3.2.2 Gewichte (W) und Biasse (B)

Ein Netzwerk ist eigentlich eine Formel mit zig Tausend Parameter. Bei mir wären es *109 386*, welche man ändern kann, um den Ausgang zu ändern. Dies ist sehr viel und kann unmöglich von Hand gemacht werden. Diese Parameter sind eigentlich der Grund, warum das Neuronale Netzwerke überhaupt lernen können.

3.2.2.1 Die Gewichte (W)

Die Gewichte sind die Verbindungsstärken zwischen den Neuronen der verschiedenen Schichten. Sie sind einfach Zahlen, die jeder Verbindung zugeordnet sind. Sie bestimmen, wie stark der Input (Neuron der vorherigen Schicht) eines Neurons die Aktivierung des nächsten beeinflusst. Gewichte können positiv oder negativ sein. Positive Gewichte sagen, dass der Input wichtig ist und der die Aktivierung stark beeinflussen soll.

```
1  # Hidden Layer 2 to Output Layer
2  Z3 = params['weight3'] @ A2 + params['bias3']
```

$Z3$ ist der Aktivierungsvektor der Output Layer und $params[´weight3´]$ ist die Gewichtsmatrix der Layer 2 zu Output Layer, dieser wird $A2$ (Aktivierungsvektor von Hidden Layer 2)

Bei Negativen ist es natürlich das Gegenteil und schwächen den Einfluss dieses Neurons auf den nächsten. Gewichte sind das, was das Netzwerk dazu bringt, Muster zu erkennen. Wenn das Netzwerk eine '2' Identifizieren muss, wird das 7-Neuron starke positive Gewichte für die Pixel haben, welche in der Regel eine 7 bilden.

Die Gewichte sind in einer Matrix gespeichert. Jede Zeile dieser Matrix entspricht einem Neuron der aktuellen Schicht, jede Spalte einem Neuron der vorherigen Schicht. Für den dritten Layer ergibt sich so eine 16×128-Matrix. Diese Matrix wird mit dem Aktivierungsvektor der vorherigen Schicht multipliziert, um den Aktivierungsvektor der aktuellen Schicht zu berechnen.

3.2.2.2 Die Biasse (B)

Der Bias ist ein zusätzlicher, konstanter Wert, welcher vor der Aktivierung noch hinzugerechnet wird. Er ist eine Art Schwelle für die Aktivierung. Der Bias ist enorm wichtig, da es dem Neuron ermöglicht seinen Schwellwert anzupassen. Ohne den Bias wäre es unmöglich, dass ein Neuron aktiviert wird, wenn die alle seine Inputs Null sind. Die Baisse werden als Vektor zusammengefasst und als diesem zu der Matrixmultiplikation von vorher addiert.

3.3 Trainingsprozess

Das Training eines Modells ist eigentlich nichts als anders alle Gewichte und Biasse so anzupassen, sodass die mathematische Formel möglichst korrekt ist. Es werden also an *109 386* Zahlen rumgeschraubt und dies geschieht in Phasen.

3.3.1 Forwardpass

Hier wird die Vorhersage getroffen. Dieser ändert noch nichts an den Gewichten und Biasse. Aber er rechnet ein Resultat basierend auf den Input (X). Er jagt diesen Input Schritt für Schritt durch alle Schichten, bis am Ende den Output kommt. Dieser ist nämlich in meinem Beispiel $A^{[3]}$. Das ist die Denkphase des Neuronales Netzwerk.

In jeder Schicht L wird zuerst die Pre-Aktivierung oder die lineare Transformation berechnet oder in meinem Code $Z^{[L]}$. Diese wird mathematisch so dargestellt:

$$Z^{[L]} = W^{[L]} \cdot A^{[L-1]} + B^{[L]}$$

Dabei ist $A^{[L-1]}$ der Aktivierungsfaktor der vorherigen Schicht. $W^{[L]}$ist die Gewichtsmatrix der aktuellen Schicht. $B^{[L]}$sind der Bias-Vektor der aktuellen Schicht. Wie schon erklärt bei den Gewichten, besteht die Gewichtsmatrix die Verbindungsstärken zu allen Neuronen der vorherigen Schicht. Durch eine Matrixmultiplikation wird dies mit dem Aktivierungsvektor verrechnet und schlussendlich wird der Bias-Vektor addiert. Direkt nach dieser linearen Transformation wird auf jedes Element des Vektors $Z^{[L]}$ eine nicht-lineare Aktivierungsfunktion angewendet. In meinem Netzwerk ist das die Sigmoid Formel, es gibt auch andere Möglichkeiten dies umzusetzen, diese transformiert jeden Wert in den Bereich von 0-1.

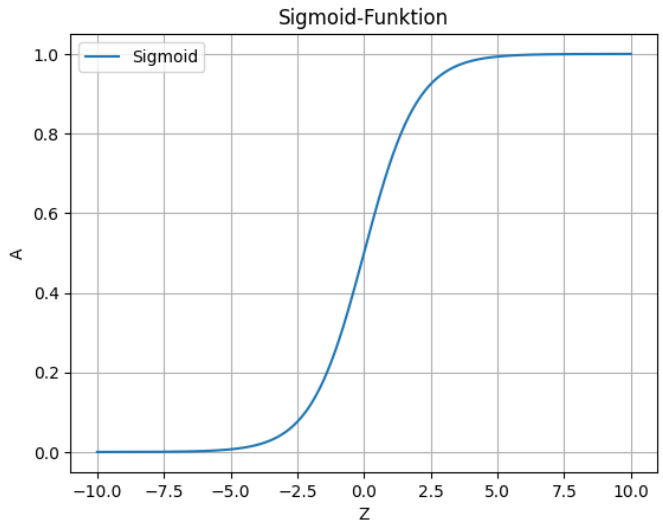
$$A^{[l]} = \sigma(Z^{[l]}) = \frac{1}{1+e^{-Z^{[l]}}}$$

```python
def sigmoid(self, x, derivatative=False):
        sig = 1/(1 + torch.exp(-x))

        if derivatative:
            return(sig * (1-sig))
        return(sig)
```

Implementierung der Sigmoid-Funktion mit Python und PyTorch

Diese Sigmoid-Funktion erfüllt mehrere wichtige Zwecke: Sie führt dazu, dass das Netzwerk nichtlinear ist, sodass es besser komplizierte Muster erkennen kann. Da sie auch Muster erkennen kann, welche über einfache Lineare Zusammenhänge hinausgehen. Ausserdem kann man somit einfacher eine Wahrscheinlichkeit daraus interpretieren, was bei Zahlen praktisch sein kann.

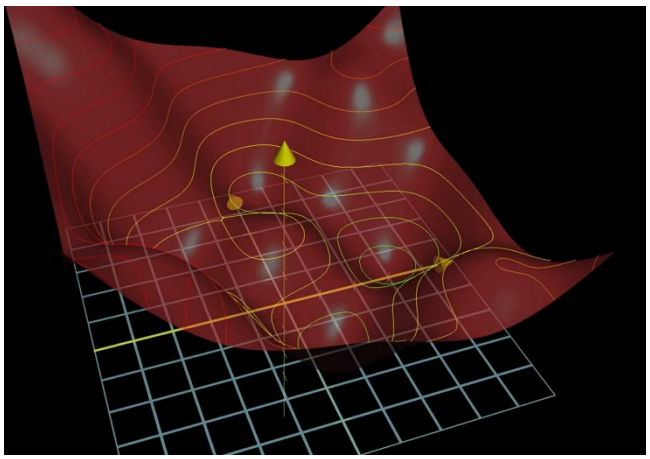


Um die Sigmoid-Aktivierung besser zu visualisieren habe ich einen Plot mit MatplotLib gemacht, welcher die Funktion schön verständlich darstellt. Z ist die Pre-Aktivierung und A ist aktuelle Aktivierung nach der Sigmoid-Funktion. Diese Bild zeigt gut, wie die Werte von $Z^{[L]}$ in den Bereichen von Null und Eins transformiert. Die grossen Z-Werte werden fast zu Null und Null ist bei 0.5.

3.3.2 Backwardpass

Nachdem im Forwardpass die Aktivierungen $A3$ berechnet wurden, kommt der Backwardpass ins Spiel. Er macht eigentlich das ein Neuronales Netzwerk lernt. Nämlich für das Anpassen der Gewichte und Biasse, dass der Fehler zwischen der Vorhersage des Forwardpass und die Tatsächliche Lösung, das Label Y (siehe Daten), minimiert wird. Ziel ist es, die Kostenfunktion $\boldsymbol{L}(\boldsymbol{A}^{[\boldsymbol{L}]}, \boldsymbol{Y})$zu minimieren.

Man sich das als Berg aus Fehlerpunkten vorstellen und das Ziel ins Tal, zu gelangen. Also den Fehlerrate so klein wie möglich zu halten.



Hier kann man sich gut den Berg vorstellen und das Ziel ins Tal zu kommen

Quelle: Hier bei 8:17

Zuerst wird der Fehler berechnet. Also wo er sich auf dem Berg befindet. Für das Output an der letzten Schicht wird folgendes angewendet: $dA^{[L]} = A^{[L]} - Y$. Dabei ist $dA^{[3]}$der Gradient der Kostenfunktion bezüglich der Aktivierung der letzten Schicht. Also wie stark die Aussage vom richtigen Wert abweicht.

Der Fehler wird nun Schicht für Schicht zurückgeschickt. Für jede Schicht: $dZ^{[l]} = dA^{[l]} \odot \sigma'(Z^{[l]})$. Das $\odot$ Zeichen steht elementweise Multiplikation, ist eigentlich nichts weiters, als dass das 1x1 der Matrix dA mit dem 1x1 von $Z^{[L]}$multipliziert wird. Jetzt haben wir den Gradienten dZ. Mit diesem Wissen wir jetzt, wie jedes Neuron den Output beeinflusst.

Jetzt können wir endlich die Gewichte und Biasse anpassen. Zuerst berechnen wird den Bias-Gradient. $\mathbf{dB^{[l]}} = \frac{1}{m}\sum_{i=1}^{m} \mathbf{dZ^{[l](i)}}$ Mit dieser Formel können wir den dB berechnen. Das m steht für Batchgrösse und wird benutzt in mehreren Bildern gleichzeitig zu verarbeiten. Wir nehmen also den Durchschnitt ein Batch (Default 32 Bilder). Der Gradient sagt uns jetzt in welcher Richtung den Bias zu drehen, also bisschen rauf oder ein runter. Nach den Biasse kommen die Gewichte. Da wird folgende Formel benutzt: $\mathbf{dW^{[l]}} = \frac{1}{\mathbf{m}}\mathbf{dZ^{[l]}} \cdot \left(\mathbf{A^{[l-1]}}\right)^{\mathsf{T}}$. Hier wird der Gewicht-Gradient berechnet, also Matrixmultiplikation von dem Neuronen-Gradient und der Aktivierung der vorherigen Schicht. Das T in der Formel steht für Transposition und vertauscht die Zeilen und Spalten einer Matrix. In meinen Code sieht das so aus:

```
error  = params['weight3'].T @ error * self.sigmoid(params['Z2'], derivatative=True)
change_weight['weight2'] = (error @ params['A1'].T) / batch_size
```

Der Fehlervektor der zweiten Hidden Layer wird berechnet, indem der Fehler der Output Layer mit der transponierten Gewichtsmatrix $weight3.\mathsf{T}$ multipliziert wird. Danach wird dieser Wert elementweise mit $\sigma(Z2)$ multipliziert und schliesslich mit den Aktivierungen $A1.T$ der ersten Hidden Layer kombiniert, um den Gradienten $change\_weight['weight2']$ für die Gewichte zu erhalten.

Schlussendlich muss das Ganze noch angepasst werden. Die Gradienten sagen jetzt einem schön in welche Richtung man anpassen muss, doch man weiss nicht um wie viel.

$$W^{[l]} := W^{[l]} - \eta \, dW^{[l]}$$

$$B^{[l]} := B^{[l]} - \eta \, dB^{[l]}$$

Da kommt die Learningrate (n) ins Spiel. Das ist die Schrittgrösse beim Anpassen der Gewichte und Biasse. Diese wird nicht berechnet, sondern beim Trainieren des Modells gesetzt. Zum Beispiel kann man drei Batchs mit 0.1 machen und dann nochmal drei mit 0.01 und für den Feinschliff zwei mit 0.001.

3.4 Programmstruktur

Mein gesamtes Projekt ist in einem einzigen Jupyter Notebook. Ich habe das Notebook extra in logische Abschnitte unterteilt, damit alles übersichtlich bleibt und man die einzelnen Schritte einzeln ausführen kann.

3.4.1 Daten

Zuerst werden die CSV-Dateien mnist_train.csv und mnist_test.csv geladen. Danach werden die Daten vorbereitet, also die Pixelwerte normalisiert, in torch.Tensor umgewandelt und für das Training bereitgestellt. Ich habe auch eine einzelne Zahl geplottet, zu sehen, ob es funktioniert.

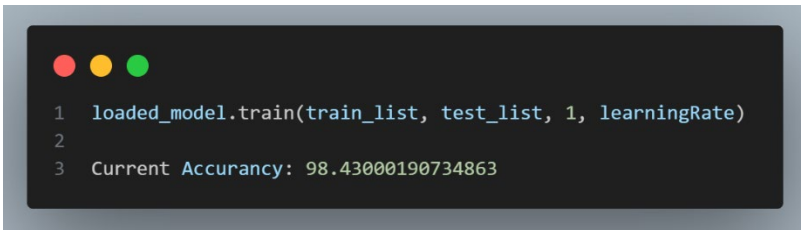
3.4.2 Netzwerk Aufbau

Das neuronale Netzwerk ist in der Klasse DenseNeuronalNetwork implementiert, bestehend aus zwei Hidden Layers und einer Output Layer. Gewichte und Biasse werden als torch.Tensor gespeichert. Es gibt folgende Methoden

- sigmoid()
- forward_pass()
- backward_pass()
- update_weights()
- accuracy()
- train()

3.4.3 Training & Speicherung

Das Netzwerk wird erstellt und anschliessend trainiert. Fertige Modelle speichere ich mit pickle, sodass ich Training unterbrechen und später fortsetzen oder Vorhersagen machen kann.



3.4.4 Visualisierung & Prediction

Für die Vorhersage eines Bildes wird es mit PIL.Image geladen, auf 28x28 skaliert und in Graustufen konvertiert. Mit tensor_plot() lassen sich die Bilder anzeigen, und predict_img() liefert die Vorhersage des Modells.

4 Resultat und Evaluation

4.1 Erfolge

Der grösste Erfolg dieses Projekts ist das erfolgreiche Programmieren eines eigenen neuronalen Netzwerks. Das Modell erreicht eine Genauigkeit von über 98%, was zeigt, dass sowohl der Forwardpass als auch der Backwardpass perfekt funktionieren. Die Kostenfunktion sinkt während des Trainings zuverlässig, und auch die selbst implementierte Gradienten-Berechnung funktioniert einwandfrei und ist optimal mit dem Batch-Processing verbunden.

Logo von PyTorch

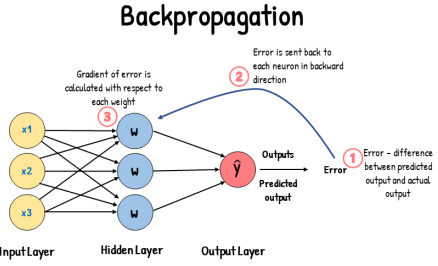
Besonders stolz bin ich darauf, dass ich das gesamte Netzwerk ohne grosse Machine-Learning-Libraries wie TensorFlow oder Keras programmiert habe. Das gesamte System könnte praktisch vollständig auf NumPy laufen, das heisst jede wesentliche Funktion wurde eigenhändig implementiert. Ich habe nur PyTorch wegen der Performanec auf Nvidia-GPU genommen.

Ein weiterer sehr grosser Erfolg ist der Lernfaktor dieses Projekts. Künstliche Intelligenz war für mich vorher etwas kompliziertes (ist es immer noch), etwas, das viele Menschen nur oberflächlich verstehen. Durch dieses Projekt habe ich die Grundlagen des Machine Learning und neuronaler Netzwerke wirklich verstanden. Heute kann ich erklären, wie ein neuronales Netzwerk aufgebaut ist, wie es lernt und wie Forward- und Backpropagation funktionieren sollten und einigermassen die Mathematik dahinter.

Ebenfalls schwierig war der Mathe-Teil dieses Projekt. Ich bin zwar gut in Mathe aber Matrizen, Vektoren, Gradienten und Aktivierungsfunktionen wie die Sigmoid-Funktion hatten wir in der Schule noch gar nicht. Deshalb musste ich mir dieses Wissenselber beibringen, um das gesamte Konzept zu verstehen und korrekt umsetzen zu können.

4.2 Misserfolge und Learnings

Trotz der vielen Erfolge gab es während des Projekts auch einige Misserfolge. Die grösste Herausforderung war eindeutig die Implementierung der Backpropagation. Besonders das korrekte Ableiten der Formeln, die passenden Matrix-Dimensionen und das richtige Anwenden von Transponierungen führten zu mehreren Fehlern im Training. Oft sah es so aus, als ob das Netzwerk funktionierte,



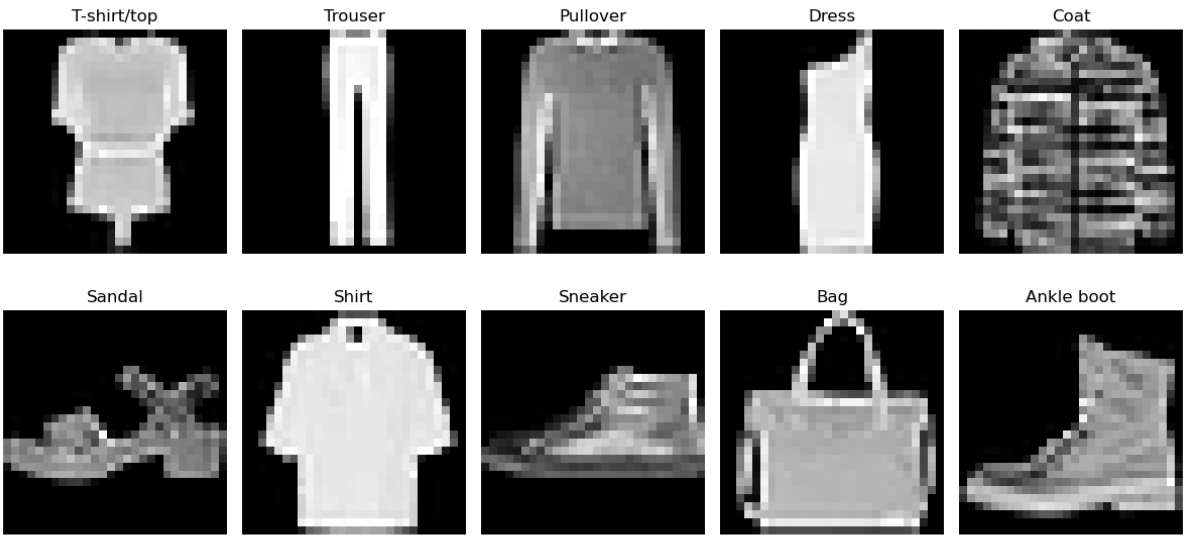
Erklärungs-Bild für Backpropagation

doch er auf einmal immer schlechter wurde, weil ich z.B. eine Reihenfolge einer Formel ausserversehen umstellte und dies das Ganze in die Luft jagte.

Ein weiterer deutlich Misserfolg war die Umstellung des gesamten Codes von NumPy auf PyTorch. Anfangs wurde alles mit NumPy programmiert, und alle Berechnungen funktionierten dort korrekt. Als ich jedoch auf PyTorch wechselte, um GPU-Unterstützung zu ermöglichen und um das Training um einiges schneller zu machen, entstanden plötzlich grosse Fehler. Manche Matrixoperationen liefen anders als erwartet, Shapes passten nicht mehr zusammen, Broadcasting verhielt sich anders, und einzelne Funktionen wie @ oder .T führten zu völlig anderen Ergebnissen als zuvor. Dadurch funktionierten sowohl Forward- als auch Backwardpass lange überhaupt nicht mehr und ich musste viele Teile von Grund auf neu aufbauen, debuggen und testen.

4.3 Nächste Features

Die MNIST Datenbank gibt es auch als Fashionvariante. Nämlich die Fashion-MNIST. Sie ist ähnlich aufgebaut und hat die gleiche Input-Grösse nur sind es Handtaschen und Kleidungsstücke. Es nahm mich wunder, was passiert, wenn man diese Datenbank als Input benutzt. Es hat super funktioniert und ich habe sehr schnell das Modell auf über 95% trainiert, ohne den Code zu umschreiben.



Das ist die Fashion-MNIST. Sie ist ebenfalls in Grayscale und ist 28x28 Pixel gross. Doch es geht um Kleider.

Auch will ich mit verschiedene Netzwerk Grössen rumprobieren. Ich denke, es wäre sehr spannend zu wissen, wie das Modell sich verhält und wie sich die Genauigkeit nach X Epochen entwickelt.

5 Quellen

- GitHub meines Projekts: Sourcecode des MNIST-Projekts
- MNIST-Daten Download: GitHub von Phoebetronic
- Fashion-MNIST Download: GitHub von dmakariev
- NIST: NIST
- Tutorial von ML for Nerds: Building a Neural Network from scratch
- Playlist von 3Blue1Brown: Neural Networks
- NotebookLM welches ich selbst gemacht habe: Neuronale Netze: Vom Pixel zur Zahl
- ChatGPT beim wechsel von NumPy auf PyTorch: ChatGPT