Backpropagation Lab

(BaPL)

Beschreibung und Bedienungsanleitung

Martin Reiche, Tübingen, Januar 2020

Inhaltverzeichnis

[1 Einführung 3](#_Toc31401807)

[2 BaPL Dateiformat 3](#_Toc31401808)

[3 Bedienung 4](#_Toc31401809)

[4 Einschränkungen 6](#_Toc31401810)

[5 Mathematische Ableitung und Programmierung mit NumPy 7](#_Toc31401811)

[5.1 Vorwärts-Propagation 7](#_Toc31401812)

[5.1.1 Mathematische Ableitung 7](#_Toc31401813)

[5.1.2 Programmierung 8](#_Toc31401814)

[5.2 Rückwärts-Propagation 8](#_Toc31401815)

[5.2.1 Mathematische Ableitung 8](#_Toc31401816)

[5.2.2 Programmierung 10](#_Toc31401817)

[6 Performance 11](#_Toc31401818)

[6.1 Trefferquote auf dem MNIST-Datensatz 11](#_Toc31401819)

[6.2 Fehlerhafte Erkennungen 11](#_Toc31401820)

[6.2.1 Fehlleistung aufgrund undeutlicher Schrift 11](#_Toc31401821)

[6.2.2 Unerklärliche Fehler 12](#_Toc31401822)

[6.2.3 „Erkennung“ aufgrund verstreuter Pixel 12](#_Toc31401823)

[7 MNIST 13](#_Toc31401824)

[7.1 Dateien 13](#_Toc31401825)

[7.2 Datenformat 13](#_Toc31401826)

[8 BaPL Programmaufbau 13](#_Toc31401827)

[9 Mögliche Verbesserungen / Erweiterungen 14](#_Toc31401828)

[10 Quellen 14](#_Toc31401829)

# Einführung

Zum Einstieg sollte man die Seite <http://www.martin-reiche.de/backpropagation-lab.html> studieren!

Eine gute Einführung in die fachliche Aufgabenstellung findet man z.B. in Tariq Rashids Buch "[Make Your Own Neural Network](http://makeyourownneuralnetwork.blogspot.com/)".

BaPL simuliert ein Neuronales Netz mit 3 Schichten (layers):

* Eingabeschicht (Input Layer)
* Zwischenschicht (Hidden Layer, optional)
* Ausgabeschicht (Output Layer)

Die MNIST-Daten im csv-Format habe ich von (bzw. über) <http://makeyourownneuralnetwork.blogspot.com/2015/03/the-mnist-dataset-of-handwitten-digits.html> heruntergeladen.

# BaPL Dateiformat

BaPL erwartet von Dateien mit den Trainings- und Testmustern ein bestimmtes Format, welches u.a. durch seine Kopfzeilen bestimmt ist. Die dort enthaltenen Parameter werden von BaPL übernommen und zum Teil auch angezeigt.

In der ersten Zeile werden Key-Value-Paare durch Leerzeichen getrennt, z.B.

application=BackPropLab version=0 type=data\_gray

Folgende Schlüssel / Wert-Paare werden benutzt:

|  |  |  |
| --- | --- | --- |
| **Schlüssel** | **Mögliche Werte** | **Bedeutung** |
| application | BackPropLab | Kennzeichnung der Applikation bzw. Version |
| version | 0… | Formatversion |
| type | 1) data\_bw  2) data\_gray | 1) Datei enthält schwarz-weiß-Daten: „-“ = weiß, „x“ = schwarz  2) Datei enthält Grauwerte von 0 = weiß bis 255 = schwarz |
| input\_rows | Natürliche Zahl | Anzahl der Zeilen pro Bild (Höhe) |
| input\_columns | Natürliche Zahl | Anzahl der Spalten pro Bild (Breite) |

In der zweiten Zeile werden die Muster (patterns) benannt, die in der Sammlung auftauchen können. Bei MNIST sind dies

patterns: 0 1 2 3 4 5 6 7 8 9

Sollen Buchstaben erkannt werden, könnte dort stehen

patterns: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Ab der dritten Zeile beginnen die eigentlichen Muster. Sie bestehen aus input\_rows + 1 Zeilen, wobei die erste Zeile das gezeigte Pattern benennt, bei MNIST also eine Ziffer 0-9.

Die folgenden input\_rows zu enthalten input\_columns die Pixel-Werte, durch Leerzeichen getrennt.

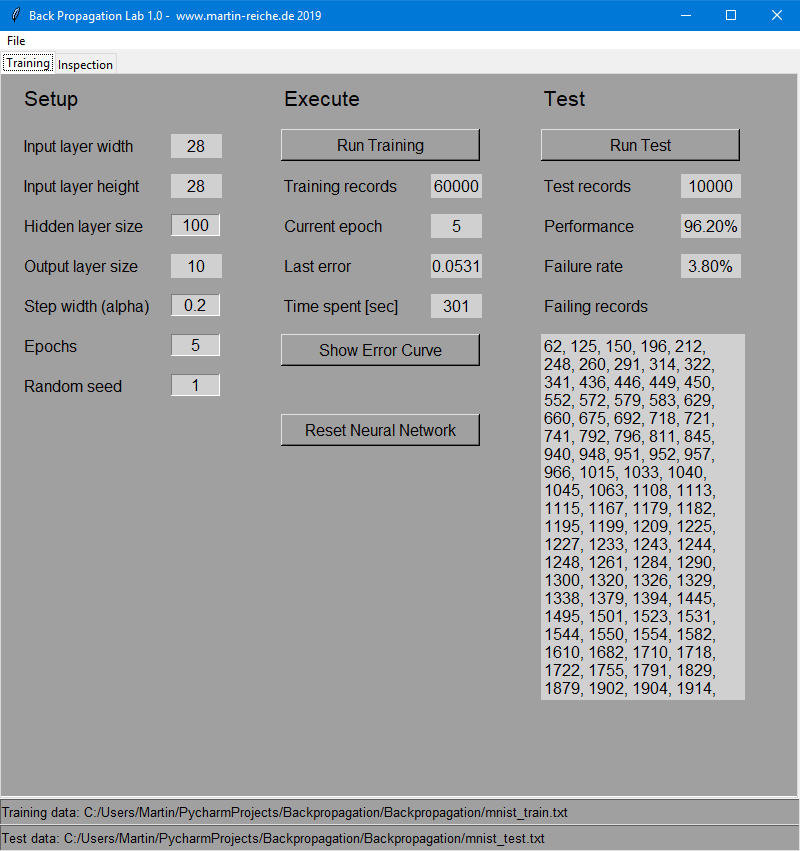
Folgende globale Variablen werden aus der Datei der Trainingsdaten gesetzt:

|  |  |  |
| --- | --- | --- |
| **Variable** | **Bedeutung** | **Kommentar** |
| g.numRows | Anzahl der Zeilen des Musters |  |
| g.numCols | Anzahl der Spalten des Musters |  |
| g.outputLayerSize | Anzahl der Neuronen in der Ausgabeschicht |  |
| g. allTrainingPattern | Arrray der Trainingsmuster | Daraus abgeleitet wird die Zahl „Training Records“ angezeigt. |

# Bedienung

Achtung: BaPL fängt nicht alle Benutzerfehler komfortabel ab. Daher sollte man bei Problemen BaPL erneut starten und sich sorgfältig vorarbeiten.

Das Training wickelt man auf der Registerkarte „Training“ ab, die beim Arbeiten mit MNIST-Daten z.B. so aussieht:



Ein Programmablauf mit BaPL gestaltet sich wie folgt:

1. Auswahl eines Trainingsmusters durch Öffnen der Datei im „File“ Menü. Achtung: Bei MNIST-Daten kann dies dauern!
2. Den gewählten Dateipfad kann man jederzeit am unteren Fensterrand nachlesen.
3. Die Anzahl der gelesenen Muster wird neben „Training records“ angezeigt.
4. Auch die Felder „Input layer width“, „Input layer height“ und „Output layer size“ füllen sich beim einlesen. „Output layer size“ bezeichnet die Anzahl der Neuronen in der Ausgabeschicht.
5. Nun trägt man die Anzahl der Neuronen in der Zwischenschicht „Hidden Layer Size“ ein. Setzt man den Wert auf Null, wird gar keine Zwischenschicht eingezogen. Die Ausgabeschicht kommuniziert also direkt mit der Eingangsschicht.
6. Mit der Schrittweite „alpha“ legt man fest, wie schnell der Gradienten-Abstieg erfolgen soll. Diese Zahl muss man experimentell ermitteln.
7. „Epochs“ legt fest, wie oft die Trainingsmuster dem Netzwerk zum Lernen vorgelegt werden sollen.
8. „Random seed“ hat Einfluss darauf, mit welchen Zufallswerten die Gewichtsmatrizen und Bias-Werte vor Beginn des Trainings belegt werden. Prinzipiell sind hier alle Startwerte gleichwertig, doch stellen sich unterschiedliche Lernverläufe ein.
9. Dann drückt man „Run Training“, um das Training zu starten. Je nach gewählten Startwerten und geladenen Mustern kann dies dauern und das Programm ist u.U. für eine Zeit nicht bedienbar. (Auf meinem Rechner dauert eine MNIST-Epoche mit 60.000 Mustern etwa eine Minute.)
10. Den Verlauf der Fehlerkurve kann man mit „Show Error Curve“ aufrufen. Backpropagation arbeitet darauf hin, diesen Fehler zu minimieren.
11. Will man das Training auf denselben Trainingsdaten mit anderen Parametern wiederholen, muss man vorher auf den Vergessensknopf „Reset Neural Network“ klicken.
12. Um den Trainingserfolg zu prüfen, lädt man im dem „File“-Menü „Load test data from file“.
13. Dann klickt man „Run Test“ und erhält eine statistische Aussage über den Anteil erfolgreicher (Performance) und fehlgeschlagener (Failure rate) Klassifizierungen. Bei welchen Mustern das Netzwerk versagt hat, wird unter „Failing records“ verbucht. Mit diesen Zahlen kann man dann auf der Inspection-Page das entsprechende Muster aufrufen und so und die Reaktion des Netzwerkes anschauen.
14. Wobei wir schon auf der Inspection-Seite bzw. Registerkarte angekommen sind, welche uns Einblicke in das Verhalten des trainierten Netzwerkes bietet:
    1. Auf der linken Seite können wir einzelne Muster („Test Record“) per Spin-Button oder direkt durch Angabe der laufenden Nummer auswählen. Schaltet man auf „Manual“ um, kann man per Mausklick auf die Felder des Musters jedes einzeln auf weiß oder Schwarz setzen. So kann man erforschen, wie das Netzwerk auf bislang unbekannte Muster reagiert. Mit „Clear“ löscht man (nur für diesen Versuch) das Testmuster und kann dann ein eigenes zeichnen (mit gedrückter Maustaste ziehen).
    2. In der mittleren Spalte werden die Erregungen der Neuronen in der Zwischenschicht durch waagerechte Balken dargestellt, maximale Länge -> Erregung = 1
    3. Auf der rechten Seite findet sich eine entsprechende Darstellung für die Neuronen der Ausgangsschicht. Deren Benennung entstammt der Datei mit den Testmustern. Unten wird noch das (laut Trainingsdatei) erwartete („Expected“) Muster und das tatsächlich erkannte („Detected“) Muster dargestellt. Bei Ungleichheit erscheint der Hintergrund bei „Detected“ in gelber Farbe.

# Einschränkungen

In BaPL ergeben sich (nur aufgrund des Layouts im Fenster) folgende Einschränkungen:

|  |  |
| --- | --- |
| maximale Zeilen- und Spaltenzahl im Input-Layer | ca. 28 |
| Maximale Anzahl der Neuronen im Hidden Layer | ca. 200 |
| Maximale Anzahl der Neuronen im Output Layer | ca. 26 |

# Mathematische Ableitung und Programmierung mit NumPy

Für eine Einführung in künstliche neuronale Netze siehe z.B. [diesen](https://de.wikipedia.org/wiki/K%C3%BCnstliches_neuronales_Netz)[[1]](#footnote-1) Wikipedia-Artikel.

## Vorwärts-Propagation

### Mathematische Ableitung

Unter Vorwärts-Propagation wollen wir die normale Funktion eines neuronalen Netzes verstehen: Die Weiterleitung einer Erregung durch die Schichten.

Für die folgende Betrachtung reicht ein Netzwerke mit jeweils 2 Neuronen pro Schicht; eine Verallgemeinerung auf n Neuronen fällt danach leicht.

Wir wählen eine Indizierung der Neuronen und Synapsengewichte, welche uns die Übertragung auf die Berechnung mit NumPy erleichtert. Dort wollen wir mit Matrizen arbeiten, die Elemente also 2 Indizes haben. Gemäß Konvention bezieht sich der erste Index auf die Zeile (row) und der zweite auf die Spalte (column):

Hidden Layer

Output Layer

h1

j = 2

k = 2

o‘1

j = 1

j = J

k = 1

k = K

…

…

o‘2

h2

w11

w22

w12

w21

h1

h2

w12

w22

w11

w21

o‘ = W · h + b

b1

b2

+

Eine Matrixmultiplikation der Gewichtsmatrix W (weight) mit dem Eingangsvektor der Ausgabeschicht ergibt die Erregung der Output-Neuronen gemäß obiger Formel d.h.

o'1 = w11 · h1 + w12 · h2 + b1

o‘2 = w21 · h1 + w22 · h2 + b2

### Programmierung

Die Neuronen einer Schicht sind in BaPL als Exemplare der Klasse *Layer* realisiert. Die Vektoren h und b kommen als eindimensionale *numpy.ndarrays* daher. w ist eine zweidimensionale Matrix. Die Berechnung in der Funktion *forward* wird wie folgt codiert:

\_sum = np.dot(self.w, self.previous.get\_excitation()) + self.b

Nach diesem Rechenschritt mit Punktmultiplikation (“dot”) muss noch auf jeden Erregungswert die Sigmoid-Funktion angewendet werden. Sie lautet

Das heißt hier:

In Python: self.excitation = scipy.special.expit(\_sum)

Man beachte, dass Python die *expit* Funktion „automatisch richtig“ auf jedes Element des Vektors \_sum anwendet!

## Rückwärts-Propagation

### Mathematische Ableitung

Unter Backpropagation verstehen wir die Fehlerrückführung während der Trainingsphase: Wir legen ein Muster auf die Eingangsschicht, und erwarten ein Ergebnis am Ausgang des Output Layers. Weil unser Netzwerk nie perfekt arbeiten wird, definieren wir einen Fehlervektor e, der sich einfach als Differenz des Trainingsvektors t und des Output-Vektors o ergibt:

Backpropagation versucht nun, alle Fehler im Vektor zu minimieren und definiert eine skalare Fehlergröße E (manchmal auch „Kosten“ oder „cost function“) genannt:

Dieser skalare Fehler E ist nun eine Funktion der Erregung hj im Hidden Layer, der Synapsengewichte wjk und den Bias-Werten bk des Output-Layers. Um einen Gradientenabstieg zu vollziehen, benötigen wir die partiellen Ableitungen von E nach den Synapsengewichte wjk und den Bias-Werten bk.. Dann können wir nach jedem Anlegen eines Trainingsmusters die Gewichte wjk und Bias-Werten bk in so anpassen, dass der Gesamtfehler verkleinert wird. Dabei hoffen wir, dass sich bei diesem Verfahren nach vielfachen Wiederholungen ein gutes Ergebnis einstellt.

Beginnen wir mit den Synapsengewichten. Für alle j und k gilt:

Man sieht, dass der Fehler ti-oi nur dann von wjk abhängt, wenn i = k ist, denn nur das k-te Neuron wird von den wjk beeinflusst – und zwar nicht-linear. Weil die Ableitung konstanter Werte Null ist, können wir vereinfacht schreiben:

Mit der Kettenregel ergibt sich

Bei

mit +

kann man wieder die Kettenregel anwenden und erhält wegen

und

schließlich

Wegen und e = t – o verkürzt sich der Ausdruck für den Gradienten weiter auf

Wollen wir gegen den Gradienten laufen d.h. talwärts, kehrt sich das Vorzeichen um und wir führen eine Schrittweite α > 0 ein, die praktischerweise den Faktor 2 mit einschließt. So erhalten wir einen Korrekturwert für :

Gl. 1

Für die Bias-Werte rechnet man ähnlich

weil ja nur von abhängt, das den zugehörigen Bias darstellt. Die weitere Berechnung verläuft ähnlich wie oben d.h.

+ ) = 1

Also verbleibt

Auch hier wollen wir gegen den Gradienten laufen und erhalten entsprechend:

Gl. 2

### Programmierung

In Python erfolgt die Berechnung mithilfe von NumPy Vektoren bzw. Matrizen gleich für alle auf einmal:

1. Berechnung des Fehlervektors e (siehe Funktion *train*):

error = self.expected[int(output\_index)] - self.output\_layer.get\_excitation()

Weiter geht’s in Funktion *train*

2. Sowohl für als auch wird der Vektor benötigt, hier val\_next genannt:

val\_next = alpha \* err \* self.excitation \* (1 - self.excitation)

Man beachte, wie elegant und korrekt Python hier die Operatoren realisiert: Bei der ersten Multiplikation müssen alle Elemente von err mit dem Skalar alpha multipliziert werden, bei den anderen Multiplikationen werden alle Elemente mit gleichem Index miteinander multipliziert. Bemerkenswert auch das 1 - …: Hier wird jedes Element ei durch 1 - ei ersetzt!

3. Der Bias ist schnell berechnet, man muss ja nur val\_next addieren:

self.b += val\_next

4. Für die Synapsengewichte in der Matrix w ist es schon etwas spannender: Wir wollen eine Matrix voller erzeugen, die wir dann zu w addieren. Damit nun alle mit den Indizes j und k belegten Werte der Multiplikation in der Gleichung (Gl. 1 oben) richtig zueinander kommen, benötigen wir eine besondere Operation. Der mit k indizierte Vektor liegt als Spalte vor, der mit j indizierte als Zeile. Somit ergibt das Produkt die gewünschte Matrix. In Python:

delta\_w = np.outer(val\_next, self.previous.get\_excitation())

self.w += delta\_w

Bleibt noch die Frage, wie wir den zu einem Fehlervektor für die verborgene Schicht kommen. Dazu teilen wir den Fehler eines Neurons k des Output Layers an die J Neuronen des Hidden Layer gemäß der Synapsengewichte rückwärts auf. Hier folgen wie dem Vorschlag von [Rashid] Seite 81.

Mit dem Präfix f für Fehler erhält man:

fh1 = w11 \* fo1 + w21 \* fo2 + …

fh2 = w12 \* fo1 + w22 \* fo2 + …

Wie man erkennt, läuft dies auf eine Multiplikation der transponierten Matrix w mit dem Fehlervektor fo hinaus. In Python schreiben wir also in *get\_error*:

ret\_val = np.dot(self.w.transpose(), self.error)

# Performance

## Trefferquote auf dem MNIST-Datensatz

Mit den Einstellungen

* Hidden Layer Size = 100
* Step width (alpha) = 0,1

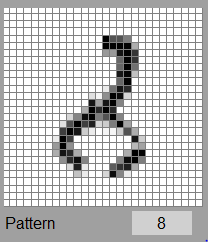
erreicht BaPL nach 10 Epochen eine Performance d.h. eine Trefferquote von über 97%.

## Fehlerhafte Erkennungen

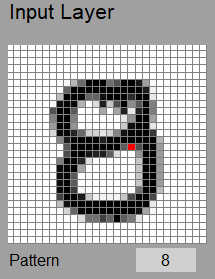
Man kann grob (und subjektiv) zwischen drei Sorten von Fehlern unterscheiden:

### Fehlleistung aufgrund undeutlicher Schrift

Auch jeder Mensch würde hier zweifeln, welche Ziffer mit diesem Gekrakel wohl gemeint sein könnte. Hier vergibt man dem neuronalen Netz gerne und sieht im Fehler keine mangelnde Leistung. Beispiel: Muster #583. Hier ist eine 8 gemeint, erkannt wird aber eine 3.

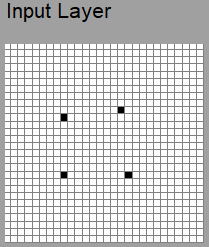


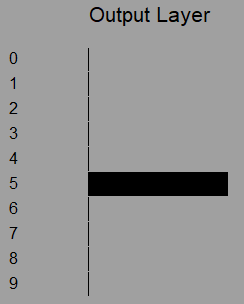
### Unerklärliche Fehler

Hier erkennt man als Mensch leicht und unzweifelhaft die „gemeinte“ Ziffer, doch das Netzwerk erkennt falsch. Beispiel: Test Record #5523. Hier sieht man deutlich eine Acht, doch erkannt wird eine Null. Fügt man manuell ein einzelnes Pixel hinzu (hier rot gekennzeichnet), springt die Erkennung um auf die erwartete Acht.

Die Erkennung durch das Netzwerk ist also nicht so robust, wie man es vielleicht erwarten würde. Ähnliche Phänomene unerwarteter Fehlleistungen werden in der Literatur häufig erwähnt.

### „Erkennung“ aufgrund verstreuter Pixel

Manuell zufällig gestreute Pixel werden als Ziffer erkannt. Hier offenbart sich das Fehlen jeglicher Plausibilitätsprüfung im Algorithmus. Beispiel: Das Muster rechts wird als eine Fünf erkannt, mit deutlichem Profil in der Ausgangsschicht:



# MNIST

## Dateien

Wir verwenden 4 Dateien:

|  |  |
| --- | --- |
| **Name** | **Inhalt** |
| mnist\_train.csv | 60.000 handgeschriebene Ziffern für Trainingszwecke |
| mnist\_test.csv | 10.000 handgeschriebene Ziffern zum Testen eines Erkennungsprogramms |
| mnist\_train\_100.csv | 100 handgeschriebene Ziffern für Trainingszwecke (für kleine Experimente) |
| mnist\_test\_10.csv | 10 handgeschriebene Ziffern für Testzwecke |

## Datenformat

Jede Ziffer ist als 28 x 28 Pixel-Muster gespeichert. Jede Ziffer ist in einer Textzeile mit komma-separierten ASCII-Zahlen kodiert . Die erste Ziffer („Label“) benennt den Wert der Ziffer also 0-9. Darauf folgen die invertierten Helligkeiten (Grauwerte, 0-255) der Pixel Zeile für Zeile. D.h. der weiße Hintergrund hat den Wert 0.

# BaPL Programmaufbau

Das Programm Backpropagation Lab besteht aus folgenden Python Modulen (Dateien), welche sich die Arbeit wie folgt teilen:

|  |  |
| --- | --- |
| **Modul** | **Funktion** |
| BplMain.py | Hauptptorgramm, Grafikinitialisierung, Menü, Hilfsfunktion zum Laden der Daten |
| BplGlobal.py | Globale Variablen |
| BplFileAccess.py | Dateizugriff zum Laden der Daten |
| BplTraining.py | Die Karteikarte (Tab) „ Train & Test“ d.h. die zugehörigen GUI-Funktionen |
| BplNeuroNet.py | Alle Berechnungen am neuronale Netz |
| BplInspection.py | Die Karteikarte (Tab) „Inspect“ d.h. die zugehörigen GUI-Funktionen |
| MnistImporter.py | Separates Modul zur Konvertierung von MNIST-Daten im csv-Format in das BaPL-Format |

Zum Datenaustausch der Module untereinander dient das globale Verzeichnis (Python dictionary) namens appGlobals bzw. app\_globals. Lediglich BplNeuroNet hat mit Absicht keinen Zugriff darauf.

Die Module bzw. die in ihnen enthaltene Klassen hängen wie folgt voneinander ab:

BplNeuroNet

BplTraining

BplInspection

BplMain

BplFileAccess

BplGlobal

# Mögliche Verbesserungen / Erweiterungen

1. Prüfe, ob die Testdaten dasselbe Format wie die Trainingsdaten haben.
2. Bei Inspektion: Button: Show next failure pattern
3. Speichern und laden trainierter Netze (Insbesondere beim XOR wäre ein tabellarischer Ausdruck schön)
4. Kurzanleitung des Programms auf das erste Tab schreiben

# Quellen

[Rashid] - Tariq Rashid: „Make Your Own Neural Network”, CreateSpace Independent Publishing Platform 2016

1. <https://de.wikipedia.org/wiki/K%C3%BCnstliches_neuronales_Netz> [↑](#footnote-ref-1)