Backpropagation Lab

(BaPL)

Beschreibung und Bedienungsanleitung

Martin Reiche, Tübingen, 21. Februar 2019

Inhaltverzeichnis

1 Einführung 3

2 Programmaufbau 3

3 Dateiformat 3

4 Bedienung 4

5 Einschränkungen 6

6 Geplante Erweiterungen 6

7 Mathematische Ableitung und Programmierung mit NumPy 6

7.1 Vorwärts-Propagation 6

7.1.1 Mathematische Ableitung 6

7.1.2 Programmierung 7

7.2 Rückwärts-Propagation 8

7.2.1 Mathematische Ableitung 8

7.2.2 Programmierung 9

8 Quellen 10

1 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".

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.

2 Programmaufbau

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

Modul	Funktion
BplMain.py	Grafikinitialisierung, Menü, Hilfsfunktion zum Laden der Daten
BplFileAccess.py	Dateizugriff zum Laden der Daten
BplTraining.py	Die Karteikarte (Tab) „ Training“ d.h. die zugehörigen GUI-Funktionen
BplNeuroNet.py	Alle Berechnungen am neuronale Netz
BplInspection.py	Die Karteikarte (Tab) „Inspektion“ 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.

3 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.

Wichtig ist die Unterscheidung der Typen:

- `data_bw` (black/white) kündigt schwarzweiß-Daten an: „-“ = weiß, „x“ = schwarz.
- `data_gray` kündigt Daten mit Grauwerten an von 0 = weiß bis 255 = schwarz

4 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. Setz 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:
    a. 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).
    b. In der mittleren Spalte werden die Erregungen der Neuronen in der Zwischenschicht durch waagerechte Balken dargestellt, maximale Länge -> Erregung = 1
    c. 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.

5 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

6 Geplante Erweiterungen

- Speichern und laden trainierter Netze

7 Mathematische Ableitung und Programmierung mit NumPy

Für eine Einführung in künstliche neuronale Netze siehe z.B. diesen[1] Wikipedia-Artikel.

7.1 Vorwärts-Propagation

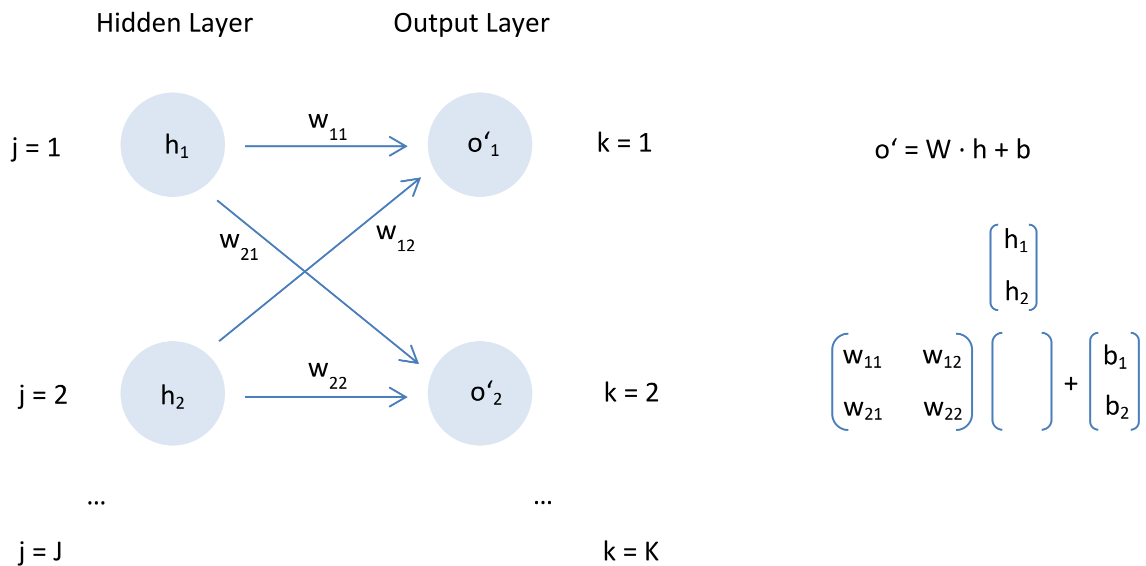
7.1.1 Mathematische Ableitung

Darunter 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):

[1] https://de.wikipedia.org/wiki/K%C3%BCnstliches_neuronales_Netz



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

$o'_1 = w_{11} \cdot h_1 + w_{12} \cdot h_2 + b_1$

$o'_2 = w_{21} \cdot h_1 + w_{22} \cdot h_2 + b_2$

7.1.2 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

$$sigm(x) = \frac{1}{1 + e^{-x}}$$

Das heißt hier:

$$o = sigm(o')$$

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

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

7.2 Rückwärts-Propagation

7.2.1 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:

$$e = t - o$$

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

$$E = \sum_{k=1}^{K} (t_k - o_k)^2$$

Dieser skalare Fehler E ist nun eine Funktion der Erregung h_j im Hidden Layer, der Synapsengewichte w_{jk} und den Bias-Werten b_k des Output-Layers. Um einen Gradientenabstieg zu vollziehen, benötigen wir die partiellen Ableitungen von E nach den Synapsengewichte w_{jk} und den Bias-Werten b_k.. Dann können wir nach jedem Anlegen eines Trainingsmusters die Gewichte w_{jk} und Bias-Werten b_k 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:

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial}{\partial w_{jk}} \sum_{i=1}^{K} (t_i - o_i)^2$$

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

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial}{\partial w_{jk}} (t_k - o_k)^2$$

Mit der Kettenregel ergibt sich

$$\frac{\partial E}{\partial w_{jk}} = -2\,(t_k - o_k)\frac{\partial o_k}{\partial w_{jk}}$$

Bei

$$o_k = sigm(o'_k) \quad \text{mit} \quad o'_k = \sum_{j=1}^{J} w_{jk}\, h_j + b_k$$

kann man wieder die Kettenregel anwenden und erhält wegen

$$\frac{d}{dx} sigm(x) = sigm(x) \cdot (1 - sigm(x))$$

und

$$\frac{\partial o'_k}{\partial w_{jk}} = h_j$$

schließlich

$$\frac{\partial E}{\partial w_{jk}} = -2\,(t_k - o_k)\ \cdot sigm(o'_k)\ \cdot\ (1\ -\ sigm(o'_k)) \cdot\ h_j$$

Wegen $sigm(o'_k)\ =\ o_k$ und e = t – o verkürzt sich der Ausdruck für den Gradienten weiter auf

$$\frac{\partial E}{\partial w_{jk}} = -2\ e_k \cdot o_k\ \cdot\ (1\ -\ o_k) \cdot\ h_j$$

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 w_{jk}:

$$\Delta w_{jk} =\ \alpha\ \cdot e_k \cdot o_k\ \cdot\ (1\ -\ o_k) \cdot\ h_j \qquad \text{Gl. 1}$$

Für die Bias-Werte rechnet man ähnlich

$$\frac{\partial E}{\partial b_k} = \frac{\partial}{\partial b_k}\ (t_k - o_k)^2$$

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

$$\frac{\partial E}{\partial b_k} = -2\ (t_k - o_k)\ sigm(\frac{\partial o'_k}{\partial b_k})$$

$$\frac{\partial o'_k}{\partial b_k} = \frac{\partial}{\partial b_k}\ (\textstyle\sum_{j=1}^{J} w_{jk}\, h_j + b_k) = 1$$

Also verbleibt

$$\frac{\partial E}{\partial b_k} = -2\ (t_k - o_k)\ o_k\ \cdot\ (1\ -\ o_k)$$

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

$$\Delta b_k =\ \alpha\ \cdot e_k \cdot o_k\ \cdot\ (1\ -\ o_k) \qquad \text{Gl. 2}$$

7.2.2 Programmierung

In Python erfolgt die Berechnung mithilfe von NumPy Vektoren bzw. Matrizen gleich für alle w_{jk}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 Δw als auch Δb wird der Vektor $\alpha \cdot e_k \cdot o_k \cdot (1 - o_k)$ 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 e_i durch 1 - e_i 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 Δw_{jk} 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)
```

8 Quellen

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