

NetLab

Back-Propagation in Python mit Tkinter

Beschreibung und Bedienungsanleitung

Martin Reiche, Tübingen, 2024

Version vom 27.8.2026

Inhaltsverzeichnis

Einführung.....	3
NetLab Dateiformat.....	3
Bedienung.....	4
Einschränkungen.....	8
Mathematische Ableitung und Programmierung mit NumPy.....	8
1.1 Vorwärts-Propagation.....	8
1.1.1 Mathematische Ableitung.....	8
1.1.2 Programmierung.....	9
1.2 Rückwärts-Propagation.....	10
1.2.1 Mathematische Ableitung.....	10
1.2.2 Programmierung.....	12
Performance.....	13
1.3 Trefferquote auf dem MNIST-Datensatz.....	13
1.4 Fehlerhafte Erkennungen.....	13
1.4.1 Fehlleistung aufgrund undeutlicher Schrift.....	13
1.4.2 Unerklärliche Fehler.....	13
1.4.3 „Erkennung“ aufgrund verstreuter Pixel.....	14
MNIST.....	14
1.5 Dateien.....	14
1.6 Datenformat.....	15
Anwendung auf andere Datensätze.....	15
1.7 7x5 Pixelmatrix Buchstaben.....	15
1.8 Folio Features.....	16
1.9 Folio Bilddaten.....	16
NetLab Programmaufbau.....	19
NeuroNet.py.....	20
1.10 Klasse Layer.....	20
1.11 Klasse NeuroNet.....	20
Die Klasse Inspection.....	20
Quellen.....	21
Siehe auch.....	21

Einführung

Zum Einstieg sollte man die Seite <https://martin-reiche.de/backpropagation> studieren!

Eine gute Einführung in die fachliche Aufgabenstellung findet man z.B. in Tariq Rashids Buch "[Make Your Own Neural Network](#)". Dieses Buch gab mir den Anstoß, NetLab zu programmieren.

NetLab 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-handwritten-digits.html> heruntergeladen.

NetLab Dateiformat

NetLab 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 NetLab übernommen und zum Teil auch angezeigt.

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

```
application=NetLab version=0 type=data_gray
```

Folgende Schlüssel / Wert-Paare werden benutzt:

Schlüssel	Mögliche Werte	Bedeutung
application	NetLab	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)

Vermutlich können bei type = data_gray irgendwelche Ganz- oder Fließkommazahlen stehen, NetLab sollte mit allen zurecht kommen. Das habe ich aber nicht geprüft.

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.

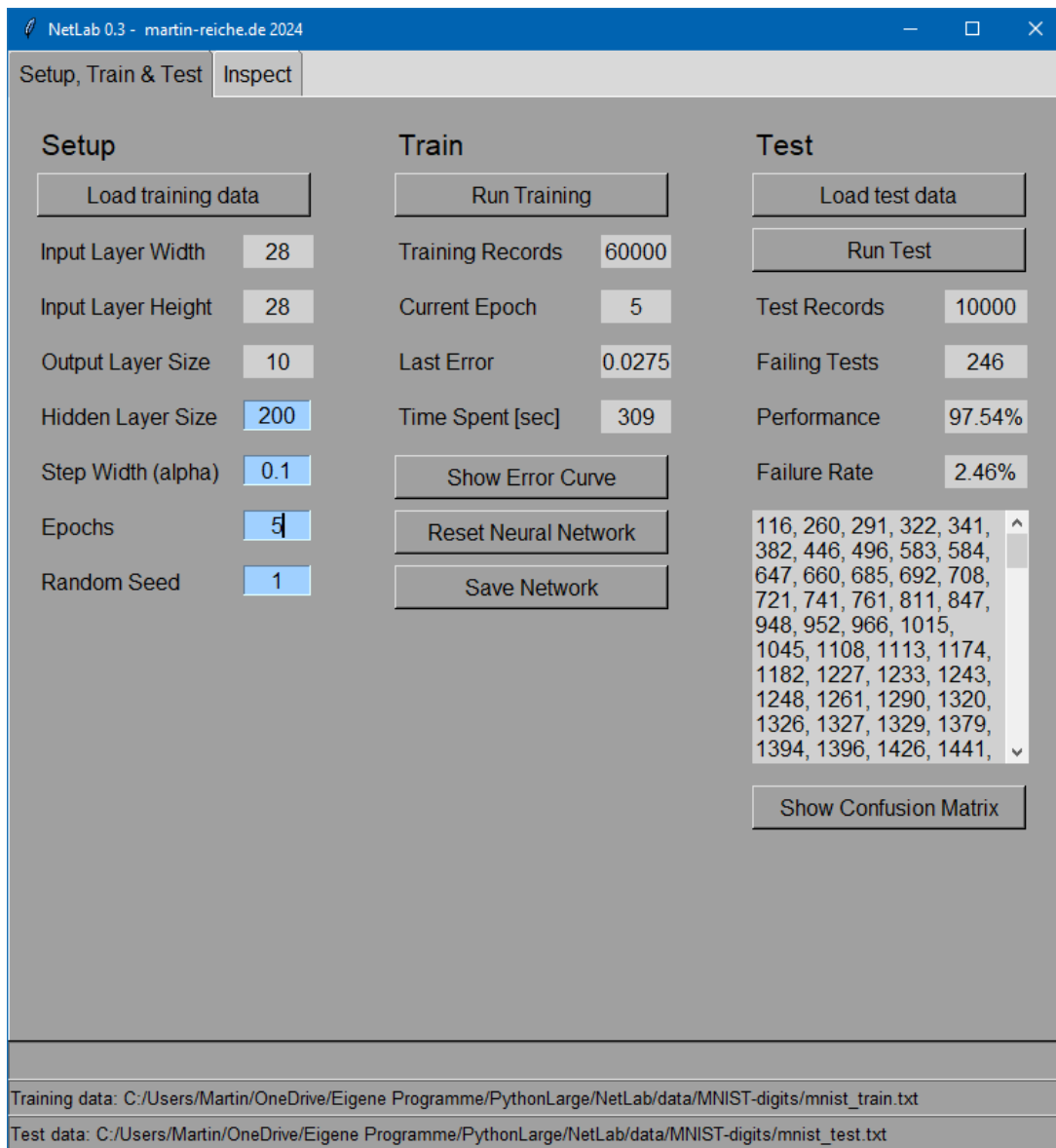
Folgende globale Variablen werden aus der Datei der Trainingsdaten gesetzt:

Variable	Bedeutung	Kommentar
<code>g.numRows</code>	Anzahl der Zeilen des Musters	
<code>g.numCols</code>	Anzahl der Spalten des Musters	
<code>g.outputLayerSize</code>	Anzahl der Neuronen in der Ausgabeschicht	
<code>g.allTrainingPattern</code>	Array der Trainingsmuster	Daraus abgeleitet wird die Zahl „Training Records“ angezeigt.

Bedienung

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

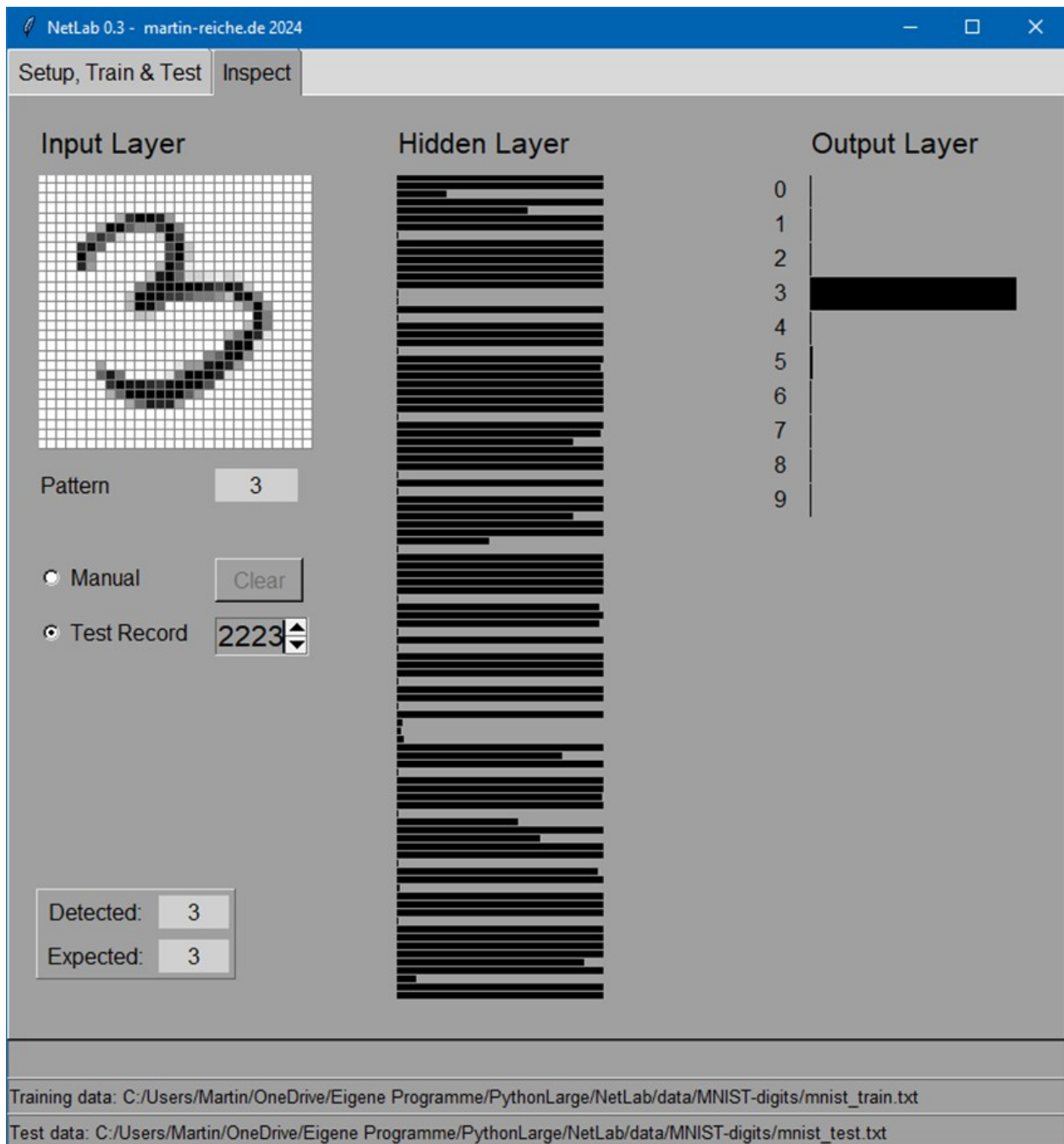
Führe die Datei „main.py“ aus! Das Training wickelt man auf der Registerkarte „Setup, Train & Test“ ab, die beim Arbeiten mit MNIST-Daten z.B. so aussieht:



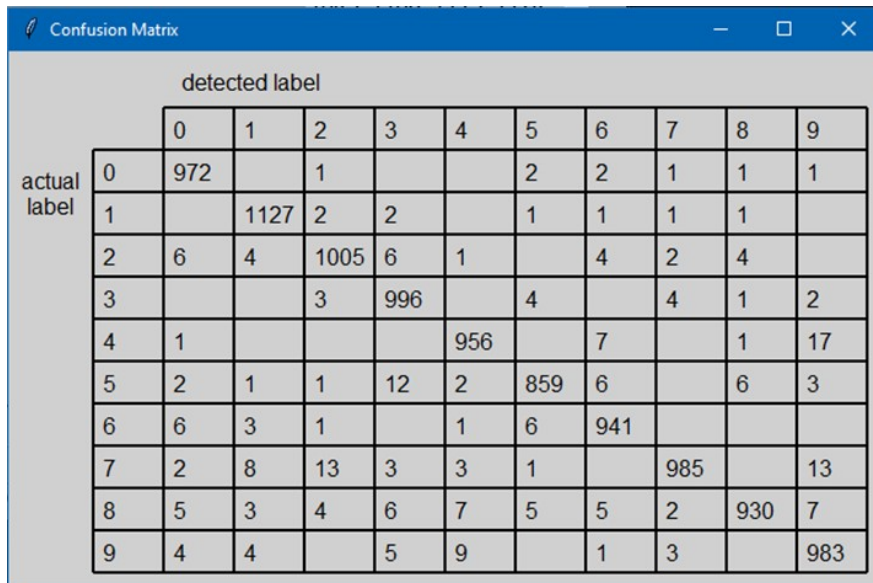
Ein Programmablauf mit NetLab gestaltet sich wie folgt:

1. Auswahl von Trainingsdaten durch Auswahl einer Datei mit dem Knopf „Load Training Data“. Achtung: Bei MNIST-Daten kann dies dauern! Ein möglicherweise vorher existentes neuronales Netz wird gelöscht.
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 ein- oder mehrmals „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. „Last Error“ bezeichnet den mittleren Trainingsfehler pro Record der letzten Epoche. 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. Der Knopf „Save Network“ speichert die Gewichte und Biases aller Layer im json-Format in eine Datei namens „network.txt“.
13. Um den Trainingserfolg zu prüfen, lädt man mit „Load Test Data“ die Testdaten aus einer Datei.
14. Dann klickt man „Run Test“ und erhält eine statistische Aussage über den Anteil erfolgreicher (Performance) und fehlgeschlagener (Failed Tests, 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 sich so die Reaktion des Netzwerkes anschauen.
15. Wobei wir schon auf der Inspection-Seite bzw. Registerkarte angekommen sind (siehe unten), 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.



Schließlich lässt sich auf dem Tab „Setup, Train Test“ die [Konfusionsmatrix](#) aufrufen (Knopf „Show Confusion Matrix“):



		detected label									
		0	1	2	3	4	5	6	7	8	9
actual label	0	972		1			2	2	1	1	1
	1		1127	2	2		1	1	1	1	
	2	6	4	1005	6	1		4	2	4	
	3			3	996		4		4	1	2
	4	1				956		7		1	17
	5	2	1	1	12	2	859	6		6	3
	6	6	3	1		1	6	941			
	7	2	8	13	3	3	1		985		13
	8	5	3	4	6	7	5	5	2	930	7
	9	4	4		5	9		1	3		983

Wie man in diesem Beispiel erkennt, wird die Eins 8 Mal falsch gelesen, während die Vier 17 Mal als Neun interpretiert wurde.

Einschränkungen

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

maximale Zeilen- und Spaltenzahl im Input-Layer	ca. 30
Maximale Anzahl der Neuronen im Hidden Layer	ca. 200
Maximale Anzahl der Neuronen im Output Layer	ca. 26

Mathematische Ableitung und Programmierung mit NumPy

Siehe [Rashid]. Für eine Einführung in künstliche neuronale Netze siehe z.B. [diesen](#)¹ Wikipedia-Artikel.

1.1 Vorwärts-Propagation

1.1.1 Mathematische Ableitung

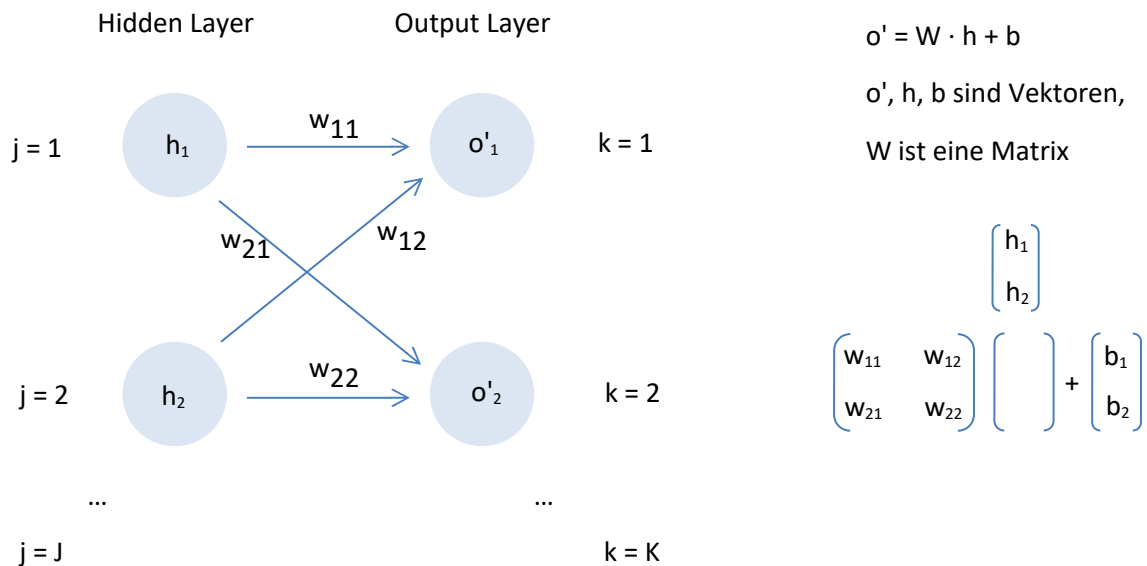
Unter Vorwärts-Propagation wollen wir die normale Funktion eines neuronalen Netzes verstehen: Die Weiterleitung einer Erregung durch die Schichten.

Für die folgende Betrachtung reicht ein Netzwerk 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

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

haben. Gemäß Konvention bezieht sich der erste Index auf die Zeile (row) und der zweite auf die Spalte (column):



Eine Matrixmultiplikation der Gewichtsmatrix W (weight) mit dem Erregungsvektor der Zwischenschicht h (hidden layer) ergibt die Erregung der Output-Neuronen gemäß obiger Formel d.h.

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

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

d.h. generell $o'_k = \sum_{j=1}^{j=J} w_{k,j} \cdot h_j + b_k$. Anschließend wird das Signal o' noch von der Sigmoid-Funktion als Aktivierungsfunktion begrenzt, sodass am Ausgang des Layers $o_k = \text{sigm}(o'_k)$ erscheint.

1.1.2 Programmierung

Die Neuronen einer Schicht sind in NetLab 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

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

Das heißt hier:

$$o = \text{sigm}(o')$$

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

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

1.2 Rückwärts-Propagation

1.2.1 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 (wegen o_k , siehe oben) 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

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

kann man wieder die Kettenregel anwenden und erhält wegen

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

und

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

schließlich

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

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

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

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

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

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.

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

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

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:

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

Und wie verfahren wir mit den Gewichten des Hidden Layer? Diese müssen beim Training ja auch angepasst werden! Dazu teilen wir den Fehler eines Neurons k des Output Layers an die J Neuronen des Hidden Layer gemäß der Synapsengewichte rückwärts auf. Wir multiplizieren also für jedes Neuron des Hidden Layer die Fehler des Output layer mit den zugehörigen Gewichten, was in Matrix-Schreibweise die Transposition der Gewichtsmatrix des Output-Layers bedeutet (Siehe [Rashid] Seite 82):

$$error_{hidden} = W^T \cdot error_{output} \quad \text{Gl. 3}$$

1.2.2 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*):

```
exp = np.zeros(self.output_layer.size)
exp[int(output_index)] = 1
error = self.expected[int(output_index)] - self.output_layer.get_excitation()
```

Weiter geht's in Funktion *Layer.backward*

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 $1 - \dots$: 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
```

Den Fehlervektor für die verborgene Schicht berechnen wir nach Gleichung 3 oben. In Python schreiben wir also in `get_error`:

```
ret_val = np.dot(self.w.transpose(), self.error)
```

Performance

1.3 Trefferquote auf dem MNIST-Datensatz

Mit den Einstellungen

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

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

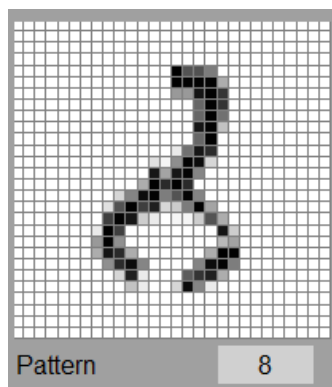
Beachte: Auch ohne Hidden-Layer wird eine beachtliche Performance von knapp 92% erreicht. Diese kann aber durch weiteres Training nicht erhöht werden.

1.4 Fehlerhafte Erkennungen

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

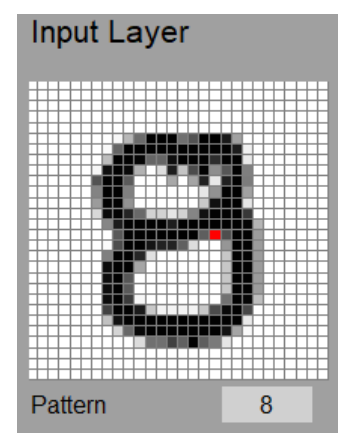
1.4.1 Fehlleistung aufgrund undeutlicher Schrift

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 2.



1.4.2 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 (bei manchen Trainingsdurchläufen)



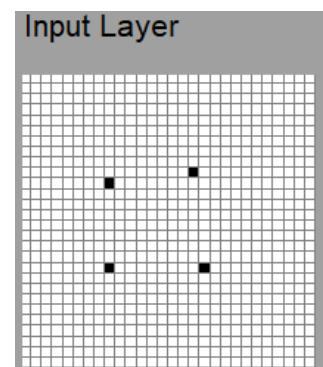
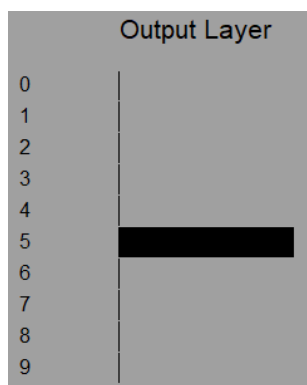
eine Null. Fügt man manuell ein einzelnes Pixel hinzu (hier rot gekennzeichnet), springt die Erkennung um auf die erwartete Acht.

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.

Warum verhält sich das Netz so seltsam? Vielleicht sollte man andersherum denken und sich wundern, warum das Netz überhaupt solche Leistungen vollbringt: Denn es gehen ja schon am Input-Layer alle räumlichen Bezüge der Pixel untereinander verloren, die für unsere Augen ja das Entscheidende sind! Jedes Neuron des Hidden-Layer summiert die gewichteten Erregungen des Input-Layer und weder in den Gewichten noch in der Summierung finden Pixel-Nachbarschaften Berücksichtigung. Wir Menschen sehen in den Bildern Striche, Bögen und Schleifen. Für all dies ist das Netz sozusagen blind. Fortschritt in dieser Hinsicht bringen erst Faltungsnetzwerke, sogenannte Convolutional Neural Networks.

1.4.3 „Erkennung“ aufgrund verstreuter Pixel

Ein „leeres“ Bild oder 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

1.5 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

1.6 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.

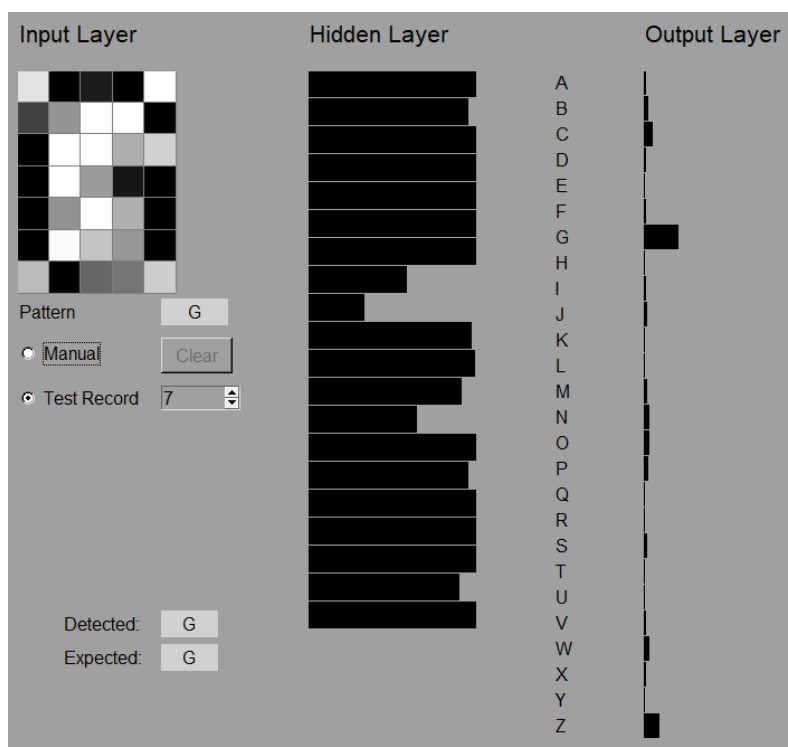
Anwendung auf andere Datensätze

1.7 7x5 Pixelmatrix Buchstaben

Man kann das Programm auch auf die Buchstaben aus NeuroLab (<https://martin-reiche.de/neurolab>) anwenden (File alfabet_gray.txt). Mit den Parametern

Hidden Layer Size	20
Step Width (alpha)	0.1
Epochs	1000
Random Seed	1

gelingt ein perfektes Training. Ein Test mit verrauschten Daten (50%, Datei alfabet_gray_gray_noise_50.txt) liefert eine fehlerfreie Erkennung:



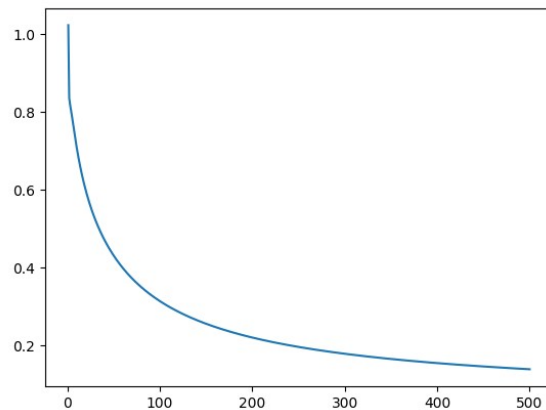
Ab ca. 60% Rauschanteil stellen sich die ersten Fehler ein. Doch auch bei 80% Rauschen werden noch 77% der Buchstaben richtig erkannt.

1.8 Folio Features

Auch mit den Feature-Daten des Folio-Projektes (<https://martin-reiche.de/folio>) gelingen gute Erkennungsleistungen. Mit den Dateien *nn_training-data.txt*, *nn_test-data.txt* und den Einstellungen

Input Layer Width	1
Input Layer Height	5
Output Layer Size	7
Hidden Layer Size	0
Step Width (alpha)	0.2
Epochs	500
Random Seed	1

Fehlerkurve:



ergibt sich eine Performance von 100% d.h. die Erkennung funktioniert perfekt ohne *hidden layer*!. Ein Testdurchlauf mit den Trainingsdaten liefert zwei Fehler; das ist vergleichbar mit dem k-means Verfahren.

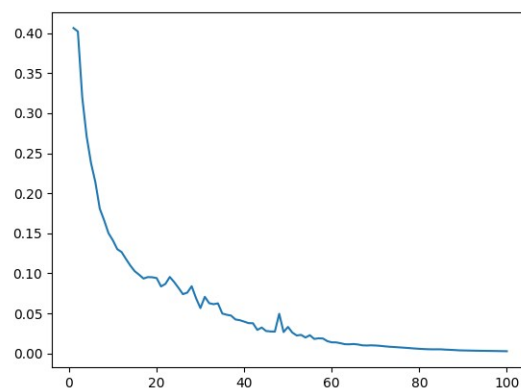
(Die Dateien *nn_training-data.txt*, *nn_test-data.txt* werden erzeugt von *backprop_features.py*)

1.9 Folio Bilddaten

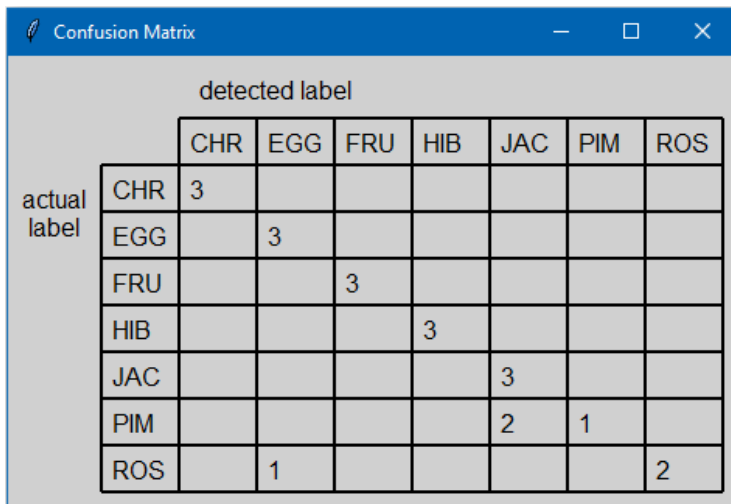
Doch funktioniert auch eine direkte Verarbeitung der Bilddaten? Zu diesem Zweck habe ich die 105 + 21 Bilder der Blätter zentriert auf ein 30x30 Pixelraster transformiert. Wenn man darauf die Backpropagation mit den Parametern

Input Layer Width	30
Input Layer Height	30
Output Layer Size	7
Hidden Layer Size	50
Step Width (alpha)	0.2
Epochs	100
Random Seed	1

Fehlerkurve:



trainiert, erzielt man eine Trefferquote von über 85% d.h. von den 21 Test-Blättern werden nur 3 falsch klassifiziert, wie auch die Konfusionsmatrix zeigt. Zwei Pimento-Blätter wurden falsch als Jackfruit erkannt, ein Rosenblatt falsch als Eggplant.



		detected label						
		CHR	EGG	FRU	HIB	JAC	PIM	ROS
actual label	CHR	3						
	EGG		3					
	FRU			3				
	HIB				3			
	JAC					3		
	PIM					2	1	
	ROS		1					2

Dieses Ergebnis bestätigt die Empfehlung, man solle sich nicht unnötig mit der Gewinnung von Features belasten, sondern das Neuronale Netz auf den Rohdaten trainieren. Das findet schon von selbst heraus, worauf es ankommt!

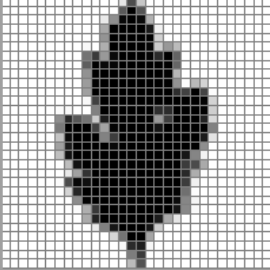
Beachte: Die Resultate könnten ein glücklicher Zufall sein. Erst mit einer [Kreuzvalidierung](#) ließe sich mehr Gewissheit verschaffen. Der Erwartungswert für einen blinden Algorithmus ist allerdings verschwindend gering. Für 6 Treffer in Reihe liegt die Wahrscheinlichkeit bei $1/7^6 = 1/117649$

(Das Programm *shrink_leaves.py* im Blattsalat-Projekt erzeugt verkleinerte Kopien der Blätter in den Verzeichnissen *shrunk_test_images* und *shrunk_training_images*. Das Programm *back_porp_images* wandelt diese Bilder um in Textdateien für NetLab.)

NetLab 0.3 - martin-reiche.de 2024

Setup, Train & Test Inspect

Input Layer



Pattern: CHR

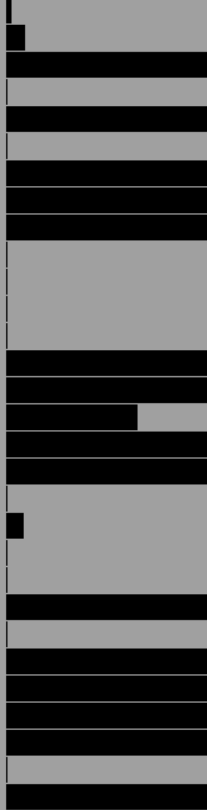
☒ Manual

☐ Test Record 2

Detected: CHR

Expected: CHR

Hidden Layer



Output Layer

Label	Value
CHR	High
EGG	Low
FRU	Low
HIB	Low
JAC	Low
PIM	Low
ROS	Low

Training data: C:/Users/Martin/OneDrive/Eigene Programme/PythonLarge/NetLab/data/Folio/nn_training_images.txt

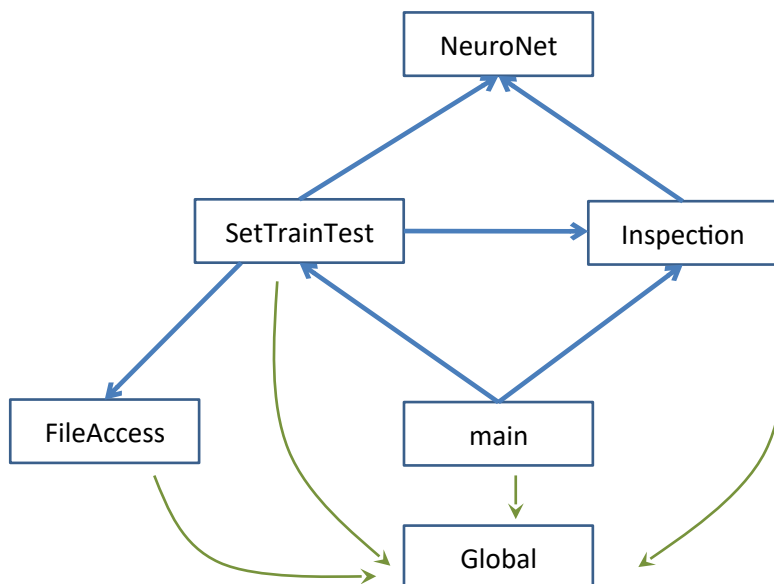
Test data: C:/Users/Martin/OneDrive/Eigene Programme/PythonLarge/NetLab/data/Folio/nn_test_images.txt

NetLab Programmaufbau

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

Modul	Funktion
main.py	Hauptprogramm, Grafikinitialisierung, Menü, Hilfsfunktion zum Laden der Daten
Global.py	Globale Variablen
FileAccess.py	Dateizugriff zum Laden der Daten
SetTrainTest.py	Die Karteikarte (Tab) „Setup, Train & Test“ d.h. die zugehörigen GUI-Funktionen
NeuroNet.py	Alle Berechnungen am neuronale Netz
Inspection.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 NetLab-Format

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



NeuroNet.py hat mit Absicht keinen Zugriff auf die globalen Variablen, damit es auch in anderen Kontexten genutzt werden kann.

NeuroNet.py

enthält die Klassen `NeuroNet` und `Layer`. Letztere ist nach außen nicht sichtbar.

1.10 Klasse `Layer`

Die Klasse `Layer` repräsentiert eine Schicht des neuronalen Netzes, sei es die Input-, Hidden-, oder Output-Schicht. Über die Klassenvariable `self.previous` kann ein `Layer` mit einer darunter liegenden Schicht (ebenfalls vom Typ `Layer`) verbunden sein. Dies trifft natürlich nur zu bei den Hidden- und Output-Layers. Konkret heißt das: Der Hidden-Layer liegt unter dem Output-Layer, aber über dem Input-Layer. Falls kein Hidden-Layer vorhanden ist, ist der `self.previous` Layer des Output-Layer direkt der Input-Layer. Diese Konfiguration wird in der Klasse `NeuroNet` festgelegt. Prinzipiell könnte ein Netzwerk aus beliebig vielen Schichten vom Typ `Layer` bestehen.

`Layer` implementiert die Funktionen zum Propagieren der Erregung (*forward*) als auch zum Trainieren, d.h. zur Backpropagation (*backward*). `Layer` macht intensiven Gebrauch von den Matrix-Operationen in Numpy.

Die Anzahl der Neuronen in der Schicht hält die Variable `self.size`. Die meisten Klassenvariablen sind vom Typ Numpy ndarrays mit den folgenden Dimensionen:

Name	Inhalt	Dimensionen	Kommentar
<code>self.w</code>	Gewichtsmatrix	Anzahl Zeilen = Anzahl der Neuronen in dieser Schicht Anzahl Spalten = Anzahl der Neuronen in der Vorgängerschicht	
<code>self.b</code>	Biasvektor	Vektor des Bias dieser Schicht. Länge = Anzahl der Neuronen in dieser Schicht	
<code>self.excitation</code>	Erregungsvektor	Länge = Anzahl der Neuronen in dieser Schicht	
<code>self.error</code>	Fehlervektor	Länge = Anzahl der Neuronen in dieser Schicht	

Siehe die Beschreibung der Funktion *forward* und *backward* in 1.1.2 [oben](#)!

1.11 Klasse `NeuroNet`

Die Klasse `NeuroNet` koordiniert das Zusammenspiel der Layers bei Training und Test bzw. einfacher Vorwärts-Propagation (*run*).

Die Klasse `Inspection`

Neben den statischen Labels und Buttons enthält die Klasse `Inspection` 3 Canvas, um die Erregungen der Input, Hidden und Output Layer grafisch darzustellen: `canvas_input`, `canvas_hidden`, `canvas_output`.

Der GUI-Aufbau erfolgt in 3 Schritten:

- Der Konstruktor wird beim Start von NetLab aus *main* heraus gerufen.

- Sobald die Testdaten geladen sind, werden alle Canvas initialisiert: *create_input_display*, *create_hidden_histogram*, *create output_histogram*.
- Erst sobald das Inspection-Tab gezeigt wird (siehe *Gui.tab_changed*), wird das erste Pattern gezeigt, durch das Neuronale Netz bewertet und die Erregungen dargestellt.

Quellen

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

Siehe auch

<https://playground.tensorflow.org/>