

1 Das ursprüngliche Programm von Koenecke

1.1 Problemstellung und Netzwerktopologie

Die Anwendung soll in der Lage sein, Klassifizierungsprobleme zu lösen, da diese zum Einstieg greifbarer sind als beispielsweise Regressionsprobleme. Zur Lösung dieser Problemstellung eignet es sich, als Topologie für das KNN das MLP festzusetzen. Bei jeder Initialisierung eines neuen MLP werden die Gewichtswerte zufällig vom Programm festgelegt. Jeder Datensatz besteht aus zwei Eingabewerten, denen eine Klasse zugeordnet wird, wobei es drei verschiedene Klassen gibt. Ziel des Trainings des KNN ist es, zu jeder Eingabemöglichkeit eine Klasse bestimmen zu können, indem ähnliche Eingabekombinationen gleich klassifiziert werden.

1.2 Technologie

Mit der Programmiersprache *Google Go (golang)* hatte Koenecke eine ausführbare Anwendung `mlpmain_windows_amd64.exe` erstellt, dessen Herzstück die Implementierung des KNN als Netzwerk-Bibliothek ist. Beim Ausführen der Anwendung wird ein Server zum Port 8080 gestartet. Der Server ist dafür zuständig, die komplexeren Berechnungen für das KNN im Hintergrund durchführen zu lassen.

Die grafische Oberfläche, im Folgenden soll hierfür der Fachbegriff Grafical User Interface (GUI) benutzt werden, wurde mit HTML, CSS und Javascript realisiert. Zur grafischen Darstellung von Daten wie die Topologie des KNN wurde die Javascript-Bibliothek D3.js verwendet, die extra darauf ausgelegt ist, Daten manipulieren und (interaktiv) visualisieren zu können. Die GUI bietet Konfigurationsmöglichkeiten und eine Visualisierung der Struktur und Ergebnisse des KNN.

Zur Kommunikation zwischen Backend und Frontend wurde ein Webserver als Schnittstelle mit `gopkg.in/igm/sockjs-go.v2/sockjs`, ein Paket zur Websocket Emulation, realisiert. Unter der Adresse <http://localhost:8080> im Webbrowser wird die GUI der Anwendung für die Daten der Netzwerk-Bibliothek aufgerufen.

1.3 Festgelegte Parameter

Um die Bedienung und die Übersichtlichkeit der GUI zu erleichtern, entschied sich Koenecke dafür, bestimmte Parameter festzulegen. Die Beschränkung auf zwei Eingabe- und drei Ausgabewerte ermöglicht eine zweidimensionale Visualisierung der Inhalte, eine dreidimensionale Visualisierung wäre deutlich aufwändiger gewesen. Zudem sind in seiner Anwendung als Aktivierungsfunktion eine sigmoide Funktion und als Lernregel die Backpropagation-Regel festgelegt, für die Lernrate wurde ebenfalls schon ein vordefinierter Wert gesetzt.

1.4 Aufbau der GUI

1.4.1 Netzwerktopologie

Das erste Element der Anwendung (siehe Abb. 1.1) dient zur Konfiguration der Topologie. Es ist möglich, bis zu 5 Hidden-Layer hinzuzufügen und sie auch wieder zu entfernen, zudem gibt es bei jedem Hidden-Layer die Möglichkeit, die Anzahl der Neuronen zwischen 1-9 festzulegen. Beim Drücken auf den Button *apply* wird direkt ein MLP bzw. Netzwerk mit der gewünschten Topologie als Graph visualisiert und kann wie in Abb. 1.2 aussehen. Die Dicke der Linien soll die Größe der Gewichtswerte andeuten, die Farbe der Linien, welches Vorzeichen die Gewichtswerte haben.

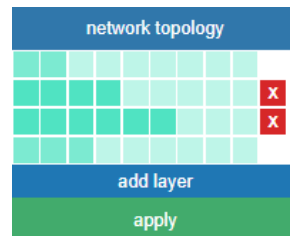


Abbildung 1.1: Konfiguration der Netzwerktopologie

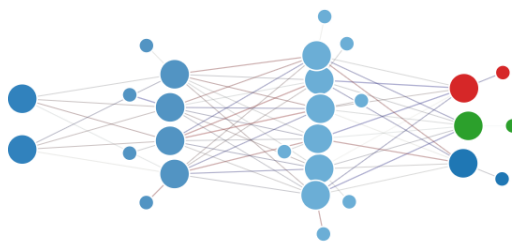


Abbildung 1.2: Darstellung eines Netzwerkgraphs (untrainiert)

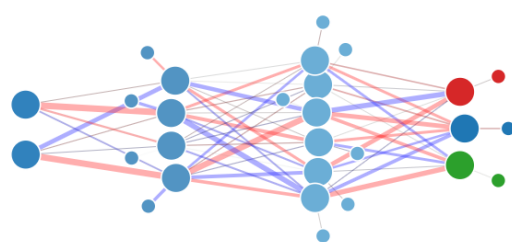


Abbildung 1.3: Darstellung eines Netzwerkgraphs (trainiert)

1.4.2 Training

Erzeugung der Trainingsdaten

Zur Erzeugung der Trainingsdaten gibt es eine Fläche von 300×300 px, die als Koordinatensystem gesehen werden kann, auf der rote, grüne und blaue Punkte eingetragen werden können (siehe Abb. 1.4). Die Wahl auf diese Farben als Klassen fiel laut Koenecke aufgrund der dadurch gegebenen Möglichkeit, so optimal den RGB-Farbraum darstellen zu können. Jeder Punkt ist eine Darstellung eines Trainingsdatensatzes, bei dem die x- und die y-Koordinate die Eingabe und die Farbe die erwartete Ausgabe bzw. Klasse darstellt.

Trainieren des Netzwerks

Sind die Trainingsdaten festgelegt, kann über den Button *train network* das Training gestartet werden (siehe Abb. 1.5). Der Slider ermöglicht es, das Training zu beschleunigen oder zu verlangsamen. Beim Trainingsprozess verändern sich die Gewichtswerte, dies ist auch bei der Visualisierung des Netzwerkgraphs zu sehen, bei dem sich die Dicken der Linien ändern (siehe Abb. 1.3).

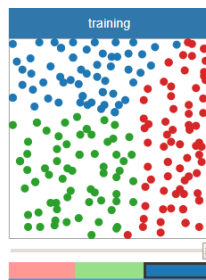


Abbildung 1.4: Erzeugung der Trainingsdaten

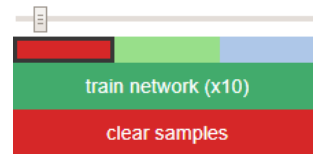


Abbildung 1.5: Start des Trainings mit 10-facher Geschwindigkeit

1.4.3 Präsentation der Netzwerkergebnisse

Vorschau der Ausgabe

Bei jedem Trainingsschritt wird für jeden Punkt des Eingabekoordinatensystems, d.h. für jede mögliche Eingabemöglichkeit der x- und der y-Koordinate,ⁱ drei Ausgabewerte berechnet, die in der Anwendung die RGB-Werte repräsentieren. Dadurch ergibt sich die Möglichkeit, die Trainingsergebnisse als Bild zu visualisieren (siehe z.B. Abb.

ⁱIn diesem Fall gibt es also $300 \cdot 300 = 90000$ Eingabemöglichkeiten.

1.6). Jeder Ausgabewert steht dafür, wie hoch die Wahrscheinlichkeit ist, dass der Punkt zu einer bestimmten Klasse bzw. Farbe gehört. Dadurch kann sich für einen Punkt eine Mischfarbe ergeben, wenn sie zwei ähnlich hohe Ausgabewerte hat, meist befindet sie sich in dem Falle nahe der Hyperebene (siehe z.B. Abb. 1.7, bei dem manche Punkte die Mischfarbe gelb-orange oder lila haben). Je mehr Trainingsschritte durchlaufen werden, desto eindeutiger kann jedem Punkt eine Farbe bzw. eine Klasse zugeordnet werden, was sich in der Vorschau dadurch bemerkbar macht, dass die Farben gesättigter und die Kanten schärfer sind (siehe z.B. Abb. 1.8).



Abbildung 1.6: Vorschau nach einer halben Million Beispieldaten



Abbildung 1.7: Vorschau nach einer Million Beispieldaten

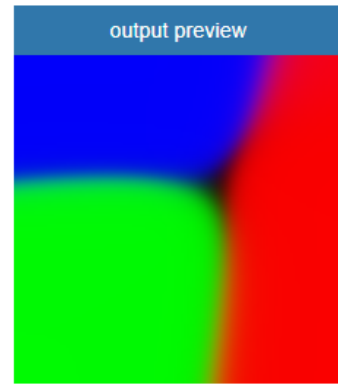


Abbildung 1.8: Vorschau nach drei Millionen Beispieldaten

Netzwerkinfo

Unter der Vorschau werden dem Anwender bestimmte Informationen des Netzes gezeigt (siehe Abb. 1.9). Die Information *samples total* gibt die Anzahl der bisher durchgeführten Trainingsschritte an, *sample coverage* wie viel Prozent der Trainingspunkte durch das KNN korrekt klassifiziert wurden und *mean weight change* wie hoch die durchschnittliche Gewichtsänderung ist.

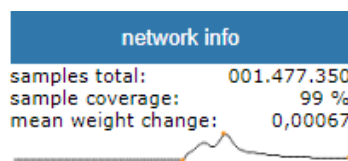


Abbildung 1.9: Netzwerkinfo während des Trainings

2 Eigene Implementierungen

2.1 Implementierung des KNN in Javascript

2.1.1 Wahl der passenden Javascript Bibliothek für das KNN

Bei der Recherche stellte sich heraus, dass es bereits einige Javascript Bibliotheken gibt, mit denen KNN erstellt und lokal im Browser trainiert werden können. In die engere Auswahl fielen *brain.js*, *Mind*, *Neataptic* und *Synaptic*. Für jede Bibliothek wurde ein Testprogramm geschrieben, um sie besser miteinander vergleichen zu können, wobei jedes Programm dieselben Trainingsbedingungen hat. Grob zusammengefasst, besteht jeder Trainingsdatensatz aus zwei Punkten, denen eine Klasse zugeordnet wurde. Für das Training wird bei jedem eine bestimmte Topologie festgelegt und eine bestimmte Anzahl an Iterationen durchgeführt. Am Ende des Trainings sollte angegeben werden, wie die berechneten Ausgabewerte der Punkte aussehen und wie lange das Training gedauert hat. Die Testprogramme befinden sich in der CD im Ordner *NNBibTests*.ⁱ

Folgende Punkte waren wichtig für die Wahl der Bibliothek gewesen:

- die Möglichkeit, ein MLP erstellen zu können
- die Handhabung mit der Bibliothek (z.B. wie aufwändig ist es, den Code zur Erstellung des MLP oder des Trainings zu schreiben)
- die Geschwindigkeit der Berechnung
- die Effektivität des Trainings
- die Dokumentation der Bibliothek
- die Übersichtlichkeit und Struktur des Codes der Bibliothek
- zu einem geringen Teil die Aktivität an der Bibliothek, also wie oft die Bibliothek aktualisiert wurde und wann die letzte Aktualisierung stattfand

ⁱEbenfalls wurde aus Interesse zum Vergleich für die Bibliothek *ConvNetJS* ein Testprogramm geschrieben, die bei der Berechnung deutlich am präzisesten und schnellsten war, allerdings werden dort keine MLP zur Berechnung genutzt, sondern sogenannte *Convolutional Neural Networks*, eine besondere Form von Feedforward Netzen.

In der Handhabung konnten `brain.js` und `Mind` ganz gut punkten, so ist es schon mit wenigen Zeilen Code möglich, ein KNN zu erstellen. Allerdings sind die Konfigurationsmöglichkeiten des KNN mit den beiden Bibliotheken eher beschränkt und beide bieten eine eher dürftige Dokumentation. Bei `Neataptic` handelt es sich um eine Bibliothek, die für bestimmte Teile des Codes die `Synaptic` Bibliothek verwendet hat. Von allen betrachteten Bibliotheken bietet sie auch mit einem Wiki aus 30 Seiten die beste Dokumentation und bietet gegenüber `Synaptic` Extras wie die Visualisierung der Topologie des KNN, mehr implementierte Aktivierungsfunktion etc. Allerdings handelt es sich um Extras, die für den eigentlichen Zweck des Programms nicht notwendig sind. Schlussendlich fiel die Wahl auf `Synaptic`, deren ausschlaggebendster Vorteil die Möglichkeit ist, das Training über `Web Worker` durchführen lassen zu können. Was `Web Worker` sind und weshalb sie einen Vorteil für die Anwendung bringen, soll im Kapitel 2.1.2 erläutert werden.

2.1.2 Performancevorteile durch Nutzung von Web Workern

Problem der Synchronität und des Single-Threadings

Allgemein kann Javascript als eine Programmiersprache betrachtet werden, die normalerweise *synchron* und *single-threaded* läuft. Single-threaded bedeutet, dass zur gleichen Zeit nur ein Prozess stattfinden kann ([Brij 2015](#)). Von synchroner Programmierung spricht man, wenn der Start von einem Prozess das ganze Programm solange zum Stoppen bringt, bis der Prozess zu Ende ausgeführt wurde. Das bedeutet also, dass bei Javascript die Prozesse sequenziell bzw. nacheinander abgearbeitet werden.

Bezogen auf die Anwendung für die Bachelorarbeit stellt diese Eigenschaft ein ungünstiges Problem da. Für das Training werden komplexe Berechnungen durchgeführt, besonders bei der Backpropagation finden viele Berechnungsschritte statt. Das Training des MLP mit der Methode `train()` der `Synaptic` Bibliothek führte dazu, dass die Bedienung der GUI während des Trainings sehr träge wirkte. Bis auf Interaktionen vom Anwender wie ein Mausklick auf einen Button reagiert wurde, vergingen teilweise mehrere Sekunden.

Multithreading mithilfe von Web Workern

Es musste eine passende Technik gefunden werden, die Nebenläufigkeit bzw. Multithreading in Javascript zu ermöglichen, d.h. mehrere Prozesse gleichzeitig ausführen lassen zu können. Denn um dem Anwender eine reaktionsschnelle GUI bieten zu können, müssen die Berechnungen für das Trainings nebenläufig laufen. Es gibt verschiedene Wege die Nebenläufigkeit in Javascript nachzuahmen, wie die Verwendung von `setTimeout()`, `setInterval()`, `XMLHttpRequest` oder Ereignis-Handlern ([Bidelman](#)

2010). Allerdings findet bei diesen Techniken alles immer noch im selben Hauptthreadⁱⁱ im Browser statt, so wechseln sich lediglich die Prozesse der Skriptausführungen mit den Aktivitäten der GUI ab. Web Worker sind eine Javascript APIⁱⁱⁱ für HTML5, mit der tatsächlich im Hintergrund Skripte über mehrere Threads ausgeführt werden können.

Zum Testen der Geschwindigkeit gibt es für die Synaptic Bibliothek im Ordner *synaptic-withoutWW* ein Testprogramm, welches das Training normal ohne einen Web Worker ausführt und im Ordner *synaptic-withWW* ein Testprogramm, welches das Training mit einem Web Worker ausführt. Auf dem getesteten Rechner fiel beim Vergleichen der Zeiten auf, dass sogar mit dem Web Worker eine bessere Zeit erzielt wurde (ca. 750 ms gegenüber ca. 1000 ms ohne Web Worker). Bei der Implementierung des MLP mit Synaptic ist auch deutlich aufgefallen, dass durch die Nutzung eines Web Workers die GUI während des Trainings deutlich schneller auf die Interaktionen des Anwenders reagiert hat.

2.1.3 Trainieren eines MLP mit Synaptic

In der Javascript Datei *neural-network.js* ist der Codeteil zu finden, in dem das MLP erstellt und trainiert wird. Es soll hier in dieser Arbeit nicht der ganze Code erläutert werden, sondern nur kurz auf den interessanten Codeteil, der für das Training eingegangen werden (siehe Listing 2.1). Bei *trainAsync()* handelt es sich um eine Methode für das Perzeptron, das Training in einem Webworker stattfinden lassen zu können. Der Parameter *rate* legt die Lernrate und *iterations* legt die maximale Anzahl an Iterationen fest, diese wurden vom Anwender zuvor in der GUI festgelegt. Der *error* gibt den Wert an, der unterschritten werden muss, bis das Training beendet werden kann. Durch *cost* wird die Fehlerfunktion festgelegt, in diesem Fall wird der MSE genommen, der im Kapitel ?? erwähnt wurde. Mit *myTrainer.trainAsync()* wird ein Promise-Objekt *promiseTrain* erstellt, der bei erfolgreicher Operation mit *promiseTrain.then()*, die Trainingsergebnisse an die Funktion *updateAndSendMessageForApp* übergibt.^{iv} Dieser führt wiederum die Operationen durchführt, damit die Ergebnisse später in der GUI visualisiert werden können.

```
1 var promiseTrain = myTrainer.trainAsync(trainingSet, {  
2   rate: messageForApp.nnConfigInfo.learningRate,  
3   iterations: iterations,
```

ⁱⁱBei einem Thread handelt es sich um eine Aktivität innerhalb eines Prozesses. Jeder Prozess besteht aus mindestens einem Thread, dem Hauptthread. Neben dem Hauptthread können im Prozess noch weitere Threads ausgeführt werden (Wolf 2009).

ⁱⁱⁱSchnittstelle zur Anwendungsprogrammierung (engl.: application programming interface)

^{iv}Das Promise-Objekt wird dazu verwendet, um asynchrone Berechnungen durchführen zu können. Genauerer zu der Technologie kann auf der Seite https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Global_Objects/Promise nachgelesen werden.

[illegible]

Listing 2.1: Codeausschnitt des Trainings eines MLP mit Synaptic

Aufbau eines Perzeptrons in Synaptic

Vereinfacht sieht der Code eines Perzeptrons in Synaptic als JSON wie in Listing 2.2 aus. Die Attribute „*input*“ und „*output*“ bekommen beide jeweils als Wert ein Layer-Objekt (siehe Listing 2.3) zugewiesen, sie stellen die Eingabeschicht und die Ausgabeschicht des MLP dar. Das Attribut „*hidden*“ bekommt als Wert ein Array zugewiesen, welches aus mehreren Layer-Objekten besteht, welches die versteckten Schichten darstellen.

```
1 {"layers" = {
2   "hidden": [...],
3   "input": {...},
4   "ouput": {...}
5 }}
```

Listing 2.2: JSON eines Perzeptron-Objekts in Synaptic

Das Listing der JSON vom Layer-Objekt ist stark vereinfacht, indem nur die Attribute und Werte angezeigt werden, die für die Visualisierung des MLP in der GUI von Bedeutung sind.

```
1 {"Layer":{:
2   "list":[
3     "Neuron":{:
4       "ID": 5,
5       "activation": 0,
6       "bias": 0.46,
7       "connections":{:
8         "projected":{:
9           "Connection":{:
10            "from":{:
11              "ID": 5,
12              ...
13            },
```



```
14         "to": {
15             "ID": 7,
16             ...
17         },
18         "weight": 6.553,
19         ...
20     },
21     ...
22 },
23 ...
24 },
25 ...
26 },
27 ...
28 ],
29 "size": 5,
30 ...
31 }
```

Listing 2.3: JSON eines Layer-Objekts

2.1.4 Schnittstelle zwischen der Netzworkebibliothek und der GUI

Dem Vorteil von Web Workern, dass rechenintensive Skripte nicht mehr im Hauptthread ausgeführt werden müssen, stehen einige Nachteile gegenüber. Im Bezug auf die Anwendung für die Bachelorarbeit sind die wichtigsten Nachteile, dass Web Worker nicht in der Lage sind, das DOM^v zu ändern und dass sie nicht auf globale Variablen und Funktionen zugreifen können ([Buckler 2013](#)). Für die Anwendung bedeutet dies, dass während des Trainingsvorgangs, indem die intensiven Berechnungen stattfinden, die GUI nicht verändert werden kann.

In seiner Bachelorthesis hat Koenecke die Vermutung aufgestellt, dass eine Implementierung des KNN in Javascript wohl keine Schnittstelle zur Kommunikation zwischen der Netzworkebibliothek und der GUI erfordert hätte ([Koenecke 2016](#): S.42). Im Grunde genommen hat er damit Recht, dies setzt jedoch voraus, dass die Berechnungen des Trainings im Hauptthread stattfinden, um währenddessen gleichzeitig die GUI verändern zu können. Wie bereits vorher erläutert, würde dies aber zu sehr die Performance verschlechtern und daher ist die Verwendung von Web Workern zu bevorzugen. Um einen Austausch der Trainingsergebnisse mit den Interaktionen der GUI zu ermöglichen, hat es sich also weiterhin angeboten die Schnittstelle zur Kommunikation beizubehalten. Zudem gewährleistet eine Trennung der Netzworkebibliothek von der GUI eine bessere Übersichtlichkeit vom Code der Anwendung. Wie auch in Koeneckes Anwendung stellt die Javascript-Datei *app.js* die Schnittstelle dar, mit

^vDOM: Document Object Model; Programmierschnittstelle für HTML und XML Dokumente

dem Unterschied, dass in dieser Anwendung kein Server und daher keine emulierten Websockets mehr benötigt werden.

Aufbau eines message-Objektes für die GUI

In der Schnittstelle wird zur Erstellung des MLP mit der Methode *newNetwork()* und zum Updaten des MLP mit der Methode *updateNetwork()* in der GUI ein message-Objekt mit einer bestimmten Struktur benötigt (siehe Listing 2.4). Sie entspricht zum größten Teil dem message-Objekt von Koenecke, aufgrund der Erweiterungen in der Anwendung wurde das message-Objekt etwas erweitert. In der *neural-network.js* wurden zur Erstellung der message-Objekte in diesem Format die Funktionen *getMessageForApp()* und *getUpdatedMessageForApp()* geschrieben.

```
1 {"message" = {
2   "bMaxIterationsReached": false,
3   "nnConfigInfo": {
4     "activationFunction": "relu",
5     "learningRate": 0.01,
6     "maxIterations": 100000
7   },
8   "id": 1,
9   "graph":{
10    "layers": [
11      {
12        "numberOfNeurons": 3,
13        "weights":{
14          "data":[
15            {0.54, 0.945, 0.435},
16            ...
17          ]
18        }
19      },
20      ...
21    ],
22    "sampleCoverage": 0,
23    "samplesTrained":0,
24    "weightChange":0
25  }
26  "output":{
27    "data": [
28      [253, 0, 0],
29      ...
30    ]
31  }
32 }}
```

Listing 2.4: JSON eines Perzeptron-Objekts in Synaptic

2.2 Erweiterungen an der GUI

2.2.1 Konfiguration der Aktivierungsfunktion und der Lernrate

Die Konfiguration des MLP wurde erweitert, dass nicht nur die Topologie, sondern auch die Aktivierungsfunktion und die Lernrate festgelegt werden kann (siehe Abb. 2.1). So kann der Anwender besser verstehen, inwiefern diese beiden Parameter das Training des Netzwerks beeinflussen können. Beide Parameter müssen beim Start festgelegt werden und können nicht während des Trainings verändert werden.

network config	
logistic ▼	?
0.001	?

Abbildung 2.1: Die Aktivierungsfunktion und die Lernrate können nun verändert werden

2.2.2 Erweiterungen beim Training

Das Feld zum Setzen der Trainingspunkte wurde um zwei Koordinatenachsen erweitert, wodurch die Trainingspunkte besser positioniert werden können (siehe Abb. 2.2). Zudem kann der Anwender durch die Achsen möglicherweise besser verstehen, dass die x - und y -Koordinate als Eingabeparameter für jede Trainingsdatensatz zu sehen sind. Ist ein Trainingspunkt ausgewählt, so hat man die Möglichkeit durch die beiden Eingabefelder x und y genaue Werte für die Koordinaten zu setzen oder den Trainingspunkt zu löschen. Die Liste darunter bietet eine weitere Übersicht, um zu veranschaulichen, welche Trainingspunkte beim Start des Trainings übergeben werden. Beim Eingabefeld *max. Iterations* kann festgelegt werden, nach wie vielen Trainingsiterationen das Training gestoppt werden kann (siehe Abb. 2.2). Es ist möglich, nach dem Stoppen des Trainings den Wert zu erhöhen und das Training fortzuführen. Auch während des Trainings kann der Wert verändert werden. Gedacht ist die Erweiterung als eine Möglichkeit, besser die Effektivität des Trainings vergleichen zu können, wenn beispielsweise eine andere Aktivierungsfunktion gewählt wurde. Durch einen Klick auf den Button *save samples* können die Trainingsdaten lokal im Webbrowser gespeichert werden, sodass man die Anwendung auch schließen kann und mit *load samples* die Trainingsdaten zu einem späteren Zeitpunkt wieder aufrufen kann.

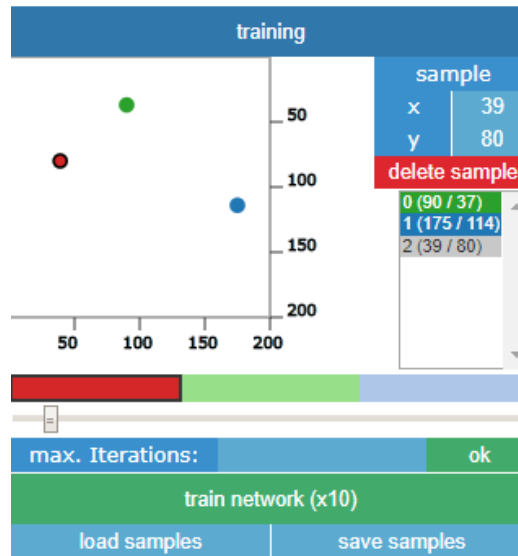


Abbildung 2.2: Das neue Trainingselement

2.2.3 Aufgaben auswählen

Der Anwender hat nun zusätzlich die Möglichkeit anhand gestellter Übungen spielerisch zu erlernen, wie das KNN für bestimmte Probleme konfiguriert werden sollte (siehe Abb. 2.3). Zu jeder Übung gibt es eine *tasklist* mit mehreren *tasks*. Wurde eine Task gelöst, so ändert sich die Textfarbe der Task (siehe Abb. 2.4). Momentan sind in der Anwendung lediglich **drei** Übungen vorhanden, die noch eher einfach gehalten sind.

select exercise

exercise 3 ▼

tasklist

logistic
relu
tanh
identity

Not all tasks solved!

reset

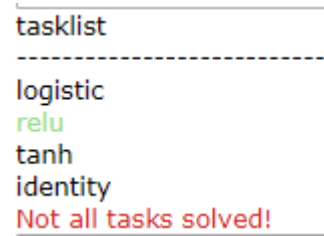
exercise description

For every activation function sample coverage should be 100%.

- logistic: after 10000 iterations-
- relu: after 400 iterations
- tanh: after 20000 iterations-
- identity: after 20000 iterations

Abbildung 2.3: Screenshot von einer Übung

2 Eigene Implementierungen



A screenshot of a tasklist interface. It features a solid horizontal line at the top, followed by the word 'tasklist'. Below this is a dashed horizontal line. Under the dashed line, the words 'logistic', 'relu', 'tanh', and 'identity' are listed vertically. The word 'relu' is highlighted in green. At the bottom of the list, the text 'Not all tasks solved!' is displayed in red. A solid horizontal line is at the very bottom of the list.

```
tasklist
-----
logistic
relu
tanh
identity
Not all tasks solved!
```

Abbildung 2.4: Der Task für die ReLu Aktivierungsfunktion wurde gelöst.

3 Konfiguration eines KNN passend zur Problemstellung

Abkürzungsverzeichnis

MLP mehrlagige Perzeptron

KI Künstliche Intelligenz

KNN künstliche neuronale Netze

GUI Grafical User Interface

Abbildungsverzeichnis

1.1	Konfiguration der Netzwerktopologie	2
1.2	Darstellung eines Netzwerkgraphs (untrainiert)	2
1.3	Darstellung eines Netzwerkgraphs (trainiert)	2
1.4	Erzeugung der Trainingsdaten	3
1.5	Start des Trainings mit 10-facher Geschwindigkeit	3
1.6	Vorschau nach einer halben Million Beispieldaten	4
1.7	Vorschau nach einer Million Beispieldaten	4
1.8	Vorschau nach drei Millionen Beispieldaten	4
1.9	Netzwerkinfo während des Trainings	4
2.1	Die Aktivierungsfunktion und die Lernrate können nun verändert werden	11
2.2	Das neue Trainingselement	12
2.3	Screenshot von einer Übung	12
2.4	Der Task für die ReLu Aktivierungsfunktion wurde gelöst.	13

Tabellenverzeichnis

Literaturverzeichnis

- Bidelman, Eric: *Web Worker-Grundlagen*, <https://www.html5rocks.com/de/tutorials/workers/basics/>, 26.07.2010, letzter Zugriff: 29. 08. 2017
- Brij: *Concurrency vs Multi-threading vs Asynchronous Programming : Explained*, <https://codewala.net/2015/07/29/concurrency-vs-multi-threading-vs-asynchronous-programming-explained/>, 29.07.2015, letzter Zugriff: 29. 08. 2017
- Buckler, Craig: *Implementing JavaScript Threading with Web Workers*, <https://www.safaribooksonline.com/blog/2013/10/24/implementing-javascript-threading-with-web-workers/>, 24.10.2013, letzter Zugriff: 29. 08. 2017
- Corves, Anna: *Nervenzellen im Gespräch*, <https://www.dasgehirn.info/grundlagen/kommunikation-der-zellen/nervenzellen-im-gespraech?gclid=C0iWg7TdzNQCFU4W0wodK74IwA>, 2012, letzter Zugriff: 1. 07. 2017
- Ertel, Wolfgang: *Grundkurs Künstliche Intelligenz : Eine praxisorientierte Einführung*, 4. Aufl., Berlin Heidelberg New York: Springer-Verlag, 2016
- Hebb, Donald Olding: *The Organization Of Behavior: A Neuropsychological Theory*, Psychology Press edition 2002, 1949
- Hillmann, Leonard: *Intelligentes Leben*, <http://www.tagesspiegel.de/themen/gehirn-und-nerven/gesund-leben-intelligentes-leben/13410564.html>, Veröffentlichungsdatum unbekannt, letzter Zugriff: 01.07.2017
- Hoffmann, Norbert *Kleines Handbuch Neuronale Netze : Anwendungsorientiertes Wissen zum Lernen und Nachschlagen*, Berlin Heidelberg New York: Springer-Verlag, 1993
- Koenecke, Finn Ole: *Realisierung eines interaktiven künstlichen neuronalen Netzwerks*, 2016
- Kramer, Oliver: *Computational Intelligence : Eine Einführung*, 1. Aufl., Berlin Heidelberg New York: Springer-Verlag, 2009.
- Manhart, Klaus: *Was Sie über Maschinelles Lernen wissen müssen*, <https://www.computerwoche.de/a/was-sie-ueber-maschinelles-lernen-wissen-muessen,3329560>, 04.05.2017, letzter Zugriff: 02.07.2017

- Minsky, Marvin; Papert, Seymour: *Perceptrons: An Introduction to Computational Geometry*, 2nd edition with corrections, first edition 1969 , The MIT Press, Cambridge MA, 1972.
- Rey, Günter Daniel ; Wender, Karl F.: *Neuronale Netze : eine Einführung in die Grundlagen, Anwendungen und Datenauswertung*, 2. vollst. überarb. und erw. Aufl., Bern: Huber, 2011.
- Riley, Tonya: *Artificial intelligence goes deep to beat humans at poker*, <http://www.sciencemag.org/news/2017/03/artificial-intelligence-goes-deep-beat-humans-poker>, 03.03.2017, letzter Zugriff: 07.07.2017
- Rimscha, Markus: *Algorithmen kompakt und verständlich : Lösungsstrategien am Computer*, 3. Aufl., Berlin Heidelberg New York: Springer-Verlag, 2014.
- Russell, Stuart ; Norvig, Peter: *Artificial Intelligence : A Modern Approach*, 3. Aufl(2010), London: Prentice Hall, 2010.
- Scherer, Andreas: *Neuronale Netze : Grundlagen und Anwendungen*. Berlin Heidelberg New York: Springer-Verlag, 1997.
- SethBling: *MarI/O - Machine Learning for Video Games*, <https://www.youtube.com/watch?v=qv6UV0Q0F44>, 13.06.2017, letzter Zugriff: 07.07.2017
- Turing, Alan M.: *Computing Machinery and Intelligence*, Mind, 59, 1950.
- Wolf, Jürgen: *C von A bis Z : das umfassende Handbuch*, http://openbook.rheinwerk-verlag.de/c_von_a_bis_z/026_c_paralleles_rechnen_003.htm Bonn: Rheinwerk Verlag GmbH, 2009, letzter Zugriff: 29.08.2017
- Wunderlich-Pfeiffer, Frank : *Alpha Go geht in Rente*, <https://www.golem.de/news/kuenstliche-intelligenz-alpha-go-geht-in-rente-1705-128059.html>, 29.05.2017, letzter Zugriff: 04.07.2017
- Zell, Andreas: *Simulation neuronaler Netze*, 4.Auflage(2003) Deutschland: Oldenbourg Wissenschaftsverlag GmbH, 1994.