HUMBOLDT-UNIVERSITÄT ZU BERLIN

Projekt 4

Neuronale Netze: Das Hopfield Modell

Felix Andreas 560403

Wissenschaftliches Rechnen / CP II

Abgabe: 31.03.2018

Inhaltsverzeichnis

Vorbemerkung

Alle Aufgaben wurden in **Python3** programmiert und können mit

```
python3 aufgabe4_x.py
```

ausgeführt werden.

Zusätzlich wurde ein **GUI** implementiert, um die Funktionsweise des Hopfield Netzes testen zu können. Das Hopfield Network GUI kann mit

```
python3 gui.py
```

gestartet werden. Nährere Informationen zum GUI sind in Unterabschnitt Python GUI gegeben.

Der Source Code befindet sich im Ordner **code**. Alle Skripte wurden auf Debian 8.10 (jessie) im HU Pool getestet.

Einführung

Ein Hopfield Netzwerk ist eine spezielle Art eines künstlichen neuronalen Netzwerkes. Ziel des Hopfield Netzes ist die assoziative Speicherung von Mustern. Dies bedeuted, dass Speicherinhalt nicht über eine Speicheradresse erreicht werden soll, sondern dass das Netzwerk auf ein Eingabemuster mit dem gespeicherten Muster antwortet, welches die größte Ähnlichkeit aufweist. Hopfield-Netze können somit genutzt werden, um ein verrauschtes oder nur in Teilen vorhandenes Muster zu rekonstruieren. Dies könnte natürlich auch über die serielle Berechnung der Abstände des Eingabemusters zu den gespeicherten Mustern realisiert werden. Hierbei geht es aber darum die genannte Funktionsweise durch ein neuronales Netz zu implementieren.

Netzwerkstruktur

Das Hopfield Netz besteht aus einer einfachen Schicht von Neuronen S_i, welche binäre Werte annehmen können. Jedes Neuron ist mit jedem anderen Neuron verbunden. Diese Verbindung ist durch die symmetrischen Gewichte $w_{ij} = w_{ji}$ gegeben. Mit jedem Zeitschritt evolviert das Netz gemäß der McCulloch-Pitts Gleichung

$$S_i(t+1) = sgn(\sum_j w_{ij} S_j(t) - \theta_i), \tag{1}$$

welche zur Minimierung der Energiefunktion

$$H = -\frac{1}{2} \sum_{ij} w_{ij} S_i S_j + \sum_i \theta_i S_i \tag{2}$$

führt.

Die Aktualisierung des Netzes kann dabei über synchron oder asynchron erfolgen. Wird das Hopfield Netz asynchron aktualisiert, ist garantiert, dass Konvergenz erreicht wird, da H im jedem Schritt kleiner wird oder gleich bleibt. Im synchronen Modus kann es zu sogenannten Oszillationen kommen. Die Ursache für diese sowie die Vorteile/Nachteile der beiden Update-Modi werden in Aufgabenteil 4.2 diskutiert.

Speichern von Mustern

Mithilfe der Hebb'schen Lernregel können Muster im Netzwerk gespeichert werden:

$$w_{ij} = \frac{1}{N} \sum_{\mu=1}^{p} xi_i^{(\mu)} xi_j^{(\mu)} \tag{3}$$

Spurious states

Zusätzlich zu den gespeichterten Bildern kommt es zu weiteren Nebenminima, den sogenannten **spurious states**. Diese lassen sich in drei Kategorien unterteilen:

- Erstens ist auch das Negative jedes Bildes ein Minimum (**reversed states**).
- Zweitens ist auch jede ungerade Linearkombination von Bildern (**mixture states**) ein Nebenminimum:

$$\xi_i^{\text{mix}} = sgn(\pm\xi_i^{(\mu_1)} \pm \xi_i^{(\mu_2)} \pm \xi_i^{(\mu_3)} \pm \dots) \tag{4}$$

- Drittens kann es bei großen p zu lokalen Minima kommen, die weder eine Linearkombination noch anders korreliert zu den gespeicherten Bildern sind (**spinglass states**).

Da die **spurious states** im Mittel einen relativ kleinen **basin of attractions** besitzen , kann das Hopfield Netz dennoch als Assoziativspeicher genutzt werden.

Endliche Temperaturen

Um die Wahrscheinlichkeit in ein Nebenminima zu konvergieren zu verringern, gibt es die Möglichkeit ein sogenanntes stochastisches Rauschen in die deterministishen McCulloh-Pitts Gleichung einzuführen. Diese sogenannten endlichen Temperaturen können dazu führen, dass die erreichten Nebenminima überwunden werden und in eines der gespeicherten Bilder konvergiert werden. Allerdings führen die endlichen Temperaturen auch dazu, dass keine exakte Konvergenz mehr erreicht werden kann, da immer eine Fluktuation möglich ist.

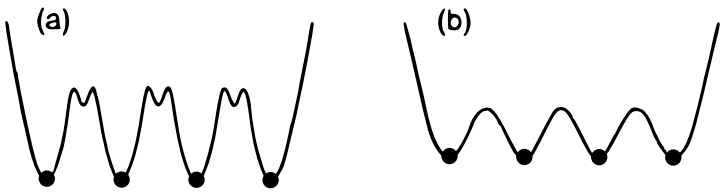

Abbildung 1: Schematische Darstellung der **energy landscapes** für $p << N$. (a) Die mixture states können sich als lokale Minima zwischen den gespeicherten Bildner vorgestellt werden. Die eigentlichen Zustände sind natürlich Eckpunkte im N-dimensionalen Hyperwürfel. (b) Bei hoher Temperature verschieden die "mixture states". (Quelle: [2])

Zur Vorbereitung die wurden [1] und [2] herangezogen.

Implementierung des Hopfield Modells

Für die Implementierung Hopfield Modells in Python wurde eine Klasse erstellt:

```
class HopfieldNetwork:
    def __init__(self, N=100, filepath=None):
        ...
```

Im Folgenden sind die wichtigsten Funktionen und die Anwendung der **HopfieldNetwork** Klasse gezeigt werden.

Wichtige Funktionen

Energiefunktion:

$$H = -\frac{1}{2}\sum_{i,j}^{N} w_{ij} S_i S_j \tag{5}$$

```
def compute_energy(self, S):
    return -0.5 * np.einsum('i,ij,j', S, self.w, S)
```

Hebb-Matrix:

$$w_{ij} = \frac{1}{N}\sum_{\mu=1}^{p} \xi_i^{(\mu)} \xi_j^{(\mu)} \tag{6}$$

```
def construct_hebb_matrix(xi):
    n = xi.shape[0]
    if len(xi.shape) == 1:
        w = np.outer(xi, xi) / n  # p = 1
    elif len(xi.shape) == 2:
        w = np.einsum('ik,jk', xi, xi) / n  # p > 1
    else:
        raise ValueError("Unexpected shape of input pattern xi: {}"\
        .format(xi.shape))
    np.fill_diagonal(w, 0)  # set diagonal elements to zero
return w
```

Hamming-Metrik:

$$H(x,y) = \#\{i : x_i \neq y_i\} \tag{7}$$

```
def hamming_distance(x, y):
    return np.sum(x != y)}
```

Anwendung

Die **HopfieldNetwork** Klasse kann wie folgt genutzt werden:

Erstelle ein neues Hopfield Netz der Größe $N = 100$:

```
hopfield_network1 = HopfieldNetwork(N=100)
```

Öffne eines bereits trainiertes Hopfield Netz:

```
hopfield_network2 = HopfieldNetwork(filepath='network2.npz')
```

Speichere ein Netzwerk als Datei:

```
hopfield_network3.save_network('path/to/file')
```

Speichere/Trainiere Bilder ins Hopfield Netz:

```
hopfield_network1.train_pattern(input_pattern)
```

Starte ein asynchrones Update mit 5 Iterationen:

```
hopfield_network1.update_neurons(iterations=5, mode='async')
```

Berechne die Energiefunktion eines Bildes:

```
hopfield_network1.compute_energy(input_pattern)
```

Test des Hopfield Modells

Zum Test der Implementierung wurde ein Hopfield Netz mit 10000 Neuronen initializiert und mit Bildner von 10 bekannten Physikern trainiert. Als Startkonfiguartion wurde einmal ein Teil eines gespeichterten Bilders und einmal eine verrauschte Version eines gespeichterten Bilders gewählt. Danach wurden zwei asynchrone Updates ausgeführt. Die Neuronenkonfigurationen zu den jeweiligen Zeitschritten (und der Hammingabstand zum gespeichtern Bild) sind in Abbildung 2 und in Abbildung 3 zu sehen.

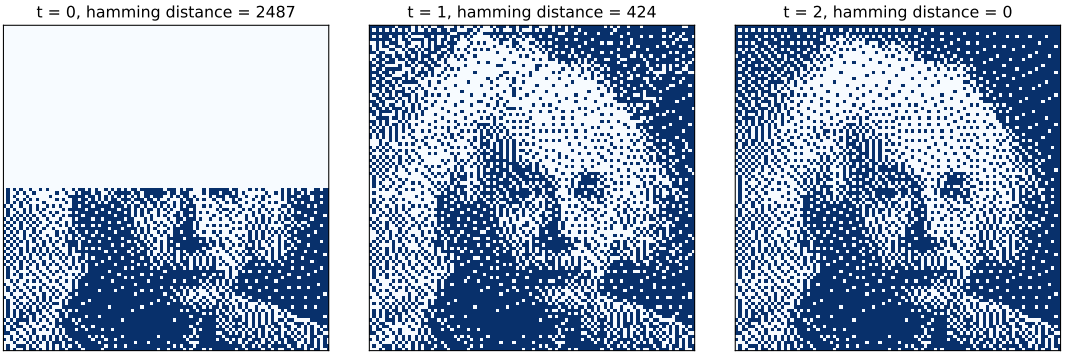

Abbildung 2: Bildrekonstruktion aus einem Teilbild. Für das Hopfield Netzwerk wird ein Teil eines gespeichterten Bildes als Startkonfiguartion der Neuronen gewählt. Nach zwei Iterationen evolviert das Netzwerk zum gespeichert Bild. Der angegebene Hammingabstand bezieht sich dabei auf das gespeicherte Bild.

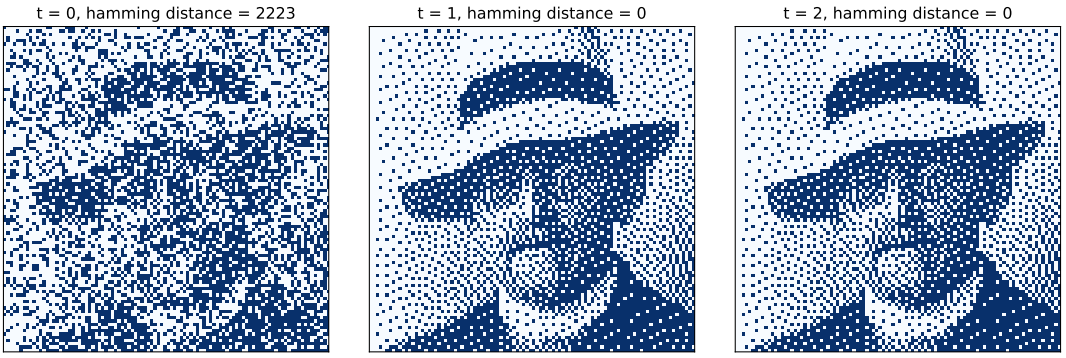

Abbildung 3: Bildrekonstruktion aus einem verauschtem Bild. Für das Hopfield Netzwerk wird ein verauschte Version eines gespeichterten Bildes als Startkonfiguartion der Neuronen gewählt. Nach zwei Iterationen evolviert das Netzwerk zum gespeichert Bild. Der angegebene Hammingabstand bezieht sich dabei auf das gespeicherte Bild.

Python GUI

Im Hopfield network GUI werden die eindimensionalen Vektoren der Neuronenzustände als zweidimensionales binäres Bild visualisiert. Der Nutzer hat die Möglichkeit verschiedene Bilder in das Netzwerk zu laden und dieses anschließend asynchron oder synchron mit oder ohne endliche Temperaturen zu aktualisieren. Um die Funktionsweise des Netzes direkt zu prüfen, können im Reiter **Examples** verschiedene vorgespeichterte Netzwerke (10 Bilder bekannter Physiker, 20 zufällige Muster, ...) geladen werden. Im Folgenden soll ein kurzer Überblick über die Bedienung sowie die möglichen Funktionen des GUIs geschaffen werden.

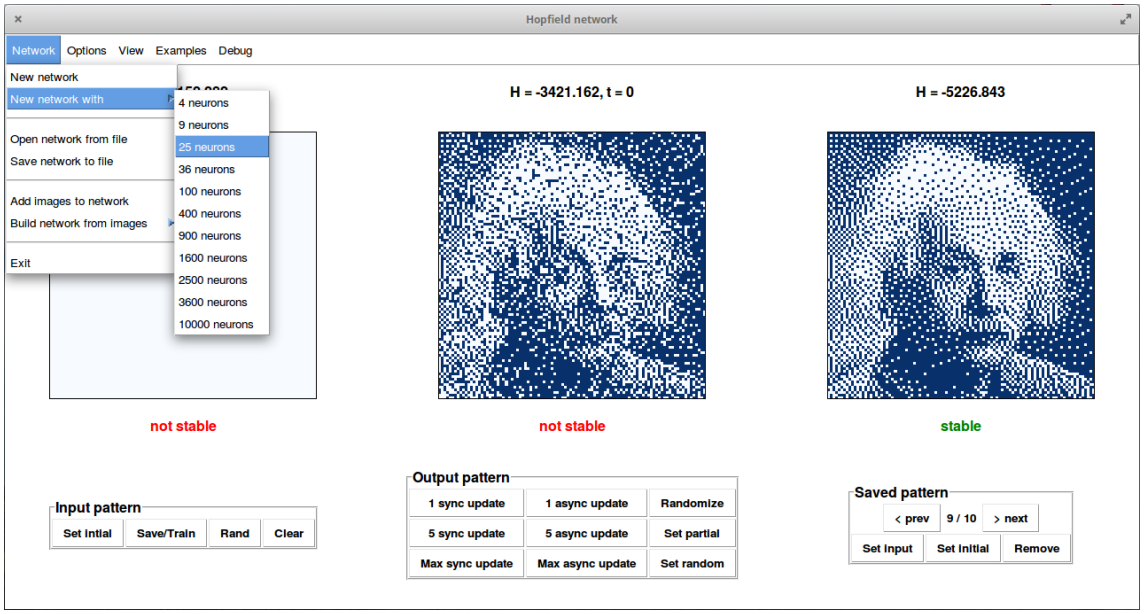

Abbildung 4: Screenshot des Hopfield network GUI.

Das Hopfield Network GUI kann mit

```
python3 gui.py
```

gestartet werden. Das Hopfield network GUI lässt sich in 3 sogenannten Frames aufteilen:

- Der **Input frame** (links) ist der Hauptinteraktionspunkt mit dem Netzwerk. Der Nutzer hat die Möglichkeit hier ein Inputmuster durch einen Linksklick auf +1, entsprechend durch einen Rechtsklick auf -1 zu setzen. Dies hat erstmal keinen Einfluss auf das Netzwerk. Erst mit den darunter liegenden Buttons kann das Netzwerk verändert werden:
  - **Set intial** setzt das momentane Inputmuster als Startkonfiguaration der Neuronen.
  - **Save/Train** speichert/trainiert das momentane Inputmuster in das Hopfield Netz.
  - **Rand** setzt ein zufälliges Inputmuster.
  - **Clear** setzt alle Punkte des Inputmusters auf -1.
- Der **Output frame** (mittig) zeigt die momentane Neuronenkonfiguartion an. Mit den darunter liegenden Buttons kann das Netzwerk asynchron oder synchron aktualisiert werden.
  - **Sync update** startet ein synchrones Update.
  - **Async update** startet ein asynchrones Update.
  - **Randomize** flippt zufällig den Zustand von einem Zehntel der Neuronen.
  - **Set partial** setzt das erste Hälfte der Neuronen auf -1.
  - **Set random** setzt einen zufälligen Neuronenzustand.
- Der **Saved pattern frame** (rechts) bietet die Möglichkeit die bereits im Netzwerk gespeicherten Bilder anzuschauen oder aus dem Netzwerk zu entfernen.
  - **Set input** setzt das momentan angezeigte Bild als neuen Neuronenzustand.

  - **Set input** setzt das momentan angezeigte Bild als Inputmuster.

Menuleiste des GUI:

- Im Reiter **Network** kann eine neues Hopfield Netz beliebiger größe initializiert werden. Darüberhinaus besteht die Möglichkeit das momentane Netz zu speichern sowie ein gespeichertes Netz zu laden. Außerdem kann eine Rastergrafik (JPG,PNG,GIF,TIF) in das Netzwerk gespeichert werden oder direkt ein neues Netzwerk aus mehreren Bildern zu erstellen.
- Im Reiter **Options** kannd das Update mit endliche Temperaturen (de)aktiviert werden.
- **View** bietet einge Möglichkeiten zur optischen Veränderung der GUI.
- Im Reiter **Examples** können einige verschieden Beispiel Netze geladen werden.

Aufgabe 4.1

Berechnung der Fehlerrate

In diesem Aufabenteil sollte ein Hopfield Modell mit 100 Neuronen implementiert werden. In Abhängigkeit von $p/N = 0.1, 0.2, 0.3$ sollte die Fehlerraten nach einer Iteration und nach Erreichen des Fixpunktes berechnet werden, indem man einmal von jedem exakten Bild startet. Dies wurde sowohl für ein asynchrones als auch für ein synchrones Update durchgeführt. Um ein statistisch aussagekräftigere Fehlerate zu berechnen, wurden dabei für jedes p/N 1000 Sets zufällige Bildern gewählt und daraus der Mittelwert bestimmt. In Abbildung 5 sind die Fehlerraten in Abhängigkeit von p/N dargestellt.

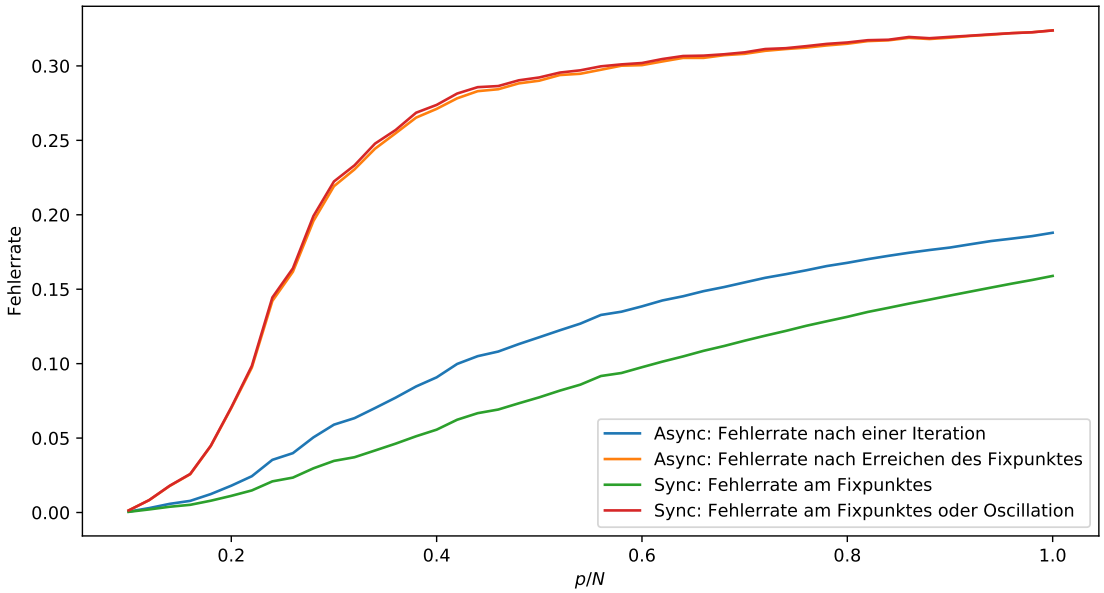

Abbildung 5: Die Fehlerrate in Abhängigkeit von $p/N = 0.1 \ldots 1.0$. Hierbei wurde einmal ein asynchrones Update und einmal ein synchrones Update gewählt und der Fehler sowohl nach einer Iteration als auch nach Erreichen eines Fixpunktes (bei synchronen Update auch nach Erreichen einer Oszillation) dargestellt. Dabei wurden für jedes p/N zufällige Sets von Bildern gewählt und der Mittelwert bestimmt.

Es ist zu sehen, dass mit zunehhmenden p/N die Fehlerrate des asynchronen und synchronen Updates sowohl nach einer Iteration als nach Erreichen des Fixpunktes zunimmt. Dies war zu erwarten, da mit Anzahl der Bilder auch die Anzahl der Nebenminima steigt. Auffällig ist außerdem, dass die Fehlerrate ab $p/N = 0.3$ für das asynchrone Update nach einer Iteration etwa 5 Prozent über der des synchronen Updates liegt.

Die Fehlerrate nach Erreichen eines Fixpunktes (bei synchronen Update auch nach Erreichen einer Oszillation) steigt für beide Update Modi von $0.2 < p/N < 0.3$ etwa linear an und konvergiert dann bei $p/N = 1.0$ gegen 0.35. Es ist außerdem zu sehen, dass die Fehlerrate des synchronen Updates nach Erreichen des Fixpunktes im Bereich $0.2 < p/N < 0.3$ minimal unter der des asynchronen Updates liegt. Die Ursache dafür konnte bei den eintretenden Oszillationen liegen, aber könnte auch an nicht auszureichender Statsitik liegen.

Außerdem wurde geprüft wie sich die Anzahl der Iterationen bis zum Erreichen eines Fixpunktes (bei synchronen Update auch nach Erreichen einer Oszillation) mit p/N ändert. Der Zusammenhang ist in Abbildung 6 geplotted.

Auch hier nimmt die Anzahl der Iterationen bis zum Erreichen des Fixpunktes/Oszillation wie erwartet zu. Zu erkennen ist, dass die durchschnittlichen Iterationen bis zum Erreichen eines Fixpunktes/Oszillation beim synchronen Update deutlich größer sind als beim asynchronen Modus. Auch dies war zu erwarten, da im synchronen Modus mehrere Neuronen gleichzeitig "denken"können, dass Sie durch ihre Änderung die Energie minimiert wird, aber es zu ein tatsächlich größeren Energie kommt. Im asynchronen Modus ist hingegen sichergestellt, dass jede Änderung zu einer Minimierung der Energiefunktion führt. Genauers dazu im folgenden Unterabschnitt 4.2.

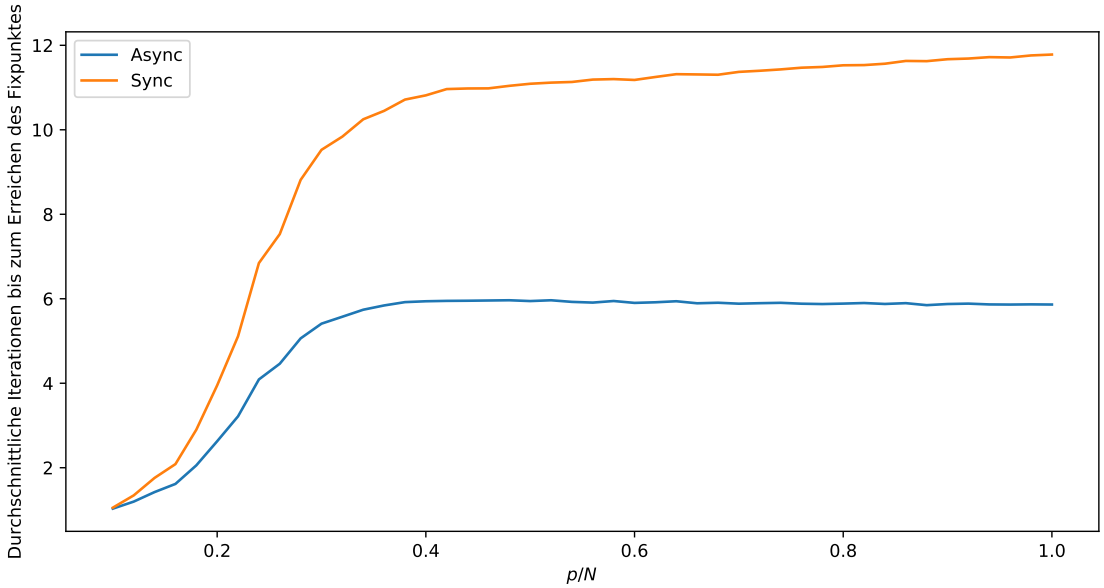

Abbildung 6: Die Anzahl der benötigten Iterationen zum Erreichen eines Fixpunktes (bei synchronen Update auch nach Erreichen einer Oszillation) in Abhängigkeit von $p/N = 0.1 \ldots 1.0$.

Synchrones Update

Nun sollte erklärt werden, was sich bei dem synchronen Update ändert: Der wesentliche Unterschied zum asynchronen Update ist, wie der Name schon sagt, dass die Neuronen synchron aktualisiert werden. Dies kann im Unterschied zum asynchronen Update (hier werden die Neuronen nacheinander aktualisiert) zu einer Oszillation der Neuronen führen, was am folgenden Beispiel verdeutlicht werden soll (siehe Abbildung 7):

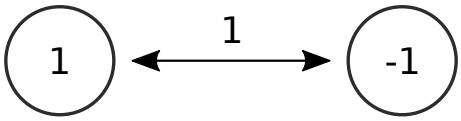

Abbildung 7: Hopfield Netz aus 2 Neuronen. Im synchronen Modus wird hier keine Konvergenz erreicht: Die Neuronen oszillieren unendlich lange mit der Periode 2.

Wir betrachten ein Netzwerk aus zwei Neuronen mit der Synapsen-Matrix $w_{12} = w_{21} = 1$. Wird das Netz asynchron aktualisiert, so wird einer der beiden stabilen Fixpunkte $(1, 1)$ und $(-1, -1)$ erreicht. Im synchronen Modus sind diese Fixpunkte zwar auch stabil, allerdings besitzen diese kein "basis of attraction". Startet das Netzwerk synchron im Zustand $(1, -1)$ und $(-1, 1)$ führt dies dazu, dass beide Neuronen die Energiefunktion gleichzeitig minimieren "wollen", aber es zu einem effektiven Anstieg der Energie kommt. Die Neuronen "flippen" ihr Vorzeichen und es kommt zu einer Unendlichen Oszillation. Diese Oszillationen besitzen dabei immer die Periode 2.

Dies ist auch nochmal für ein Hopfield Netzwerk im 4 Neuronen simuliert worden und ist in Abbildung 8 dargestellt:

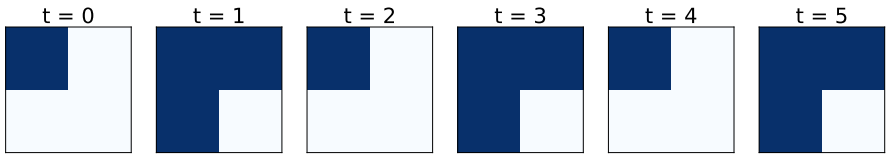

Abbildung 8: Hopfield Netz aus 4 Neuronen mit den trainierten Zuständen $(1, 1, -1, -1)$ und $(1, -1, 1, -1)$. Im synchronen Modus wird hier keine Konvergenz erreicht: Die Neuronen oszillieren unendlich lange mit der Periode 2.

Der Code zu Aufgabe 4.1 befindet sich in **aufgabe4_1.py**.

Aufgabe 4.2

Spurious states

In diesem Aufgabenteil sollten für zufällige Neuronenkonfigurationen die Anzahl der **spurious sates** in Abhängigkeit von p/N gezählt werden. Dabei wurde eine Fehlerrate unter 5 Prozent als Konvergenz zu einem gespeicherten Bild gezählt. Außerdem wurde auch das negative Bild $\xi_i^{(\mu)}$ gezählt. Die Wahrscheinlichkeit in ein spurious state zu konvergieren P(spurious states) wurde in Abhängigkeit von p/N in Abbildung 9 geplotted. Hierbei wurde für jedes p/N 100 verschiedene 100-Neuronen-Netzwerke mit jeweils 1000 verschiedenen Neuronenkonfigurationen gestartet.

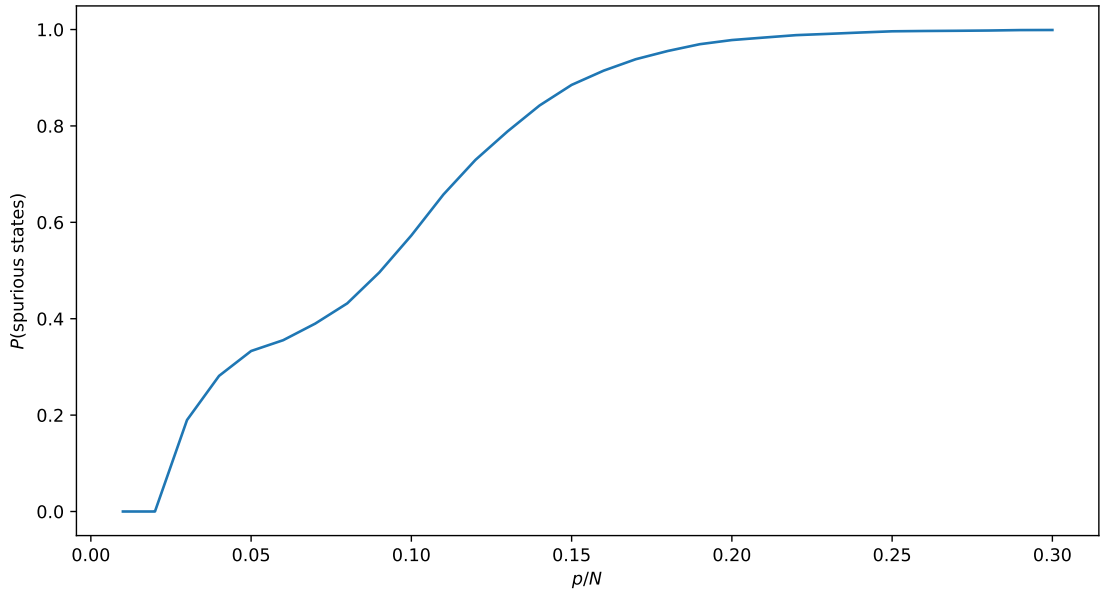

Abbildung 9: Wahrscheinlichkeit P(spurious states) in ein spurious state zu konvergieren in Abhängigkeit von p/N.

Wie zu erkennen, steigt mit zunehmender Anzahl gespeicherter Bilder auch die Wahrscheinlichkeit in ein spurious state zu konvergieren. Bei $p/N = 0.1$ liegt diese schon bei 50 Prozent und konvergiert ab etwa $p/N = 0.2$ gegen 100 Prozent. Dies lässt sich damit erklären, dass mit p/N die Anzahl der Nebenminima steigt und somit der basin of attraction der gespeicherten Bilder verkleinert wird. Bei sehr großer Anzahl von Bildern sind die gespeicherten Bilder selbst auch keine Minima mehr. Dies führt dazu, dass die Wahrscheinlichkeit zur Konvergenz in ein spurious state auf 100 Prozent steigt. Hierbei sei noch anzumerken, dass die genaue Form des Graphen natürlich von der willkürlichen Definition, wann ein Fixpunkt als spurious state zu zählen ist (hier über 5 Prozent Fehlerrate), abhängig ist.

Der Code zu Aufgabe 4.2.1 befindet sich in **aufgabe4_2_1.py**.

Endliche Temperaturen

In diesem Aufgabenteil sollten endliche Temperatur für $\beta < \infty$ so implementiert werden, dass die Tendenz zu den 'spurious states' abnimmt. Da bei endlichen Temperaturen keine exakte Konvergenz mehr einsetzt, muss ein anderes Abbrechkriterium gewählt werden. Um einen sinnvollen Vergleich zu dem Update ohne endliche Temperaturen zu ziehen sollten beide Varianten mit der selben Anzahl von Iterationen ausgeführt werden. Wie man in Abbildung 6 sieht, liegt die durchschnittliche Dauer bis zur Konvergenz bei maximal 12 Iterationen. Deshalb wurde entschieden die Fehlerrate nach 20 Iterationen zu messen, da somit nahezu sichergesellt ist, dass das Netzwerk ohne endliche Temperaturen in ein Fixpunkt konvergieren kann. Auch hier wurde eine Fehlerrate unter 5 Prozent als Konvergenz zu einem gespeicherten Bild gezählt.

Die Wahrscheinlichkeit in ein spurious state P(spurious states) nach 20 Iterationen zu erreichen wurde für verschiedene inversere Temperaturen β in Abhängigkeit von p/N in Abbildung 9 geplotted.

Wie zu erkennen ist die Wahrscheinlichkeit in ein spurious state zu konvergieren für ein Update mit $\beta = 2$ für den Großteil der Verhältnisse p/N deutlich größer als bei einem Update ohne

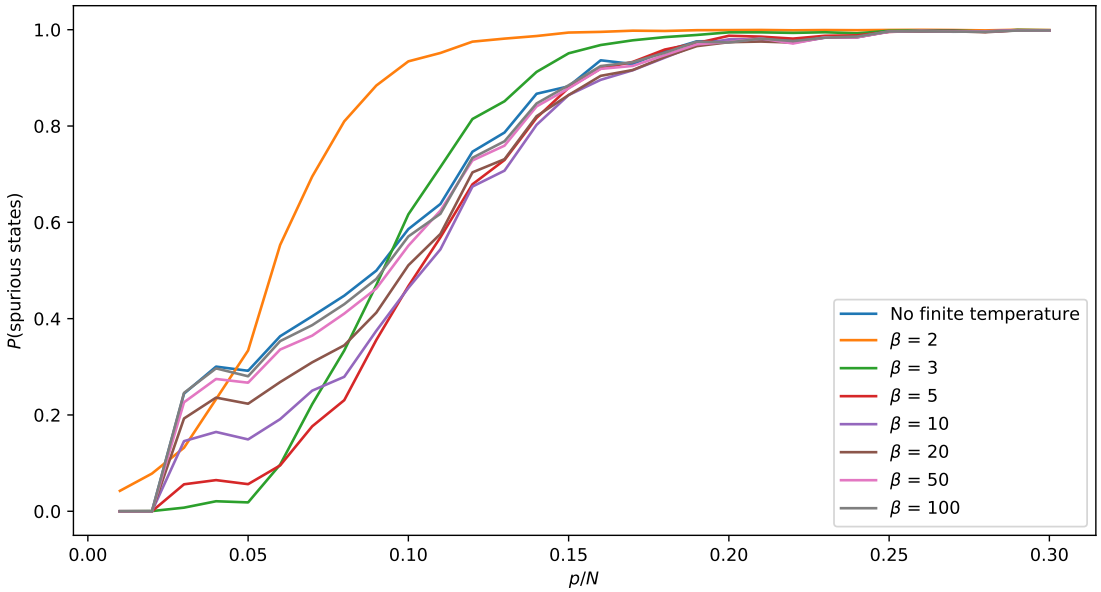

Abbildung 10: Wahrscheinlichkeit P(spurious states) in ein spurious state zu konvergieren in Abhängigkeit von p/N.

endliche Temperaturen. Dies lässt sich damit erklären, dass die Fluktuationen noch zu stark sind. Für $\beta = 3$ ist die Wahrscheinlichkeit bis $p/N < 0.08$ schon deutlich unter der ohne endlicher Temperatur. Für $\beta = 3$ wird ein Optimum erreicht: Hier liegt die Wahrscheinlichkeit in ein spurious state zu konvergieren immer unter der ohne endliche Temperaturen. Bei $p/N = 0.05$ beträgt diese bei etwa nur ein Zehntel und bei $p/N = 0.1$ bei immerhin noch 80 Prozent der des Updates ohne endliche Temperaturen. Für $p/N > 0.15$ sind beide identisch und konvergieren ab $p/N = 0.25$ gegen 100 Prozent.

Mit zunehmneden β nähert sich der Verlauf der Wahrscheinlichkeit in ein spurious state zu konvergieren asymptotisch dem ohne endliche Temperaturen. Für $\beta = 100$ sind beide Verläufe nahezu identisch. Dies lässt sich noch einmal am Graphen der Sigmoid-Funktion

$$P(S_i = \pm 1) = \frac{1}{1 + exp(\mp 2\beta h_i)} \tag{8}$$

verdeutlichen (siehe Abbildung 11): Für kleine β ist der Graph relativ flach, was ein großen Fluktuation entspricht. Im Grenzfall $\beta \to 0$ wird das Update komplett zufällig. Im anderen Grenzfall $\beta \to 0$ wird die Sigmoid-Funktion zur Signum-Stufenfunktion, was der deterministischen McCulloch-Pitts Gleichung entspricht.

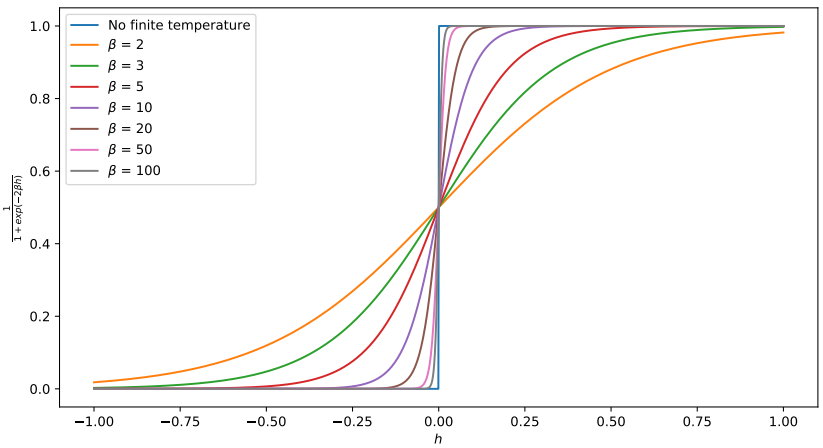

Abbildung 11: Die Sigmoidfunktion für verschiedene Werte von β.

Der Code zu Aufgabe 4.2.2 befindet sich in **aufgabe4_2_2.py**.

Literatur

[1] Ulli Wolf und Burkhard Bunk, Computational Physis II, Humboldt University of Berlin, 2003 (siehe S. 3).

[2] A. Krogh J. Hertz und R.G.Palmer, Introdution to the Theory of Neural Computation, Addison-Wesley, 1990 (siehe S. 3).