Universität Augsburg

Mathematisch-Naturwissenschaftlich-Technische Fakultät

Lehr- und Forschungseinheit Angewandte Analysis

Masterarbeit zum Thema:

Tiefes bestärkendes Lernen: Grundlagen, Approximationseigenschaft und Implementierung multimodaler Erklärungen

von

Tobias Huber

Sommersemester 2018
Alter Postweg 6, 86159 Augsburg
Master Mathematik, 4tes Fachsemester
Matrikelnummer: 1322474
Betreut von Prof. Dr. Malte Peter

Inhaltsverzeichnis

Notation

Bevor ich beginne, möchte ich kurz einige Notationsgrundlagen klären, die nicht a priori eindeutig sind.
Mit $\mathbb{N}$ bezeichne ich in dieser Arbeit die natürlichen Zahlen ohne 0, also $\{1, 2, 3, 4...\}$.
Seien X und Y topologische Räume, dann bezeichnet $C(X, Y) = C^0(X, Y)$ die stetigen Funktionen von X nach Y und $C^n(X, Y)$, mit $n \in \mathbb{N}$, steht für die n-mal stetig differenzierbaren Funktionen. Falls $Y = \mathbb{R}$ gilt, schreibe ich auch $C^n(X, \mathbb{R}) = C^n(X)$.
Weiter sind $C_0^n(X, Y)$ die Funktionen aus $C^n(X, Y)$, welche kompakte Träger besitzen.
Ist X ein metrischer Raum mit Metrik d und $x \in X$, dann beschreibt $B_\epsilon(x) = \{y \in X | d(x, y) < \epsilon\}$ den offenen ϵ-Ball um x.
Seien $v, w \in \mathbb{R}^n$ dann schreibe ich $v \cdot w$ für das Skalarprodukt $\sum_{i=1}^n v_i w_i$. Mit $\mathbb{R}^{n \times m}$ bezeichne ich den Vektorraum der Matrizen mit n-Reihen, m-Spalten und reellen Einträgen.
Für zwei Mengen X, Y schließe ich mit $X \subset Y$ den Fall $X = Y$ nicht aus. Echte Teilmengen beschreibe ich mit $X \subsetneq Y$.
Alle Begriffe, die ich mathematisch präzise definieren kann, werden **fett** markiert und Namenskonventionen *kursiv* geschrieben.

1 Einleitung

Tiefes bestärkendes Lernen bezeichnet das Trainieren von künstlichen neuronalen Netzen mit Methoden des bestärkenden Lernens.
Ein künstliches neuronales Netz ist ein lernendes System, welches, motiviert durch das menschliche Gehirn, aus künstlichen Neuronen zusammengesetzt wird. Anhand einer vorgegebenen Menge an korrekten Input-Output Paaren „lernt“ ein solches Netz, wie es die einzelnen Neuronen gewichten muss, um ein vorgegebenes Problem zu lösen.
Bestärkendes Lernen beschreibt Methoden, bei denen einem lernenden System nicht direkt vorgegeben wird, welche Aktionen in einer bestimmten Situation richtig sind. Stattdessen erhält das System in jedem Zeitschritt eine Belohnung und versucht die gesamte, innerhalb einer Episode erhaltene, Belohnung zu maximieren.
Beim tiefen bestärkenden Lernen speichert man hierzu die Gewichte des bisher gelernten Netzes in einem zweiten Zielnetz, welches periodisch, nach einer bestimmten Anzahl an Trainingsschritten, durch das trainierte Netz ersetzt wird. In jedem Trainingsschritt ist dann die momentane Situation der Input. Als richtigen Output verwendet man diejenige Aktion, welche die Belohnung der gesamten Episode maximieren würde, wenn das System im weiteren Verlauf der Episode jeweils die Aktionen wählt, die das Zielnetz vorgibt.

1.1 Geschichtlicher Hintergrund

Ein Beispiel für die erfolgreiche Anwendung tiefer bestärkender Lernsysteme ist „Computational Intelligence in Games“, eine Disziplin, bei der unter anderem Systeme entwickelt werden, die verschiedene Spiele selbständig auf menschlichem Niveau spielen können.
Einer der ersten großen Erfolge in diesem Gebiet war der Sieg des von IBM entwickelten Programmes *Deep Blue* gegen den Schachweltmeister Garri Kasparow im Jahr 1997. Dazu simulierte das System mehrere Millionen mögliche Spielverläufe und wählte daraus den besten Zug aus (vgl. [CHJH02]).
Die nächste bedeutende Herausforderung war das Brettspiel *Go*. Dieses weist, zum Beispiel aufgrund seines größeren Spielfeldes, eine höhere Komplexität als Schach auf und kann somit nicht mehr auf dieselbe Art und Weise gelöst werden. Erst durch die Integration eines tiefen bestärkenden Lernsystems konnte das von Google DeepMind entwickelte *AlphaGo* System im Jahr 2016 einen der weltbesten Go Spieler besiegen (vgl. [SSS+17]).
Nachdem mit Go das komplexeste klassische Brettspiel gelöst wurde, rücken auch Computerspiele in den Fokus der Forschung. Bereits 2015 präsentierte DeepMind in [MKS+15] ein System, welches 49 verschiedene Atari 2600 Spiele auf mindestens demselben Niveau, wie professionelle menschliche Spieltester, spielen konnte.

Auch in diesem Jahr gab es einige beeindruckende Fortschritte bei der Anwendung tiefer bestärkender Lernmethoden auf die Lösung von Computerspielen. Im Juni 2018 endete der von der Non-Profit-Organisation OpenAi veranstaltete Wettbewerb „Retro Contest“ [SPN+]. Dieser Wettbewerb testete die Fähigkeit der lernender Systeme, gelerntes Wissen auf neue Aufgaben zu transferieren. Konkret sollten die Teilnehmer ein System entwickeln, welches das Computerspiel „Sonic the Hedgehog“ lösen kann. Dazu konnten sie ihr System auf allen bekannten Leveln dieses Spiels trainieren. Die Bewertung der Teilnehmer fand jedoch anhand von Leveln statt, die extra für diesen Wettbewerb entwickelt wurden und somit komplett neu waren.
Im selben Monat schaffte es ein OpenAI Forscherteam, ein System zu trainieren, das ein menschliches Team im Echtzeitstrategiespiel DOTA 2 schlagen konnte [DPF+]. Dies ist nicht nur erstaunlich, weil dieses System fünf unterschiedliche Agenten enthält, die miteinander kooperieren können. Sondern auch, da dasselbe System in [ABC+18] ohne große Umstellungen auf die Steuerung einer Roboterhand mit fünf Fingern angewandt werden konnte. Die Anwendung desselben Systems auf zwei so unterschiedliche Probleme lässt hoffen, dass man hier einer allgemeinen Intelligenz näher kommt.
Einem Team von Google DeepMind gelang es in [JCD+18] ein System zu entwickeln, das seine menschlichen Mitspieler bei dem First-Person Shooter „Quake“ übertrumpfen konnte. Besonders erstaunlich ist hierbei, dass das System auch mit menschlichen Team Partnern hervorragend zusammenspielen konnte. So waren die Ergebnisse von gemischten Mensch-Maschine Teams besser, als die von rein menschlichen Teams. Dies zeigt die Möglichkeit solcher Systeme mit Menschen zusammenzuarbeiten.

1.2 Zielsetzung und Motivation

Der größte Vorteil des bestärkenden Lernens ist gleichzeitig auch ein Nachteil. Dadurch, dass das System selbständig lernt, welcher Output zu einem gegebenem Input passt, ist es oft schwierig nachzuvollziehen, warum sich das System für einen bestimmten Output entscheidet.
Beim tiefen bestärkenden Lernen kommt noch hinzu, dass auch neuronale Netze oft eine „Black Box“ darstellen, da man meistens nicht genau weiß, wie sich die gelernten Gewichte auf den Output auswirken. Experten können zwar die gespeicherten Gewichte eines Netzes durch verschiedene Methoden visualisieren und versuchen dadurch das Verhalten der Netze zu analysieren, doch dafür ist einiges an Hintergrundwissen notwendig.
Gerade die Beweggründe dieser Systeme sind jedoch oft sehr interessant. So wurde zum Beispiel Kasparow, der 1997 von Deep Blue geschlagen wurde, in einem Interview mit Chess.com nach seiner Meinung zu AlphaZero gefragt, dem Nachfolgesystem von AlphaGo, welches zusätzlich zu Go auch Schach und Shogi meistern konnte. Darauf antwortete er, dass Systeme wie AlphaZero eine einzigartige neue Gelegenheit bieten diese Spiele zu verstehen, auch wenn wir bisher dachten, zum Beispiel Schach bereits vollständig analysiert zu haben (vgl. [Kas17]).
Das Ziel dieser Arbeit ist es, tiefe bestärkende Lernsysteme für eine breitere Allgemeinheit verständlich zu machen. Dazu werde ich mich zuerst mit den theoretischen Grund-

lagen beschäftigen und danach ein System vorstellen, welches multimodale Erklärungen zu einem bereits trainierten tiefen neuronalen Netz erstellt. Diese Erklärungen sollen aus zwei Teilen bestehen. Zuerst berechne ich die visuelle Aufmerksamkeit. Das heißt ich hebe diejenigen Teile des Bildschirms hervor, die für die Entscheidung relevant waren. Anschließend generiere ich sprachliche Erklärungen, die rechtfertigen, warum eine bestimmte Aktion in einer bestimmten Situation ausgeführt wurde.
Diese Erklärungen können zum Beispiel Menschen helfen, die mit Robotern zusammenarbeiten. So verwenden beispielsweise Modares et al . in [MRLP16] bereits bestärkendes Lernen, um die Zusammenarbeit von Menschen und Robotern in Fertigungsanlagen zu verbessern. Hierbei werden zwar noch keine neuronalen Netze benutzt, aber DeepMind zeigte in [JCD+18] am Beispiel von *Quake*, dass auch tiefe bestärkende Lernsysteme mit Menschen kooperieren können. Außerdem benutzen zum Beispiel Lathuilière et al. in [LMMH17] neuronale Netze, die sie durch bestärkendes Lernen trainiert haben, um die Kooperation von Menschen und Robotern zu verbessern. Wenn nun Roboter, die durch tiefes bestärkendes Lernen trainiert wurden, direkt mit Menschen zusammenarbeiten, dann ist es notwendig, dass diese Roboter ihre Handlungen gegenüber ihren menschlichen Mitarbeitern rechtfertigen können.
Auch in der Forschung zu autonom fahrenden Autos werden bestärkende Lernalgorithmen diskutiert (vgl. [FJT18]). Im Straßenverkehr würde es somit zu einer direkten Interaktion zwischen Menschen und Systemen kommen, die durch tiefes bestärkendes Lernen trainiert wurden. In diesem Zusammenhang wäre es insbesondere dann interessant zu wissen, wie das System seine Entscheidungen begründet, falls es zu einem Unfall kommt. Dies könnte zum Beispiel dabei helfen rechtliche Fragen zu klären oder Fehler in dem System zu beheben.

1.3 Aufbau

Ich unterteile diese Arbeit in drei Abschnitte. Zunächst fasse ich in Kapitel 2 die Grundlagen von neuronalen Netzen zusammen und lege dar, wie diese mit bestärkendem Lernen trainiert werden können.
In Kapitel 3 werde ich beweisen, dass es zumindest theoretisch möglich ist, schon mit einfach gehaltenen neuronalen Netzen beliebige stetige Funktionen zu approximieren.
Zuletzt stelle ich in Kapitel 4 ein System vor, das multimodale Erklärungen, im oben beschriebenen Sinn, zu einem tiefen bestärkenden Lernsystem erstellt, welches einfache Atari 2600 Spiele löst.

2 Grundlagen des tiefen bestärkenden Lernens

In diesem Kaptiel möchte ich kurz darlegen, was künstliche neuronale Netze sind. Hierbei setze ich Grundwissen aus der Analysis und der linearen Algebra voraus, welche zum Beispiel in [For16],[For13] und [Fis13] behandelt wird.
In der allgemeinsten Beschreibung sind künstliche neuronale Netze Funktionen der Form $f : \mathbb{R}^n \times \Theta \to \mathbb{R}^m$. Dabei nennen wir $\theta \in \Theta \subset \mathbb{R}^k$ die *Parameter* des Netzes.
Das Ziel eines solchen Netzes ist es, gegebene Funktionen zu approximieren. Sei also eine Funktion $f^* : \mathbb{R}^n \to \mathbb{R}^m$ gegeben, welche *Ground Truth* genannt wird. Dann versuchen wir Parameter $\theta \in \Theta$ zu finden, sodass $f(\,\cdot\,, \theta)$ diese Funktion möglichst gut approximiert. In Kapitel 3 werden wir genauer definieren, was wir unter Approximation verstehen. Weiter werden wir dort zeigen, dass schon sehr einfache Netzstrukturen beliebige stetige Funktionen approximieren können.
Künstliche neuronale Netze wurden ursprünglich von einem Modell des menschlichen Gehirns, sogenannten biologischen neuronalen Netzen, inspiriert. Obwohl sich die heutige Forschung zunehmend von dieser Vorstellung löst, hilft es beim Verständnis der Definitionen, wenn man die Grundlagen des biologischen Modells kennt. Deshalb möchte ich an dieser Stelle die Grundzüge dieses Modells wiedergeben. Eine ausführlichere Beschreibung findet man zum Beispiel in [Sim09] auf den Seiten 6 - 10.
Bei modernen Modellen des menschlichen Nervensystems geht man davon aus, dass externe Reize, wie zum Beispiel Berührungen, in Form von elektrischen Impulsen durch die Nerven zum Gehirn geleitet werden. Dort wird die Information verarbeitet. Die dabei im Gehirn stattfindenden Vorgänge werden mit biologischen neuronalen Netzen erklärt. Diese Netze bestehen aus Neuronen und Synapsen. Dabei ist jedes Neuron über Synapsen mit anderen Neuronen verbunden, wodurch die namensgebende Netzstruktur entsteht.
Die im Gehirn ankommende Information wird zunächst an einige Neuronen weitergeleitet. Überschreiten die elektrischen Impulse, die bei einem Neuron ankommen, einen bestimmten Schwellwert, so „feuert“ das Neuron selbst einen elektrischen Impuls ab. Dieser Impuls wird von den Synapsen an andere Neuronen weitergeleitet. Einige dieser Neuronen „feuern“ daraufhin ebenfalls einen Impuls ab. Dies setzt sich fort bis die Information vollständig verarbeitet ist. Das Ergebnis wird dann über die Nerven an den Körper zurück geleitet.

2.1 Neuronen

Inspiriert von biologischen neuronalen Netze, wollen wir auch künstliche neuronale Netze auf Neuronen zurückführen. Ab dieser Stelle werden wir nur noch künstliche neuronale Netze betrachten, deshalb werde ich ab jetzt den Zusatz „künstliche“ weglassen und nur noch neuronale Netze schreiben.
Zunächst müssen wir also Neuronen, die Grundbausteine von neuronalen Netzen, definieren. Die folgende Definition findet man zum Beispiel in [Sim09] Seite 11 und in [Gol17] Seite 41.

Definition 2.1.1. Ein **Neuron** ist eine Funktion $f : \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R} \to \mathbb{R}$ der Form

$$f(x; \omega, b) = \sigma(\omega \cdot x + b)$$

für eine sogenannte **Aktivierungsfunktion** $\sigma : \mathbb{R} \to \mathbb{R}$ und $n \in \mathbb{N}$.
Weiter nennen wir $x \in \mathbb{R}^n$ den **Input**, $\omega \in \mathbb{R}^n$ die **Gewichte (engl. weigths)** und $b \in \mathbb{R}$ den **Bias** des Neurons. Das Ergebnis eines Neurons wird oft als **Output** bezeichnet.

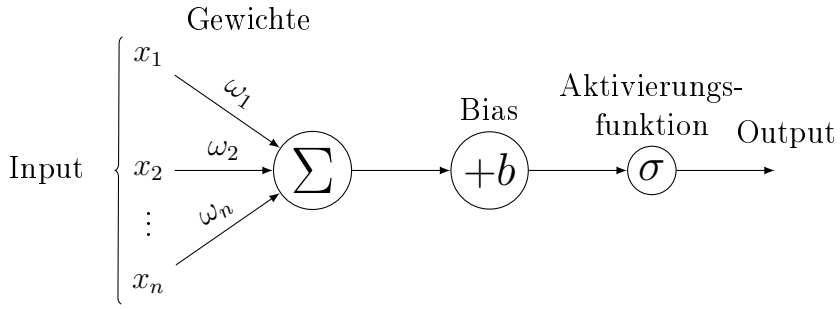


Abbildung 2.1: Das Innere eines Neurons.

Wählt man $m = 1$ und $\Theta = \mathbb{R}^n \times \mathbb{R}$, so erfüllt ein Neuron die allgemeine Form $f : \mathbb{R}^n \times \Theta \to \mathbb{R}^m$ eines neuronalen Netzes. Um zu verdeutlichen, dass die Parameter in Gewichte und Bias aufgeteilt werden, trennt man diese dabei oft mit einem Strichpunkt, anstatt nur mit einem Komma, von dem Input x.
Die Motivation hinter dieser Definition ist es, dass die Gewichte ω_i „lernen“, wie wichtig die jeweiligen Inputs x_i für das Neuron sind. Die Aktivierungsfunktion σ repräsentiert das „Feuern“ des Neurons. Zuletzt sorgt der Bias b dafür, dass der Input eines Neurons Werte in ganz $\mathbb{R}$ annehmen kann, auch wenn er a priori beschränkt sein sollte.
Anstatt die gewichteten Inputs zu addieren, wie es beim Skalarprodukt der Fall ist, könnte man sie theoretisch auch anders verarbeiten (vgl. [Gol17] S 41). Die Addition wird jedoch bei Weitem am häufigsten verwendet, deshalb möchte ich mich in dieser Arbeit auf diesen Fall beschränken.

Die Parameter (ω, b) werden im Laufe des Trainings, welches in Kapitel 2.4 genauer beschrieben wird, angepasst. Deshalb ist die Aktivierungsfunktion der entscheidende Faktor bei der Wahl eines Neurons.

Da beinahe alle Aktivierungsfunktionen nicht linear sind, werden sie oft auch *Nonlinearties* genannt. Heuristisch kann man sich vorstellen, dass die Aktivierungsfunktion dafür sorgt, dass man auch nicht lineare Funktionen approximieren kann.
In Kapitel 3 dieser Arbeit werden wir zeigen, dass stetige Aktivierungsfunktionen, die nicht polynomiell sind, ausreichen, um bereits mit sehr simplen Netzstrukturen beliebige Funktionen zu approximieren.
Die folgenden Aktivierungsfunktionen werden wir in dieser Arbeit benötigen (Formeln aus [Gol17] Seite 44):

Beispiel 2.1.2. *Die **rectified linear Unit (ReLU)** Funktion*

$$\sigma(x) = max(0, x).$$

Beispiel 2.1.3. *Die **Sigmoid** Funktion*

$$\sigma(x) = \frac{e^x}{e^x + 1}.$$

Beispiel 2.1.4. *Der **Tangens Hyperbolicus***

$$tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1}.$$

Um Neuronen zu einem Netz zusammenzusetzen, benötigen wir das Konzept eines gerichteten Graphen (vgl. [Sim09] Seite 15).

Definition 2.1.5. Ein **gerichteter Graph** besteht aus einer endlichen Menge V, den sogenannten **Knoten (engl. vertices)**, und einer Menge von **Kanten (engl. edges)** $E \subset V \times V$. Wir sagen eine Kante $(v_1, v_2) \in E$ ist von v_1 nach v_2 **gerichtet**.

Mit diesem Konzept können wir die Verbindungen der Neuronen präzise beschreiben.
Sei V eine Menge von Neuronen, dann beschreibt $(v_1, v_2) \in V \times V$, dass der Output der Neurons v_1 als Teil des Input des Neurons v_2 verwendet wird. Der Output von v_2 ist also gegeben durch

$$v_2(\dots v_1(x; \theta_1) \dots; \theta_2).$$

Definition 2.1.6. Sei (V, E) ein gerichteter Graph, dessen Knoten V nur aus Neuronen bestehen.
Die Komposition aller Neuronen aus V entsprechend der durch E im obigen Sinne gegebenen Verknüpfungen nennen wir ein **(künstliches) neuronales Netz**.
Den Graphen (V, E) möchte ich in dieser Arbeit als den **darstellenden Graphen** dieses Netzes bezeichnen.

Bemerkung 2.1.7. *Die Neuronen der Menge V müssen nicht alle dieselbe Aktivierungsfunktion benutzen.*

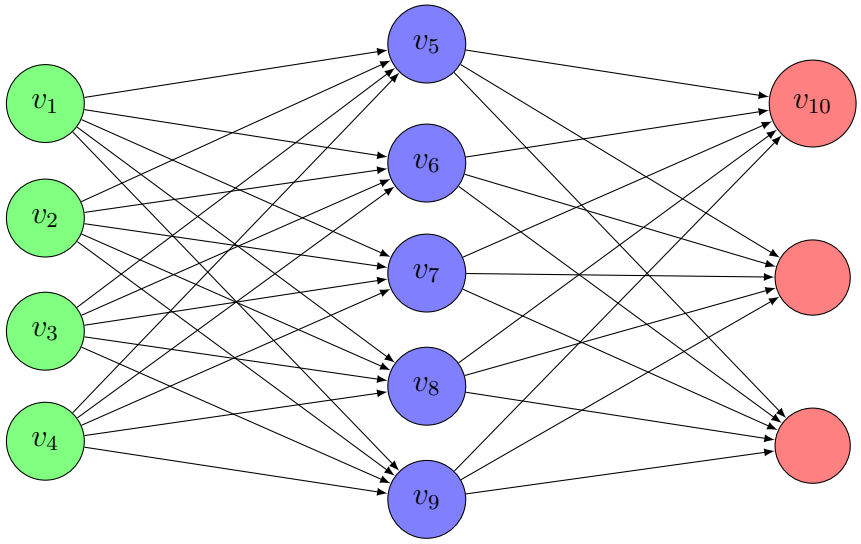


Abbildung 2.2: Die Visualisierung eines neuronalen Netzes durch einen darstellenden Graphen.

Beispiel 2.1.8. *In Abbildung 2.2 wird der Output von v_{10} gegeben durch*

$$v_{10}((v_5((v_1(x;\theta_1),\ldots,v_4(x;\theta_4));\theta_5),\ldots,v_9(\ldots;\theta_9));\theta_{10}).$$

Bemerkung 2.1.9. *In manchen Büchern wird nicht zwischen dem Graphen und dem Netz unterschieden.*

Diese Darstellung bezeichnet man auch als einen *Architectural Graph*. Im Gegensatz dazu kann man ein neuronales Netz auch über einen *Signal-flow Graphen* definieren (vgl. [Sim09] Seite 16).
Bei einem Signal-flow Graphen werden den Knoten und Kanten Rechenoperationen zugeordnet. Somit ist es möglich auch das Innere eines Neurons als gerichteten Graphen darzustellen. Dadurch kann man sich die explizite Definition von Neuronen sparen und der darstellende Graph ist vollständiger.
Ein Nachteil von Signal-flow Graphen ist, dass man sehr spezifische Kanten innerhalb des Graphen definieren muss, was die Definition meiner Meinung nach verkompliziert. Da die beiden Definitionen äquivalent sind, möchte ich daher nicht weiter auf Signal-flow Graphen eingehen.

2.2 Layer

Da neuronale Netze oft aus weit über 1000 Neuronen bestehen, ist es sehr schwierig die Struktur eines Netzes nur durch Neuronen zu vermitteln. Aus diesem Grund verwendet man bei der Beschreibung von neuronalen Netzen oft Netz-Strukturen, welche in mehreren Netzen vorkommen und welche durch Funktionen dargestellt werden, die schneller verstanden werden können.

Motiviert von Darstellungen wie Abbildung 2.2, in denen die Neuronen in Reihen oder Schichten angeordnet werden, nennt man ein solches Unternetz ein *Layer* (vgl. [GBC16] Seite 168f, [Gol17] Seite 41f).
Wenn wir ein neuronales Netz als ein Layer bezeichnen, möchten wir also betonen, dass das Netz Teil eines größeren neuronalen Netzes ist.
In diesem Sinne kann man sich ein neuronales Netz als Komposition von Layern vorstellen, welche gemäß eines gerichteten Graphen verknüpft werden.
Im Folgenden halte ich mich an [GBC16] Seite 168f . Bei einem aus Layern zusammengesetzten Netz hat es sich durchgesetzt, das letzte Layer das *Output Layer* zu nennen und den Input als eigenes Layer, das *Input Layer*, zu betrachten. Alle anderen Layer heißen *Hidden Layer*.

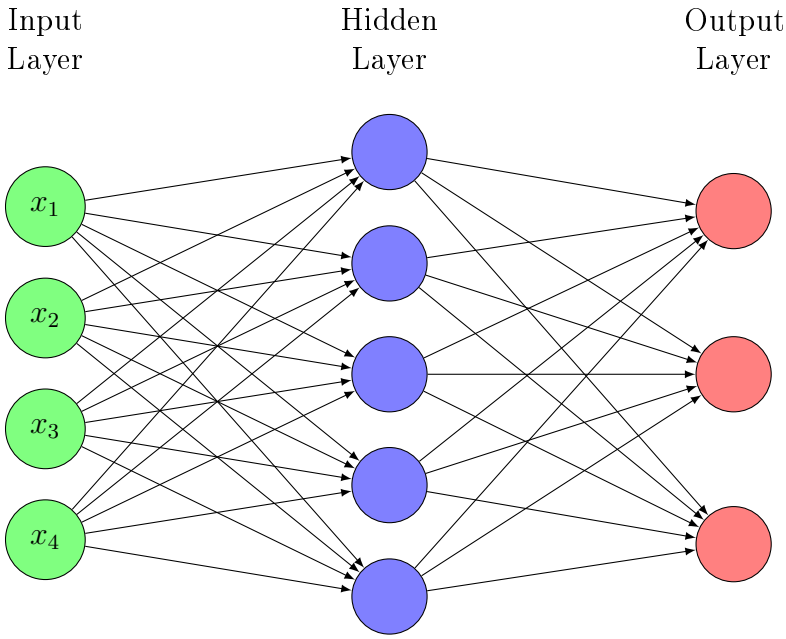


Abbildung 2.3: Beispiel der Anordnung des darstellenden Graphen in Layern.

Definition 2.2.1. Sei $f : \mathbb{R}^n \times \Theta \to \mathbb{R}^m$ ein neuronales Netz und sei $\pi_i : \mathbb{R}^m \to \mathbb{R}$ für $i = 1, ..., m$ die Projektion auf die i-te Komponente, dann bezeichnen wir $\pi \circ f$ als eine **Einheit (engl. unit)** von f.

Diese Definition ist zwar theoretisch für alle Netze möglich, aber sie ist nur für Layer innerhalb eines größeren Netzes sinnvoll.
Bei einem solchen Layer spielen die Einheiten eine ähnliche Rolle wie die Neuronen. Sie sind Funktionen von höher dimensionalen reelen Räumen nach $\mathbb{R}$, aus denen das Netz zusammengesetzt wird. Aus diesem Grund wird in manchen Büchern nicht zwischen Neuronen und Einheiten unterschieden.

Um Netze zu vergleichen, bezeichnet man die Anzahl der Einheiten eines Layers, also die Dimension seines Outputs, als die *Breite (engl. width)* des Layers. Die Anzahl der Layer in einem Netz wird *Tiefe (engl. depth)* genannt. Diese Nomenklatur inspirierte den Begriff des *tiefen neuronalen Netzes* für neuronale Netze mit mehr als einem Layer. Für eine allgemein geltende Theorie von neuronalen Netzen ist die Vorstellung als Komposition von Layern nicht sinnvoll. Es ist nicht einheitlich definierbar, welche Strukturen als Layer gelten, da beinahe jede Funktion als Layer verwendet werden kann (vgl. [GBC16] Seite 376). Somit sind Begriffe wie die Tiefe eines Netzes stark von der Konvention desjenigen abhängig, der sie verwendet (vgl. [GBC16] Seite 7).
In der Praxis ist diese Vorstellung jedoch sehr nützlich. Gerade weil ein Layer keine klar definierten Vorlagen erfüllen muss, ist es möglich verschiedene Funktionen zu testen. So werden in machen Anwendungen Layer verwendet, die Funktionen enthalten, welche sich nicht mit den in dieser Arbeit definierten Neuronen darstellen lassen. Erweitert man den Begriff eines Neurons, sodass die gewichteten Inputs nicht nur addiert, sondern auch anders verrechnet werden können (zum Beispiel durch das Auswählen des höchsten Wertes vgl. [Gol17] Seite 41), dann kann man auch diese Layer durch Neuronen darstellen.

2.3 Konkrete Strukturen

In diesem Kapitel stelle ich einige konkrete neuronale Netze vor. Dabei möchte ich nicht ausführlich auf die Entwicklung und Motivation dieser Netze eingehen. Ich beschränke mich darauf, die Netze mathematisch zu definieren und einige Gebiete zu beschreiben, in denen diese Netze bisher gute Ergebnisse erzielt haben.

2.3.1 Feed-forward Netze

Die einfachste Art von neuronalen Netzen sind solche, deren Verbindungen alle in eine Richtung ausgerichtet sind, bei denen die Information also immer nach vorne gereicht wird.
Dieses Konzept kann man mathematisch präzise formulieren.

Definition 2.3.1. Ein gerichteter Graph (V, E) heißt **zyklisch**, wenn er einen Zykel enthält. Ein **Zykel** ist ein Teilemenge $\{(v_1, v_2), ..., (v_{m-1}, v_m)\} \subset E$ mit $v_1 = v_m$. Falls ein Graph nicht zyklisch ist, nennen wir ihn **azyklisch**.

Definition 2.3.2. Ein neuronales Netz heißt **feed-forward** Netz, wenn der darstellende Graph des Netzes azyklisch ist.

Vollständig verbundene Netze

Die einfachste Klasse von feed-forward Netzen sind solche, bei denen mehrere Neuronen parallel ausgeführt werden.
Dieser Spezialfall eines feed-forward Netzes wird *Perzeptron* genannt und entspricht einem Vektor $(f_1, ..., f_n)$, wobei jede Komponente ein Neuron der Form $f_i = f(x; \omega_i, b_i) =$

$\sigma(\omega_i \cdot x + b_i)$ ist. Hierbei verwendet man bei allen Neuronen dieselbe Aktivierungsfunktion σ. Die Gewichte ω_i und Biase b_i der einzelnen Neuronen kann man dann zu einer Matrix W bzw. einem Vektor b zusammenfassen.

Definition 2.3.3. Ein **Perzeptron** ist eine Funktion $f : \mathbb{R}^n \times (\mathbb{R}^{m \times n} \times \mathbb{R}^m) \to \mathbb{R}^m$ mit

$$f(x; W, b) = \sigma(Wx + b),$$

wobei σ die komponentenweise Anwendung einer Aktivierungsfunktion $\sigma : \mathbb{R} \to \mathbb{R}$ beschreibt. Wie bei den einzelnen Neuronen nennen wir die beiden Parameter die **Gewichts** Matrix $W \in \mathbb{R}^{m \times n}$ und den **Bias** Vektor $b \in \mathbb{R}^m$.

In der Anfangszeit wurde die Aktivierungsfunktion häufig mit einem diskreten Wertebereich gewählt, um das Feuern bzw. nicht-Feuern der Neuronen zu beschreiben (vgl. [Sim09] Seite 50).
Heute wird die Aktivierungsfunktion manchmal sogar weggelassen, um allgemeinere Funktionen zu approximieren, besonders wenn das Perzeptron als Output Layer eines größeren Netzes verwendet wird (vgl. [Gol17] Seite 43).
Ich habe mich für die Variante entschieden, in der Definition eine allgemeine Aktivierungsfunktion zu erlauben. Somit lässt man sowohl die Identität als auch diskrete Funktionen als Aktivierungsfunktion zu. Außerdem macht dieses Verständnis eines Perzeptrons die folgende Definition einfacher und klarer verständlich.

Definition 2.3.4. Ein **Multi Layer Perzeptron(MLP)** mit n (Hidden) Layern ist eine Komposition aus $n + 1$ Perzeptronen. Das heißt ein MLP ist eine Funktion

$$f(x; \theta) = P_n(P_{n-1}(P_{n-2}(...); \theta_{n-1}); \theta_n),$$

wobei $P_i(x; \theta_i)$ für $i = 0, ..., n$ ein Perzetpron ist.

Man definiert MLP's also über die Anzahl der Hidden Layer und zählt das Output Layer P_n nicht mit. Den darstellenden Graphen eines MLP's haben wir bereits in Abbildung 2.2 gesehen.
Hierbei werden in jedem Layer alle Inputs mit allen Neuronen verbunden. Deswegen werden MLP's auch *vollständig verbundene (engl. fully connected)* oder *dichte (engl. dense)* Netze genannt und für gewöhnlich versteht man unter einem *vollständig verbundenen Layer* ein Perzeptron.

Convolutionale neuronale Netze

Im Gegensatz zu vollständig verbundenen MLP's gibt es auch Arten von feed-forward Netzen, bei denen jedes Neuron nur mit einer Teilmenge des Inputs verbunden ist. Dadurch können diese Neuronen lernen, Ausschnitte des Inputs zu beschreiben.
Indem man den gesamten Input durch Teilmengen dieser Form überdeckt, kann das Netz somit die auf den Ausschnitten gelernten Muster innerhalb des Inputs erkennen. Durch dieses Verfahren können zum Beispiel Gesichter in einem Foto identifiziert werden.

In Anlehnung an die mathematische Faltungsoperation werden solche Netze *konvolutional* genannt.
Die Motivation hinter konvolutionalen Netzen ist, dass man Muster auf Teilen des Inputs erkennen kann.
Zunächst müssen wir also formalisieren, was wir unter einer Teilmenge des Inputs verstehen. Dabei schreibe ich den Input als $\prod_{j=1}^{k} \mathbb{R}^j$, da konvolutionale Netze oft auf Input in Matrixform angewandt werden. Zum Beispiel auf Bilder, bei denen jeder Pixel einem Eintrag einer Matrix entspricht.

Definition 2.3.5. Sei $m < \prod_{j=1}^{k} j$. Ein **Fenster** ist eine Projektion $p : \prod_{j=1}^{k} \mathbb{R}^j \to \mathbb{R}^m$ auf bestimmte Komponenten, d. h. es existiert ein injektives $\pi : \{1, ..., m\} \hookrightarrow \prod_{j=1}^{k} \{1, ..., j\}$, sodass

$$p(x) = (x_{\pi(1)}, ..., x_{\pi(m)}).$$

Eine weitere Besonderheit von konvolutionalen Netzen ist, dass mehre Neuronen dieselben Parameter teilen. Es wird dasselbe Neuron auf alle gewählten Fenster angewandt.

Definition 2.3.6. Ein **konvolutionales neuronales Netz (CNN)** besteht aus einer endlichen Menge $W = \{w_1, ..., w_s\}$ von Fenstern $w_i : \prod_{j=1}^{k} \mathbb{R}^j \to \mathbb{R}^m$ und einer Menge $\{k_1, ..., k_r\}$ aus Neuronen $k_i : \mathbb{R}^m \to \mathbb{R}$, welche alle dieselbe Aktivierungsfunktion verwenden und **Filter** oder **Kerne (engl. kernels)** genannt werden.
Das Netz wird als Funktion $\prod_{j=1}^{k} \mathbb{R}^j \to \mathbb{R}^s \times \mathbb{R}^r$ dann durch

$$f(x, \theta)_{ij} = k_j(w_i(x), \theta_j)$$

gegeben.

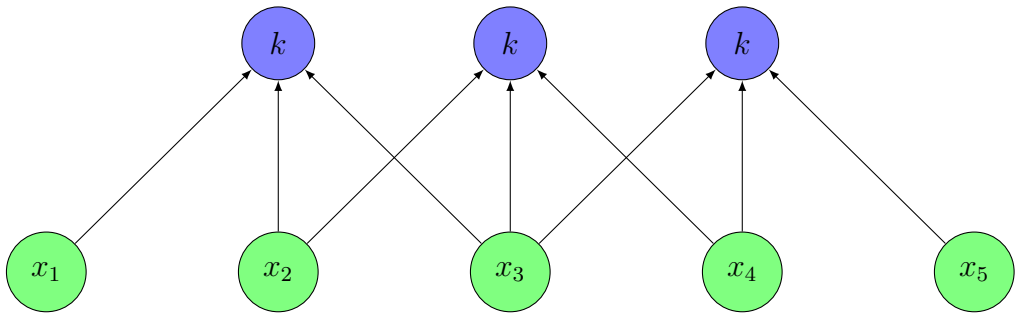


Abbildung 2.4: Der darstellende Graph eines stark vereinfachten konvolutionalen Netzes mit nur einem einzigen Filter k. Dieser Filter wird durch drei verschiedene Neuronen dargestellt, die sich alle dieselben Parameter teilen.

Die Wahl der Fenster hängt dabei stark von der Art von Daten ab, die man auswerten möchte.
Bei der Analyse von natürlicher Sprache zum Beispiel, werden Sätze oft als Vektoren in $\mathbb{R}^n$ dargestellt, deren Einträge jeweils ein Wort repräsentieren. In diesem Fall wählt man

als Fenster innerhalb des Satzes meistens k aufeinander folgende Wörter. Eine genauere Beschreibung hierzu findet man in [Gol17] Kapitel 13.1.
Visuelle Daten, wie zum Beispiel Bilder, werden, wie oben bereits erwähnt, oft als Matrizen gespeichert, wobei jeder Eintrag einem Pixel entspricht. Bei einer sogenannten *2D Konvolution* wählt man die Fenster als $k_1 \times k_2$ Untermatrizen. Diese werden mit einem *Schritt (engl. stride)* über das Bild „geschoben". Wählt man (s_1, s_2) als Schritt, dann ist das Fenster, welches den Output in der i-ten Reihe und der j-ten Spalte erzeugt, als Untermatrix von der Form

$$p_{i,j} = \begin{bmatrix} x_{is_1,js_2} & \dots & x_{is_1,js_2+k_2} \\ \vdots & \ddots & \vdots \\ x_{is_1+k_1,js_2} & \dots & x_{is_1+k_1,js_2+k_2} \end{bmatrix}$$

für $i, j \in \mathbb{N}_0$ mit $is_1 + k_1 \leq n$ und $js_2 + k_2 \leq m$.
Diese Untermatrix wird dann durch Untereinanderschreiben ihrer Spalten in eine Vektorform gebracht.
Um schneller zu sehen, auf welche Untermatrizen die Filter angewandt werden, schreibt man die Gewichte, der Filter einer 2D Konvolution, oft ebenfalls in Matrixform.
Das Skalarprodukt mit den Gewichten $\begin{bmatrix} k_{1,1} & \dots & k_{1,k_2} \\ \vdots & \ddots & \vdots \\ k_{k_1,1} & \dots & k_{k_1,k_2} \end{bmatrix}$ eines solchen Filters k erhalten wir durch

$$\sum_{r=1}^{k_1} \sum_{t=1}^{k_2} x_{is_1+(r-1),js_2+(t-1)} k_{r,t}.$$

Versteht man p und k als Funktionen $p(i,j) = x_{i,j}, k(i,j) = k_{i,j}$, so sieht man in dieser Darstellung die Ähnlichkeit zur namensgebenden mathematischen Faltung $(p*k)(i,j) = \sum_{r=1}^{k_1} \sum_{t=1}^{k_2} p(i-r, j-t)k(r,t)$.
Diese allgemeinen Beschreibungen sind sehr undurchsichtig. Um das Verfahren besser zu verstehen, enthält Abbildung 2.3.1 eine beispielhafte Berechnung eines konvoluten Netzes mit nur einem Filter.

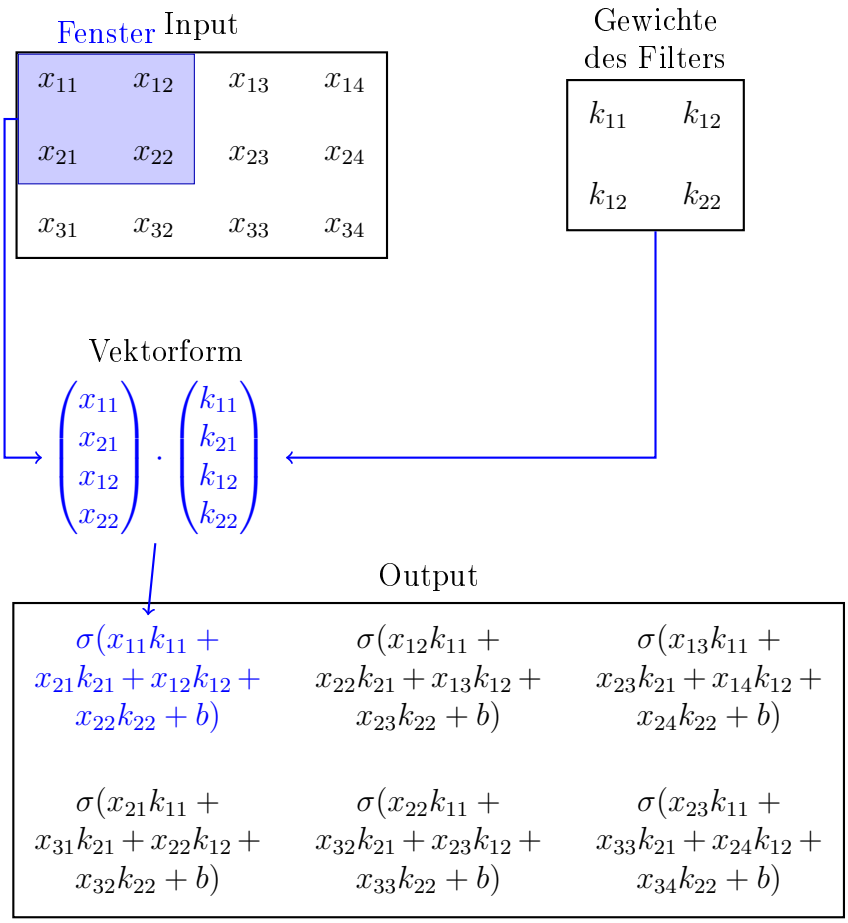


Abbildung 2.5: Beispiel für eine 2D Konvolution mit nur einem Filter k mit Bias b und Aktivierungsfunktion σ.

Bei mehr als einem Filter enthält der Output eines konvolutionalen Layers für jeden Filter einen eigenen Kanal. Ebenso werden Bilder oft mit drei Kanälen für rote, blaue und grüne Töne gespeichert.
Wir wollen nun beschreiben, wie man eine 2D Konvolution auf diese Art von Daten anwendet. Sei also $x \in \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R}^c$ eine $n \times m$ Matrix mit c Kanälen.
In diesem Fall wählen wir die Fenster, wie im Fall von nur einem Kanal, als $k_1 \times k_2$ Untermatrizen, schieben diese jedoch über alle c Kanäle des Inputs gleichzeitig. Das heißt für den Schritt (s_1, s_2) ist das Fenster, welches den i,j-ten Eintrag der Output Matrix bestimmt, von der Form:

$$\boldsymbol{p_{i,j}} = \begin{pmatrix} x_{is_1,js_2,1} & \cdots & x_{is_1,js_2+k_2,1} \\ \vdots & \ddots & \vdots \\ x_{is_1+k_1,js_2,1} & \cdots & x_{is_1+k_1,js_2+k_2,1} \end{pmatrix}$$

Auch hier schreiben wir die Spalten des Fensters untereinander und erhalten den Vektor $p_{i,j} = (x_{is_1,js_2,1}, \ldots, x_{is_1+k_1,js_2,1}, \ldots, x_{is_1+k_1,js_2+k_2,1}, x_{is_1,js_2,2}, \ldots, x_{is_1+k_1,js_2+k_2,3})$.

2.3.2 Rekurrente neuronale Netze

Rekurrente Netze werden oft benutzt, um Folgen von Daten $x_1, ..., x_n$ zu analysieren. Die grundlegende Idee dabei ist, eine Art Gedächtnis zu simulieren, indem man das Ergebnis der bisher verarbeiteten Teile $x_1, \ldots, x_{i-1}$ der Folge nicht einfach „vergisst“, sondern als zusätzlichen Input für die Verarbeitung des nächsten Teils x_i verwendet. Anstatt die Information nur nach vorne weiterzugeben, kann der Output eines Neurons also auch an frühere Schichten zurückgegeben werden. In der Sprache von gerichteten Graphen können wir dies folgendermaßen definieren (vgl. [Sim09] S23):

Definition 2.3.7. Ein neuronales Netz heißt **rekurrentes neuronales Netz (RNN)**, wenn der darstellende Graph des Netzes zyklisch ist.

Im Folgenden halte ich mich an [Gol17] Kapitel 14 und an [Ola15].
Wenn ein rekurrentes Netz verwendet wird, um eine Folge von Daten $x_1, ..., x_n$ zu analysieren, dann werden dieselben gelernten Parameter auf jeden Teil der Folge x_i angewandt. Dieses Verhalten haben wir bereits bei den Filtern von CNN's gesehen.
Anders als bei solchen Netzen, geht das Ergebnis jedes Teils dieser Folge bei einem RNN in die Entscheidung der folgenden Teile mit ein. Dies ist in Abbildung 2.6 zu sehen. Das sich wiederholende Modul, welches die einzelnen Teile der Folge verarbeitet, wird oft als eine *Zelle* bezeichnet.
Bei einem RNN bezeichnet man die Verarbeitung einer ganzen Input-Folge $x = x_0, \cdots, x_n$ durch ein Modul als ein Layer. Entsprechend bezeichnet man rekurrente Netze, bei denen mehrere Module hintereinander geschaltet werden, als *tiefe rekurrente Netze*, ein solches Netz ist in Abbildung 2.7 abgebildet (vgl. [Gol17] Kapitel 14.5).

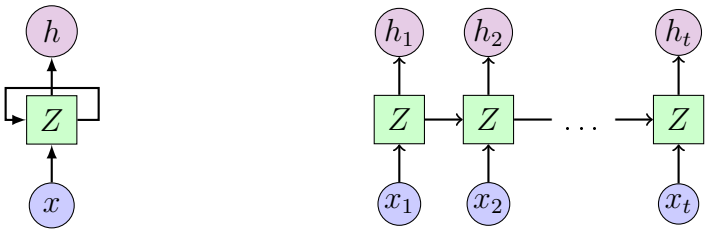


Abbildung 2.6: Ein RNN mit einem Layer, bestehend aus der Zelle Z. Links ist die Verarbeitung des gesamten Inputs $x = x_1, \ldots, x_n$ dargestellt. Rechts sieht man dasselbe Netz *ausgerollt*, wodurch deutlicher wird, wie die einzelnen Teile h_i des Outputs entstehen.

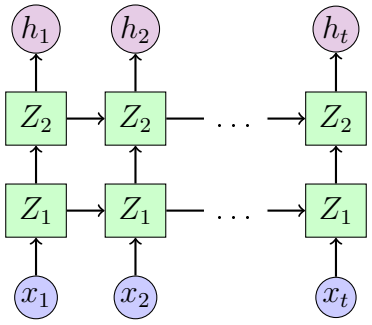


Abbildung 2.7: Ein tiefes RNN mit zwei Layern.

Die in der Praxis erfolgreichsten RNN's sind die von Hochreiter und Schmidhuber in [HS97] vorgeschlagenen *long short-term memory (LSTM)* Netze.
Eine LSTM Zelle enthält mehrere Perzeptronen, welche wir folgendermaßen definieren:

Definition 2.3.8. Sei σ die Sigmoid Aktivierungsfunktion $\frac{e^x}{e^x+1}$. Wir definieren die folgenden Perzeptronen:

$$\begin{aligned} \text{Forget Gate} \quad & f(x) := \sigma(W_f \cdot x + b_f) \\ \text{Input Gate} \quad & i(x) := \sigma(W_i \cdot x + b_i) \\ \text{Output Gate} \quad & o(x) := \sigma(W_o \cdot x + b_o) \\ & \widetilde{C}(x) := tanh(W_C \cdot x + b_C), \end{aligned}$$

mit $W_k \in \mathbb{R}^{m \times n}$ und $b_k \in \mathbb{R}^m$ für $k = f, i, o, C$.

Aus diesen Bestandteilen können wir die LSTM Zelle zusammensetzten.
Das besondere an LSTMs im Vergleich zu anderen rekurrenten Netzen ist, dass eine LSTM Zelle nicht nur ihren Output h an die nächste Zelle weitergibt, sondern auch einen *Zellen-Zustand (engl. cell state)* C. Über diesen Zellen-Zustand kann reguliert werden, welche Informationen an die nächste Zelle weitergegeben werden.
Eine LSTM Zelle besteht also aus zwei Funktionen $C(\cdot, \cdot)$ und $h(\cdot)$. Die erste Funktion C liefert einen internen Zellen-Zustand, der nur zwischen den Zellen weitergereicht wird:

$$C(x, y) = f(x) * y + i(x) * \widetilde{C}(x)$$

Hierbei beschreibt $*$ das komponentenweise Produkt zweier Vektoren.
Auf das Ergebnis dieser Funktion wird komponentenweise die *tanh* Aktivierungsfunktion angewandt und daraus wird der tatsächliche Output berechnet.

$$h(x, y) = o(x) * tanh(C(x, y))$$

Auf eine Folge $x_1, ..., x_n$ angewandt erhält man rekursiv für den t-ten intern weitergereichten Zellen-Zustand

$$C_t := C([h_{t-1}, x_t], C_{t-1}),$$

wobei $[h_{t-1}, x_t]$ die Konkatenation der beiden Vektoren h_{t-1} und x_t beschreibt.
Der Output h_t and der t-ten Stelle wird dann durch

$$h_t := h([h_{t-1}, x_t], C_{t-1})$$

berechnet, die gesamte Struktur ist in Abbildung 2.8 dargestellt.

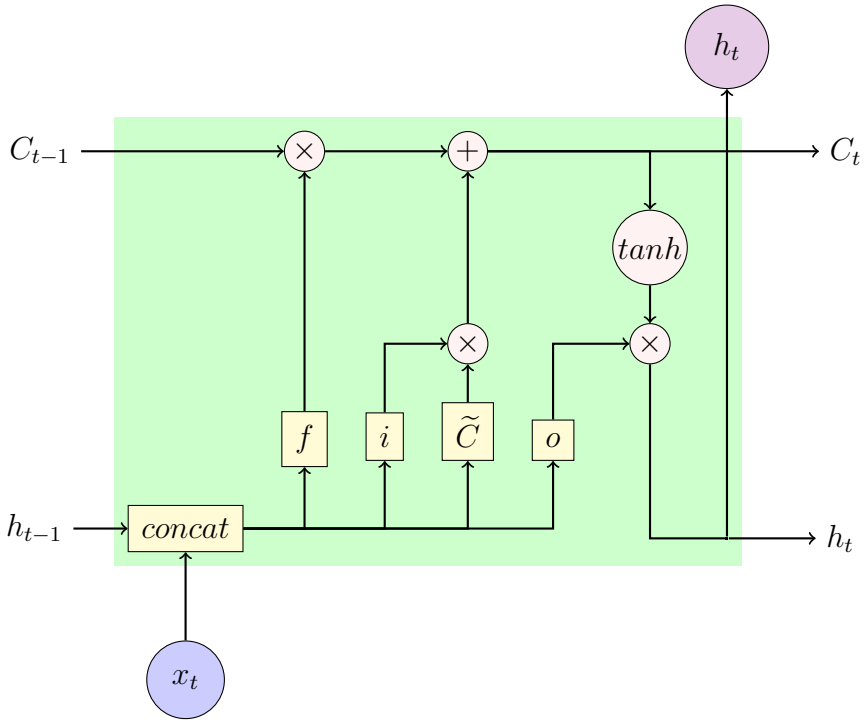


Abbildung 2.8: Darstellung einer LSTM Zelle. Dabei beschreiben die Rechtecke Perzeptronen und die Kreise komponentenweise Operationen.

Soweit zur reinen Definition einer LSTM Zelle.
Um zu verstehen, warum ein LSTM funktioniert, verfolgen wir den Zellen-Zustand C. Nachdem der Zellen-Zustand C_{t-1} an die Zelle weitergegeben wurde, wird er zunächst mit dem Ergebnis des Forget Gates $f_t := f([h_{t-1}, x_t])$ multipliziert. Da f in jeder Komponente nur Werte zwischen 0 und 1 annehmen kann, stellt man sich f als eine Art Tor vor, das entscheidet, welche Informationen nach dem Input x_t vergessen werden können. Enthält $f_t = f([h_{t-1}, x_t])$ eine 0 Komponente, so wird der Inhalt dieser Komponente von C_{t-1} durch die komponentenweise Multiplikation ebenfalls zu 0 und wird somit „vergessen“.
Danach wird ein vorläufiger Zellen-Zustand $\widetilde{C}_t := \widetilde{C}([h_{t-1}, x_t])$ berechnet, der den Beitrag des momentanen Inputs $[h_{t-1}, x_t]$ misst. Bevor dieser zu dem bisherigen Zellen-Zustand $f([h_{t-1}, x_t]) * C_{t-1}$ addiert wird, entscheidet auch hier ein „Tor“, das Input Gate $i_t := i([h_{t-1}, x_t])$, welche Komponenten von $\widetilde{C}_t$ in dieser Situation gespeichert werden sollen.

Das Ergebnis dieser Addition $C_t = f_t * C_{t-1} + i_t * \widetilde{C}_t$ enthält nun alle Informationen, die sich das LSTM Netz für die nächsten Teile des Inputs „merken“ möchte, und wird an die nächste Zelle weitergereicht.
Wir möchten jedoch nicht alle Informationen als Output h_t dieser Zelle ausgeben. Außerdem wollen wir, dass alle Teile unseres Outputs eine gemeinsame Skalierung aufweisen. Dafür wenden wir zunächst die *tanh* Aktivierungsfunktion komponentenweise auf C_t an, um alle Einträge in den Wertebereich$[-1, 1]$ zu skalieren. Danach entscheidet das letzte „Tor“, das Output Gate $o_t = o([h_{t-1}, x_t])$, welche Informationen von C_t für den Output h_t relevant sind.
Dadurch, dass C_t auch Informationen enthalten kann, die für den Output h_t komplett irrelevant sind, kann sich ein LSTM Netz Informationen vom Anfang des Inputs solange merken, bis sie relevant werden. Dies ist zum Beispiel erforderlich, wenn man einen Satz übersetzen möchte und die Übersetzung eines Wortes, das am Anfang des ursprünglichen Satzes steht, im übersetzten Satz erst am Ende auftaucht.

Die LSTM-Struktur ist ein gutes Beispiel dafür, dass sich die Forschung von der biologischen Motivation durch Neuronen entfernt. Genau genommen ist eine LSTM-Zelle kein neuronales Netz im Sinne unserer Definition, da sich die komponentenweise Multiplikation * nicht aus Neuronen zusammensetzen lässt, die ihren gewichteten Input nur addieren.
Die komponentenweise Anwendung von *tanh* können wir durch klassische Neuronen mit Aktivierungsfunktion *tanh* darstellen, welche nur einen Input verarbeiten und deren einziges Gewicht auf 1 und deren Bias auf 0 festgelegt sind. Für die Darstellung der komponentenweisen Addition benötigen wir Neuronen, welche jeweils nur zwei Inputs verwenden, diese addieren und dann die Identität als Aktivierungsfunktion anwenden, wobei wir auch hier die Gewichte auf 1 und den Bias 0 setzen.
Möchten wir aber die komponentenweise Multiplikation $*$ durch Neuronen darstellen, so müssen wir auch Neuronen zulassen, die ihren gewichteten Input multiplizieren anstatt zu addieren. In diesem Fall können wir für die komponentenweise Multiplikation analog zur komponentenweisen Addition vorgehen.

2.4 Training neuronaler Netze

In diesem Abschnitt möchte ich kurz darauf eingehen, wie ein neuronales Netz lernt. Dabei werden wir uns auf die Grundlagen der bekanntesten Optimierungsverfahren beschränken und uns an [GBC16] Kapitel 4.3 halten.
Um zu messen, wie gut das Ergebnis eines neuronalen Netzes für eine bestimmte Eingabe ist, muss man zunächst festlegen, wann sich zwei Ereignisse ähneln. Dazu definiert man eine sogenannte *Loss Funktion* L, welche angibt, wie sehr sich das Ergebnis des neuronalen Netzes $f(x;\theta)$ von der Ground Truth $f^*(x)$ unterscheidet.
Eine beliebte Wahl für eine solche Funktion ist die mittlere quadratische Abweichung

(engl. mean squared error)

$$L(v, w) = \frac{1}{n} \sum_{i=1}^{n} (v_i - w_i)^2.$$

Halten wir einen bestimmten Input x_0 fest, so erhalten wir durch $F(\theta) = L(f(x_0; \theta), f^*(x_0))$ eine reellwertige Funktion $F : \mathbb{R}^n \to \mathbb{R}$ in θ.
Um lokale Minima solcher Funktionen zu finden, schlug bereits Cauchy in [Cau47] den sogenannten *Gradient Descent* vor.
Bei diesem Verfahren verwendet man, dass man sich eine partiell differenzierbare Funktion $F : \mathbb{R}^n \to \mathbb{R}$ als Graphen über $\mathbb{R}^n$ vorstellen kann, wobei $F(x)$ die „Höhe“ über $x \in \mathbb{R}^n$ beschreibt. Die Steigung dieses Graphen an der Stelle $x \in \mathbb{R}^n$, in eine Richtung $v \in \mathbb{R}^n$, ist dann durch die Richtungsableitung $D_v F(x) = \nabla F(x) \cdot v$ gegeben.
Diese Steigung ist genau dann minimal, wenn die Richtung v dem negativen Gradienten $-\nabla F(x)$ entspricht (vgl. [Sch14] S93).
Um ein lokales Minimum von F zu finden und somit den Fehler $L(f(x_0; \theta), f^*(x_0))$ zu minimieren, ist es also sinnvoll, die Funktion ein Stück in Richtung $-\nabla F(\theta)$ zu verschieben. Dazu wählen wir eine sogenannte *Lernrate* $\epsilon > 0$ und erhalten neue Parameter $\tilde{\theta}$ durch

$$\tilde{\theta} = \theta - \epsilon \nabla F(\theta).$$

Falls man mit diesem Schritt nicht gerade das lokale Minimum überschreitet, ist $F(\tilde{\theta})$ somit in jeder Iteration kleiner als $F(\theta)$.

Möchte man dieses Verfahren auf beliebige neuronale Netze anwenden, ist zuerst nicht klar, wie man den Gradienten $\nabla F(\theta)$ allgemein berechnen kann. Um dieses Problem zu lösen, hat man den sogenannten *Back-Propagation Algorithmus* entwickelt, welcher zum Beispiel in [GBC16] Kapitel 6.5 beschrieben wird.
Bei dieser Methode macht man sich zu Nutze, dass neuronale Netze oft aus Layern zusammengesetzt werden. Im Folgenden sei f ein neuronales Netz, welches aus den Layern $g_0, \ldots, g_n$ besteht, wobei g_0 das Input- und g_n das Output Layer ist.
Während des Trainings halten wir in jedem Schritt einen Input x_0 fest. Um die Notation zu vereinfachen, schreiben wir $\text{in}_i(\theta)$ für den Input des Layers g_i innerhalb der Berechnung von $f(x_0, \theta)$.
In diesem Sinne ist $f(x_0, \theta) = g_n(\text{in}_n(\theta))$ und $\text{in}_0 = (x_0, \theta)$. Da ein Layer mehrere andere Layer als Input verwenden kann, gilt allgemeiner $\text{in}_i = (g_{i_1}(\text{in}_{i_1}(\theta)), \cdots, g_{i_m}(\text{in}_{i_m}(\theta)), \theta)$ für bestimmte $i_j \in \mathbb{N}$. Setzen wir $\widetilde{g_{i_j}} = (0, \ldots, 0, g_{i_j}, 0, \ldots, 0)$ können wir dies auch als Summe

$$\text{in}_i = \sum_{j=1}^{m} \widetilde{g_{i_j}}(\text{in}_{i_j}(\theta)) + (0, \ldots, 0, \theta)$$

schreiben.

Für den Fall, dass f ein MLP ist, würde jedes Layer nur ein Layer als Input verwenden und der Input des i-ten Perzeptrons würde durch $\text{in}_i(\theta) = (g_{i-1}(\text{in}_{i-1}(\theta), \theta)$ gegeben werden.
Kehren wir nun zur Beschreibung des Algorithmus zurück. Zunächst berechnet man die Ergebnisse aller Layer $g_i(\text{in}_i(\theta))$, wobei wir x_0 immer noch festhalten. Dies wird *Forward Propagation* genannt, da wir hierbei bei dem Input Layer g_0 beginnen und uns zum Output Layer g_n vorarbeiten.
Nun wollen wir den Gradienten $\nabla F(\theta)$ berechnen. Da F eine reellwertige Funktion ist, entspricht ihr Gradient $\nabla F(\theta)$ ihrer, an θ ausgewerteten, Jakobimatrix $DF(\theta)$. Aus der Kettenregel erhalten wir, wenn wir uns an die Definition von L als Funktion $L(v, w)$ erinnern,

$$\begin{aligned} DF(\theta) &= D_\theta(L(f(x_0, \theta), f^*(x_0))) \\ &= D_\theta(L(g_n(\text{in}_n(\theta)), f^*(x_0)) \\ &= D_v L(g_n(\text{in}_n(\theta)), f^*(x_0)) \;\; D_\theta(g_n(\text{in}_n(\theta))). \end{aligned}$$

Wir beginnen also damit, die Ableitung des Output Layers g_n zu berechnen, und arbeiten uns zum Input Layer vor. Aus diesem Grund nennt man diesen Schritt die *Back Propagation*.
Nach der Kettenregel erhalten wir die Ableitung von $g_i(\text{in}_i)$ durch

$$D(g_i \circ \text{in}_i)(\theta) = Dg_i(\text{in}_i(\theta)) \;\; D\,\text{in}_i(\theta) \tag{2.1}$$

$$= Dg_i(g_{i_1}(\text{in}_{i_1}(\theta)), \cdots, g_{i_m}(\text{in}_{i_m}(\theta)), \theta) \;\; D\,\text{in}_i(\theta). \tag{2.2}$$

Gibt man die Ableitungen Dg_i bereits bei der Konstruktion des neuronalen Netzes an, so können wir $Dg_i(g_{i_1}(\text{in}_{i_1}(\theta)), \cdots, g_{i_m}(\text{in}_{i_m}(\theta)), \theta)$ direkt ausrechnen, da wir alle $g_i(\text{in}_i(\theta))$ bereits bei der Forward Propagation berechnet haben.
$D\,\text{in}_i(\theta)$ erhalten wir, indem wir die einzelnen Summanden ableiten, also

$$\begin{aligned} D\,\text{in}_i(\theta) &= D_\theta(\sum_{j=1}^{m} \widetilde{g_{i_j}}(\text{in}_{i_j}(\theta)) + (0, \ldots, 0, \theta)) \\ &= \sum_{j=1}^{m} D_\theta(\widetilde{g_{i_j}}(\text{in}_{i_j}(\theta))) + D_\theta(0, \ldots, 0, \theta). \end{aligned}$$

Da die Jakobimatrix von $\widetilde{g_{i_j}}(\text{in}_{i_j}(\theta)) = (0, \ldots, 0, g_{i_j}(\text{in}_{i_j}(\theta)), 0, \ldots, 0)$ nur 0-en und die Jakobimatrix von $g_{i_j}(\text{in}_{i_j}(\theta))$ enthält, müssen wir nun $D(g_{i_j} \circ \text{in}_{i_j})(\theta)$ berechnen.
Damit sind wir wieder in derselben Situation wie in 2.1 und indem wir induktiv immer weiter so vorgehen erhalten wir die gesamte Ableitung von $f(x_0, \theta)$.
Diese Methode bildet die Grundlage aller Optimierungsverfahren für neuronale Netze. Moderne Verbesserungen dieses Verfahrens konzentrieren sich hauptsächlich darauf, nicht nur ein lokales Minimum, sondern ein globales Minimum zu finden.
Außerdem verwenden neuere Methoden Sub-Gradienten, welche eine Verallgemeinerung des Gradienten für nicht differenzierbare Funktionen darstellen. Dadurch lässt sich das Verfahren zum Beispiel auch auf die ReLU Aktivierungsfunktion anwenden, welche in 0 nicht differenzierbar ist (vgl. [DHS11]).

2.5 Bestärkendes Lernen

In diesem Abschnitt möchte ich die Grundlagen des bestärkenden Lernens darlegen, die für tiefes bestärkendes Lernen notwendig sind. Dabei halte ich mich an [SB+98] für die Grundlagen und [MKS+15] für den Spezialfall des tiefen Lernens.
Beim tiefen bestärkenden Lernen wird ein neuronales Netz nicht anhand eines vorgegebenen Datensatzes trainiert, sondern mit Hilfe von bestärkenden Lernmethoden. Das Besondere am bestärkenden Lernen ist, dass das trainierte System selbständig lernt, welcher Output zu einem bestimmten Input passt. Dazu muss man lediglich eine Belohnungsfunktion festlegen, die beurteilt, wie gut oder schlecht die gewählten Input-Output Paare sind.
Trainiert man ein neuronales Netz mit bestärkendem Lernen, so kann man die in Zukunft erhaltenen Belohnungen abschätzen, indem man davon ausgeht, dass sich das System bei jedem Schritt so verhalten wird, wie es das bisher gelernte Netz tun würde. Somit kann man bei jedem Schritt den Output auswählen, der die zukünftigen Belohnungen maximieren würde.

Im Kontext des bestärkenden Lernens bezeichnet man das System, welches das Problem lösen soll, als *Agent* und alles, womit dieser Agent interagieren kann, wird *Umgebung* genannt. Um die abstrakte Umgebung fassbar zu machen, definiert man *Zustände* $\{s_1, \ldots, s_m\}$, in denen sich die Umgebung befinden kann und fasst diese im sogenannten *Zustandsraum* S zusammen.
Beim bestärkenden Lernen ist es oft sinnvoll eine Simulation als Umgebung zu wählen, anstatt den Agenten anhand der realen Welt zu trainieren. Da man die Geschwindigkeit einer Simulation stark erhöhen kann und es möglich ist, den Agenten gegen sich selbst antreten zu lassen, ist es möglich sehr viele Trainingsdaten in kurzer Zeit zu sammeln. Außerdem sind die Zustände innerhalb einer Simulation meistens einfacher zu bestimmen, als die Zustände in einem realen Szenario, welches beliebig kompliziert werden kann.
Der Agent besitzt außerdem eine Menge A an *Aktionen* $\{a_1, ..., a_n\}$, mit denen er die Umgebung beeinflussen kann.
Für bestärkendes Lernen geht man meistens davon aus, dass das Model einen oder mehrere *finale Zustand* besitzt, nach denen die Simulation beendet ist. Bei einem Computerspiel würde dieser Fall zum Beispiel eintreten, wenn man das Spiel verliert oder gewinnt. Eine Abfolge von aufeinanderfolgenden Zuständen und Aktionen $s_1, a_1, ...a_{n-1}, s_T$, welche mit einem finalen Zustand s_T endet, bezeichnet man als eine *Episode*.
Nach jeder Aktion erhält der Agent eine *Belohnung* $r(s_t, a_t, s_{t+1})$, die beschreibt, wie gut die gewählte Aktion in dieser Situation war. Dazu definiert man eine *Belohnungsfunktion* $r : S \times A \times S \to \mathbb{R}$ und bestimmt dadurch das Ziel des Agenten.

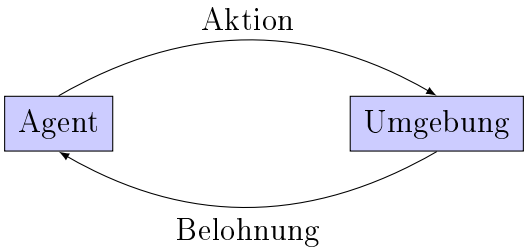


Abbildung 2.9: Interaktion des Agenten mit der Umgebung.

Das Verhalten des Agenten legen wir mithilfe einer *deterministischen Strategie* fest. Dies ist eine Abbildung $\pi : S \to A$, wobei $\pi(s)$ die Aktion festlegt, die im Zustand $s \in S$ ausgeführt wird. Theoretisch wäre auch eine *stochastische Strategie* möglich, bei der $\pi(s, a)$ die Wahrscheinlichkeit beschreibt, mit der im Zustand $s \in S$ die Aktion $a \in A$ gewählt wird. Ich möchte mich in dieser Arbeit jedoch auf Agenten mit deterministischen Strategien beschränken.
Das Ziel des Agenten ist es, eine optimale Strategie π^* zu lernen. Hierbei bedeutet optimal, dass der Agent im Laufe einer Episode die maximale Menge an Belohnungen erhält, wenn er dieser Strategie folgt.
Um zu beurteilen, wie erfolgreich ein Zustand s_t innerhalb einer bestimmten Episode $s_1, a_1, ..., a_{T-1}, s_T$ war, definiert man eine von der Episode abhängige Belohnung R_t. Die Idee hierbei ist, dass dieser Zustand zu einem gewissen Teil auch an der Belohnung der nächsten Zustände mitgewirkt hat. Dabei nimmt der Anteil des Zustandes s_t an den nachfolgenden Belohnungen immer weiter ab, da diese immer weniger von dem Zustand s_t beeinflusst wurden.
Um dies zu formulieren, wählt man einen sogenannten *Diskont-Wert* $\gamma \in [0, 1]$ und setzt

$$R_t(s_1, a_1, ..., a_{T-1}, s_T) := \sum_{i=t}^{T-1} \gamma^{i-t} r(s_i, a_i, s_{i+1}).$$

Daraus berechnet man eine *Aktion-Wert- oder Q-Wert-Funktion* $Q^\pi : S \times A \to \mathbb{R}$, die jedem Zustands-Aktions Paar $(s, a) \in S \times A$ die geschätzte Belohnung zuordnet, die der Agent erreicht, wenn er in Zustand s die Aktion a ausführt und dann der Strategie π folgt. Dazu setzen wir

$$Q^\pi(s, a) = \mathbb{E}_\pi\{R_t(s_1, a_1, ..., a_{T-1}, s_T) | s_t = s, a_t = a\}.$$

Wobei $\mathbb{E}_\pi\{R_t(s_1, a_1, ..., a_{T-1}, s_T) | s_t = s, a_t = a\}$ den Erwartungswert aller Episoden berechnet, die mit der Strategie π gespielt wurden und bei denen s, gefolgt von a, an der t-ten Stelle der Episode erscheint. Den Erwartungswert benötigen wir, da das Problem nicht deterministisch sein muss, das heißt, es können verschiedene Nachfolge-Zustände s_{t+1} auf das Zustand-Aktion Paar (s_t, a_t) folgen.
Wenn wir die optimale Aktion-Wert-Funktion $Q^*(s, a) = \max_\pi Q^\pi(s, a)$ kennen würden, könnten wir in jedem Zustand $s \in S$ diejenige Aktion $a \in A$ auswählen, für die $Q^*(s, a)$

am größten ist. Eine solche Strategie $\pi(s) = \text{argmax}_{a \in A} Q(s,a)$ für eine Q-Wert-Funktion Q nennen wir eine *gierige Strategie*.
Um Q^* zu approximieren, geht man beim bestärkenden Lernen davon aus, dass das zu lösende Problem die sogenannte *Markow-Eigenschaft* besitzt.

Definition 2.5.1. Sei S eine Menge an Zuständen und A eine Menge von Aktionen. S und A erfüllen die **Markow-Eigenschaft**, wenn, für einen beliebigen Ablauf $s_1, a_1, ...s_t, a_t$ von aufeinander folgenden Zuständen $s_i \in S$ und Aktionen $a_i \in A$, der nächste Zustand nur vom momentanen Zustand s_t und der gerade gewählten Aktion a_t abhängt. Das heißt für alle $s' \in S$ gilt

$$P(s_{t+1} = s' | s_t, a_t) = P(s_{t+1} = s' | s_1, a_1, ...s_t, a_t).$$

Hierbei bezeichnet $P(s_{t+1} = s' | s_1, a_1, ...a_t, s_t)$ die bedingte Wahrscheinlichkeit, dass der Agent im Zustand s' landet, unter der Bedingung, dass er bis jetzt die Zustände $s_1, \ldots, s_t$ durchlaufen hat und dort die Aktionen $a_1, \ldots, a_t$ ausgeführt hat.

Indem man zusätzlich davon ausgeht, dass jede Episode nach endlich vielen Schritten terminiert, erhält man ein endliches Markow-Entscheidungsproblem. Bei einem solchen Problem erfüllt Q^* die *Bellman Gleichung*

$$Q^*(s,a) = \sum_{s' \in S} P(s'|s,a)\big(r(s,a,s') + \gamma \max_{a' \in A} Q^*(s',a')\big).$$

Siehe [SB+98] Gleichung (3.20).
Eine beliebte Methode des bestärkenden Lernens ist es, diese Gleichung als Vorschrift zu verwenden, um die Q-Werte iterativ anzupassen. Das heißt in jedem Schritt i wählt man

$$Q^{i+1}(s,a) = \sum_{s' \in S} P(s'|s,a)\big(r(s,a,s') + \gamma \max_{a' \in A} Q^i(s',a')\big).$$

wobei man für Q^0 oft alle Werte auf 0 setzt.
Dies konvergiert aufgrund der Bellman Gleichung für $i \to \infty$ gegen Q^* (vgl. [SB+98] Kapitel 4.4).
In der Praxis ist es meistens nicht möglich diesen Limes zu bilden. Deswegen werden beim bestärkenden Lernen verschiedene Näherungsfunktionen $Q(s,a;\theta)$ verwendet, bei denen wir lediglich die Parameter θ anpassen, um Q^* zu approximieren.
Mnih et. al. schlagen hierzu in [MKS+15] ein neuronales Netz $Q(s,a;\theta)$ mit Parametern θ vor. Im Kontext des tiefen bestärkenden Lernens nennen sie ein solches neuronales Netz ein *Q-Netz*.
Dieses Netz wollen wir mit Hilfe der mittleren quadratische Abweichung trainieren. Die große Herausforderung dabei ist, dass wir die Werte der optimalen Q-Wert-Funktion Q^*, welche wir als Ground Truth für das Training des Q-Netzes benötigen, nicht kennen.
Deshalb verwenden wir die Bellman Gleichung, um uns iterativ der optimalen Aktion-Wert-Funktion Q^* zu nähern. Dabei berechnen wir in jeder Iteration des Gradient-Descent-Algorithmus (vgl. Kapitel 2.4) des Q-Netzes die erwartete mittlere quadratische Abweichung, indem wir das Q-Netz mit bereits in vorherigen Iterationen gelernten

Parametern $\tilde{\theta}$ als Ground Truth verwenden. Der Fehler in einer solchen Iteration ist dann

$$L_i(\theta_i) = \mathbb{E}_{s,a}\Bigg(\big(\sum_{s' \in S} P(s'|s,a)(r(s,a,s') + \gamma \max_{a' \in A} Q(s', a', \tilde{\theta})) \big) - Q(s,a;\theta_i) \Bigg)^2$$
$$= \sum_{s \in S, a \in A} P_\pi(s,a) \Bigg(\big(\sum_{s' \in S} P(s'|s,a)(r(s,a,s') + \gamma \max_{a' \in A} Q(s', a', \tilde{\theta})) \big) - Q(s,a;\theta_i) \Bigg)^2 .$$

Nach einer bestimmten Anzahl von Iterationen werden die Ziel Parameter $\tilde{\theta}$ durch die gelernten Parameter θ_i ersetzt.
Die genauen Übergangswahrscheinlichkeiten $P(s'|s,a)$ sind in den meisten Fällen nicht bekannt. Deshalb ist die Berechnung dieses Erwartungswertes in der Praxis für gewöhnlich nicht durchführbar.
Als Ersatz lässt man den Agenten die Simulation mit einer gierigen Strategie bezüglich der momentan gelernten Q-Wert-Funktion $Q(s,a;\theta_i)$ durchlaufen und betrachtet in jedem Zeitschritt den momentanen Zustand s_t, die gewählte Aktion a_t und den nächsten Zustand s_{t+1}. Daraus berechnet man den Fehler

$$L_i(\theta_i) = \Bigg((r(s_t, a_t, s_{t+1}) + \gamma \max_{a' \in A} Q(s_{t+1}, a', \tilde{\theta})) \big) - Q(s_t, a_t; \theta_i) \Bigg)^2$$

und verwendet dann einen Gradient-Descent Schritt, um diesen Fehler zu verringern (vgl. Kapitel 2.4).
Durchläuft man hierbei jede Episode schrittweise vom Anfang bis zum Ende, so bestehen starke Abhängigkeiten zwischen den einzelnen Zuständen. Befindet man sich zum Beispiel bei einem Computerspiel auf der linken Seite des Bildschirms, so werden nur die Aktionen bestärkt, die auf der linken Seite sinnvoll sind. Kommt der Agent danach auf die rechte Seite, so werden die Aktionen belohnt, die dort sinnvoll sind. Aktionen, die auf der linken Seite erfolgreich waren, werden vom Agenten eventuell immer noch bevorzugt, aber nun auf der rechten Seite vielleicht sogar bestraft. Dadurch würde es zu einer Oszillation kommen, welche die Konvergenz des Algorithmus unmöglich macht.
Als Gegenmaßnahme verwenden Minh et. al. die sogenannte *action replay* Methode. Dabei speichert man in jedem Schritt ein Tupel $e_t = (s_t, a_t, r_t, s_{t+1})$, bestehend aus dem momentanen Zustand s_t, der gewählten Aktion a_t, dem nächsten Zustand s_{t+1} und der erhaltenen Belohnung $r_t := r(s_t, a_t, s_{t+1})$, in einer *Erinnerungs-Menge* $D = \{e_1, ..., e_t\}$. Nachdem diese Menge durch einige zufällig ausgeführte Episoden gefüllt wurde, wird in jedem Schritt ein *Stapel (engl. batch)* aus zufälligen Tupeln e_i gewählt und auf diesem Stapel wird das Q-Netz mit der oben gegebenen Fehler-Funktion trainiert. Dies hat den zusätzlichen Vorteil, dass die meisten modernen Trainingsmethoden neuronaler Netze für das Training auf Stapeln optimiert sind. Außerdem wird bei diesem Verfahren jeder Zustand mehrmals verwendet, was die Effizienz des Algorithmus erhöht.

Zuletzt ist noch anzumerken, dass es nicht unbedingt sinnvoll ist, während des Trainingsvorgangs strikt nach einer gierigen Strategie vorzugehen. Wählt man immer nur

die Aktionen, bei denen man bisher die besten Ergebnisse erzielt hat, so erforscht man nie neue Aktionen, die eventuell besser geeignet wären. Somit läuft man Gefahr, nur ein lokales Minimum der Loss Funktion zu finden. Um dies zu umgehen, wählt man während des Trainings mit einer gewissen Wahrscheinlichkeit $\epsilon > 0$ eine zufällige Aktion. Dass wir während des Trainings nicht dieselbe Strategie verwenden, die wir trainieren, bezeichnet man als *off-policy* Training.

3 Approximationseigentschaft des MLP Modells

In diesem Kapitel wollen wir die universelle Approximationseigenschaft von MLP's mit einem Hidden Layer und bestimmten Aktivierungsfunktionen beweisen.
Da die meisten neuronalen Netze ein fully connected Layer als Output Layer verwenden, reicht dies oft aus, um die theoretische Approximationseignschaft des Netzes zu beweisen.
Im Abschnitt 2.3.1 haben wir gesehen, dass ein MLP mit einem Hidden Layer eine Funktion

$$f(x; W, b) = W_2\sigma(W_1x + b_1) + b_2,$$

mit $W := (W_1, W_2) \in \mathbb{R}^{r\times n} \times \mathbb{R}^{m\times r}$ und $B := (b_1, b_2) \in \mathbb{R}^r \times \mathbb{R}^m$ ist. Dabei beschreibt σ die komponentenweise Anwendung einer Aktivierungsfunktion $\sigma : \mathbb{R} \to \mathbb{R}$.
Hierbei haben wir als Aktivierungsfunktion des output-Layers die Identität gewählt, da andere Aktivierungsfunktionen eine unnötige Einschränkung bedeuten würden. Eine Aktivierungsfunktion mit Wertebereich $[-1, 1]$ zum Beispiel würde die Approximation von Funktionen mit größerem Wertebereich unmöglich machen.
Wir wollen nun beweisen, dass Funktionen dieser Form jede stetige Funktion $g \in C(\mathbb{R}^n, \mathbb{R}^m)$ approximieren können. Obwohl wir uns hierbei auf stetige Funktionen beschränken ist diese Aussage für viele praktische Anwendungen interessant. Bei der Analyse von menschlichen Emotionen mit Hilfe neuronaler Netze zum Beispiel, werden die Emotionen meist nicht diskret im Sinne von „glücklich“ oder „nicht glücklich“ ausgewertet. Stattdessen verwendet man Werte im Bereich $[0, 1]$, um kontinuierlich zu beschreiben, wie sehr eine bestimmte Emotion momentan empfunden wird.
Unter Approximation verstehen wir eine beliebig nahe Annäherung bezüglich der Topologie der kompakten Konvergenz. Das heißt für jedes kompakte $K \subset \mathbb{R}^n$ und jedes $\epsilon > 0$ sollen Parameter W, b existieren, sodass

$$\|f(x; W, b) - g(x)\|_K < \epsilon,$$

für die Supremumsnorm $\|.\|_K$ auf K, gilt.
Da f und g beide stetig sind, wollen wir also:

$$\sup_{x\in K} \|f(x; W, b) - g(x)\| = \max_{x\in K} \|f(x; W, b) - g(x)\| < \epsilon.$$

Wir können das MLP Modell für diesen Beweis ohne Einschränkungen auf den Fall $b_2 = 0$ und $m = 1$ reduzieren. Um die Notation zu vereinfachen teilen wir diese Aussage in zwei kurze Lemmata auf.

Lemma 3.0.1. *Seien $f = (f_1, ..., f_m)$ und $g = (g_1, ..., g_m) \in C(\mathbb{R}^n, \mathbb{R}^m)$ und sei $K \subset \mathbb{R}^n$ kompakt. Gilt für $\epsilon > 0$ und für alle $i = 1, ..., m$*

$$\|f_i - g_i\|_{sup_K} < \epsilon,$$

so gilt

$$\|f - g\|_{sup_K} < m\epsilon.$$

Beweis. Aus $\|f_i - g_i\|_{sup_K} < \epsilon$ folgt

$$\forall x \in K : \|e_i(f_i(x) - g_i(x)\| < \epsilon.$$

Für die Standardbasisvektoren e_i gilt $\sum_{i=1}^m e_i f_i = f$ und $\sum_{i=1}^m e_i g_i = g$. Mit der Dreiecksungleichung erhalten wir somit

$$\forall x \in K : \|f(x) - g(x)\| \leq \sum_{i=1}^{m} \|e_i(f_i(x) - g_i(x))\| < m\epsilon.$$

□

Lemma 3.0.2. *Sei $K \subset \mathbb{R}^n$ kompakt. Angenommen für alle $g \in C(\mathbb{R}^n, \mathbb{R})$ und für alle $\epsilon > 0$ existieren $W_1 \in \mathbb{R}^{r \times n}, W_2 \in \mathbb{R}^{1 \times r}$ und $b_1 \in \mathbb{R}^r$ mit $r \in \mathbb{N}$, sodass*

$$\max_{x \in K} \|W_2 \cdot \sigma(W_1 x + b_1) - g(x)\| < \epsilon,$$

dann existieren auch für alle $g \in C(\mathbb{R}^n, \mathbb{R}^m)$ Parameter W und b eines MLP mit einem Hidden Layer $f(x; W, b)$, sodass

$$\max_{x \in K} \|f(x; W, b) - g(x)\| < \epsilon.$$

Beweis. Sei $K \subset \mathbb{R}^n$ kompakt, $g = (g_1, ..., g_m) \in C(\mathbb{R}^n, \mathbb{R}^m)$ und $\epsilon > 0$.
Nach der Voraussetzung finden wir für alle $i = 1, ..., m$ Parameter $W_1^i \in \mathbb{R}^{r_i \times n}, W_2^i \in \mathbb{R}^{1 \times r_i}$ und $b_1^i \in \mathbb{R}^{r_i}$, sodass

$$\max_{x \in K} \left\|W_2^i \cdot \sigma(W_1^i x + b_1^i) - g_i(x)\right\| < \frac{\epsilon}{m}.$$

Wählen wir nun W_2 als

$$W_2 = \begin{bmatrix} W_2^1 & 0 & \dots & 0 \\ 0 \dots 0 & W_2^2 & \dots & 0 \\ \vdots & 0 \dots 0 & \ddots & \vdots \\ 0 & \dots & 0 & W_2^m \end{bmatrix},$$

also die Matrix, die in jeder Zeile jeweils den entsprechenden Vektor W_2^i und sonst nur Nullen enthält. Außerdem setzen wir für W_1 und b_1 die W_1^i und b_1^i folgendermaßen zusammen:

$$W_1 = \begin{bmatrix} W_1^1 \\ \vdots \\ W_1^m \end{bmatrix} \quad \text{und} \quad b_1 = \begin{pmatrix} b_1^1 \\ \vdots \\ b_1^m \end{pmatrix}.$$

Somit gilt für die Parameter $W = (W_1, W_2)$ und $b = (b_1, 0)$

$$f(x; W, b) = \begin{pmatrix} W_2^1 \cdot \sigma(W_1^1 x + b_1^1) \\ \vdots \\ W_2^m \cdot \sigma(W_1^m x + b_1^m) \end{pmatrix}.$$

Nach dem vorherigen Lemma erhalten wir also

$$\|f - g\|_{sup_K} < m\frac{\epsilon}{m} = \epsilon,$$

wie gefordert. □

Für den Beweis der Approximationseigenschaft beschränken wir uns also auf Funktionen der Form

$$f(x; (W, w), b) = w \cdot \sigma(Wx + b) = \sum_{i=1}^{r} w_i \sigma(W_i \cdot x + b_i),$$

mit $r \in \mathbb{N}, w, b \in \mathbb{R}^r$ und $W \in \mathbb{R}^{r \times n}$. Hierbei beschreibt W_i die i-te Zeile von W.
Dabei legen wir die Anzahl der Einheiten des Hidden Layers r nicht fest. Tatsächlich ist mir kein Beweis für die Approximationseigenschaft von MLP's mit festgelegter Breite des Hidden Layers bekannt.
Wir können die Menge der Funktionen, die uns interessieren, folgendermaßen zusammenfassen:

Definition 3.0.3. Sei σ eine Aktivierungsfunktion und $X \subset \mathbb{R}^n$. Definiere:

$$M(\sigma; X) := span\{f \in C(X) | f(x) = \sigma(\omega \cdot x + \theta) \quad \text{mit} \quad \omega \in \mathbb{R}^n, \theta \in \mathbb{R}\}$$

Für $X = \mathbb{R}^n$, schreiben wir auch nur $M(\sigma)$.

Hierbei verwende ich θ anstatt b für den Bias, da dies in längeren Beweisen schneller ins Auge fällt.
Damit können wir das Ziel von Kapitel 3 formulieren.
Wir wollen den folgenden Satz für bestimmte Aktivierungsfunktionen σ beweisen.

Satz 3.0.4 (Approximationstheorem)**.** *Die Menge $M(\sigma)$ liegt, bezüglich der Topologie der kompakten Konvergenz, dicht in $C(\mathbb{R}^n)$.*

3.1 Erste Versuche

Einer der ersten Beweise des Approximationstheorems erschien 1989 in [Cyb89]. Professor Cybenko beweist dort das Approximationstheorem für eine Art von Aktivierungsfunktionen, welche er *discriminatory* nennt. Dies definiert er folgendermaßen:

Definition 3.1.1. Eine Funktion $\sigma \in C(\mathbb{R})$ heißt **discriminatory**, wenn

$$\forall \omega \in \mathbb{R}^n, \quad \forall \theta \in \mathbb{R} : \quad \int_{I^n} \sigma(\omega \cdot x + \theta) d\mu(x) = 0$$

für ein Borel-Maß μ impliziert, dass $\mu = 0$.

Zunächst führt er dann den Beweis nur auf dem kompakten Einheitsintervall I^n.

Satz 3.1.2. *Sei σ eine stetige, discriminatory Funktion. Dann liegt $M(\sigma; I^n)$ dicht in $C(I^n)$*

Beweis. $M(\sigma; I^n)$ ist per Definition ein Untervektorraum von $C(I^n)$. Somit ist auch der Abschluss $\overline{M(\sigma; I^n)}$ von $M(\sigma; I^n)$ ein Untervektorraum von $C(I^n)$.
Nehmen wir an, dass $\overline{M(\sigma; I^n)} \neq C(I^n)$ gilt. Dann gibt es ein $x \in C(I^n)$, welches nicht in $\overline{M(\sigma; I^n)}$ liegt.
Nach dem Satz von Hahn-Banach 3.3.1 erhalten wir demnach ein stetiges lineares Funktional L, welches auf $M(\sigma, I^n)$ verschwindet, aber nicht auf ganz $C(I^n)$.
Da L ein stetiges lineares Funktional zwischen normierten Vektorräumen ist, ist es auch beschränkt. Nach dem Riesz'schen Darstellungssatz 3.3.2 gilt für L somit

$$\forall f \in C(I^n) : \quad L(f) = \int_{I^n} f(x) d\mu(x)$$

für ein positives Borel-Maß μ.
Da $\sigma(\omega \cdot x + \theta)$ für alle $\theta \in \mathbb{R}$ und $\omega \in \mathbb{R}^n$ in $M(\sigma; I^n)$ liegt, erhalten wir

$$0 = L(\sigma(\omega \cdot x + \theta)) = \int_{I^n} \sigma(\omega \cdot x + \theta) d\mu(x)$$

für alle $\theta \in \mathbb{R}$ und $\omega \in \mathbb{R}^n$.
Da wir σ discriminatory angenommen haben, gilt also $\mu = 0$. Das würde aber bedeuten, dass L auf ganz $C(I^n)$ verschwindet, was im Widerspruch zu dem von Hahn-Banach gelieferten L steht.
Demnach muss $\overline{M(\sigma; I^n)}$ schon ganz $C(I^n)$ sein und somit liegt $M(\sigma; I^n)$ dicht in $C(I^n)$. □

Cybenko begründet im weiteren Verlauf des Artikels, dass man von dem kompaktem Einheitsintervall auf jede kompakte Teilmenge schließen kann und dass zum Beispiel die Sigmoid Funktion discriminatory ist. Da wir im Anschluss eine stärkere Version des Approximationstheorems beweisen werden, möchte ich hier aber nur diese Grundzüge darlegen.

3.2 Approximationstheorem für nicht polynomielle Aktivierungsfunktionen

Genau 10 Jahre nach Cybenko veröffentlichte Allan Pinkus in [Pin99] einen noch stärkeren Beweis für die Approximationseigenschaft von MLP's mit einem Hidden Layer. In diesem zeigt er, dass es theoretisch ausreicht eine nicht polynomielle stetige Funktion als Aktivierungsfunktion zu wählen, um alle stetigen Funktionen approximieren zu können. Dabei leitet Pinkus die Dichte der MLP's von der Dichte sogenannter ridge Funktionen ab.

Definition 3.2.1. Eine **ridge Funktion** ist eine Funktion $F : \mathbb{R}^n \to \mathbb{R}$ der Form

$$F(x) = f(a \cdot x)$$

für ein $a \in \mathbb{R}^n \setminus \{0\}$ und ein $f \in C(\mathbb{R})$. Wir schreiben

$$R := span\{f(a \cdot x) | a \in \mathbb{R}^n, f \in C(\mathbb{R})\}$$

für den von den ridge Funktionen aufgespannten Untervektorraum von $C(\mathbb{R}^n)$.

Diese Menge ist interessant für uns, da die Vektoren der Erzeugendensysteme der Vektorräume $M(\sigma)$ ridge Funktionen sind. Dass ein solcher erzeugender Vektor $\sigma(\omega \cdot x + \theta)$, mit $\omega \in \mathbb{R}^n$ und $\theta \in \mathbb{R}$ eine ridge Funktion ist, sieht man, wenn man $a = \omega$ und $f(x) = \sigma(x + \theta)$ setzt.
Damit es überhaupt möglich ist, dass ein $M(\sigma)$ dicht in $C(\mathbb{R}^n)$ liegt, muss zunächst also R dicht in $C(\mathbb{R}^n)$ sein.
Wir können in folgendem Satz sogar zeigen, dass die Menge der ridge Funktionen mit Vektoren a aus bestimmten Teilmengen $A \subset \mathbb{R}^n$ dicht in $C(\mathbb{R}^n)$ liegt.
Davor benötigen wir jedoch noch ein Hilfslemma.

Lemma 3.2.2. *Sei $\mathbb{K}$ ein Körper und X ein normierter $\mathbb{K}$-Vektorraum. Ein Untervektorraum $X_0 \subset X$ liegt genau dann dicht in X, wenn für jedes stetige lineare Funktional $\phi : X \to \mathbb{K}$ gilt:*

$$\phi|_{X_0} \equiv 0 \quad \Rightarrow \quad \phi \equiv 0$$

Beweis. „⇒"
Sei $\phi : X \to \mathbb{K}$ ein stetiges lineares Funktional, welches auf $X_0 \subset X$ verschwindet. Nehmen wir weiter an, dass ein $x_0 \in X \setminus X_0$ mit $\phi(x_0) \neq 0$ existiert, dann können wir $0 < \epsilon < \|\phi(x_0)\|_{\mathbb{K}}$ wählen.
Aufgrund der Dichtheit von X_0 in X, finden wir für jedes $\delta > 0$ ein $y \in X_0$, sodass $\|x_0 - y\|_X < \delta$. Wegen $y \in X_0$ gilt $\phi(y) = 0$, also $\|\phi(x_0) - \phi(y)\|_{\mathbb{K}} = \|\phi(x_0)\|_{\mathbb{K}} > \epsilon$.
Somit haben wir ein $\epsilon > 0$ gefunden, für das kein $\delta > 0$ existiert, welches

$$\|x_0 - y\|_X < \delta \Rightarrow \|\phi(x_0) - \phi(y)\|_{\mathbb{K}} < \epsilon$$

erfüllt. Dies widerspricht der Stetigkeit von ϕ.

„⇐“
Angenommen es existiert ein $x \in X \setminus \overline{X_0}$, dann gibt es nach dem Satz von Hahn-Banach 3.3.1 ein stetiges lineares Funktional ϕ mit $\phi(X_0) = 0$ aber $\phi(x) \neq 0$. Dies widerspricht der Voraussetzung. □

Für den nächsten Beweis benötigen wir neue Notationen.

Definition 3.2.3. Sei $n \in \mathbb{N}$. Mit $\mathbb{Z}_+^n$ bezeichnen wir die Menge der Mulitindizes $\{(m_1, ..., m_n) | m_i \in \mathbb{N}_0\}$. Für $m = (m_1, ..., m_n) \in \mathbb{Z}_+^n$ setzen wir

$$|m| := \sum_{i=1}^{n} m_i.$$

Definition 3.2.4. Mit H_k^n bezeichnen wir die Menge der homogenen Polynome $p \in \mathbb{R}[X_1, ..., X_n]$ vom Grad k.

Nun können wir die Dichte der ridge Funktionen beweisen. Dabei orientiere ich mich an [LP93], wobei der Beweis dort sogar für allgemeinere ridge Funkionen der Form $f(Ax)$ für Matrizen $A \in \mathbb{R}^{m \times n}$ geführt wird. Um nicht zu weit vom Thema abzuweichen, beschränke ich mich hier jedoch auf den Fall $A = a \in \mathbb{R}^n$. Der Beweis im allgemeineren Fall verläuft analog.

Satz 3.2.5. *Die Menge der ridge Funktionen auf $A \subset \mathbb{R}^n$*

$$R(A) := span\{f(a \cdot x) | a \in A, f \in C(\mathbb{R})\}$$

liegt genau dann dicht in $C(\mathbb{R}^n)$, bezüglich der Topologie der gleichmäßigen Konvergenz auf Kompakta, wenn kein nicht triviales homogenes Polynom $p \in \mathbb{R}[x_1, ..., x_n]$ existiert, welches auf A verschwindet.

Beweis. „⇒“
Angenommen es existiert ein homogenes Polynom $p \not\equiv 0$ vom Grad k, das auf A verschwindet, dann ist dieses von der Form

$$p(x) = \sum_{|m|=k} b_m x^m,$$

mit $b_m \in \mathbb{R}$ und $x^m = \prod_{i=1}^n x_i^m$.
Nun wählen wir ein $\phi \in C_0^\infty(\mathbb{R}^n)$ mit $\phi \not\equiv 0$ (siehe 3.3.6 für Beispiele solcher Funktionen). Für $m \in \mathbb{Z}_+^n$ mit $|m| = k$ setzen wir

$$D^m = \frac{\partial^k}{\partial x_1^{m_1} ... \partial x_n^{m_n}}$$

und können damit folgende Funktion definieren:

$$\psi(x) = \sum_{|m|=k} b_m D^m \phi(x).$$

Da ϕ außerhalb seines Trägers konstant 0 ist, verschwinden dort auch alle $D^m\phi(x)$. Somit gilt $supp(\psi) \subset supp(\phi)$. Insbesondere ist dadurch der Träger von ψ als abgeschlossene Teilmenge des Kompaktums $supp(\phi)$ wieder kompakt, also $\psi \in C_0(\mathbb{R}^n)$.
Angenommen $D^m\phi$ verschwindet überall, für ein $m \in \mathbb{Z}_+^n$ mit $|m| = k$. Dann gibt es eine partielle Ableitung $\frac{\partial}{\partial x_i} D^{\tilde{m}}\phi$, welche als erstes verschwindet. Das bedeutet, dass $D^{\tilde{m}}\phi$ nicht von x_i abhängt und dass $supp(D^{\tilde{m}}\phi)$ nicht leer ist. Also ist $supp(D^{\tilde{m}}\phi)$ von der Form $\mathbb{R} \times U$ für ein $U \subset \mathbb{R}^{n-1}$ und somit insbesondere nicht beschränkt. Analog zur Begründung für den Träger von ψ müsste $supp(D^{\tilde{m}}\phi)$ aber kompakt sein.
Es verschwindet also keines der $D^m\phi(x)$ auf ganz A.
Da außerdem die b_m die Koeffizienten des nicht trivialen Polynoms p sind, gilt $\psi \not\equiv 0$.
Weiter wissen wir über die Fouriertransformation $\widehat{\psi}$ von ψ, aufgrund der Linearität und der Ableitungseigenschaft der Fouriertransformation:

$$\widehat{\psi}(x) = \sum_{|m|=k} b_m \widehat{D^m\phi}(x) = \sum_{|m|=k} b_m i^k x_1^{m_1} ... x_n^{m_n} \widehat{\phi}(x) = i^k \widehat{\phi}(x) \sum_{|m|=k} b_m x^m = i^k \widehat{\phi}(x) p(x).$$

Im Folgenden wollen wir

$$\int_{\mathbb{R}^n} g(a \cdot x)\psi(x)dx = 0$$

für alle $g \in C(\mathbb{R})$ und $a \in A$ zeigen. Daraus werden wir mit Hilfe von Lemma 3.2.2 einen Widerspruch herleiten.
Sei $a \in A$, dann können wir a zu einer orthogonal Basis von $\mathbb{R}^n$ erweitern. In dieser Basis können wir jedes $x \in \mathbb{R}^n$ als $x = (x', x'')$, mit $x' \in \mathbb{R}$ und $x'' \in \mathbb{R}^{n-1}$, schreiben. Dabei bezeichnet $(x', 0) = x'a$ die Projektion von x auf $span(a)$ und $(0, x'')$ sein orthogonales Komplement.
Nach dem Satz von Fubini gilt für jedes $f \in C_0^\infty(\mathbb{R}^n)$:

$$\int_{\mathbb{R}^n} f(x)dx = \int_{\mathbb{R}\times\mathbb{R}^{n-1}} f(x', x'')d(x', x'') = \int_{\mathbb{R}} [\int_{\mathbb{R}^{n-1}} f(x', x'')dx'']dx'$$

Sei nun $c' \in \mathbb{R}$, dann ist $c = (c', 0) \in span(a)$. Da p auf A verschwindet, gilt $p(a) = 0$, und weil p außerdem homogen ist, gilt $p(span(a)) = 0$. Somit erhalten wir

$$0 = i^k \widehat{\phi}(c)p(c) = \widehat{\psi}(c) = \frac{1}{(2\pi)^{\frac{n}{2}}} \int_{\mathbb{R}^n} \psi(x)e^{-ic\cdot x}dx = \frac{1}{(2\pi)^{\frac{n}{2}}} \int_{\mathbb{R}} [\int_{\mathbb{R}^{n-1}} \psi(x', x'')dx'']e^{-ic'\cdot x'}dx'.$$

Wenn wir nun

$$H(x') := \int_{\mathbb{R}^{n-1}} \psi(x', x'')dx''$$

definieren, dann ist H wieder in $C_0^\infty(\mathbb{R})$. Außerdem erhalten wir aus der vorherigen Gleichung

$$0 = \frac{1}{(2\pi)^{\frac{1}{2}}} \int_{\mathbb{R}} H(x')e^{-ic'\cdot x'}dx' = \widehat{H}(c')$$

für alle $c' \in \mathbb{R}$. Da die Fouriertransformation ein lineares Funktional ist, gilt somit $H \equiv 0$.

Für $x = (x', x'') \in \mathbb{R}^n$ ist $(0, x'')$ orthogonal zu $span(a)$, demnach gilt:

$$a \cdot x = a \cdot (x', 0) + a \cdot (0, x'') = a \cdot (x', 0).$$

Somit erhalten wir für jedes $g \in C(\mathbb{R})$

$$\int_{\mathbb{R}^n} g(a \cdot x)\psi(x)dx = \int_{\mathbb{R}} [\int_{\mathbb{R}^{n-1}} \psi(x', x'')dx'']g(a \cdot (x', 0))dx' = \int_{\mathbb{R}} H(x')g((a \cdot (x', 0))dx' = 0,$$

wie gewünscht.
Demnach ist $L(f) := \int_{\mathbb{R}^n} f(x)\psi(x)dx$ ein nicht triviales stetiges lineares Funktional, welches auf ganz $R(A)$ verschwindet. Im Beweis von Lemma 3.2.2 haben wir für die zweite Richtung nur den Satz von Hahn-Banach benötigt. Da wir diesen Satz auch auf $C(\mathbb{R}^n)$, versehen mit der Topologie der kompakten Konvergenz anwenden können, gilt auch hier zumindest die zweite Richtung des Lemmas.
Über Kontraposition erhalten wir aus dieser Richtung, dass $R(A)$ bezüglich der Topologie der kompakten Konvergenz nicht dicht in $C(\mathbb{R}^n)$ liegt, wenn es ein solches nicht triviales stetiges lineares Funktional gibt, welches auf ganz $R(A)$ verschwindet.
Da wir in der Voraussetzung angenommen haben, dass $R(A)$ dicht in $C(\mathbb{R}^n)$ liegt, kann es also kein nicht triviales homogenes Polynom existieren, welches auf A verschwindet.

„$\Leftarrow$“
Für $k \in \mathbb{N}$ gibt es nach der Voraussetzung kein nicht triviales homogenes Polynom $p \in H_k^n$, welches auf ganz A verschwindet. Wir wollen nun zeigen, dass dann ganz H_k^n in $R(A)$ liegt. Daraus werden wir mit dem Approximationssatz von Weierstraß die Dichtheit von $R(A)$ in $C(\mathbb{R})$ ableiten.
Sei $d \in span(A) = span\{a \in \mathbb{R}^n | a \in A\}$, dann existiert ein $\lambda \in \mathbb{R}$ und ein $a \in A$, sodass $d = \lambda a$ gilt. Setzen wir nun $f(a \cdot x) := (\lambda a \cdot x)^k = (d \cdot x)^k$, so sieht man, dass $(d \cdot x)^k$ in $R(A)$ liegt.
Jedes lineare Funktional L auf H_k^n ist durch seine Werte auf den Basisvektoren eindeutig definiert, also durch $L(x^m)$ mit $|m| = k$. Wenn wir D^m genauso wie im ersten Teil des Beweises definieren, dann gilt für $|m| = |\tilde{m}|$:

$$D^m x^{\tilde{m}} = \begin{cases} 0 & \text{falls} \quad m \neq \tilde{m} \\ \prod_{i=1}^n m_i! & \text{falls} \quad m = \tilde{m} \end{cases}.$$

Wählen wir nun $q = \sum_{|m|=k} q_m x^m \in H_k^n$ mit $q_m := L(x^m)(\prod_{i=1}^n m_i!)^{-1}$, dann repräsentiert q das lineare Funktional L, da

$$q(D)x^m = L(x^m)(\prod_{i=1}^n m_i!)(\prod_{i=1}^n m_i!)^{-1} = L(x^m),$$

für alle $m \in \mathbb{Z}_+^n$ mit $|m| = k$ gilt.
Angenommen $span\{(d \cdot x)^k | d \in span(A)\} \neq H_k^n$. Dann finden wir linear unabhängige Vektoren $v_i \in H_k^n$, welche zu den $(d \cdot x)^k$ linear unabhängig sind und diese zu einem erzeugenden System von H_k^n ergänzen.

Mit diesem erzeugenden System können wir ein nicht triviales lineares Funktional auf H_k^n definieren, welches auf $span\{(d\cdot x)^k | d \in span(A)\}$ verschwindet, indem wir folgendes setzen:

$$\begin{aligned} L(v_1) &= 1 \\ L(v_i) &= 0 \quad \text{für} \quad i > 0 \\ L((d\cdot x)^k) &= 0 \quad \forall d \in span(A). \end{aligned}$$

Sei q das homogene Polynom, welches L, wie oben beschrieben, repräsentiert.
Die Monome von $(d\cdot x)^k$ sind nach der Multinomialformel von der Form $\frac{k!}{\prod_{i=1}^n m_i!} d^m x^m$ für ein $m \in \mathbb{Z}_+^n, |m| = k$. Für diese Monome gilt

$$q(D)\frac{k!}{\prod_{i=1}^n m_i!} d^m x^m = q_m \frac{k!}{\prod_{i=1}^n m_i!} d^m \prod_{i=1}^n m_i! = k! q_m d^m.$$

Insgesamt also

$$q(D)(d\cdot x)^k = k! q(d).$$

Für alle $d \in span(A)$ gilt nach unserer Definition $L((d\cdot x)^k) = 0$. Mit der gerade hergeleiteten Formel erhalten wir also

$$0 = L((d\cdot x)^k) = q(D)(d\cdot x)^k = k! q(d)$$

und somit verschwindet q auf ganz A.
Nach der Voraussetzung sind dann aber q und dadurch auch L trivial. Das steht im Widerspruch zu unserer Wahl von L, also muss $span\{(d\cdot x)^k | d \in span(A)\} = H_k^n$ gelten. Insbesondere erhalten wir dadurch $H_k^n \subset R(A)$.
Da diese Argumentation für beliebige $k \in \mathbb{N}_0$ funktioniert, sind alle homogenen Polynome in $R(A)$.
Der Vektorraum $R(A)$ enthält also insbesondere alle Monome und somit alle Polynome. Nach dem Satz von Weierstrass 3.3.4 liegt $R(A)$ demnach dicht in $C(\mathbb{R}^n)$, bezüglich der Topologie der kompakten Konvergenz. □

Da nur das triviale Polynom auf ganz $\mathbb{R}^n$ verschwindet, gilt dieser Satz insbesondere für die globalen ridge Funktionen R.
Unser Ziel ist es nun auszunutzen, dass es kein homogenes nicht triviales Polynom gibt, welches auf ganz S^{n-1} verschwindet. Mit Hilfe dieser Tatsache werden wir zeigen, dass es genügt den Approximationssatz für $x \in \mathbb{R}$ anstatt für $x \in \mathbb{R}^n$ zu zeigen.

Definition 3.2.6. Seien $\Lambda, \Theta \subset \mathbb{R}$ und $A \subset \mathbb{R}^n$. Für $\sigma \in C(\mathbb{R})$, definiere:

$$\begin{aligned} \Lambda \times A &:= \{\lambda a | \lambda \in \Lambda, a \in A\} \\ N(\sigma; \Lambda, \Theta) &:= span\{\sigma(\lambda t + \theta) | \lambda \in \Lambda, \theta \in \Theta\} \\ M(\sigma; \Lambda \times A, \Theta) &:= span\{\sigma(w\cdot x + \theta) | w \in \Lambda \times A, \theta \in \Theta\} \end{aligned}$$

Proposition 3.2.7. *Seien Λ, Θ, σ wie in der vorherigen Definition 3.2.6 mit der zusätzlichen Voraussetzung, dass $N(\sigma; \Lambda, \Theta)$ dicht in $C(\mathbb{R})$ bezüglich der Topologie der kompakten Konvergenz liegt. Nehmen wir weiter an, dass ein $A \subset S^{n-1}$ existiert, sodass $R(A)$ in derselben Topologie dicht in $C(\mathbb{R}^n)$ liegt.*
Dann ist $M(\sigma; \Lambda \times A, \Theta)$ dicht in $C(\mathbb{R}^n)$ bezüglich der Topologie der gleichmäßigen Konvergenz auf Kompakta.

Beweis. Sei $K \subset \mathbb{R}^n$ kompakt, $g \in C(K)$ und $\epsilon > 0$ gegeben.
Da $R(A)$ bezüglich der Topologie der gleichmäßigen Konvergenz dicht in $C(\mathbb{R}^n)$ liegt, gibt es $f_i \in C(\mathbb{R}), \rho_i \in \mathbb{R}$ und $a_i \in A$ für $i = 1, ..., r$ und $r \in \mathbb{N}$, sodass

$$\forall x \in K: \quad \left| g(x) - \sum_{i=1}^{r} \rho_i f_i(a_i \cdot x) \right| < \frac{\epsilon}{2}.$$

Für $i \in 1, ..., r$ ist $\{a_i \cdot x | x \in K\} \subset \mathbb{R}$ kompakt als Bild des Kompaktums K unter der stetigen Multiplikation mit a_i.
Da $N(\sigma; \Lambda, \Theta)$ bezüglich der Topologie der gleichmäßigen Konvergenz dicht in $C(\mathbb{R})$ liegt, finden wir $c_{ij} \in \mathbb{R}, \lambda_{ij} \in \Lambda$ und $\theta_{ij} \in \Theta$ für $j = 1, ..., m_i$ und $m_i \in \mathbb{N}$, sodass

$$\left| \rho_i f_i(t) - \sum_{j=1}^{m_i} c_{ij} \sigma(\lambda_{ij} t - \theta_{ij}) \right| < \frac{\epsilon}{2r}$$

für alle $t \in \{a_i \cdot x | x \in K\}$ und $i = 1, ..., r$.
Aufgrund der Dreiecksungleichung gilt somit

$$\begin{aligned}
&\left| \sum_{i=1}^{r} \rho_i f_i(a_i \cdot x) - \sum_{i=1}^{r} \sum_{j=1}^{m_i} c_{ij} \sigma(\lambda_{ij} a_i \cdot x - \theta_{ij}) \right| \\
&= \left| \sum_{i=1}^{r} [\rho_i f_i(a_i \cdot x) - \sum_{j=1}^{m_i} c_{ij} \sigma(\lambda_{ij} a_i \cdot x - \theta_{ij})] \right| \\
&\leq \sum_{i=1}^{r} \left| \rho_i f_i(a_i \cdot x) - \sum_{j=1}^{m_i} c_{ij} \sigma(\lambda_{ij} a_i \cdot x - \theta_{ij}) \right| \\
&\leq \sum_{i=1}^{r} \frac{\epsilon}{2r} = r \frac{\epsilon}{2r} = \frac{\epsilon}{2},
\end{aligned}$$

für alle $x \in K$.

Schließlich erhalten wir durch Einfügen einer nahrhaften 0

$$\begin{aligned}
&\left| g(x) - \sum_{i=1}^{r} \sum_{j=1}^{m_i} c_{ij}\sigma(\lambda_{ij} a_i \cdot x - \theta_{ij}) \right| \\
&= \left| g(x) - \sum_{i=1}^{r} \rho_i f_i(a_i \cdot x) + \sum_{i=1}^{r} \rho_i f_i(a_i \cdot x) - \sum_{i=1}^{r} \sum_{j=1}^{m_i} c_{ij}\sigma(\lambda_{ij} a_i \cdot x - \theta_{ij}) \right| \\
&\leq \left| g(x) - \sum_{i=1}^{r} \rho_i f_i(a_i \cdot x) \right| + \left| \sum_{i=1}^{r} \rho_i f_i(a_i \cdot x) - \sum_{i=1}^{r} \sum_{j=1}^{m_i} c_{ij}\sigma(\lambda_{ij} a_i \cdot x - \theta_{ij}) \right| \\
&< \frac{\epsilon}{2} + \frac{\epsilon}{2} = \epsilon,
\end{aligned}$$

für alle $x \in K$.
Da K, g und ϵ beliebig gewählt waren, ist die Behauptung somit gezeigt. □

Bevor wir weiter machen, benötigen wir noch ein Lemma. Der Beweis dieses Lemmas folgt im Wesentlichen [Don69] Seite 53.

Lemma 3.2.8. *Sei $I \subset \mathbb{R}$ ein offenes Intervall und $\sigma \in C^\infty(I)$. Wenn für alle $x \in I$ ein $N \in \mathbb{N}_0$ existiert, sodass $\sigma^{(N)}(x) = 0$ gilt, dann entspricht σ auf I einem Polynom.*

Beweis.
Wir betrachten folgende Mengen:

$$\begin{aligned}
X &:= \{x \in I | \exists (a,b) \subset I : x \in (a,b) \quad \& \ \sigma \text{ entspricht auf (a,b) einem Polynom } \} \\
S_n &:= \{x \in I | \sigma^{(n)}(x) = 0\}
\end{aligned}$$

X ist offen in $\mathbb{R}$. Sollte X leer sein, ist dies klar. Falls X nicht leer ist, so existiert für jedes $x \in X$ nach der Definition von X eine offene Umgebung (a,b), auf der σ einem Polynom entspricht. Diese offene Umgebung (a,b) von x liegt also selbst in X.
Wir wollen nun zeigen, dass das Komplement von X in I (also X^c) leer ist. Denn dann gilt $X = I$ und σ entspricht auf ganz I einem Polynom. Um dies zu erreichen, nehmen wir $X^c \neq \emptyset$ an und werden daraus einen Widerspruch konstruieren.
X^c ist als abgeschlossene Teilmenge des vollständigen metrischen Raums $\mathbb{R}$ selbst ein vollständiger metrischer Raum. S_n ist als Urbild der Ein-Punkt Menge $\{0\}$ unter der stetigen Abbildung $\sigma^{(n)}$ abgeschlossen. Somit ist $X^c \cap S_n$ eine abgeschlossene Teilmenge von X^c.
Nach der Voraussetzung gilt $\bigcup_{i \in \mathbb{N}_0} S_n = I$. Demnach bilden die Mengen $X^c \cap S_n$ mit $n \in \mathbb{N}_0$ eine Überdeckung von X^c.
Da wir außerdem $X^c \neq \emptyset$ angenommen haben, können wir Baire's Kategoriensatz 3.3.5 anwenden. Nach diesem Satz gibt es ein $N \in \mathbb{N}_0$, sodass $(X^c \cap S_N)^o$ nicht leer ist. Hierbei bezeichnet $(X^c \cap S_N)^o$ das Innere von $X^c \cap S_N$ bezüglich der Teilraumtopologie auf X^c. Das bedeutet, dass ein $x_0 \in (X^c \cap S_N)^o$ existiert. Für dieses x_0 gibt es aufgrund der Offenheit in X^c ein $\epsilon > 0$, sodass

$$B_\epsilon(x_0)|_{X^c} = \{x \in X^c | \, |x_0 - x| < \epsilon\} \subset (X^c \cap S_N)^o.$$

Für eine abgeschlossene Umgebung $[c,d]$ von x_0 in I mit einem Radius kleiner ϵ gilt somit

$$[c,d] \cap X^c \subset B_\epsilon(x_0)|_{X^c} \subset S_N.$$

Nach der Definition von S_N verschwindet demnach $\sigma^{(N)}$ auf ganz $[c,d] \cap X^c$.

Nun wollen wir zeigen, dass auch alle höheren Ableitungen von σ auf $[c,d] \cap X^c$ verschwinden. Dazu müssen wir zunächst zeigen, dass X^c keine isolierten Punkte enthält. Angenommen $x_1 \in X^c$ ist ein isolierter Punkt, dann existieren Intervalle $(a,x_1),(x_1,b) \subset X$, auf denen σ einem Polynom entspricht. Die Koeffizienten der Taylorreihenentwicklung von σ um x_1 entsprechen dann den Koeffizienten des Polynoms, mit dem σ auf (a,x_1) übereinstimmt, und den Koeffizienten des Polynoms, mit dem σ auf (x_1,b) übereinstimmt. Somit entspricht σ auf ganz (a,b) demselben Polynom. Dies ist aber ein Widerspruch zu $x_1 \in X^c$.
X^c ist also eine perfekte Menge und damit ist auch $[c,d] \cap X^c$ eine perfekte Menge. Deshalb existiert für jedes $y \in [c,d] \cap X^c$ eine Folge $\{y_n | y_n \in [c,d] \cap X^c, n \in \mathbb{N}\}$, die gegen y konvergiert. Somit gilt

$$\sigma^{(N+1)}(y) = \lim_{n\to\infty} \frac{\sigma^{(N)}(y_n) - \sigma^{(N)}(y)}{y_n - y} = \lim_{n\to\infty} \frac{0-0}{y_n - y} = 0.$$

Induktiv erhalten wir für alle $m \geq N$ und alle $x \in [c,d] \cap X^c$

$$\sigma^{(m)}(x) = 0. \tag{3.1}$$

Nun können wir den versprochenen Widerspruch konstruieren.
Dazu betrachten wir zuerst den Fall $[c,d] \cap X = \emptyset$, also $[c,d] \cap X^c = [c,d]$. In diesem Fall verschwindet $\sigma^{(N)}$ auf dem gesamten offenen Intervall (c,d) und somit entspricht σ dort einem Polynom. Dies bedeutet aber, dass (c,d) in X liegt. Dies steht im Widerspruch dazu, dass $(X^c \cap S_N)^o$ nach Baire's Kategoriensatz nicht leer ist.
Für den Fall $[c,d] \cap X \neq \emptyset$ gibt es eine Überdeckung von $[c,d] \cap X$ aus Mengen der Form $[c,d] \cap (a,b)$, wobei (a,b) in X liegt und mindestens einer der beiden Punkte a,b in $[c,d] \cap X^c$ liegt.
Für jedes $(a,b) \subset X$ aus dieser Überdeckung existiert ein Polynom p, sodass $p|_{(a,b)} = \sigma|_{(a,b)}$ gilt. Die Koeffizienten dieses Polynoms sind dann durch die Taylorreihenentwicklung von σ um a bzw. b gegeben.
Sei nun o.B.d.A. $a \in [c,d] \cap X^c$, dann wissen wir aus 3.1, dass $\sigma^{(k)}(a)$ für alle $k \geq N$ verschwindet. Somit ist der Grad des Polynoms p kleiner als N. Dies bedeutet, dass $p^{(N)}$ und somit auch $\sigma^{(N)}$ auf (a,b) verschwindet.
Da wir dies für ein beliebiges (a,b) aus unserer Überdeckung von $[c,d] \cap X$ gezeigt haben und da wir schon $[c,d] \cap X^c \subset S_N$ wissen, verschwindet $\sigma^{(N)}$ also auf ganz $[c,d]$. Dies ist aber, wie im vorherigen Fall, ein Widerspruch zu $(X^c \cap S_N)^o \neq \emptyset$.

Wir erhalten also in beiden Fällen einen Widerspruch und die einzige Annahme, die wir getroffen haben, ist, dass X^c nicht leer ist. Demnach muss X^c leer sein und die Aussage ist gezeigt. □

Nun können wir den Approximationssatz für glatte Aktivierungsfunktionen beweisen. Dabei können wir sogar die Räume, aus denen wir unsere Parameter wählen, sehr stark einschränken. Dies ist kein Grund sich in der Praxis bei der Suche nach passenden Parametern unnötig einzuschränken. Trotzdem ist es interessant, dass die theoretische Approximation schon mit sehr kleinen Parameterräumen möglich ist.

Proposition 3.2.9. *Sei $\Theta \subset \mathbb{R}$ ein offenes Intervall und $\Lambda \subset \mathbb{R}$ eine offene Umgebung der 0. Sei weiter $\sigma \in C(\mathbb{R})$ nicht polynomiell auf Θ und mit $\sigma \in C^\infty(\Theta)$, dann ist $N(\sigma; \Lambda, \Theta)$ dicht in $C(\mathbb{R})$.*

Beweis. Da σ auf Θ keinem Polynom entspricht, existiert nach Lemma 3.2.8 ein Punkt $\theta_0 \in \Theta$, sodass für alle $k \in \mathbb{N}_0$ gilt:

$$\sigma^{(k)}(\theta_0) \neq 0.$$

Nach Voraussetzung enthält Λ eine Nullfolge $\{h_n\}_{n\in\mathbb{N}}$. Da Λ offen ist, finden wir für jedes $\lambda \in \Lambda$ ein $m \in \mathbb{N}$, sodass für alle $n \geq m$ gilt:

$$\lambda + h_n \in \Lambda.$$

In diesem Fall erhalten wir

$$f(h, \lambda) := \frac{1}{h_n}\sigma((\lambda + h_n)t + \theta_0) - \frac{1}{h_n}\sigma(\lambda t + \theta_0) \in N(\sigma; \Lambda, \Theta),$$

für $n \geq m$.
Somit liegt der Grenzwert

$$\lim_{n\to\infty} f(h_n, \lambda_0) = \frac{d}{d\lambda}\sigma(\lambda t + \theta_0)|_{\lambda=\lambda_0}$$

für alle $\lambda_0 \in \Lambda$ im Abschluss $\overline{N(\sigma; \Lambda, \Theta)}$. Da der Abschluss eines Untervektorraums wieder ein Untervektorraum ist, gilt für dieselben λ und h_n, wie im vorherigen Fall,

$$f'(h, \lambda) := \frac{1}{h_n}\sigma'((\lambda + h_n)t + \theta_0) - \frac{1}{h_n}\sigma'(\lambda t + \theta_0) \in N(\sigma; \Lambda, \Theta),$$

für $n \geq m$.
Somit liegt auch der Grenzwert

$$\lim_{n\to\infty} f'(h_n, \lambda_0) = \frac{d^2}{d\lambda^2}\sigma(\lambda t + \theta_0)|_{\lambda=\lambda_0}$$

für alle $\lambda_0 \in \Lambda$ im Abschluss $\overline{N(\sigma;\Lambda,\Theta)}$. Induktiv erhalten wir insbesondere für $\lambda_0 = 0$

$$\frac{d^k}{d\lambda^k}\sigma(\lambda t + \theta_0)|_{\lambda=0} = t^k\sigma^{(k)}(\theta_0) \in N(\sigma;\Lambda,\Theta).$$

Da wir θ_0 so gewählt haben, dass $\sigma^{(k)}(\theta_0)$ für kein $k \in \mathbb{N}$ verschwindet, liegen alle Monome und somit alle Polynome in $N(\sigma;\Lambda,\Theta)$. Nach dem dem Satz von Weierstraß 3.3.4 liegt $N(\sigma;\Lambda,\Theta)$ dann dicht in $C(\mathbb{R})$, bezüglich der Topologie der gleichmäßigen Konvergenz auf Kompakta. □

Da $\mathbb{R}$ die Bedingungen für Λ und Θ erfüllt, erhalten wir aus diesem Satz insbesondere den Approximationssatz für glatte Aktivierungsfunktionen, die nicht polynomiell sind.

Korollar 3.2.10. *Sei $\sigma \in C^\infty(\mathbb{R})$ nicht polynomiell, dann liegt $N(\sigma;\mathbb{R},\mathbb{R})$ dicht in $C(\mathbb{R})$.*

Nun werden wir auf dem Beweis von Proposition 3.2.9 aufbauen, um den Approximationssatz für stetige Aktivierungsfunktionen zu zeigen, die nicht polynomiell sind.

Satz 3.2.11. *Sei $\sigma \in C(\mathbb{R})$ und sei σ kein Polynom. Dann ist $N(\sigma;\mathbb{R},\mathbb{R})$ dicht in $C(\mathbb{R})$.*

Beweis. Sei $\phi \in C_0^\infty(\mathbb{R})$. Da der Träger von ϕ beschränkt ist, liegt er in einem Intervall $[a,b] \subset \mathbb{R}$. Indem wir das Intervall weiter vergrößern, können wir annehmen, dass das Intervall von der Form $[-a,a]$, mit $a \in \mathbb{R}^+$, ist.
Der Träger von $\sigma(t-y)\phi(y)$ als Funktion in y ist dann für alle $t \in \mathbb{R}$ ebenfalls eine Teilmenge von $[-a,a]$. Da $\sigma(t-y)\phi(y)$ als Funktion in y außerdem für alle $t \in \mathbb{R}$ stetig ist, nimmt sie auf $[-a,a]$ ihr Maximum an und ist dort beschränkt. Somit konvergiert

$$\sigma_\phi(t) := \int_{-\infty}^{\infty} \sigma(t-y)\phi(y)dy = \int_{-a}^{a} \sigma(t-y)\phi(y)dy$$

für alle $t \in \mathbb{R}$.
σ_ϕ ist die Faltung $\sigma * \phi$. Somit gilt nach der Ableitungsregel für Faltungen $D\sigma_\phi = \sigma * D\phi$. Induktiv erhalten wir daraus, zusammen mit $\phi \in C_0^\infty$, dass $\sigma_\phi \in C^\infty(\mathbb{R})$ gilt.
In dem Beweis von 3.2.9 haben wir bereits gesehen, dass aus $\sigma_\phi \in C^\infty(\mathbb{R})$ folgt, dass $t^k\sigma_\phi^{(k)}(\theta)$ für alle $k \in \mathbb{N}$ und $\theta \in \mathbb{R}$ in $\overline{N(\sigma_\phi;\mathbb{R},\mathbb{R})}$ liegt.
Setzen wir nun

$$y_i^m = -a + 2a\frac{i}{m} \quad \text{mit} \quad i = 1,...,m \quad \text{und} \quad m \in \mathbb{N},$$

dann erhalten wir durch $[y_{i-1}^m, y_i^m]$ für alle $m \in \mathbb{N}$ eine Zerlegung von $[-a,a]$, deren Intervalle die Länge $\triangle y_i^m = y_i^m - y_{i-1}^m = 2a\frac{1}{m}$ haben.
Für $m \in \mathbb{N}$ und $\theta, \lambda \in \mathbb{R}$ gilt außerdem

$$\sum_{i=1}^{m} \sigma(\lambda t + \theta - y_i^m)\phi(y_i^m)\triangle y_i^m \quad \in N(\sigma;\mathbb{R},\mathbb{R}).$$

Da dies eine Riemansumme ist, liegt

$$\sigma_\phi(\lambda t - \theta) = \int_{-a}^{a} \sigma(\lambda t - \theta - y)\phi(y)dy = \lim_{m\to\infty} \sum_{i=1}^{m} \sigma(\lambda t - \theta - y_i^m)\phi(y_i^m)\triangle y_i^m$$

für alle $\lambda, \theta \in \mathbb{R}$ im Abschluss $\overline{N(\sigma; \mathbb{R}, \mathbb{R})}$. Dadurch erhalten wir $\overline{N(\sigma_\phi; \mathbb{R}, \mathbb{R})} \subset \overline{N(\sigma; \mathbb{R}, \mathbb{R})}$.
Nehmen wir nun an, dass $N(\sigma; \mathbb{R}, \mathbb{R})$ nicht dicht in $C(\mathbb{R})$ liegt, dann existiert nach dem Satz von Weierstraß 3.3.4 ein $k \in \mathbb{N}$, sodass t^k nicht in $\overline{N(\sigma; \mathbb{R}, \mathbb{R})}$ und somit insbesondere nicht in $\overline{N(\sigma_\phi; \mathbb{R}, \mathbb{R})}$ liegt.
Da wir bereits $t^k \sigma_\phi^{(k)}(\theta) \in \overline{N(\sigma_\phi; \mathbb{R}, \mathbb{R})}$ für alle $\theta \in \mathbb{R}$ gezeigt haben, muss $\sigma_\phi^{(k)}$ also auf ganz $\mathbb{R}$ verschwinden. Somit ist σ_ϕ für alle $\phi \in C_0^\infty(\mathbb{R})$ ein Polynom vom Grad maximal $k-1$.
Nach Lemma 3.3.7 existiert eine Folge $(\phi_n)_{n\in\mathbb{N}}$ in $C_0^\infty(\mathbb{R})$, sodass $\sigma * \phi_n$ auf jedem Kompaktum $K \subset \mathbb{R}$ gleichmäßig gegen σ konvergiert.
Der Raum der Polynome vom Grad maximal $k-1$ ist als endlicher Untervektorraum topologisch abgeschlossen. Somit entspricht σ auf jedem kompakten $K \subset \mathbb{R}$ einem Polynom vom Grad maximal $k-1$.
Für $n \in \mathbb{N}$ bezeichne p_n das Polynom, mit dem σ auf $\overline{B_n(0)}$ übereinstimmt. Aus $\overline{B_1(0)} \subset \overline{B_n(0)}$ folgt für alle $x \in \overline{B_1(0)}$

$$p_1(x) - p_n(x) = \sigma(x) - \sigma(x) = 0.$$

Da $\overline{B_1(0)}$ unendlich viele Elemente enthält, ist $p_1 - p_n$ also ein Polynom mit unendlich vielen Nullstellen. Dies ist nur möglich, wenn $p_1 - p_n$ das konstante 0-Polynom ist. Somit erhalten wir für alle $n \in \mathbb{N}$

$$p := p_1 = p_n.$$

Sei nun $K \subset \mathbb{R}$ eine andere kompakte Menge mit nicht leerem Maß, dann enthält K unendlich viele Elemente und es existiert ein $n \in \mathbb{N}$, sodass K in $\overline{B_n(0)}$ liegt. Analog zu der Begründung für $p = p_n$ entspricht demnach auch das Polynom, mit dem σ auf K übereinstimmt, dem Polynom p.
Sei $E \in \mathbb{R}$ eine messbare beschränkte Menge mit nicht leerem Maß, dann hat auch $\overline{E}$ ein nicht leeres Maß. Da $\overline{E}$ außerdem kompakt ist, gilt $\sigma(x) - p(x) = 0$ für alle $x \in \overline{E}$ und insbesondere für alle $x \in E$. Wir erhalten also

$$\int_E \sigma(x) - p(x)dx = 0$$

für alle messbaren beschränkten Mengen $E \subset \mathbb{R}$.
Nach dem Fundamentallemma der Variationsrechnung 3.3.8 gilt somit $\sigma - p = 0$ fast überall. Da σ und p stetig sind, können wir daraus $\sigma - p = 0$ auf ganz $\mathbb{R}$ folgern.
Wir haben somit gezeigt, dass σ ein Polynom ist, falls $N(\sigma; \mathbb{R}, \mathbb{R})$ nicht dicht in $C(\mathbb{R})$ liegt. Dies beweist unsere Behauptung durch Kontraposition.

□

Wir haben nun alle Aussagen gezeigt, die wir benötigen, um das Approximationstheorem für MLP's mit einem Hidden Layer zu beweisen. Jetzt müssen wir nur noch alle Bausteine zusammensetzen.

Satz 3.2.12 (Approximationstheorem)**.** *Sei $\sigma \in C(\mathbb{R})$ nicht polynomiell, dann liegt die Menge $M(\sigma)$ bezüglich der Topologie der kompakten Konvergenz dicht in $C(\mathbb{R}^n)$.*

Beweis. Sei $\sigma \in C(\mathbb{R})$ kein Polynom, dann haben wir in Satz 3.2.11 gezeigt, dass $N(\sigma; \mathbb{R}, \mathbb{R})$ bezüglich der Topologie der kompakten Konvergenz dicht in $C(\mathbb{R})$ liegt.
Da es kein homogenes Polynom gibt, welches auf der ganzen Sphäre S^{n-1} verschwindet, wissen wir aus Satz 3.2.5, dass die Menge der ridge Funktionen $R(S^{n-1})$ bezüglich der Topologie der kompakten Konvergenz dicht in $C(\mathbb{R}^n)$ liegt.
Nach Proposition 3.2.7 liegt somit auch $M(\sigma; \mathbb{R} \times S^{n-1}, \mathbb{R})$ bezüglich der Topologie der kompakten Konvergenz dicht in $C(\mathbb{R}^n)$.
Weiter gilt $\mathbb{R} \times S^{n-1} = \{\lambda a | \lambda \in \mathbb{R}, a \in S^{n-1}\} = \mathbb{R}^n$.
Daraus erhalten wir

$$M(\sigma; \mathbb{R} \times S^{n-1}, \mathbb{R}) = M(\sigma; \mathbb{R}^n, \mathbb{R}) = span\{\sigma(w \cdot x + \theta) | w \in \mathbb{R}^n, \theta \in \mathbb{R}\} = M(\sigma).$$

Damit ist der Satz gezeigt.

□

Tatsächlich ist dies sogar die größte Klasse an stetigen Aktivierungsfunktionen, die das Approximationstheorem erfüllen.

Satz 3.2.13. *Angenommen $\sigma \in C(\mathbb{R})$ ist ein Polynom, dann liegt $M(\sigma)$ bezüglich der Topologie der kompakten Konvergenz nicht dicht in $C(\mathbb{R}^n)$.*

Beweis. Wenn σ ein Polynom vom Grad $k \in \mathbb{N}_0$ ist, dann ist auch $\sigma(w \cdot x + \theta)$ für $w \in \mathbb{R}^n$ und $\theta \in \mathbb{R}$ ein Polynom vom Grad maximal k in n Variablen. Die Polynome vom Grad maximal k liegen jedoch nicht dicht in $C(\mathbb{R}^n)$, bezüglich der Topologie der kompakten Konvergenz.
Um dies zu sehen, sei K eine kompakte Teilmenge von $\mathbb{R}^n$. In diesem Fall ist $C(K)$ versehen mit der Topologie der gleichmäßigen Konvergenz ein unendlich dimensionaler Hausdorff-Raum. Die Menge der Polynome vom Grad maximal k ist ein endlich dimensionaler Untervektorraum von $C(K)$ und somit abgeschlossen bezüglich der Topologie der gleichmäßigen Konvergenz. Da die Polynome vom Grad maximal k also eine abgeschlossene echte Teilmenge von $C(K)$ sind, liegen sie nicht dicht in $C(K)$ bezüglich der Topologie der gleichmäßigen Konvergenz. □

3.3 Addendum

Wir haben in Kapitel 3 einige aus der Funktionalanalysis bekannte Sätze verwendet. Oft werden diese Sätze von verschieden Autoren unterschiedlich formuliert oder es existieren mehrere Versionen eines Satzes, die den gleichen Namen tragen. Deshalb möchte ich in diesem Abschnitt alle Aussagen in der Formulierung, die ich verwende, aufzählen.

Dabei entspricht die Reihenfolge dem Auftreten der Sätze in Kapitel 3.
Sofern ich eine gute Referenz für die Beweise der Sätze gefunden habe, werde ich lediglich auf diese verweisen. Falls ich aber keine geeignete Quelle für die von mir benötigten Versionen der Sätze gefunden habe, werde ich diese aus bekannteren Versionen, für die ich eine Referenz gefunden habe, herleiten.

Satz 3.3.1 (Satz von Hahn-Banach)**.** *Angenommen M ist ein Untervektorraum eines lokal konvexen topologischen Vektorraumes X. Falls $x \in X$ nicht im Abschluss von M liegt, dann gibt es ein stetiges lineares Funktional L auf X, welches auf M verschwindet und für das $L(x) = 1$ gilt.*

Beweis. Siehe [Rud91] Theorem 3.5. □

Dies können wir insbesondere auf jeden normierten Vektorraum M anwenden, da die offenen Bälle der von der Norm induzierten Topologie konvex sind.
Außerdem können wir diesen Satz auf $C(\mathbb{R}^n)$, versehen mit der Topologie der kompakten Konvergenz, anwenden.
Diese Topologie wird von offenen Bällen der Form

$$B_{\epsilon,K}(f) = \{g \in C(\mathbb{R}^n) | \sup_{x \in K} \|f(x) - g(x)\| < \epsilon\},$$

mit $\epsilon > 0, f \in C(\mathbb{R}^n)$ und $K \subset \mathbb{R}^n$ kompakt, erzeugt.
Sind $g, h \in B_{\epsilon,K}(f)$, dann gilt für $0 \leq t \leq 1$ und für alle $x \in K$:

$$\begin{aligned}
&\|tg(x) + (1-t)h(x) - f(x)\| \\
&= \|t(g(x) - f(x)) + (1-t)(h(x) - f(x)\| \\
&\leq t\,\|g(x) - f(x)\| + (1-t)\,\|h(x) - f(x)\| \\
&\leq t\epsilon + (1-t)\epsilon = \epsilon.
\end{aligned}$$

Somit liegt auch $tg + (1-t)h$ in $B_{\epsilon,K}(f)$. Demnach sind die offenen Bälle von $C(\mathbb{R}^n)$ konvex und $C(\mathbb{R}^n)$ ist lokal konvex.

Satz 3.3.2 (Riesz'scher Darstellungssatz)**.** *Sei X ein kompakter Hausdorff-Raum und seien $C(X)$ die stetigen Funktionen auf X. Für jedes beschränkte lineare Funktional L auf $C(X)$ existiert genau ein positives Borel-Maß μ, sodass*

$$\forall f \in C_0(X): \quad L(f) = \int_X f d\mu.$$

In diesem Fall sagt man, dass μ das Funktional L repräsentiert.

Beweis. [Rud87] Theorem 2.14 beweist diese Aussage für lokal kompakte Räume X und beschränkte lineare Funktionale auf den stetigen Funktionen mit kompaktem Träger $C_0(X)$.
Wenn X nun sogar kompakt ist, dann ist der Träger jeder stetigen Funktion $f \in C(X)$, als abgeschlossene Teilmenge eines kompakten Raums, selbst kompakt. Somit gilt $C_0(X) = C(X)$ und der Satz ist bewiesen. □

Satz 3.3.3 (Stone-Weierstraß). *Sei S ein kompakter Hausdorff-Raum. Sei A eine abgeschlossene Unteralgebra der Menge der stetigen komplexwertigen Funktionen $C(S, \mathbb{C})$, versehen mit der Topologie der gleichmäßigen Konvergenz. Gilt außerdem:*

- *A ist selbstadjungiert, d.h. $\forall f \in A : \overline{f} \in A$*
- *A separiert Punkte*
- *$\forall p \in S \quad \exists f \in A : f(p) \neq 0$,*

dann ist $A = C(S, \mathbb{C})$.

Beweis. [Rud91] Theorem 5.7. □

Der wichtigste Spezialfall dieses Satzes ist der Approximationssatz von Weierstraß.

Korollar 3.3.4. *Sei $n \in \mathbb{N}$. Die Menge der Polynome $P(\mathbb{R}^n) = \{f : \mathbb{R}^n \to \mathbb{R} | f$ ist ein Polynom $\}$ liegt dicht in den stetigen reellwertigen Funktionen $C(\mathbb{R}^n)$ bezüglich der Topologie der gleichmäßigen Konvergenz auf Kompakta.*

Beweis. Sei $K \subset \mathbb{R}^n$ kompakt, dann ist $P(K, \mathbb{C}) = \{f : \mathbb{R}^n \to \mathbb{C} | f$ ist ein Polynom $\}$ eine Unteralgebra der Banachalgebra der stetigen Funktionen $C(K, \mathbb{C})$. Der Abschluss $\overline{P(K)}$ von $P(K)$ in $C(K, \mathbb{C})$, versehen mit der Topologie der gleichmäßigen Konvergenz, ist demnach eine abgeschlossene Unteralgebra von $C(K, \mathbb{C})$.
Sei $f \in P(K, \mathbb{C})$, dann entspricht f einem Polynom mit komplexen Koeffizienten. Zerlegen wir f in seinen Imaginär- und Realteil, so erhalten wir $f = p + iq$ für zwei Polynome p, q mit reellen Koeffizienten. Die komplex konjugierte Funktion $\overline{f} = p - iq$ ist dann wieder ein Polynom mit komplexen Koeffizienten und liegt somit in $P(K, \mathbb{C})$.
Sei nun $f \in \overline{P(K, \mathbb{C})}$, dann gibt es komplexe Polynome $f_n \in P(K, \mathbb{C})$, sodass $f = \lim_{n \to \infty} f_n$ gilt. Demnach liegt auch $\overline{f} = \lim_{n \to \infty} \overline{f_n}$ in $P(K, \mathbb{C})$.
Seien $v \neq w \in K$, dann gibt es ein $0 < k \leq n$, sodass $v_k \neq w_k$ ist. Wählen wir nun das Polynom $f(x_1, ..., x_n) = x_k$, dann gilt $f(v) \neq f(w)$.
Außerdem verschwindet das konstante Polynom $f(x_1, ..., x_n) = 1$ für keinen Punkt aus K.
Somit gilt nach dem Satz von Stone-Weierstraß 3.3.3

$$\overline{P(K, \mathbb{C})} = C(K, \mathbb{C}).$$

Sei nun eine reellwertige Funktion $f \in C(K)$ gegeben, dann liegt f auch in $C(K, \mathbb{C})$. Es existieren also Polynome $f_n \in P(K, \mathbb{C})$, die gegen f konvergieren.
Insbesondere konvergieren die Realteile der f_n gegen den Realteil von f. Da f schon reell ist und die Realteile der f_n reellen Polynomen entsprechen, haben wir somit

$$\overline{P(K)} = C(K)$$

gezeigt.

□

Satz 3.3.5 (Bair'scher Kategoriensatz). *Sei $X \neq \emptyset$ ein vollständiger metrischer Raum. Wenn es ein Überdeckung*

$$X = \bigcup_{k \in \mathbb{N}} A_k$$

aus abgeschlossenen Mengen $A_k \subset X$ gibt, dann gibt es ein $k_0 \in \mathbb{N}$, sodass das Innere $A_{k_0}^o$ nicht leer ist.

Beweis. [Alt06] Satz 5.1. □

Für ein beliebiges Polynom $p \in \mathbb{R}[X]$ definieren wir

$$f_p(x) := \begin{cases} p(\frac{1}{x})e^{-\frac{1}{x}} & \text{falls} \quad x > 0 \\ 0 & \text{falls} \quad x \leq 0 \end{cases}.$$

Da $e^{-\frac{1}{x}}$ für $x \to 0$ exponentiell gegen 0 konvergiert, während $p(\frac{1}{x})$ lediglich polynomiell wächst, ist f_p stetig.
Für $x > 0$ gilt außerdem

$$f_p'(x) = p(\tfrac{1}{x})e^{-\frac{1}{x}}\tfrac{1}{x^2} - p'(\tfrac{1}{x})\tfrac{1}{x^2}e^{-\frac{1}{x}} = \tfrac{1}{x^2}(p(\tfrac{1}{x}) - p'(\tfrac{1}{x}))e^{-\frac{1}{x}},$$

wobei $\frac{1}{x^2}(p(\frac{1}{x}) - p'(\frac{1}{x}))$ wieder ein Polynom in $\frac{1}{x}$ ist. Induktiv erhalten wir somit $f_p \in C^\infty(\mathbb{R})$.
Wählen wir nun das Polynom $\mathbf{1}(x) = 1$ und $g(x) = 1 - x^2$, dann ist

$$f(x) := f_{\mathbf{1}} \circ g(x) = \begin{cases} e^{-\frac{1}{1-x^2}} & \text{falls} \quad |x| < 1 \\ 0 & \text{falls} \quad |x| \geq 1 \end{cases}$$

als Komposition glatter Funktionen selbst in $C^\infty(\mathbb{R})$.
Setzen wir weiter $c = \int_{-1}^{1} f(x)dx$, dann gilt

$$\frac{1}{c}f \geq 0 \quad \text{und} \quad \int_{-\infty}^{\infty} \tfrac{1}{c}f(x)dx = \int_{-1}^{1} \tfrac{1}{c}f(x)dx = \frac{1}{c}\int_{-1}^{1} f(x)dx = 1.$$

Definition 3.3.6. Seien c, f wie oben und sei $\epsilon > 0$, dann nennen wir $\{\frac{1}{\epsilon^n c}f(\frac{x}{\epsilon})\}_{n \in \mathbb{N}}$ eine **Standard Dirac-Folge**.

Satz 3.3.7. *Sei ϕ_n eine Standard Dirac-Folge und ist $\sigma \in C(\mathbb{R})$, so konvergiert $\sigma * \phi_n$ auf jedem Kompaktum $K \subset \mathbb{R}$ gleichmäßig gegen σ.*

Beweis. [Eva98] Appendix C Theorem 6. □

Satz 3.3.8 (Fundamentallemma der Variationsrechnung)**.** *Sei $\Omega \subset \mathbb{R}^n$ offen und Y ein Banachraum. Für eine integrierbare Funktion $g : \Omega \to Y$ gilt $g = 0$ fast überall in Ω genau dann, wenn*

$$\int_E g d\mathfrak{L}^n = 0$$

für alle messbaren beschränkten Mengen E mit $\overline{E} \subset \Omega$ gilt.

Beweis. [Alt06] 2.21 . □

4 Multimodale Erklärungen beim tiefen bestärkendem Lernen

Wie in der Einleitung beschrieben werden Systeme, die von tiefen bestärkenden Lernverfahren trainiert wurden, in Zukunft immer mehr in das Alltagsleben der Menschen integriert werden. Zum Beispiel im Zusammenhang mit autonom fahrenden Autos (vgl. [FJT18]) oder durch eine direkte Zusammenarbeit mit solchen lernenden Systemen, sowie Google DeepMind es anhand des Computerspiels *Quake* in [JCD+18] demonstrierte. Da man beim bestärkenden Lernen nicht direkt angibt, welche Aktionen in bestimmten Situationen ausgeführt werden sollen, ist es nicht immer einfach die Aktionen des Agenten nachzuvollziehen. Hinzu kommt, dass auch neuronale Netze oft eine Art „Black Box“ darstellen. Experten können zwar die gespeicherten Gewichte eines Netzes durch verschiedene Methoden visualisieren und versuchen dadurch das Verhalten der Netze zu analysieren, doch dafür ist bereits einiges an Hintergrundwissen notwendig.
Damit solche Systeme von der breiten Bevölkerung akzeptiert werden, muss es jedoch auch für Menschen ohne entsprechenden Hintergrund möglich sein, die Aktionen des Agenten nachzuvollziehen. Zu diesem Zweck stelle ich in diesem Kapitel ein System vor, das die Handlungen eines Agenten in natürlicher Art und Weise erklärt.
Dazu werde ich zum Einen die visuelle Aufmerksamkeit des Agenten hervorheben, indem ich die Teile des Inputs ermittle, die für die Entscheidung relevant sind. Zum Anderen werde ich kurze sprachliche Erklärungen generieren, die rechtfertigen, warum sich der Agent in einer bestimmten Situation für eine bestimmte Aktion entscheidet.
Dieses System gestalte ich so, dass es auch auf bereits trainierte Netze anwendbar ist.

4.1 Problemstellung

Computerspiele sind eine beliebte Testumgebung für bestärkende Lernsysteme, da bei einem Spiel der objektive Vergleich der Leistung des künstlichen Agenten mit der Leistung eines Menschen möglich ist.
Als „proof of concept“ möchte ich deshalb in dieser Arbeit das Verhalten eines Agenten erklären, der das Atari 2600 Spiel *Breakout* löst. Bei diesem Spiel kontrolliert man einen Schläger, der sich an der unteren Hälfte des Spielfeldes horizontal nach rechts und links bewegen kann.
Als menschlicher Spieler benutzt man einen Joystick mit einem einzigen roten Knopf, um Atari 2600 Spiele zu steuern (vgl. Abbildung 4.3). Drückt man diesen Knopf, so erscheint ein Ball, der sich auf die untere Hälfte des Spielfeldes zu bewegt.

Trifft der Spieler den Ball mit dem Schläger, so prallt der Ball davon ab und bewegt sich nach oben. In der oberen Hälfte des Spielfeldes befinden sich mehrere Reihen von Blöcken. Berührt der Ball einen dieser Blöcke, dann prallt er davon ab und der Block wird zerstört. Für jeden zerstörten Block bekommt der Spieler einen Punkt. An den Seiten und am oberen Rand des Spielfeldes befinden sich Wände, von denen der Ball ebenfalls abprallt.
Schafft der Spieler es nicht den Ball am unteren Ende des Spielfeldes mit seinem Schläger zu treffen, dann verlässt der Ball den unteren Rand des Bildschirms und man muss durch Drücken des Knopfes einen neuen Ball anfordern. Nachdem man fünf Bälle verloren hat, ist das Spiel zu Ende. Das Ziel des Spiels ist es, möglichst viele Blöcke zu zerstören und somit Punkte zu erhalten, bevor man alle fünf Bälle verloren hat.

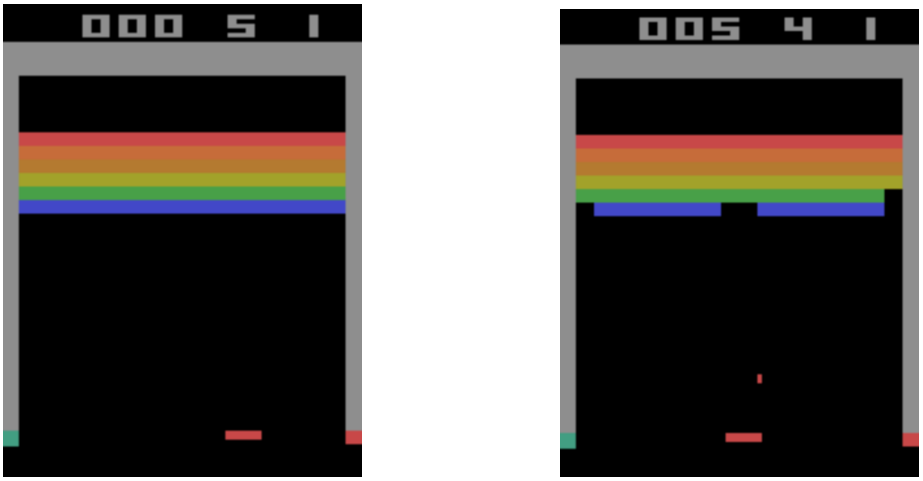


Abbildung 4.1: Links sieht man den Startzustand eines Breakoutspiels und rechts das Spiel nach einigen Sekunden.

Um dieses Spiel zu lösen, möchte ich das *Deep Q Network (DQN)* aus [MKS+15] benutzen und verwende dazu die Implementierung von Taehoon Kim [Tae17].
Das DQN erhält dabei den momentanen Bildschirmzustand, ein sogenanntes *Frame*, und den momentanen Punktestand als Input. Daraus generiert es sogenannte *Q-Werte*, das heißt der Output des Netzes ist ein Vektor, dessen Einträge den möglichen Aktionen des Agenten entsprechen und beschreiben, wie gut die jeweilige Aktion im momentanen Zustand ist.
Da die Farben und eine hohe Auflösung für das Spiel nicht relevant sind, wird jedes Frame in Graustufen umgewandelt und auf ein 84x84 Pixelbild skaliert. Um zeitliche Zusammenhänge zu verstehen, werden außerdem immer die letzten vier Frames zusammengefasst. Der Input lebt also in $\mathbb{R}^{84} \times \mathbb{R}^{84} \times \mathbb{R}^4$.

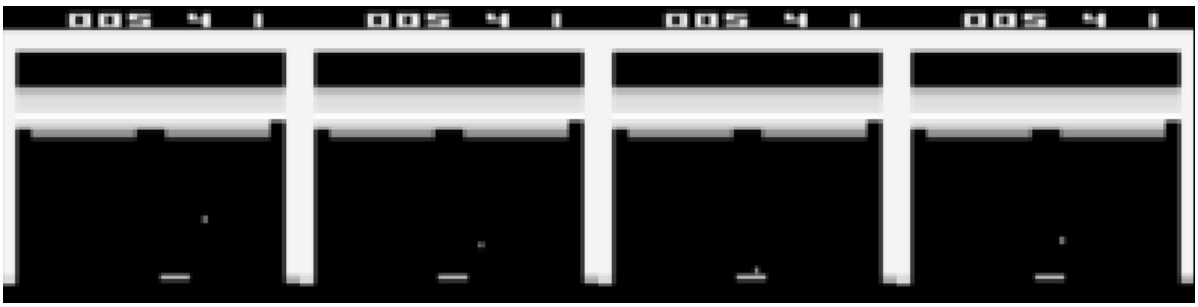


Abbildung 4.2: Ein Breakout Bildschirm, so wie ihn der Agent „sieht“. Die zusammengefassten Zustände beginnen in dieser Darstellung links. Ganz rechts befindet sich das aktuelle Frame.

Dieser Input wird dann von einem konvolutionalen Netz mit drei 2D-Konvolution Schichten verarbeitet, dabei besteht die erste Schicht aus 32 Filtern und die Fenster sind 8×8 Untermatrizen mit Schritt $(4, 4)$. Die zweite Schicht verwendet 64 Filter, 4×4 Fenster und einen Schritt von $(2, 2)$ und die letzte Schicht benutzt erneut 64 Filter, 3×3 Fenster und einen Schritt von $(1, 1)$.
Das Ergebnis des konvolutionalen Netzes wird in Vektorform geschrieben und an ein MLP mit einer versteckten Schicht weitergereicht. Hierbei besteht die versteckte Schicht $fc1$ aus 512 Neuronen. Alle versteckten Schichten, also auch alle Schichten des konvolutionalen Netzes, verwenden die ReLU Aktivierungsfunktion und das Output Layer $fc2$ benutzt keine Aktivierungsfunktion. Da die Output Schicht $fc2$ des MLP's auch die Output Schicht des gesamten DQN ist, bestimmt sie die Q-Werte der Aktionen des Agenten. Somit entspricht die Anzahl der Neuronen von $fc2$ der Anzahl an Aktionen, die der Agent ausführen kann.
Da wir ein Atari 2600 Spiel lösen wollen, ist die Anzahl der Aktionen durch die Menge an möglichen Befehlen beschränkt, die man mit einem Atari 2600 Joystick geben kann. Aus diesen insgesamt 18 möglichen Befehlen wählen wir nur die Aktionen aus, die in dem zu lösenden Spiel überhaupt Sinn ergeben. Im Fall von Breakout passiert zum Beispiel nie etwas, wenn man den Joystick nach oben bewegt. Insgesamt gibt es bei Breakout nur vier sinnvolle Aktionen: sich nach links oder rechts bewegen, den Knopf drücken oder nichts tun.

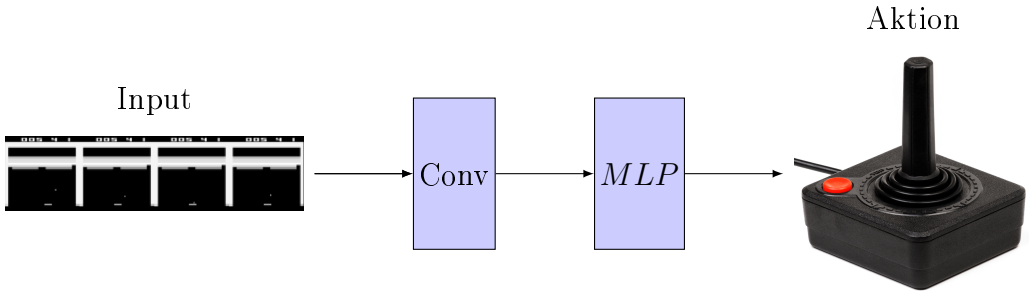


Abbildung 4.3: Das Layout des DQN. Das Bild des Joystick entstammt [Wik18].

Da wir am Ende des DQN ein MLP mit einer versteckten Schicht und der nicht polynomiellen ReLU Aktivierungsfunktion verwenden, ist es nach Satz 3.2.12 zumindest theoretisch möglich, mit diesem Netz eine optimale Strategie zu finden, solange wir das versteckte Layer groß genug wählen.

Das besondere Ziel dieser Arbeit ist, dass der Agent seine Entscheidung zusätzlich begründet. Dabei soll er mit einem Satz in natürlicher Sprache kurz erklären, warum er die Aktion wählt und dabei auf dem Bildschirm „zeigen“, welche Objekte für seine Entscheidung relevant waren. Die Erklärung soll dabei nicht besonders ausführlich sein. Es ist eher als eine Antwort auf die Frage „Warum tust du das?“ zu sehen.
Bei Breakout könnte ein Zustand so aussehen, dass sich der Ball auf die untere Seite des Spielfeldes zubewegt, aber an der rechten Seite des Schlägers vorbeifliegen wird. In diesem Fall würde man von dem vorgeschlagenen System erwarten, dass zumindest der Ball und der Schläger als relevant markiert werden. Außerdem würde man eine Erklärung wie „Ich bewege mich nach rechts, weil der Ball sich zur rechten Seite des Schlägers bewegt.“ erwarten.

4.2 Verwandte Projekte

Das Problem, ein gegebenes Bild in natürlicher Sprache zu beschreiben, wird in den letzten Jahren vermehrt erfolgreich mit neuronalen Netzen gelöst.
Hierbei wird ein Verfahren verwendet, das ursprünglich zur Übersetzung von Sätzen aus einer Sprache in eine andere entwickelt wurde. Man „übersetzt“ also ein Bild in einen Satz in natürlicher Sprache. Die zu grundlegende Idee bei diesem Verfahren ist, dass man einen *Encoder* verwendet, der die wichtigen Merkmale (engl. *Features*) aus dem Input extrahiert. Die dabei entstehenden Merkmalsvektoren werden dann an einen sogenannten *Decoder* weitergegeben, der daraus Sätze in der gewünschten Sprache generiert.
In [VTBE16] zum Beispiel wird ein konvolutionales Netz mit einem MLP als Output Layer als Encoder verwendet, um einen Merkmalsvektor aus einem Bild zu extrahieren. Dieser wird dann durch ein LSTM Netz in natürliche Sprache übersetzt. Da der Merkmalsvektor selbst keine Folge von Inputs darstellt, erhält das LSTM-Netz bei der Generierung jedes Wortes den Merkmalsvektor als Input x_i (vgl. Kapitel 2.3.2 für meine Notation einer LSTM Zelle).
Xu et al. verwenden in [XBK+15] ebenfalls ein LSTM Netz als Decoder. Um das Bild zu encodieren, verwenden sie jedoch ein konvolutionales Netz ohne eine vollständig verbundene Schicht am Ende. Außerdem teilen sie das Bild in mehrere Gebiete auf und berechnen einen Merkmalsvektor a_i für jedes dieser Gebiete.
Während das Decoder LSTM Netz den Satz generiert, der das Bild beschreiben soll, wird nach jedem Wort h_i die Aufmerksamkeit α_i berechnet, mit der das Netz den Merkmalsvektor a_i jedes Gebietes „betrachtet“. Dazu verwenden Xu et al. ein MLP, welches jeweils ein encodiertes Gebiet a_i und den bisherigen Zellenzustand C_{i-1}(vgl. Kapitel 2.3.2) als Input verwendet und daraus ein positives Gewicht α_i berechnet. Die Merkmalsvektoren a_i aller Gebiete, zusammen mit ihren entsprechenden Gewichten α_i, werden dann

von einem Aufmerksamkeitsmechanismus ϕ zu einem Vektor zusammengefasst. Für die Berechnung des nächsten Wortes h_i wird dann der von ϕ zusammengefasste Vektor als Input x_i verwendet.
Die so durch ϕ zusammengefasste Aufmerksamkeit kann außerdem verwendet werden, um für jedes Wort des beschreibenden Satzes eine visuelle Hervorhebung der relevanten Gebiete zu generieren. Als Aufmerksamkeitsmechanismen ϕ haben Xu et al. hierbei zwei verschiedene Methoden implementiert. Eine „harte“ Aufmerksamkeit, bei der nur das Gebiet markiert wird, das am relevantesten für die Entscheidung war und eine „weiche“ Aufmerksamkeit, bei der die einzelnen Gebiete entsprechend ihrer Gewichte α_i hervorgehoben werden.
Sowohl [VTBE16] als auch [XBK+15] verwenden zum Training ihrer Netze das von Microsoft zur Verfügung gestellte COCO Datenset [LMB+14]. Die neueste Version (2017) dieses Datensets enthält über 200 000 Bilder mit jeweils mindestens fünf sprachlichen Beschreibungen.

Park et al. stellen in [PHA+18] ein Encoder-Decoder System vor, welches Fragen zu Bildern beantworten kann und diese Antworten zusätzlich begründet. Die Frage wird hierbei mit einem LSTM Netz und das Bild mit einem konvolutionalen Netz encodiert. Aufgrund der so gewonnen Merkmale wird ein Antwortsatz in natürlicher Sprache generiert. Auf den Teil des Models, der diese Antwort generiert, möchte ich hierbei nicht weiter eingehen. Für unser Ziel, die Aktionen eines bestärkenden Lernagenten zu erklären, ist die Begründung der Antworten wesentlich relevanter.
Park et al. begründen die Antwort, indem sie den Antwortsatz in einen endlich dimensionalen reellen Raum einbetten und diese Einbettung mit dem encodierten Bild und der encodierten Frage zusammenfassen. Aufgrund dieser Daten generieren sie mit einem konvolutionalen Netz Aufmerksamkeitsregionen, welche sie auf dem Bild hervorheben können, um die Antwort visuell zu begründen.
Danach gewichten sie das encodierte Bild mit den gerade berechneten Aufmerksamkeitsregionen. Diese gewichtete Summe, welche die für die Entscheidung wichtigen visuellen Daten enthält, wird mit der eingebetteten Antwort und der encodierten Frage verbunden. Aus diesen Daten wird dann mit einem LSTM Netz eine sprachliche Erklärung der Antwort generiert.
Um dieses Netz zu trainieren, haben Park et al. ein eigenes Datenset erstellt. Dazu bauen sie auf dem VQA Datenset von [ALA+17] auf, welches aus 200 000 Bildern des COCO Datensets [LMB+14] besteht. Zu jedem dieser Bilder enthält VQA drei Fragen und zu jeder Frage zehn Antworten. Da die meisten dieser Antworten zu linear sind, um interessante Begründungen zu ermöglichen, beschränken sie sich dabei auf Bilder mit komplexen Antworten. Außerdem verwenden sie Bilder aus dem Datenset [GKSS+17], welches Paare aus ähnlichen Bildern enthält, die auf dieselbe Frage verschiedene Antworten geben, da die Begründungen bei solchen Frage-Bild-Paaren besonders interessant sind.
Diese Bilder werden von Menschen annotiert. Dazu werden die Probanden gebeten, zu jedem Frage-Antwort-Paar eine sprachliche Begründung zu schreiben und die Teile des Bildes zu markieren, welche die Antwort am besten rechtfertigen.

Keines der bisher vorgestellten Systeme beschäftigt sich explizit mit der Erklärung von tiefen bestärkenden Lernsystemen. Eine der ersten Arbeiten, die sich insbesondere darauf konzentriert, tiefes bestärkendes Lernen zu erklären, ist [ILL+18].
Dort schlagen Iyer et al. eine Erweiterung von tiefen bestärkenden Lernmethoden vor, die sie *Objekt-orientiertes* tiefes bestärkendes Lernen nennen. Dabei verwenden sie *Template Matching*, um Objekte auf dem letzten Input Frame zu erkennen. Bei Template Matching werden verschiedene Vorlagen über das Bild geschoben, um Bereiche zu identifizieren, die dem Objekt auf der Vorlage ähnlich sind.
Dies erinnert an das Prinzip von konvolutionalen neuronalen Netzen. Der Unterschied ist jedoch, dass die Vorlagen beim Template Matching schon vorher bestimmt und nicht trainiert werden. Für jede Vorlage erhält man dadurch ein Bild, bei dem jeder Pixel entweder 1 oder 0 ist, je nachdem ob der Pixel Teil eines Objektes ist, das auf der Vorlage abgebildet ist oder nicht.
Beim Objekt-orientierten tiefen bestärkenden Lernen verwendet man die Ergebnisse von Template Matching mit k Vorlagen als zusätzliche Kanäle für den Input des neuronalen Netzes und ergänzt dadurch die vier zeitlich versetzten Frames des klassischen tiefen Q-Lernens.
Nun muss man bestimmen, wie wichtig ein Objekt o aus einem der durch Template Matching generierten Kanäle des Zustandes s für die Entscheidung des Q-Netzes ist. Dazu bildet man einen Zustand s_0, bei dem das Objekt o durch die Hintergrundfarbe ersetzt wird. Da das Q-Netz das Objekt o im Zustand s_0 nun nicht mehr „erkennt“, zeigt die Differenz $Q(s,a) - Q(s_0,a)$, wie sehr das Objekt o die Entscheidung des Q-Netzes beeinflusst. Entsprechend dieses Wertes $Q(s,a) - Q(s_0,a)$ heben Iyer et al. das Objekt o im Input Bild farblich hervor, um eine für Menschen verständliche visuelle Erklärung der Entscheidung zu generieren.
Dadurch, dass man explizit nach vorgegebenen Objekten innerhalb des Zustandes sucht, sind die durch dieses Verfahren entstehenden Bilder für Menschen intuitiv verständlich. Ein Nachteil dieser Methode ist, dass Objekt-orientiertes tiefes bestärkendes Lernen zwar auf alle etablierten Lernmethoden angewandt werden kann, aber die Netze trotzdem neu trainiert werden müssen. Außerdem kann die Berechnung der Aufmerksamkeit bei einer hohen Anzahl an Objekten recht rechenintensiv werden.

4.3 Visuelle Aufmerksamkeit

In diesem Kapitel möchte ich darauf eingehen, wie ich die Teile des Input Bildes ermittle, die für die Entscheidung relevant sind.
Das DQN Netz, das wir erklären wollen, wird nicht anhand von annotierten Daten trainiert, sondern mit Hilfe einer Simulation, wie zum Beispiel einem Computerspiel. Da eine solche Simulation für gewöhnlich keine visuelle Aufmerksamkeit enthält, können wir im Gegensatz zu [XBK+15] und [PHA+18] kein spezielles Datenset verwenden, bei dem menschliche Experten die relevanten Bereiche des Bildes annotiert haben.
Aus diesem Grund müssen wir uns darauf beschränken, die Aufmerksamkeit aus den

Parametern zu extrahieren, die das Netz gelernt hat. Dazu verwende ich eine leichte Abwandlung des Verfahrens in [MGR17].
Bei dieser Methode arbeitet man sich, ähnlich wie bei der Back-Propagation (vgl. Kapitel 2.4), nach einer Forward-Propagation von der Output Schicht zum Input Layer vor und bestimmt in jeder Schicht die Einheiten, die zum endgültigen Ergebnis beigetragen haben. Jedoch wenden wir dies nicht während des Trainings an, sondern während der Laufzeit, nachdem das Netz trainiert wurde.
Unser DQN f besteht aus drei konvolutionalen Schichten $conv_1, ..., conv_3$ und zwei vollständig verbundenen Schichten fc_1 und fc_2. Für einen Input x möchte ich mit $fc_i(x)$ bzw. $conv_i(x)$ das Ergebnis des Layers fc_i bzw. $conv_i$ während der Berechnung von $f(x)$ bezeichnen.
Für einen Input x berechnen wir innerhalb einer Forward-Propagation alle diese Werte $fc_i(x)$ und $conv_i(x)$. Dies erfordert keine zusätzliche Rechenleistung, da wir diese Werte ohnehin bestimmen müssen, um den Output $fc_2(x)$ des gesamten Netzes zu erhalten.
Bei der Auswahl einer Aktion betrachtet der Agent die Q-Werte, also die Ergebnisse der Neuronen der Output Schicht, und wählt die Aktion, die dem höchsten Q-Wert entspricht. Nehmen wir an, dass der Agent sich für die j-te Aktion entscheidet, dann beschreibt das j-te Neuron des Output Layers fc_2, wie positiv das Netz die j-te Aktion in diesem Zustand einschätzt.
Um zu begründen, warum der Agent die j-te Aktion sinnvoll findet, reicht es also die Teile des Inputs zu bestimmen, welche für das Ergebnis des j-ten Neurons des Output Layers relevant waren. Dazu ermitteln wir zunächst, welche Einheiten des vorletzten Layers fc_1 dieses Neuron positiv beeinflusst haben. Gehen wir für jede so ermittelte Einheit von fc_1 ebenso vor, dann erhalten wir alle Einheiten der Schicht vor fc_1, die die Wahl der j-ten Aktion positiv beeinflusst haben. Setzen wir diese Vorgehensweise induktiv auf alle Schichten fort, so erhalten wir nach endlichen Schritten alle Einheiten des Input Layers, welche sich positiv auf die Wahl der j-ten Aktionen ausgewirkt haben. Da die Einheiten des Input Layers gerade den Pixeln des Inputs entsprechen, haben wir somit alle Teile des Bildes ermittelt, die für die j-te Aktion relevant sind. Das Layout dieses Verfahrens ist in Abbildung 4.4 dargestellt.
Nun müssen wir nur noch beschreiben, wie man die Einheiten berechnet, die für eine bestimmte Einheit relevant sind.
Dazu beginne ich mit dem Fall, dass N eine Einheit aus einer der vollständig verbundenen Schichten fc_i ist. In diesem Fall ist N ein Neuron mit einem Gewichtsvektor ω und einem Bias b. Der Wert von N während der Berechnung von $f(x)$ entspricht dann $\sigma(\omega \cdot fc_{i-1}(x) + b)$ (vgl. Kapitel 2.1), wobei ich mit $fc_0(x)$ den in Vektorform geschriebenen Output von $conv_3(x)$ bezeichne. Da auf jeden Input derselbe Bias und dieselbe Aktivierungsfunktion angewandt werden, unterscheiden sich die verschieden Inputs nur durch $\omega \cdot fc_{i-1}(x) = \sum_{k=1}^{n} \omega_k fc_{i-1}(x)_k$. In diesem Sinne hat die k-te Einheit von fc_{i-1} genau dann positiv zu dem Ergebnis von N beigetragen, wenn $\omega_k fc_{i-1}(x)_k > 0$ gilt.
Es kann in anderen Situationen sinnvoll sein, nur diejenigen Einheiten zu betrachten, die besonders zu dem Ergebnis beigetragen haben, indem man einen Schwellenwert C festlegt und $\omega_k fc_{i-1}(x)_k > C$ fordert. Außerdem könnte man auch die Einheiten ermitteln, die sich negativ auf das Ergebnis auswirken, indem man $\omega_k fc_{i-1}(x)_k < 0$ betrachtet.

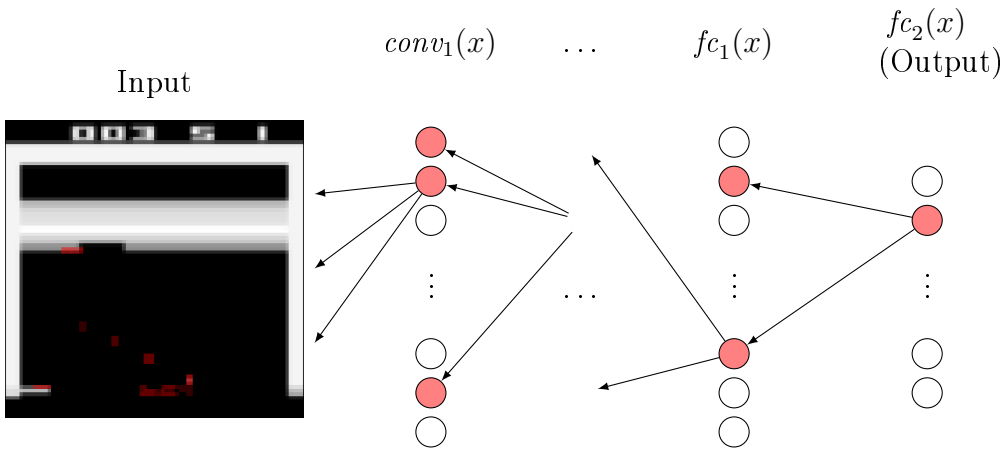


Abbildung 4.4: Berechnung der visuellen Aufmerksamkeit, indem man, beginnend beim Output Layer, die relevanten Einheiten (rot markiert) berechnet. Hierbei wird in jedem Schritt nur ein Neuron zurückverfolgt. Tatsächlich würde man in jeder Schicht alle relevanten Einheiten zurückverfolgen. Der Input wird hier nur durch das aktuellste Frame dargestellt und die relevanten Regionen werden rot markiert. Dabei zeigt die Transparenz der Markierung, in welchem Frame die Relevanz tatsächlich festgestellt wurde. Je weiter das Frame in der Vergangenheit liegt, desto transparenter ist die Markierung.

Für dieses Projekt möchte ich mich jedoch auf die Einheiten festlegen, für die $\omega_k \, fc_{i-1}(x)_k > 0$ gilt.
Da das DQN nur aus Perzeptronen und konvolutionalen Layern besteht, müssen wir noch den Fall betrachten, dass N eine Einheit aus einer der konvolutionalen Schichten $conv_i$ ist.
In diesem Fall ist N ebenfalls ein Neuron, jedoch verarbeitet dieses Neuron nicht den gesamten Output $conv_{i-1}(x)$, sondern nur ein Fenster von $conv_{i-1}(x)$ (vgl. Kapitel 2.3.1). Sei also m die Zeile, n die Spalte und l der Kanal von N in $conv_i(x)$, dann ist N ein Neuron auf dem Fenster $p_{m,n}$, welches wie am Ende von Kapitel 2.3.1 definiert ist. Die relevanten Einheiten innerhalb dieses Fensters $p_{m,n}$ können wir dann genauso bestimmen, wie im vollständig verbundenen Fall. An dieser Stelle unterscheide ich mich von [MGR17], da dort nur die am meisten beitragende Einheit des Fensters ausgesucht wird, anstatt wie im vollständig verbundenen Fall alle relevanten Einheiten zu betrachten.

Auf diese Art und Weise finden wir alle Pixel des Input Bildes, die für die Entscheidung des Agenten relevant waren. Ein Nachteil dieser Methode ist, dass diese Pixel eventuell schwer zu interpretieren sind, da sie nicht anhand von menschlich annotierten Aufmerksamkeiten trainiert wurden. Außerdem kann diese Methode nur auf Netze angewandt werden, bei denen klar ist, welche Einheiten der Output Schicht für die Entscheidung

verantwortlich sind.
Falls jedoch klar ist, welche Einheiten des Output Layers für das Ergebnis relevant sind, kann man diese Vorgehensweise auf beinahe alle Netz-Strukturen anwenden. In [MGR17] ist zum Beispiel noch beschrieben, wie man diese Methode auf LSTM Zellen anwendet. Ein weiterer Vorteil ist, dass dieses Verfahren lediglich eine Forward-Propagation benötigt und somit fast ohne Verlust an Performanz, zur Laufzeit, angewendet werden kann.

4.4 Erklärungen in natürlicher Sprache

In diesem Kapitel beschreibe ich, wie die sprachliche Erklärung der vom Agenten gewählten Aktion generiert wird.
Als Grundgerüst verwende ich, wie [VTBE16], [XBK+15] und [PHA+18], ein Encoder-Decoder System mit einem LSTM Netz als Decoder.
Bei der Struktur des LSTM Erklärungsnetzes orientiere ich mich an der Implementierung in [Zhe18]. Dabei wird auf den Output jeder LSTM Zelle die *softmax* Funktion angewandt, sodass die Einträge des daraus resultierenden Vektors der Wahrscheinlichkeit entsprechen, dass das entsprechende Wort an dieser Stelle gewählt werden sollte.
Das heißt, dass die Dimension des Outputs einer LSTM Zelle der Anzahl an Wörtern entspricht, die das Erklärungsnetz verwenden kann. Das Wörterbuch, aus dem das LSTM Netz die Erklärungssätze zusammensetzten kann, fülle ich hierbei mit allen Wörtern, die bei den Beschreibungen der annotierten Zustände verwendet wurden. Wird diese Trainingsmenge später erweitert, so füge ich neue Wörter automatisch zu dem Wörterbuch hinzu und passe das Erklärungsnetz entsprechend an.
Um die generierten Wörter innerhalb des Erklärungsnetzes an die nächste LSTM Zelle weiterzugeben (vgl. Kapitel 2.3.2), müssen sie in einen endlichen reellen Raum eingebettet werden. Dazu verwende ich keine trainierbare Einbettung, sondern benutze die vortrainierte „Glove“ Einbettung der Universität Stanford [PSM14]. Dadurch muss das Erklärungsnetz nicht mehr selbst lernen, welche Wörter sich ähnlich sind, und benötigt somit weniger annotierte Zustände, um sinnvolle Sätze zu generieren.

Als Encoder bietet es sich an das DQN zu verwenden, wobei man das Output Layer fc_2 weglässt. Somit erhält man einen 512 dimensionalen Merkmalsvektor, welcher alle Informationen enthält, die der Agent für die Entscheidung verwendet.
Wenn man diesen Merkmalsvektor benutzt, läuft man jedoch Gefahr, dass die von dem Decoder erzeugten Erklärungen lediglich den Zustand x beschreiben, anstatt zu begründen, warum der Agent eine bestimmte Aktion in dieser Situation sinnvoll findet.
Um sicherzustellen, dass die Begründung auch von der gewählten Aktion abhängt, verwende ich deshalb die im vorherigen Kapitel 4.3 vorgestellte Methode, um die relevanten Teile des Merkmalsvektors zu ermitteln.
Für den Input x führt der Agent also zunächst eine Forward-Propagation durch, um die richtige Aktion in diesem Zustand zu bestimmen. Danach ermittele ich, wie in Abschnitt 4.3 beschrieben, diejenigen Komponenten von $fc_1(x)$, die für diese Entscheidung relevant

waren.
Nun setze ich alle anderen Einträge von $fc_1(x)$, die sich nicht positiv auf die Entscheidung ausgewirkt haben, auf 0. Somit erhalte ich einen Merkmalsvektor, der genau die Informationen des Inputs zusammenfasst, welche für die Wahl dieser spezifischen Aktion relevant waren.

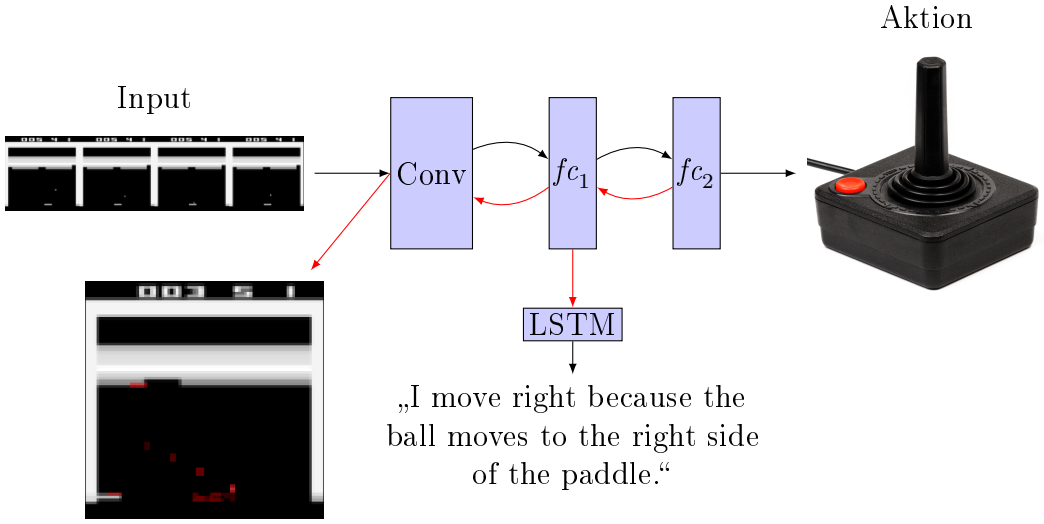


Abbildung 4.5: Das Layout des Erklärungsmodels. Hierbei bezeichnen schwarze Pfeile eine Forward Propagation und rote Pfeile die Berechnung von Aufmerksamkeit nach dem Verfahren in Kapitel 4.3 .

Wenn man nun das LSTM Netz auf diesen Merkmalsvektoren trainiert, so besteht dasselbe Problem, wie bei der visuellen Aufmerksamkeit. Da das DQN bestärkendes Lernen verwendet, trainiert man es nicht auf einem vorgegebenen Datenset. Somit können diese Daten nicht um sprachliche Erklärungen erweitert werden, wie Park et al. es in [PHA+18] tun.
Die Ground Truth Erklärungen müssen also anders erzeugt werden. Hierbei ist die Bezeichnung „Ground Truth“ etwas irreführend, da es oft mehrere passende Erklärungen zu einer bestimmten Aktion gibt. Die Ground Truth Erklärungen sind in diesem Fall lediglich als Beispiele für richtige Erklärungen zu sehen, nicht als einzige objektiv richtige Erklärung.
Um solche Beispiele zu erhalten, trainiere ich das DQN zunächst ohne Erklärungen, bis es das Spiel zufriedenstellend lösen kann. Nach diesem Training soll das System einen Experten danach fragen, warum eine gewählte Aktion in einer bestimmter Situation sinnvoll war. Damit dieser Experte keine Unmenge an Zuständen annotieren muss, verwende ich Methoden des *aktiven Lernens* (siehe [Set09] für eine ausführliche Einführung in dieses Thema), um besonders aussagekräftige Zustände zu identifizieren.
Eine Grundlegende Idee des aktiven Lernens ist es, die Erklärungen zu ermitteln, bei denen sich das System besonders unsicher ist. Dabei entspricht die Sicherheit eines Wortes in dem Erklärungssatz, dem Wert des entsprechenden Eintrags im Output-Vektor

des LSTM Netzes bei der Erzeugung dieses Wortes und die Sicherheit des ganzen Satzes berechne ich als Produkt der Sicherheiten aller Wörter des Satzes. Da die Ergebnisse des Erklärungsnetzes für jedes einzelne Worte zwischen 1 und 0 liegen, befindet sich auch die Sicherheit des ganzen Satzes innerhalb dieser Grenzen.
Um diese Sicherheiten zu bestimmen, muss das Netz zuerst initial trainiert werden. Dazu verwende ich entweder die Erinnerungsmenge, die während des Trainings des DQN angelegt wurde (vgl. Kapitel 2.5), oder erzeuge eine neue Datenmenge, indem ich den Agenten eine Zeit lang das Spiel spielen lassen. Aus dieser Menge wähle ich dann eine bestimmte Menge (10 bei Breakout) an zufälligen Zuständen aus und lege sie dem Experten vor.
Damit dieser die Entscheidung des Agenten nachvollziehen kann, obwohl er den bisherigen Verlauf der Simulation nicht kennt, präsentiere ich dabei die letzten vier Frames nebeneinander. Außerdem berechne ich nach Kapitel 4.3 die visuelle Aufmerksamkeit des Agenten und hebe diese farblich hervor. Dies ist in Abbildung 4.6 dargestellt. Dadurch gründen die Erklärungen des Experten auf genau den Informationen, die der Agent für diese Entscheidung verwendet hat, also denselben Daten, die im Input Vektor des LSTM encodiert sind.
Dem Experten wird dann mitgeteilt, welche Aktion in diesem Zustand ausgeführt wurde und er wird aufgefordert diese Aktion zu erklären. Dabei wird ihm der erste Teil der Erklärung „Ich führe die gewählte Aktion aus, weil“ vorgegeben und er muss nur noch den letzten Teil formulieren.

Abbildung 4.6: Beim initialen Training und beim Pool-basierten aktiven Lernen sieht der Annotator alle vier Frames des Zustandes und die visuelle Aufmerksamkeit des Agenten.

Nachdem das Erklärungsnetz auf den so annotierten Daten initial trainiert wurde, können wir aktive Learnmethoden anwenden. Hierbei habe ich zwei verschiedene Verfahren implementiert.
Beim sogenannten *Stream-basierten* aktiven Lernen lasse ich die Simulation ablaufen und generiere in jedem Schritt visuelle und sprachliche Erklärungen, basierend auf dem bisher trainierten Erklärungsnetz. Bei jedem Schritt überprüfe ich dabei, wie sicher sich das Erklärungsnetz bei der erzeugten Begründung ist.
Liegt diese Sicherheit unter einem von mir festgelegten Schwellwert $C > 0$, dann fragt das System den Experten nach einer neuen Erklärung dieser Entscheidung. Wie beim

initialen Training wird der Experte dabei gebeten, den Satz „Ich führe die gewählte Aktion aus, weil“ zu vervollständigen.
Da der Annotator den bisherigen Verlauf der Simulation beobachten konnte, ist es hierbei nicht nötig alle vier Frames des Zustandes darzustellen. Deshalb zeige ich nur das aktuellste Frame und markiere Aufmerksamkeitsregionen, die in älteren Frames erkannt wurden, mit einer transparenteren roten Markierung (siehe Abbildung 4.5).

Im Gegensatz dazu berechnet man beim *Pool-basierten* aktiven Lernen die Sicherheit des Erklärungsnetzes in jedem einzelnen Zustand der Datenmenge, die bereits für das initiale Training verwendet wurde. Danach wählt man diejenigen Zustände aus, bei denen das Erklärungsnetz am unsichersten ist, und lässt diese auf dieselbe Art und Weise, wie beim initialen Training, von einem Experten annotieren.
Verwendet man die Unsicherheit als einzigen Faktor, um wertvolle Zustände zu identifizieren, dann werden oft Ausreißer (engl. *outlier*) ausgewählt, die den anderen Zuständen überhaupt nicht ähnlich sind. Durch solche Zustände wird das Erklärungsnetz eher „verwirrt“ und es kommt, wenn überhaupt, erst sehr spät zu einer Konvergenz der Erklärungen.
Um dies zu verhindern, schlug Settles in seiner Doktorarbeit [Set08] Kapitel 4 vor, auch die *Dichte oder Ähnlichkeit* eines Zustands zu allen anderen Zuständen des Datensets zu verwenden. Als eine solche Dichte verwende ich die von Settle vorgeschlagene *Cosinus Ähnlichkeit*

$$sim(x, y) = \frac{x \cdot y}{\|x\| \, \|y\|}.$$

Sei x der Merkmalsvektor eines Zustandes s, dann berechne ich die Dichte dieses Vektors x in der Menge aller Merkmalsvektoren x_i der N Zustände aus der Datenmenge des initialen Trainings durch

$$den(x) = \phi(x) * (\frac{1}{N} \sum_{i=1}^{N} sim(x, x_i))^{\beta}.$$

Hierbei bestimmt der Faktor β, wie sehr die Dichte in die Berechnung eingehen soll und $\phi(x)$ beschreibt die Sicherheit des Erklärungsnetzes bei der Verarbeitung des Merkmalsvektors x.
In jedem Trainingsschritt wähle ich also die Zustände mit der geringsten Dichte aus und lasse diese von einem Experten annotieren. Die so annotierten Merkmalsvektoren werden zu den bisher annotierten Merkmalsvektoren hinzugefügt und das Netz wird auf dem daraus resultierenden Datensatz neu trainiert. Danach wird die Dichte mit dem neuen ϕ entsprechend des aktualisierten Erklärungsnetzes neu berechnet und ich wähle wieder die Merkmalsvektoren mit der geringsten Dichte. Um den Rechenaufwand dabei gering zu halten, berechne ich am Beginn des Trainings einmalig $\frac{1}{N} \sum_{i=1}^{N} sim(x, x_i)$ für jeden Merkmalsvektor x der Datenmenge. Indem man diese Ergebnisse speichert, benötigt das Berechnen der Dichte wesentlich weniger Zeit und der Annotator muss nicht lange auf die nächsten zu annotierenden Zustände warten.

4.5 Ergebnisse

In einigen ersten Testläufen habe ich drei verschiedene zufällig ausgewählte und von mir gelabelte initiale Datensets jeweils mit Pool-basiertem und Stream-basierten Lernen annotiert.
Bei beiden Methoden hat das System nach 50-100 zusätzlich zu der initialen Datenmenge gelabelten Zuständen beinahe ausschließlich grammatikalisch richtige Sätze generiert, die in den meisten Fällen eine sinnvolle Erklärung der gewählten Aktion darstellen. Diese Zahl ist sehr gering im Vergleich zu den riesigen Datensets, die beispielsweise in [XBK+15] verwendet wurden, um Bilder zu beschreiben. Dies liegt hauptsächlich daran, dass Breakout ein relativ simples Spiel ist. Dadurch gibt es nicht viele verschiedene Erklärungen, die die Aktionen des Agenten sinnvoll begründen. Ich habe das System zum Beispiel auch auf das Atari Spiel *StarGunner* angewandt, bei dem man sich frei in einem zweidimensionalen Raum bewegen kann und gegnerische Raumschiffe abschießen muss. Dort konnte das System nach 100 annotierten Zuständen schon gut beschreiben, warum der Agent beispielsweise schießt, wenn sich ein Raumschiff vor ihm befindet. Die Bewegungen des Agenten im zweidimensionalen Raum konnten mit dieser Menge an Trainingsdaten jedoch noch nicht verlässlich erklärt werden.
Ein weiterer Grund für die geringe Menge an Trainingsdaten bei Breakout ist, dass das System anders als bei [XBK+15] nicht mehr lernen muss, wie die einzelnen Wörter zusammenhängen, da wir die Glove-Einbettung [PSM14] verwenden.
Bei meinen Tests ist mir aufgefallen, dass das Pool-basierte Lernen zunächst etwas länger braucht, um sinnvolle Sätze zu erzeugen. Ich denke das liegt daran, dass beim Pool-basierten Lernen zunächst alle interessanten Fälle durchgegangen werden, wohingegen beim Stream-basierten Lernen zunächst die am häufigsten vorkommenden Zustände verarbeitet werden.
Ein weiterer Nachteil der Pool-basierten Methode ist, dass die vier angezeigten Frames manchmal nicht genug sind, um wirklich zu verstehen, warum der Agent eine Aktion ausführt. Da man beim Stream-basierten Verfahren den gesamten bisherigen Verlauf der Episode beobachten kann, tritt dieses Problem dort seltener auf.
Im späteren Verlauf des Trainings (ab ungefähr 50 zusätzlich gelabelten Zuständen) dauert es beim Stream-basierten Lernen dafür zunehmend länger, bis man einen Zustand erreicht, bei dem sich das System noch nicht sicher ist. Dadurch muss der Annotator immer länger warten, bis er einen Zustand labeln kann, wohingegen die Pausen beim Pool-basierten Lernen immer gleich lang sind.
Aufgrund dieser Beobachtungen würde ich empfehlen, zunächst Stream-basiertes Lernen zu verwenden und danach auf Pool-basierte Methoden zu wechseln.

Die folgenden Abbildungen wurden nach 50 zusätzlich mit Stream-basierten aktiven Lernmethoden annotierten Zuständen generiert.
Man sieht in Abbildung 4.7, dass der Agent in einfachen aber verständlichen Sätzen erklären kann, warum er sich zum Beispiel in eine bestimmte Richtung bewegt.

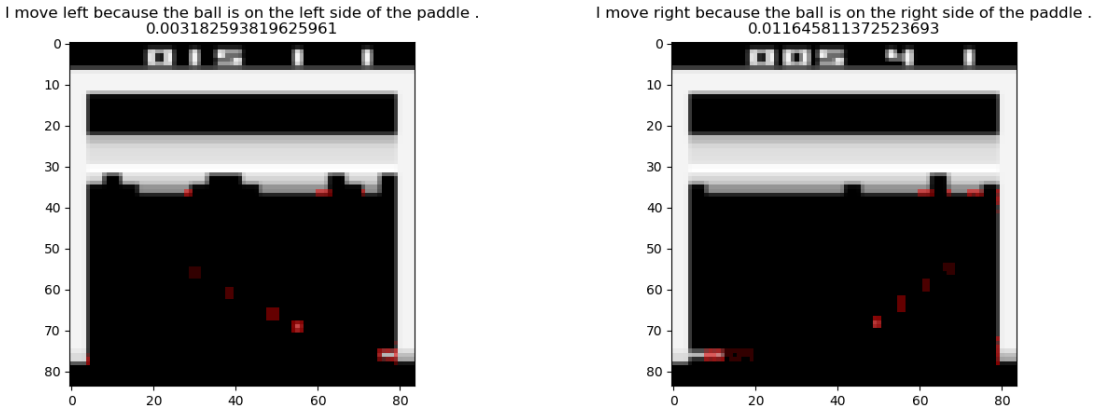
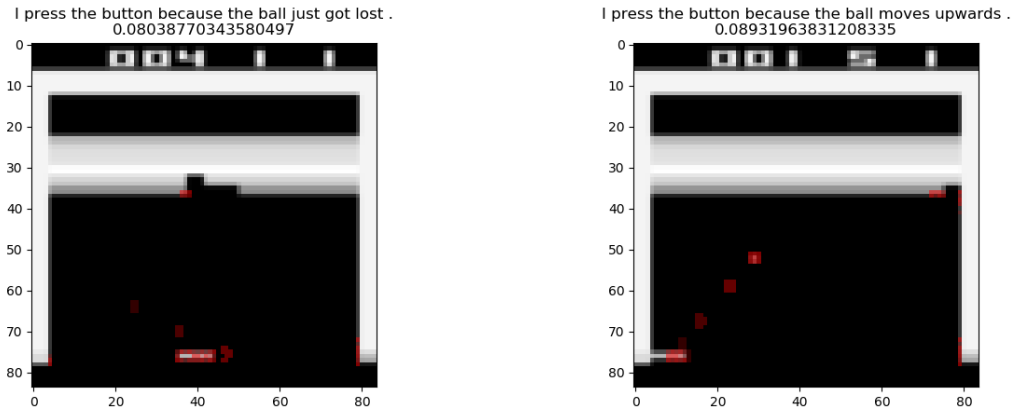


Abbildung 4.7: Das System kann erklären, warum sich der Agent in eine bestimmte Richtung bewegt. Dabei beschreibt die Zahl unter der Erklärung die Sicherheit des Agenten bei der Erklärung.

Das System kann außerdem verlässlich beschreiben, wenn sich der Ball vom Schläger wegbewegt, verloren geht oder den Schläger treffen wird.



Hierbei fällt auf, dass der Agent den Knopf drückt, obwohl er eigentlich gar nichts tun müsste. Dies liegt daran, dass sich die Aktion „den Knopf zu drücken“ nicht von der Aktion „nichts zu tun“ unterscheidet, solange der Ball im Spiel ist. Deshalb kann der Agent keinen Unterschied zwischen den Aktionen feststellen und tendiert sogar eher dazu den Knopf zu drücken, denn dies ist zumindest manchmal die einzige Möglichkeit den Ball wiederzubekommen. Dieses Phänomen sieht man schön in Abbildung 4.8.

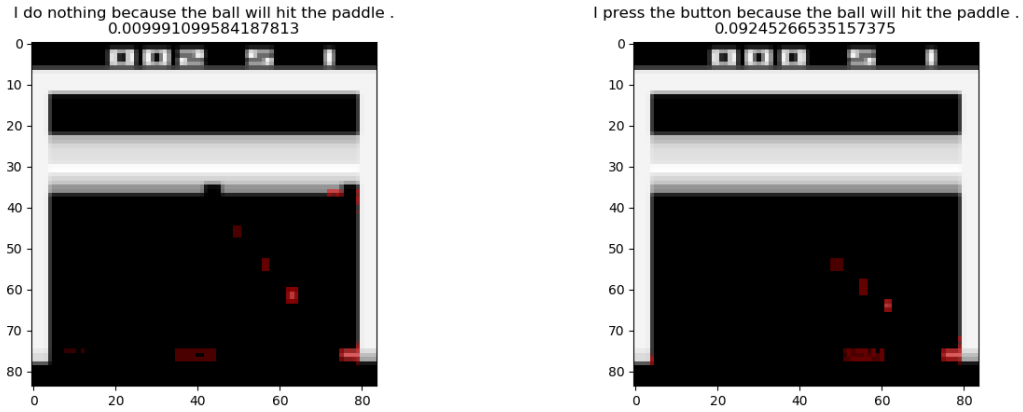


Abbildung 4.8: Ein Beispiel dafür, dass der Agent nicht wirklich zwischen „nichts tun“ und „Knopf drücken“ unterscheidet.

Ich möchte auch ein Beispiel anbringen, bei dem man gut erkennen kann, dass sich multimodale Erklärungen eignen, um Fehler in tiefen bestärkenden Lernen besser zu verstehen. Die beiden Beschreibungen in Abbildung 4.9 lassen darauf schließen, dass der Agent in manchen Situationen nicht unterscheiden kann, ob der Ball auf den Schläger oder auf die Wand treffen wird. In beiden Zuständen bewegt sich der Ball auf eine der Wände zu und die farbige Hervorhebung zeigt, dass der Agent den entsprechenden Teil der Wand erkennen kann. Obwohl es für den Agenten besser wäre sich zu bewegen tut er dies in beiden Fällen nicht. Auf dem linken Bild wird der Agent den Ball deshalb sogar verlieren. Die Erklärung zeigt, dass der Agent nicht handelt, da er fälschlicherweise annimmt, dass der Ball den Schläger treffen wird. Da die Hervorhebung des Wandteils, den der Ball treffen wird, sehr ähnlich aussieht, wie die Markierung des Schlägers, ist es gut möglich, dass der Agent die Wand in diesem Zustand für einen Schläger hält. Ob dies wirklich stimmt, müsste man noch explizit testen, aber die multimodalen Erklärungen ermöglichen es in diesem Fall zumindest eine Hypothese aufzustellen, die man überprüfen kann.

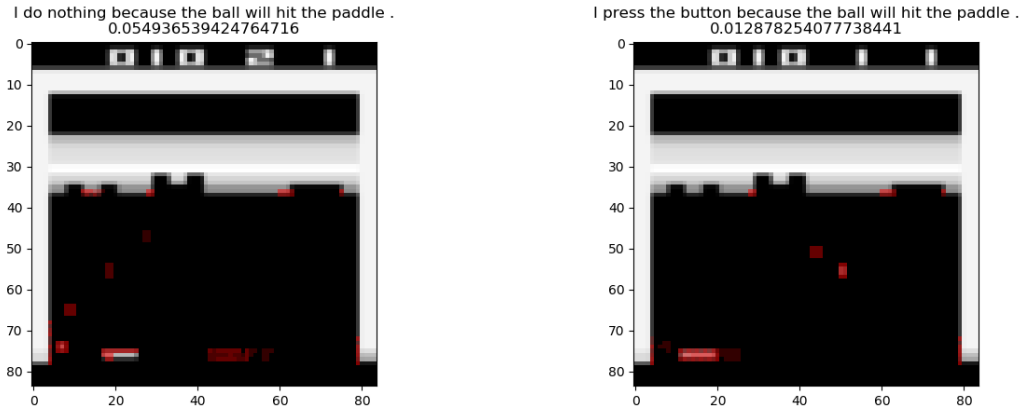


Abbildung 4.9: Beispiele für Fälle, bei denen multimodale Erklärungen helfen können, einen Fehler des Agenten zu analysieren.

4.6 Konklusion

In dieser Arbeit habe ich ein System implementiert, das die Handlungen eines Agenten erklärt, der gelernt hat das Atari 2600 Spiel Breakout zu lösen.
Dabei verwende ich eine Verbindung aus Aufmerksamkeitsmechanismen und Encoder-Decoder Übersetzungsmethoden, um die relevanten Teile des Bildschirms zu markieren und die gewählte Aktion in einem kurzen Satz zu begründen.
Ein ähnliches Vorgehen wurde zum Beispiel in [PHA+18] bereits benutzt, um die Entscheidungen eines neuronalen Netzes zu rechtfertigen.
Das besondere an dieser Arbeit ist, dass ich ein neuronales Netz erkläre, welches mit bestärkendem Lernmethoden trainiert wurde. Im Kontext des tiefen bestärkenden Lernens ist mir keine andere Arbeit bekannt, die multimodale Erklärungen erzeugt.
Außerdem teste ich Stream-basierte und Pool-basierte aktive Lernmethoden, um interessante Trainingsdaten für diese Erklärungen zu identifizieren. Mit beiden Verfahren konnte ein einziger Annotator, in weniger als einer Stunde, genug Trainingsdaten annotieren, um das Erklärungsnetz soweit zu trainieren, dass es in der Lage war kontinuierlich sinnvolle Erklärungssätze zu generieren.
Diese Verbindung visueller und sprachlicher Erklärungen hilft Menschen, die kein Vorwissen im Bereich des maschinellen Lernens besitzen, die Aktionen eines tiefen bestärkenden Lernsystems zu Verstehen.
Außerdem können auch Entwickler diese Begründungen verwenden, um ihre eigenen Systeme zu debuggen. Im Fall von Breakout zeigen die generierten Erklärungen zum Beispiel, dass der Agent nicht zwischen dem Drücken eines Knopfes und der Aktion „nichts zu tun“ unterscheidet. Dem könnte man entgegenwirken, indem man dem Agenten jedes mal eine negative Belohnung gibt, wenn er eine andere Aktion als „nichts tun“ ausführt.

4.7 Anwendungsgrenzen und zukünftige Arbeit

Die Ergebnisse dieser Arbeit sind bis jetzt nur als „proof of concept“ zu sehen.
Um die tatsächliche Nützlichkeit dieses Systems zu beweisen, muss man ausführlicher evaluieren, ob die so generierten Erklärungen tatsächlich helfen, die Aktionen des Agenten nachzuvollziehen.
Dazu kann man einen ähnlichen Versuchsaufbau wie in [ILL+18] verwenden. Hierbei werden einer Menge von Probanden jeweils einzelne Zustände der Simulation vorgelegt. Die Probanden sollen dann anhand der multimodalen Erklärungen vorhersagen, welche Aktion der Agent in dieser Situation wählen wird. Falls die generierten Erklärungen das Verhalten des Agenten tatsächlich ausreichend begründen, sollte dies in den meisten Fällen möglich sein.

Ein weitere Einschränkung des hier vorgestellten Erklärungssystems ist, dass für kompliziertere Probleme immer mehr menschliche annotierte Erklärungen benötigt werden. Während Breakout noch simpel genug war, dass ich selbst die Ground Truth Begründungen annotieren konnte, müsste man diese Aufgabe für komplexere Probleme auf mehrere Personen aufteilen.

Des Weiteren können die sprachlichen Erklärungen nur dann generiert werden, wenn das zu erklärende tiefe bestärkende Lernsystem in der Lage ist das Problem gut zu lösen. Falls sich der Agent unsinnig verhält, ist es für die Annotatoren schwierig eine sinnvolle Erklärung zu dieser Aktion zu formulieren. Dies habe ich zum Beispiel bei dem Atari 2600 Spiel StarGunner beobachtet, dort bewegt sich der Agent oft sehr zufällig durch den zweidimensionalen Raum, bis er einen Zustand erreicht, den er eindeutig zuordnen kann. Diese zufälligen Bewegungen sind sehr schwer zu erklären.
Indem man in Zukunft neuere tiefe bestärkende Lernsysteme implementiert, die auch komplexere Probleme lösen können, könnte man diesem Problem vorbeugen.
Hinzukommt, dass man bei dem Deep Q Network, welches ich in dieser Arbeit implementiert habe, nur vier Frames als Input verwendet um zeitliche Zusammenhänge zu analysieren. Dadurch können die generierten Erklärungen jede Aktion nur kurzfristig begründen. Das LSTM Netz kann keine langfristigen Pläne lernen, da der Merkmalsvektor nur die letzten vier Frames encodiert.
Neuere Ansätze des tiefen bestärkenden Lernens, wie zum Beispiel [JCD+18], verwenden zusätzlich zu konvolutionalen auch rekurrente neuronale Netze, um weiter in die Zukunft planen zu können.
Es wäre interessant zu beobachten, welche Erklärungen zu einem solchen Netz generiert werden, wenn man auch rekurrente Netze verwendet, um einen Merkmalsvektor für die Erklärungen zu erzeugen. Meine Hoffnung wäre, dass sich dadurch auch Erklärungen für längerfristige Pläne erstellen lassen.

Literaturverzeichnis

[ABC+18] Andrychowicz, Marcin ; Baker, Bowen ; Chociej, Maciek ; Józefowicz, Rafał ; McGrew, Bob ; Pachocki, Jakub ; Petron, Arthur ; Plappert, Matthias ; Powell, Glenn ; Ray, Alex ; Schneider, Jonas ; Sidor, Szymon ; Tobin, Josh ; Welinder, Peter ; Weng, Lilian ; and, Wojciech Z.: Learning Dexterous In-Hand Manipulation. In: *ArXiv e-prints* (2018). `https://arxiv.org/abs/1808.00177v1`

[ALA+17] Agrawal, Aishwarya ; Lu, Jiasen ; Antol, Stanislaw ; Mitchell, Margaret ; Zitnick, C L. ; Parikh, Devi ; Batra, Dhruv: Vqa: Visual question answering. In: *International Journal of Computer Vision* 123 (2017), Nr. 1, S. 4–31

[Alt06] Alt, Hans W.: *Lineare Funktionalanalysis: Eine anwendungsorientierte Einführung*. Springer-Verlag, 2006

[Cau47] Cauchy, Augustin: Méthode générale pour la résolution des systemes d'équations simultanées. In: *Comp. Rend. Sci. Paris* 25 (1847), Nr. 1847, S. 536–538

[CHJH02] Campbell, Murray ; Hoane Jr, A J. ; Hsu, Feng-hsiung: Deep blue. In: *Artificial intelligence* 134 (2002), Nr. 1-2, S. 57–83

[Cyb89] Cybenko, George: Approximation by superpositions of a sigmoidal function. In: *Mathematics of control, signals and systems* 2 (1989), Nr. 4, S. 303–314

[DHS11] Duchi, John ; Hazan, Elad ; Singer, Yoram: Adaptive subgradient methods for online learning and stochastic optimization. In: *Journal of Machine Learning Research* 12 (2011), Nr. Jul, S. 2121–2159

[Don69] Donoghue, William F.: *Distributions and Fourier transforms*. Academic Press, 1969

[DPF+] Dennison, Christy ; Pachocki, Jakub ; Farhi, David ; Raiman, Jonathan ; Petrov, Michael ; Tang, Jie ; Dębiak, Przemysław ; Brockman, Greg ; Zhang, Susan ; Józefowicz, Rafał ; Wolski, Filip ; Pondé, Henrique ; Sidor, Szymon ; Chan, Brooke ; Hesse, Christopher ; Gray, Scott ; Radford, Alec ; Clark, Jack ; Hashme, Shariq ; Fischer, Quirin ; Sutskever, Ilya ; Schulman, John ; Berner, Christopher ; Schneider, Jonas ; Sigler, Eric ; Christiano, Paul ; Schiavo,

Larissa ; Yoon, Diane ; Luan, David: OpenAI Five. In: *OpenAi blog* https://blog.openai.com/openai-five/. – [Online; Stand 28.08.2018]

[Eva98] Evans, Lawrence C.: *Partial Differential Equations*. American Mathematical Society, 1998 (Graduate Studies in Mathematics, V. 19 GSM/19). – ISBN 9780821807729,0821807722

[Fis13] Fischer, G.: *Lineare Algebra: Eine Einführung für Studienanfänger*. Springer Fachmedien Wiesbaden, 2013 (Grundkurs Mathematik). – ISBN 9783658039455

[FJT18] Fridman, Lex ; Jenik, Benedikt ; Terwilliger, Jack: DeepTraffic: Driving Fast through Dense Traffic with Deep Reinforcement Learning. In: *CoRR* abs/1801.02805 (2018). http://arxiv.org/abs/1801.02805

[For13] Forster, O.: *Analysis 2: Differentialrechnung im IRn, gewöhnliche Differentialgleichungen*. Springer Fachmedien Wiesbaden, 2013 (Grundkurs Mathematik). – ISBN 9783658023577

[For16] Forster, O.: *Analysis 1: Differential- und Integralrechnung einer Veränderlichen*. Springer Fachmedien Wiesbaden, 2016 (Grundkurs Mathematik). – ISBN 9783658115456

[GBC16] Goodfellow, Ian ; Bengio, Yoshua ; Courville, Aaron: *Deep Learning*. MIT Press, 2016. – http://www.deeplearningbook.org

[GKSS+17] Goyal, Yash ; Khot, Tejas ; Summers-Stay, Douglas ; Batra, Dhruv ; Parikh, Devi: Making the V in VQA matter: Elevating the role of image understanding in Visual Question Answering. In: *CVPR* Bd. 1, 2017, S. 3

[Gol17] Goldberg, Yoav: Neural network methods for natural language processing. In: *Synthesis Lectures on Human Language Technologies* 10 (2017), Nr. 1, S. 1–309

[HS97] Hochreiter, Sepp ; Schmidhuber, Jürgen: Long short-term memory. In: *Neural computation* 9 (1997), Nr. 8, S. 1735–1780

[ILL+18] Iyer, Rahul ; Li, Yuezhang ; Li, Huao ; Lewis, Michael ; Sundar, Ramitha ; Sycara, Katia: Transparency and Explanation in Deep Reinforcement Learning Neural Networks. In: *Aies Conference - Artificial Intelligence, Ethics, and Society*, 2018

[JCD+18] Jaderberg, M. ; Czarnecki, W. M. ; Dunning, I. ; Marris, L. ; Lever, G. ; Garcia Castaneda, A. ; Beattie, C. ; Rabinowitz, N. C. ; Morcos, A. S. ; Ruderman, A. ; Sonnerat, N. ; Green, T. ; Deason, L. ; Leibo, J. Z. ; Silver, D. ; Hassabis, D. ; Kavukcuoglu, K. ; Graepel, T.: Human-level performance in first-person multiplayer games with

population-based deep reinforcement learning. In: *ArXiv e-prints* (2018). https://arxiv.org/abs/1807.01281v1

[Kas17] Kasparow, Garri: *Kasparov Exclusive: His MasterClass, St. Louis, AlphaZero*. https://www.twitch.tv/videos/217167956?t=17m55s, 2017. – [Online; Stand 07. September 2018]

[LMB+14] Lin, Tsung-Yi ; Maire, Michael ; Belongie, Serge ; Hays, James ; Perona, Pietro ; Ramanan, Deva ; Dollár, Piotr ; Zitnick, C L.: Microsoft coco: Common objects in context. In: *European conference on computer vision* Springer, 2014, S. 740–755

[LMMH17] Lathuilière, Stéphane ; Massé, Benoît ; Mesejo, Pablo ; Horaud, Radu: Neural Network Reinforcement Learning for Audio-Visual Gaze Control in Human-Robot Interaction. In: *CoRR* abs/1711.06834 (2017). http://arxiv.org/abs/1711.06834

[LP93] Lin, Vladimir Y. ; Pinkus, Allan: Fundamentality of ridge functions. In: *Journal of Approximation Theory* 75 (1993), Nr. 3, S. 295–311

[MGR17] Mopuri, Konda R. ; Garg, Utsav ; Radhakrishnan, Venkatesh B.: CNN Fixations: An unraveling approach to visualize the discriminative image regions. In: *arXiv preprint arXiv:1708.06670v2* (2017)

[MKS+15] Mnih, Volodymyr ; Kavukcuoglu, Koray ; Silver, David ; Rusu, Andrei A. ; Veness, Joel ; Bellemare, Marc G. ; Graves, Alex ; Riedmiller, Martin ; Fidjeland, Andreas K. ; Ostrovski, Georg u. a.: Human-level control through deep reinforcement learning. In: *Nature* 518 (2015), Nr. 7540, S. 529

[MRLP16] Modares, Hamidreza ; Ranatunga, Isura ; Lewis, Frank L. ; Popa, Dan O.: Optimized assistive human–robot interaction using reinforcement learning. In: *IEEE transactions on cybernetics* 46 (2016), Nr. 3, S. 655–667

[Ola15] Olah, Christopher: Understanding lstm networks. In: *GITHUB blog* (2015). http://colah.github.io/posts/2015-08-Understanding-LSTMs. – [Online; Stand 28.08.2018]

[PHA+18] Park, Dong H. ; Hendricks, Lisa A. ; Akata, Zeynep ; Rohrbach, Anna ; Schiele, Bernt ; Darrell, Trevor ; Rohrbach, Marcus: Multimodal Explanations: Justifying Decisions and Pointing to the Evidence. In: *CoRR* abs/1802.08129 (2018). http://arxiv.org/abs/1802.08129

[Pin99] Pinkus, Allan: Approximation theory of the MLP model in neural networks. In: *Acta numerica* 8 (1999), S. 143–195

[PSM14] Pennington, Jeffrey ; Socher, Richard ; Manning, Christopher D.: GloVe: Global Vectors for Word Representation. In: *Empirical Methods in Natural Language Processing (EMNLP)*, 2014, 1532–1543

[Rud87] Rudin, Walter: *Real and complex analysis*. McGraw-Hill, 1987

[Rud91] Rudin, Walter: Functional analysis 2nd ed. In: *International Series in Pure and Applied Mathematics. McGraw-Hill, Inc., New York* (1991)

[SB+98] Sutton, Richard S. ; Barto, Andrew G. u. a.: *Reinforcement learning: An introduction*. MIT press, 1998

[Sch14] Schmidt, Bernd: *Analysis 2*. Vorlesungsskript. `https://www.math.uni-augsburg.de/prof/ana/arbeitsgruppe/schmidt/skripten/ana2.pdf`. Version: Sommersemester 2014

[Set08] Settles, Burr: *Curious machines: Active learning with structured instances*, University of Wisconsin–Madison, Diss., 2008

[Set09] Settles, Burr: Active Learning Literature Survey / University of Wisconsin–Madison. 2009 (1648). – Computer Sciences Technical Report

[Sim09] Simon, Haykin: *Neural networks and learning machines*. Bd. 3. Pearson Upper Saddle River, NJ, USA:, 2009

[SPN+] Schulman, John ; Pfau, Vicki ; Nichol, Alex ; Hesse, Christopher ; Klimov, Oleg ; Schiavo, Larissa: Retro Contest: Results. In: *OpenAI blog* `https://blog.openai.com/first-retro-contest-retrospective/`. – [Online; Stand 28.08.2018]

[SSS+17] Silver, David ; Schrittwieser, Julian ; Simonyan, Karen ; Antonoglou, Ioannis ; Huang, Aja ; Guez, Arthur ; Hubert, Thomas ; Baker, Lucas ; Lai, Matthew ; Bolton, Adrian u. a.: Mastering the game of Go without human knowledge. In: *Nature* 550 (2017), Nr. 7676, S. 354

[Tae17] Taehoon, Kim: *DQN-tensorflow*. `https://github.com/devsisters/DQN-tensorflow`, 2017. – [commit: c7b1f1051dfa152530322445fc8febb9a2ea078b]

[VTBE16] Vinyals, Oriol ; Toshev, Alexander ; Bengio, Samy ; Erhan, Dumitru: Show and Tell: Lessons learned from the 2015 MSCOCO Image Captioning Challenge. In: *CoRR* abs/1609.06647 (2016). `http://arxiv.org/abs/1609.06647`

[Wik18] Wikipedia: *Atari 2600 — Wikipedia, Die freie Enzyklopädie*. `https://de.wikipedia.org/w/index.php?title=Atari_2600&oldid=177885895`. Version: 2018. – [Online; Stand 29. August 2018]

[XBK+15] Xu, Kelvin ; Ba, Jimmy ; Kiros, Ryan ; Cho, Kyunghyun ; Courville, Aaron C. ; Salakhutdinov, Ruslan ; Zemel, Richard S. ; Bengio, Yoshua: Show, Attend and Tell: Neural Image Caption Generation with Visual Attention. In: *CoRR* abs/1502.03044 (2015). http://arxiv.org/abs/1502.03044

[Zhe18] Zhenguo, Chen: *Neural-Network-Image-Captioning*. https://github.com/ZhenguoChen/Neural-Network-Image-Captioning, 2018. – [commit: 7463a2cd167e263215491e1728b3f2062b4be0f5]