Erklärung

Hiermit erkläre ich, dass ich diese schriftliche Abschlussarbeit selbstständig verfasst habe, keine anderen als die angegebenen Hilfsmittel und Quellen benutzt habe und alle wörtlich oder sinngemäß aus anderen Werken übernommenen Aussagen als solche gekennzeichnet habe.

Tübingen, den 14.06.2017

(Steffen Schnürer)

Inhaltsverzeichnis

1 Einleitung

TODO Aus der Psychologie kennen wir Ansätze wie die Event Segmentation Theory (Zacks et al. 2007), die beschreiben wie es Teil der menschlichen Wahrnehmung ist kontinuierliche Vorgänge in diskrete, bedeutungsvolle Ereignisse zu unterteilen [**?**]. Die Motivation hinter einer entsprechenden automatisierten Event Segmentation ist vielfältig. Ein möglicher, aber auch sehr ambitionierter Anwendungsbereich ist Event Segmentation von menschlichen Aktionen. Aktionen können so unterteilt, klassifiziert und eventuell sogar vorhergesagt werden. So ist zum Beispiel die Aktion des Kaffee Trinkens aus einer Tasse unterteilbar in "Hand zu Tasse führen", "Hand umgreift Henkel", "Tasse wird zum Mund geführtünd "Kaffee wird getrunken", jeweils getrennt durch Ereignisgrenzen wie z.B. "Hand erreicht Tasse".

Ein sehr einfaches Beispiel einer solchen unterteilbaren kontinuierlichen Aktivität ist das in dieser Arbeit betrachtete Bouncing Ball Szenario. Ein Ball gleitet mit steter Geschwindigkeit in einer quadratischen Ebene. Erreicht er eine Kante, prallt er von ihr ab und gleitet in die entsprechende Richtung weiter.

2 Grundlagen

In diesem Kapitel möchte ich einige Grundlagen für die weiteren Kapitel dieser Arbeit legen. Zunächst erläutere ich den Aufbau eines neuronalen Netzes im allgemeinen, die Bedeutung von rekurrenten neuronalen Netzen (RNN), erkläre das Vanishing Gradient Problem und mache so die Notwendigkeit der LSTM-Technik deutlich.

2.1 Neuronale Netze

Als neuronales Netz bezeichnet man eine verwobene Struktur zwischen vielen einzelnen Zellen von meistens gleichem - aber keineswegs darauf beschränktem - Aufbau, den sogenannten Neuronen. Eine solche Zelle hat immer die Eigenschaft, dass sie Signale von anderen Zellen empfängt, diese gewichtet aufakkumuliert und abhängig von einer internen Aktivierungsfunktion ein entsprechendes Signal an andere Zellen weitergibt, die damit ihrerseits diesselbe Prozedur durchlaufen. Ein neuronales Netz im Gehirn einer Ameise hat ca. 250.000 Neuronen, ein menschliches 86 Milliarden [1] und wir haben lediglich eine wage Vorstellung, wozu diese imstande sind. In der Informatik werden solche Strukturen als künstliche neuronale Netze nachgebildet und je nach Problemstellung abgewandelt. Zur Vereinfachung meinen wir ab sofort, sofern nicht explizit anders angegeben, mit neuronalen Netzen künstliche neuronale Netze.
Es gibt viele verschiedene Arten von neuronalen Netzen, die einfachste von Ihnen ist das Multilayer Perceptron (MLP). Die Zellen werden zu mehreren Ebenen(Layer) zusammengefügt. Erst das Inputlayer, in das die Daten hineingegeben werden, dann die versteckten Layer, versteckt weil hier wie in einer Blackbox die Berechnungen passieren und dann an das Outputlayer weiterleitet, aus welchem das Ergebnis des Netzes kommt.Es gibt eine geordnete Datenflussrichtung, eine Zelle erhält ihren Input von jeder Zelle der vorigenen Ebene und gibt ihren Output entsprechend an jede

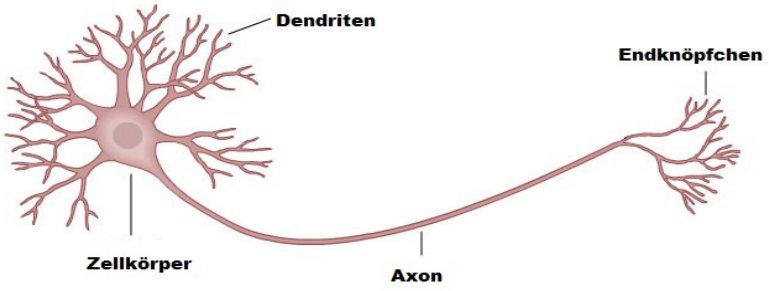

Abbildung 2.1: Schematische Darstellung eines biologischen Neuron [5]

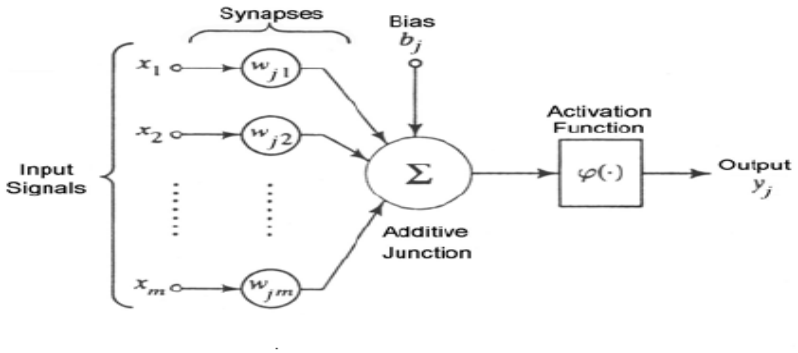

Abbildung 2.2: Schematische Darstellung einer Möglichkeit, ein künstliches Neuron zu simulieren [3]

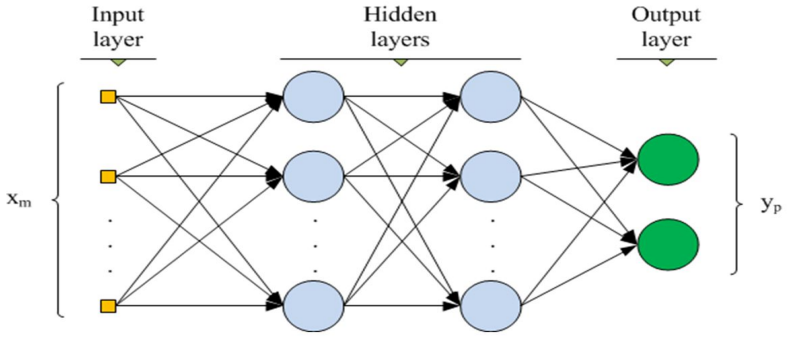

Abbildung 2.3: Ein Multilayer Perceptron mit m Inputneuronen(gelb), zwei Hidden Layern(blau) und zwei Outputneuronen(grün). [2]

Zelle der folgenden Ebene weiter, Feedforward genannt. Ein solches Netz sieht man in Abbildung 2.3.

$$x_j = \varphi_h(net_h) = \varphi_h(\sum_{i=1}^{n} x_i w_{ij}) \tag{2.1}$$

In jeder Zelle j werden die Inputsignale x_i, also die Aktivierungen der vorigen Zellen mit Index i unterschiedlich gewichtet w_{ij} aufsummiert net_h (Abbildung 2.2). Eben diese Gewichte sind der Kern des maschinellen Lernens, da bei einmal geschickt gefundenen Gewichten, komplexe Aufgabenstellungen und Probleme mit (vergleichsweise) wenig Rechen- sowie Programmieraufwand gelößt werden können. Im Wesentlichen versucht ein MLP immer eine Funktion zu berechnen welches einen Inputvektor der Größe n auf einen Outputvektor der Größe m abbildet.

$$f_{MLP} : R^n \to R^m$$

Ein Beispiel für eine solche Funktion wäre für ein gegebenes Bild zu entscheiden, ob darauf ein Gesicht zu erkennen ist. Die Länge des Inputs wäre hier z.B. mehrere Millionen, einfach die Pixelwerte, als Output wäre hier nur eine Zahl, nämlich ob ein Gesicht zu sehen ist oder nicht. Der Output y des Netzes ist also abhängig vom Input x, aber auch von den Gewichten w. Die Gewichte werden entweder mit Hilfe von Trainingsdaten, bestehend aus Inputdaten und entsprechenden Outputdaten, also den zugehören Lösungen, oder mit einer zu erlenenden Zielfunktion, die einem aus gegebenen Inputdaten die gewünschten Lösungen liefert, über mehrere Trainingsläufe (Epochen) justiert. Fügt man in die Inputebene entsprechende Inputdaten x ein, berechnet die Aktivierungen aller Zellen, vergleicht die Aktivierungen y des Outputlayers des Netzes mit den gegebenen Lösungen z und erhält eine Fehlerabweichung $E(z, y)$:

$$E(z, y) =_{def} \frac{1}{2} \sum_{i=1}^{m} (z_i - y_i)^2 \tag{2.2}$$

Die Herausforderung ist es nun, diejenigen Gewichte w zu finden, für die die Fehlerabweichung über alle Trainingsdaten minimal ist.

$$arg_w min(\sum_{(x,z) \in trainset} E(z, f_{MLP}(w, x))) \tag{2.3}$$

Bei der Anpassung der Gewichte der versteckten Zellen hat man aber das Problem, das man keinen direkten Fehler für sie kennt, da man nur für die Outputschicht den gewünschten Output hat und so einen Fehler ermitteln kann. Als Lösung ist hier der Backpropagation-Algorithmus geläufig. Dieser besteht aus 3 Schritten, die entweder solang durchlaufen werden bis der Fehler einen festgesetzten Schwellenwert(Threshold) unterschreitet, oder eine vorgegebene Anzahl an Epochen durchlaufen wurde.

Feedforward

Das Inputlayer wird mit der Eingabe des jeweiligen Testsatzes befüllt und die Aktivierungen vorwärts Layer für Layer, Zelle für Zelle berechnet.

Backward-Pass

Das Ergebnis am Outputlayer wird mit der gewünschten Lösung des Testsatzes verglichen und der Fehler berechnet. Diese werden entsprechend gewichtet die unteren Layer rückwärts Zelle für Zelle rückpropagiert. Oft werden hier statt dem tatsächlichen Fehler direkt der Gradient bzw. ein Zwischenwert für dessen Berechnung propagiert.

Anpassen der Gewichte

Dies ist der wichtige Schritt, hier werden nun Zelle für Zelle die Gewichte entsprechend des Gradienten angepasst. Mit den neuen Gewichten ist in der Regel der Output des Netzes genauer, der Fehler also geringer und der Algorithmus lässt die Fehlerfunktion im Idealfall gegen 0 konvergieren. [**?**]

Die Gewichte werden anhand eines absteigenden Gradienten der Fehlerfunktion, abgeleitet nach den Gewichten w_{ij}, aktuallisiert. Hierzu werden die δ, die aufsummierten Fehler für jede Zelle wie folgt berechnet:

$$\delta_k = \varphi'_k(net_k)(x_k - z_k) \tag{2.4}$$

$$\delta_h = \varphi'_h(net_h) \sum_{k \in K} w_{hk}\delta_k \tag{2.5}$$

Wobei erst nach der oberen Gleichung für die Outputlayer die δ_k gebildet und in die jeweils unteren Layer nach der unteren Gleichung beim bilden der δ_h gewichtet aufsummiert werden (Backpropagation). K sind hier die Neuronen aus dem jeweils oberen Layer. Die Formel für das Gewichtsupdate mit einer Lernrate η lautet dann [2]:

$$\Delta w_{ij} =_{def} -\eta \frac{\partial E}{\partial w_{ij}} = -\eta x_i \delta_j \tag{2.6}$$

Es gibt nun noch verschiedene Techniken um das Ergebnis weiter zu verbessern und das lernen noch schneller und stabiler zu machen, an dieser Stelle wollen wir unser Augenmerk aber in eine andere Richtung lenken.

2.2 Rekurrente Neuronale Netze

Neuronale Netzwerke haben einen für einen 1-Dimensionalen Inputvektor einen eindeutig zu berechnenden 1-Dimensionalen Outputvektor und für die meisten Problemstellungen ist auch genau diese Kompetenz gefordert. Es gibt aber auch Probleme, bei denen der Output nicht nur von diesem, sondern auch von allen beziehungsweise einigen vorigen Inputs und Zuständen abhängt. Ein anschauliches Beispiel wäre hier die Klassifizierung von Videoausschnitten, wo die Bedeutung eines einzelnen Frames vom Kontext der vorigen Frames abhängt, hingegen ein einfacheres Beispiel wäre das in dieser Arbeit betrachtete Bouncing Ball Szenario. Befindet sich ein Ball im Punkt $p_1 = (0/0)$ und der nächste Ort p_2 soll vorhergesagt werden, ist es natürlich entscheidend ob der Ball sich zuvor beispielsweise im Punkt $p_0 = (0.1/0, 1)$ oder im Punkt $p_0 = (-0.1/-0.1)$ befand. Hierzu werden den Zellen aus dem verstecktem

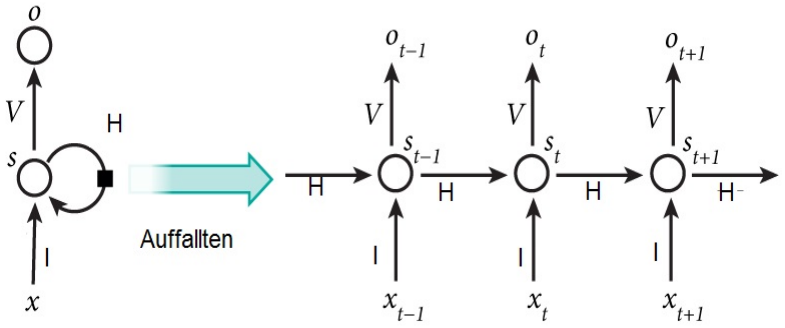

Abbildung 2.4: Um Backpropagation für ein rekurrentes Netz zu benutzen, muss man es durch die Zeit auffallten. [?]

Layer des neuronalen Netzes rekurrente (lat. recurrere:zurücklaufen) Verbindungen hinzugefügt. Eine Zelle h bekommt nun im Zeitpunkt t seinen Input immernoch von allen Zellen des unteren Layers I, zusätzlich nimmt sie aber auch noch als Input den Output aller Zellen desselben Layers, aber aus dem vorigen Zeitschritt H', die Formel für die Aktivierung lautet also:

$$x_h^t = \varphi_h(net_h^t) = \varphi_h(\sum_{i \in I} w_{ih} x_i^t + \sum_{i \in H'} w_{h'h} x_{h'}^{t-1}) \tag{2.7}$$

Da diese neuen rekurrenten Inputs auch wieder gewichtet verarbeitet werden, müssen diese erst noch geschickt gefunden werden. Dies macht man mit Backpropagation durch die Zeit (BPTT). Dies passiert analog zur Backpropagation im MLP, jedoch werden die einzelnen Zeitschritte aufgeklappt (Abbildung 2.4). Die Backpropagation through time berechnet sich also wiefolgt:

$$\delta_k^t = \varphi_k'(net_k^t)(x_k^t - z_k^t) \tag{2.8}$$

$$\delta_h^t = \varphi_h'(net_h^t)(\sum_{k \in K} w_{hk} \delta_k^t + \sum_{h' \in H'} w_{hh'} \delta_{h'}^{t+1}) \tag{2.9}$$

Wobei hier die obere Formel wieder für jedes Neuron des Outputlayers und die untere für die Hiddenlayer. Für das Gewichtsupdate werden die Deltas noch über den betrachteten Zeitraum aufsummiert und analog zum Multilayer Perzeptron berechnet. Den Backpropagation Algorithmus anwenden um das Netz zu trainieren und passende Gewichte, auch für die rekurrenten Verbindungen zu erhalten. Nun hat man ein Neuronales Netz mit einer Art Kurzzeitgedächtnis, welches beim Auswerten des Inputs die Nahe Vergangenheit mit in Betracht zieht. Hat es in der Anwendung aber tiefere Abhängigkeiten, wo ein vergangener Input einen Einfluss auf die fernere Zukunft hat, hat auch das RNN noch Schwierigkeiten:

2.3 Das Vanishing Gradient Problem

Wir haben gesehen wie beim Training eines rekurrenten Netzes über die Bildung des Gradienten geschickt Gewichte gefunden werden, die Probleme lösen können

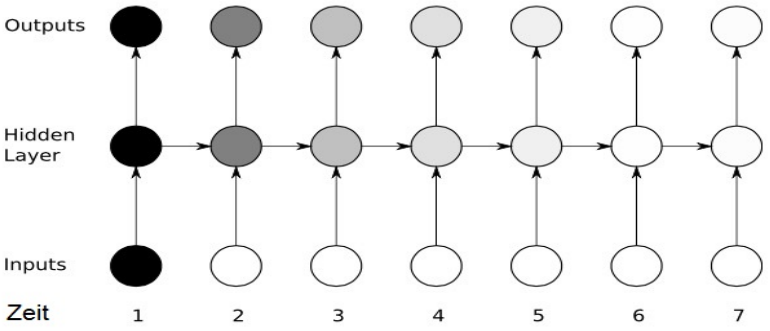

Abbildung 2.5: Das Problem vom verschwindendem Gradienten in einem rekurrenten Netz. Ein Fehler im Zeitschritt 1 erzeugt einen Gradienten, der aber auf die folgenden Zeitschritte immer weniger Einfluss hat. [?]

bei denen Abhängigkeiten über die Zeit auftreten. Erstrecken sich diese Abhängigkeiten aber über zu große Zeitraume, tritt das Problem vom verschwindendem Gradienten auf. Mit den üblichen Aktivierungsfunktionen wie z.B der *tanh*-Funktion treten Gradienten mit Betrag ≤ 1 auf, bei der *sigmoid*-Funktion sogar ≤ 0.25, die bei Backpropagation per Kettenregel miteinander multipliziert werden. Wird nun im Frontlayer ein Gradient berechnet, werden für n-Layer im Netz, also auch für ein rekurrentes Netz das um n Zeitschritte aufgefalltet wird, n dieser kleinen Zahlen miteinander multipliziert, wodurch der Gradient exponentiell schnell sinkt. Hat man hingegen eine Aktivierungsfunktionen mit Gradienten Betrag ≥ 1, wächst der Gradient exponentiell schnell, das Exploding Gradient Problem.

Es bleibt aber zu erwähnen, dass dieses Problem nur beim automatisierten Training eines rekurrenten Netzes auftritt. Wenn man die Gewichte vorgibt und Aktivierungsfunktionen geschickt wählt, kann ein rekurrentes Netz Zellen als Speicherzellen verwenden, z.B. indem es immer den Wert den es im letzten Zeitschritt als Output hatte als Input nimmt, indem es nur dem rekurrenten Gewicht zur eigenen Zelle den Wert 1 gibt. Sinnvoll ist die Speicherzelle natürlich nur, wenn sie auch befüllt werden kann, durch sogenannte Gate-Zellen. Eine Inputgate-zelle, welches reguliert ob, und mit was die Speicherzelle befüllt wird, eine Forgetgate-Zelle, welche die Speicherzelle wieder leeren kann und auch eine Outputgate-Zelle die steuert ob die Speicherzelle ihren Inhalt ausgibt, sind Strukturen die mit rekurrenten Neuronalen Netzen möglich sind und im kleinen Maß auch einfach zu erzeugen, wenn man die Gewichte manuell auswählen würde. Praktisch ist ein Neuronales Netz aber natürlich nur, wenn die Gewichte automatisch durch geschicktes Training gefunden werden, dass solche Strukturen aber zufällig erzeugt werden ist aber mit den bisher bekannten Trainingsmethoden quasi unmöglich. Wenn man jedoch eine geschickte Struktur fest vorgibt, funktioniert das Training und das Vanishing Gradient Problem wird überwunden. Eine mögliche solche Struktur sind die LSTM Zellen.

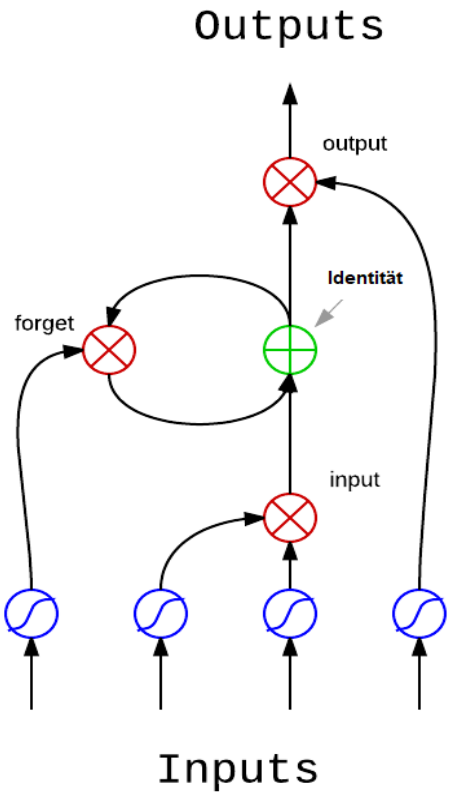

Abbildung 2.6: Ein Aufbau einer LSTM-Speicherzelle. 4 rekurrente Neuronen (blau), die die 3 multiplikativen Gates lenken (rot), die das Verhalten der Speicherzelle steuern. [?]

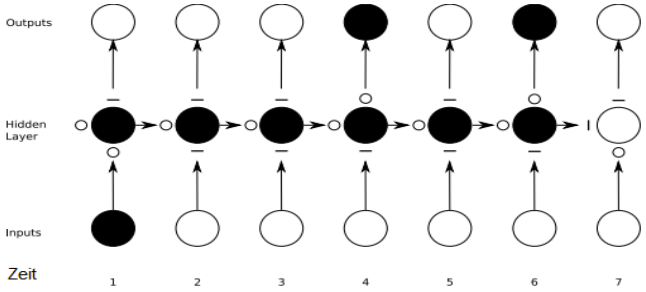

Abbildung 2.7: Der Inhalt der Zelle wird durch die verschiedenen Zeitschritte durch die Gates gelenkt. Kreis bedeutet offenes Gate, ein Querstrich stellt ein geschlossenes Gate dar. [?]

2.4 LSTM

LSTM steht für long short-term memory, zu deutsch, langes Kurzzeitsgedächtnis, ist eine Struktur die man in rekurrente neuronale Netze einbaut um Speicherzellen zu haben, die Werte stabil über beliebig viele Zeitschritte speichern können. Üblich ist hierfür ein Aufbau wie in Abbildung 2.6. 4 normale rekurrente Neuronen, die als Input also Aktivierungen des darunterliegenden Layers sowie die desselben Layers, aber des vorrigen Zeitschritts, bekommen und daraus ihre eigene Aktivierung berechnen, steuern das Verhalten der Speicherzelle. Eines steuert das Forget-Gate, also ob die Speicherzelle ihren gespeicherten Wert behalten soll. Im Normalzustand hat das Forgetgate den Wert 1, sodass der gespeicherte Wert multipliziert mit 1 der Wert selbst bleibt. Soll ein Wert vergessen, bzw. nichts gespeichert werden, muss das Forgetgate den Wert 0 annehmen. 2 Zellen steuern das Inputgate, eine gibt den Wert an, welcher gespeichert werden soll und die andere steuert mit einem Wert zwischen 0 und 1, ob die Zelle einen neuen Wert speichert. Es gibt Varianten, wo das Input- und das Forgetgate gekoppelt sind, sodass immer wenn ein neuer Wert gespeichert werden soll der alte gelöscht wird und umgekehrt, ist dies nicht der Fall, wird das Ergebnis des Inputgates auf den bisher gespeicherten Wert aufaddiert. Die 4. Zelle steuert das Outputgate, interaggiert also nicht mit dem Wert der Speicherzelle selbst, aber regelt ob das gespeicherte ausgegeben wird oder nicht. [?]

In Abbildung 2.7 sieht man, wie der Inhalt der Speicherzellen sauber durch die Gates gelenkt werden können. Das funktioniert im Forwardpass, aber auch genauso beim Training im Backwardpass mit dem Gradienten, da der Inhalt der Zelle per Identitätsfunktion durch die Zeit auf sich selbst abgebildet wird, und da der Gradient der Identität den Betrag = 1 hat, geht jetzt auch nichts mehr verloren. [?]. LSTM Netze werden wie rekurrente Netze trainiert, mit Backpropagation through time.

LSTM wird in aktuellen Produkten von bedeutenden Technologie Firmen wie zum Beispiel Apple [?], Amazon [?] und Google als Kernkomponente verwendet, von Google zum Beispiel für die Spracherkennung für den Smartphoneassistenten Allo [?].

Nun wollen wir in dieser Arbeit untersuchen, wie das LSTM in seinen Zellen mit Events umgeht. Für die Unterteilung durch Events gibt es aus der Psyhologie bereits

einen Ansatz, die Event Segmantation Theory (EST).

2.5 Event Segmentation Theory

Ein Weg etwas zu verstehen, ist es in kleinere Teile zu Unterteilen. Nach der Event Segmentation Theory (Zacks et al. 2007) ist das Unterteilen von kontinuirlicher Aktivität durch bedeutungsvolle Ereignisse eine Kernkompnente der Wahrnemung. [?] Dies ist sowohl ein Top-Down also auch ein Bottom-Up Vorgang, also sowohl das Wissen über die Pläne des beobachteten Akteurs als auch der direkte sensorische Input haben Einfluss auf diese Unterteilung. [?] Die Planung zum erreichen eines Ziels ist in Events unterteilt, z.B. wenn man jemandem zuschaut sein Zelt aufzubauen, erwartet man, wenn man den Vorgang kennt, dass dieser erst das Zelt aus der Verpackung holt, das Zelt ausbreitet, die Stangen zusammensteckt, usw.. Diese Unterteilung ist für uns ganz natürlich und dazu macht es keinen Unterschied, ob man sich zum Zusammenstecken der Stangen z.B. auf den Boden setzt, auch wenn das physich ein ganz anderer Vorgang ist. Die Bottom-Up Seite von EST ist greift auf ein Model zurück, bei dem unser Gehirn ständig Vorhersagen über die Umwelt macht und resultierene entdeckte Fehler verarbeitet [?]. Befindet sich der beobachtete Akteur in einem Event, werden sich dessen folgenden Bewegungen und Aktionen stetig an seine vergangenen Aktionen anschließen. Hat man hingegen eine hohe Unsicherheit in der Vorhersage der nächsten Aktionen, wird eine Eventunterteilung vorgenommen. Am Beispiel: Ist jemand gerade beschäftigt mit dem zusammenstecken der Zeltstangen, wird er das erfahrungsgemäß auch weitertun, ist er damit aber fertig, ist unklar ob er z.B. erst das Vorzelt aufbaut, uns als Beobachter um Hilfe bittet oder sich erstmal einen Kaffee holt. Entgegen dem alltäglichen Sprachgebrauch meint EST mit Ereignissen immer ganze Vorgänge, also z.B. das zusammenstecken der Zeltstangen, nicht aber das Beginnen oder das Fertigwerden, diese werden als Ereignisgrenzen bezeichnet. Ereignisse sind verschieden Granuliert und hierarchisch, so ist z.B. das zusammenstecken der Zeltstangen teil des Ereignisses des Zeltaufbauens welches wiederrum teil des Ereignisses Campingurlaub ist.

In Abbildung 2.8 sieht man, wie eine Unterteilung nach Bedeutung dem Verstehen hilft. EST sieht seine Gültigkeit sowohl beim beurteilen und vorhersagen der Aktionen anderer, als auch beim durchführen und planen der eigenen.

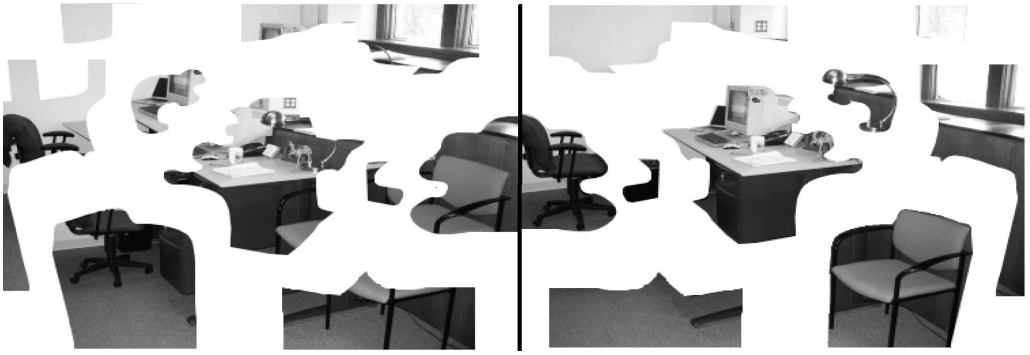

Abbildung 2.8: Ein Bild einer Büroszene in 2 verschiedenen Weisen zerschnitten. Für die linke Version benötigt man mehr Anstrengung um die Szene zu verstehen. Es veranschaulicht, wie eine Zerteilung nach Bedeutung die Wahrnehmung unterstützt. [?]

3 Implementierung

Nun da die Grundlagen gelegt sind können wir anfangen zu untersuchen wie ein LSTM Netz mit Events umgeht. Wir betrachten dazu das Bouncing Ball Szenario, da es klare Events hat und sich dadurch auch gut segmentieren lässt. Zur Implementation wurde JANNLab (Otte et al. 2013), ein Java Framework für Neuronale Netze verwendet [4]. Dieses Kapitel soll die im Zuge dieser Arbeit verwendeten Techniken erklären, die betrachteten Testfälle definieren und aufzeigen wie die im nächsten Kapitel beschriebenen Erkenntnisse gewonnen wurden, sowie welche Probleme sich dabei in den Weg gestellt haben.

3.1 Das Bouncing Ball Szenario

Das Bouncing Ball Szenario beschreibt einen Ball, der gleichförmig durch die Ebene gleitet bis er an entsprechenden Begrenzungen abprallt. Wir haben die 1- und die 2-dimensionale Version verwendet.

Im 1D Fall haben wir einen Ball ohne Ausdehnung (also eigentlich ein Punkt aber diese Unterscheidung ist für unsere Zwecke irrelevant) mit einer Position x_b und einer konstanten Geschwindigkeit v. Dieser springt zwischen den Grenzen $x_1 = -1$ und $x_2 = 1$ hin und her, also immer wenn die Position des Balls eine der Grenzen annimmt, wird die Geschwindigkeit invertiert $v := -v$. Also genau so wie man es sich vorstellt. Der 1D Fall unterteilt sich also in 2 Ereignisse, die "Bewegung nach linksünd die "Bewegung nach rechts", wenn man das Bouncen zwischen -1 und 1 als Bewegung auf einer horizontalen Linie nimmt.

Der 2D Fall funktioniert analog, der Ball hat eine Position (x_b/y_b), eine konstante Geschwindigkeit (v_x/v_y), welcher nun zwischen den 4 Grenzen $x_1 = -1$ und $x_2 = 1$, $y_1 = -1$ und $y_2 = 1$ hin und her springt. Nimmt nun eine Koordinate des Balls eine der Grenzen an wird die entsprechende Koordinate invertiert, also $(v_x/v_y) = (-v_x/v_y)$ beziehungsweise $(v_x/-v_y)$. Der 2D Fall unterteilt sich also in 4 Ereignisse, die "Bewegung in Richtung oben-rechts", etc. wenn man das Bouncen zwischen dem Quadrat von (-1/-1) und (1/1) im 2-dimensionalen als Bewegung in einer stehenden Ebene nimmt.

Das 1D-Szenario habe ich hauptsächlich verwendet um mich einzuarbeiten und zurechtzufinden, da es unter anderem mit weniger Neuronen erlernt werden kann und so die Aktivierungen der einzelnen Gates auch übersichtlicher sind. Der Grund warum das 1D-Szenario auch für diese Ausarbeitung wichtig ist, ist das unter Hinzunahme der Zeitachse aussagekräftige Diagramme erstellt werden können, anhand derer einige Trainingseffekte und Verläufe veranschaulicht werden können. Entsprechende Analysen konnten vom 2D-Fall auch gemacht werden, wenn man sah wie sich eine Linie oder ein Ball mit Verzögerung entsprechend über den Bildschirm bewegt, die resultierenden Bilder sind jedoch eher unübersichtlich. Das 2D Szenario hingegen

ist das für unsere Untersuchungen interessantere, da es mehr Events gibt, die auch in komplexernen Beziehungen zueinander stehen. In Abbildung 3.1 sieht man jeweils 2 Beispiele, zu dem genauen Aufbau des dazu benutzten LSTM Netzes kommen wir in der folgenden Sektion.

3.2 Parameter und Testfälle

Für unser LSTM-Netz haben sich folgende Parameter als sinnvoll erwiesen:

Aktivierungsfunktionen:
Zur Ansteuerung der Gates der LSTM-Zellen im Hiddenlayer werden Neuronen mit *Tangens hyperbolicus* als Aktivierungsfunktion verwendet. Für das Outputlayer wird zur Aktivierung die lineare Funktion verwendet, da die Position des Balles gleichverteilt Werte zwischen -1 und 1 annimmt, soll unser Netzoutput dasselbe tun. Anfangs war für das Outputlayer per Defaultoption die *sigmoid*-Funktion zur Aktivierung eingestellt, deren Bild aber nur von 0 bis 1 reicht. Das hatte natürlich nicht funktioniert und für Verwirrung gesorgt, warum denn nur die Hälfte des Problems erlernt wird.

Trainingsoptimierung:
Als Optimierungsmethode für das Traing wurde die Adaptive Moment Estimation Methode(ADAM), also eine adaptive Momentumsschätzung, verwendet. Diese baut die übliche Lernmethode mit Lernrate und Momentumrate dahingehend aus, dass diese durchgehend angepasst werden, um noch schneller zu genaueren Ergebnissen zu kommen. Hierzu wurden die Standard Parameter verwendet, also $\beta_1 = 0.9, \beta_2 = 0.99$ und $\epsilon = 10^{-8}$. Wobei β_1 und β_2 die Abklingraten des Einflusses der vergangenen Momenti auf den jetzigen Zeitschritt sind. Sind sie Nahe 1, klingt der Einfluss langsamer ab. ϵ ist ein glättender Term, der Division durch 0 verhindert. [?] Es wurde für Debugging-Zwecke an diesen Werten herumgespielt, dadurch wurden aber nie sichtlich bessere Ergebnisse erzielt.

Netzgröße:
Die passende Netzgröße ist fallabhängig, in Abbildung 3.1 z.B. wurden für die 1D Fälle 1-2-1 Netze verwindet (also 1 Inputneuron, 1 Hiddenlayer mit 2 Neuronen und 1 Outputneuron), für die 2D Fälle 2-4-2 Netze. An anderer Stelle werden andere Netzgrößen verwendet, dann wird das dort, falls relevant, auch nochmal aufgegriffen.

Außerdem:
Alle Layer sind mit Bias-Unit, um mit möglichst wenig Neuronen möglichst viel Funktionalität zu erreichen, damit deren Analyse wiederum möglichst ausgiebig verläuft. Peepholes [?] wurden getestet, hatten in diesem Beispiel keinen sichtbaren Effekt auf das Training und wurden dann weggelassen.

Außerdem werden verschiedene Testfälle mit verschiedenen Intentionen betrachtet, die ich an dieser Stelle vorgreifend definieren möchte. Es wurde mit fast allen Kombinationen experimentiert:

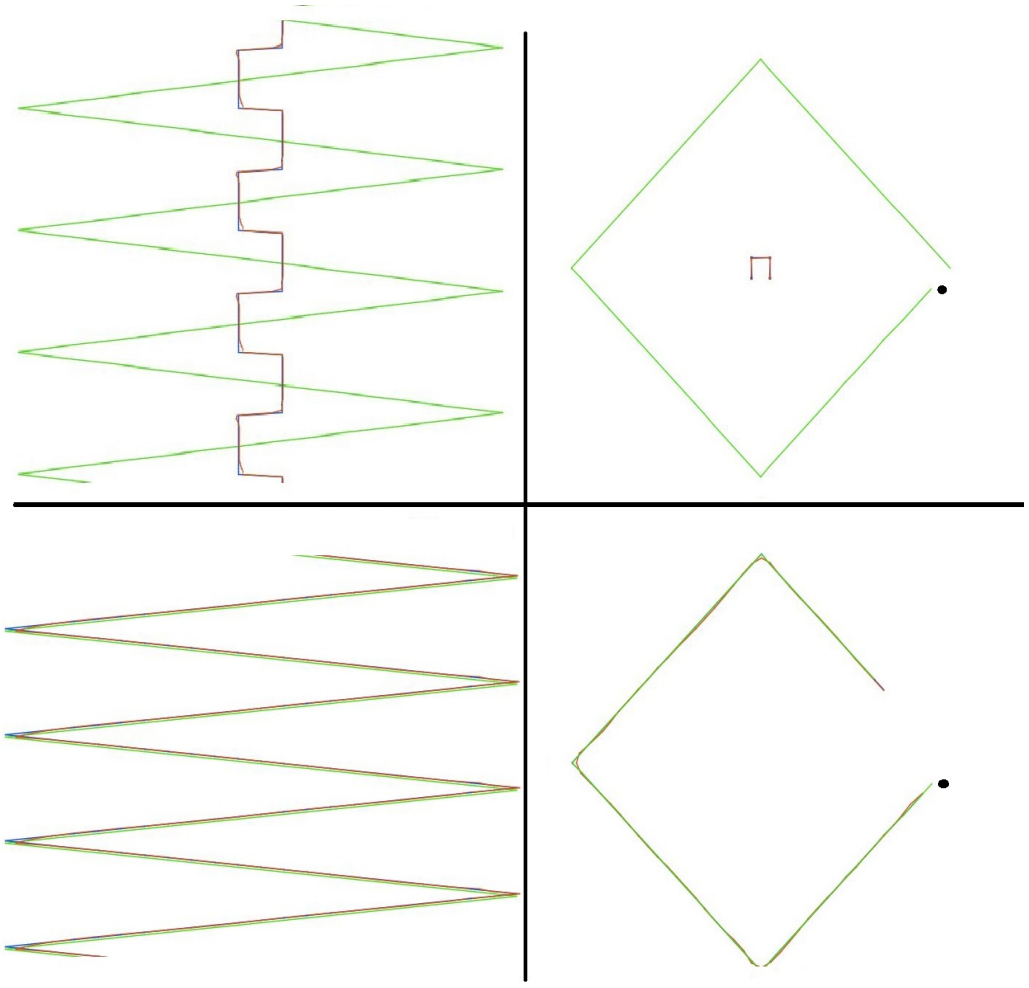

Abbildung 3.1: 4 Versionen des Bouncing Ball Szenario erlernt durch LSTM Netze. Zu sehen sind der Netzinput (grün), der Netzoutput (rot) und das Trainingsziel (blau). Links sieht man 2 Beispiele des 1D Falls, zur besseren Veranschaulichung sind die Daten nach den Zeitschritten vertikal versetzt. Rechts sieht man 2 Beispiele des 2D Falls, die Abbildung zeigt jeweils den Verlauf des Balls bis kurz vor dem vollenden einer Runde, um Überlagerungen in der Abbildung zu vermeiden. Der Startpunkt ist durch den schwarzen Punkt gekennzeichnet. In den jeweils oberen Fällen wurde als Netzinput die Position des Balles und als Trainingsziel die Geschwindigkeit des Balles für den nächsten Schritt. In den jeweils unteren Fällen wurde als Netzinput die Position des Balles und als Trainingsziel die vorauszuberechnende nächste Position des Balles.

Verschiedene Inputs:

- Position: Der Standard Input
- Noisy Position: Hier wird für den Input auf die Position des Balles noch ein zufälliger Wert zwischen -0.02 und 0.02 hinzu addiert, das Trainingsziel bleibt davon aber unbetroffen. Dieses "rauschenßoll das Netz dazu zwingen, eine innere Repräsentation von der Position des Balles zu haben, anstatt stur auf den Input einen konstanten Wert (die Geschwindigkeit) aufzuaddieren. Tatsächlich kann hier beobachtet werden, wie schon bei geringerer Epochenzahl das Bounce-Verhalten des Balles schneller erlernt wird.
- Null: Hier werden im 1. Schritt eine 1, dann aber nur noch 0-en in das Netz gespeist. Der Gedanke hierbei ist, das das Netz eine vom Input unabhängige Repräsentation des Balles hat, die sich auch als solche analysieren lassen kann. Im Standardfall mit fixer Startposition und Startgeschwindigkeit hat dies gut funktioniert, im zufälligen Fall natürlich nicht, das Netz kann ohne Input ja die anderen Position und Geschwindigkeit nicht kennen. Experimente mit einer "Lern-Phase", also das z.B. 100 Zeitschritte als Input die Position des Balles gegeben wird, bis das Netz ein Model von der Bewegung aufgebaut hat und ab dann den Input wegzulassen, schlugen fehl.
- Geschwindigkeit: Ein Kompromiss zwischen dem Netz direkt die Position zu verraten und ihm keine Information zu liefern. Wichtig ist zu beachten, dass die Geschwindigkeit im Input sich unterscheidet von der Geschwindigkeit im Output indem sie einen Zeitschritt nach hinten versetzt ist. Sonst müsste das Netz auf die Position wirklich nur die Geschwindigkeit aufaddieren und müsste sonst keine Repräsentation des ganzen haben. Die Hoffnung war so auch die Fälle mit zufälligen Geschwindigkeiten zu erlernen (die Startposition muss hierfür natürlich fix sein), aber trotzdem das Modell unabhängiger vom Input zu haben. Dieser Fall hätte eine Repräsentation der Position des Balles gefordert und auch ein sauberes Handling der Eventgrenzen, da einfach nur Schritte Zählen nicht gereicht hätte, da bei verschiedenen Geschwindigkeiten die Events nach unterschiedlich vielen Schritten auftreten. Am vielen Konjunktiv in diesem Abschnitt merkt man aber, das dieser Trainingsfall leider nicht geklappt hat und es hat sich auch nicht mehr aufgeklärt was am Training hätte geändert werden müssen, um diesen Fall zu erlernen. Es wurden verschiedene fixe Startpunkte getestet, es wurden nur kleine Varianzen in der Geschwindigkeit getestet, der Output, die nächste Position des Balles, hat sehr schnell sehr falsches Verhalten aufgezeigt, wie z.B. in Abbildung 3.2.

Verschiedene Trainingsziele:

- Nächste Position: Der Standard Output.

- Geschwindigkeit: Da die Geschwindigkeit bis auf Vorzeichen konstant ist, kann sich das Netz hier darauf konzentrieren zu klassifizieren, in welchem Ereignis, also Bewegungsabschnitt es sich befindet, also ob sich der Ball momentan z.B. nach links oben bewegt.
- Nächstes Event: Es wurde auch getestet wie sich das Netz verhält wenn es selbst vorhersagen soll, welches Event als nächstes ansteht. War erfolgreich auch bei zufälligen Startposition und Geschwindigkeiten, was aber nicht weiter Überraschend ist, das sich das LSTM Netz auf ein lineares Problem, welche Grenze als nächstes getroffen wird, trainieren lässt. Auch aus den Aktivierungen im Netz konnte nichts interessantes gefolgert werden, weswegen das nächste Event als Trainingsziel nicht mehr aufgegriffen wird.

Startposition und Geschwindigkeit:

- Feste Startposition und Geschwindigkeit: Der Standard Fall, leicht zu erlernen. Der Ball wird so gesendet, dass er jede Grenze in der Mitte trifft und die Flugbahn wieder ein Quadrat beschreibt, wie z.B. in Abbildung 3.1 rechts. Dadurch ist das Trainingsziel aber auch immer dieselbe Zahlenfolge, und kann (und wird auch) auf andere Weise gelernt als die Eventgrenzen sauber zu lernen. Das Netz kann eine periodische Struktur erlernen, die gelernt hat das z.B. bei einer Geschwindigkeit von 0.1 alle 20 Schritte eine Eventgrenze vorliegt und die Geschwindigkeit invertiert werden muss. Oder aus dieser periodischen Struktur werden direkt die Netzausgaben generiert und eine Repräsentation des Bewegung des Balles wird komplett Übergangen.

  Außerdem ist im 2D Version in diesem Fall die Ereignisabfolge immer dieselbe, was die Vorhersage des nächsten, bzw. die Einteilung des derzeitigen Ereignisses, nicht nur sehr vereinfacht, sondern auch entweder unabhängig von der derzeitigen Geschwindigkeit des Balles oder von der derzeitigen Position des Balles macht. Also um zu wissen, dass der nächste Bounce an der unteren Grenze erfolgt, genügt entweder das man sich gerade um Quadranten unten rechts befindet, oder das die Geschwindigkeit die Richtung nach unten-links hat. Im Allgemeinen ist dies aber nicht der Fall.
- Zufällige Startposition und Geschwindigkeit: Um zu umgehen, dass das Netz das Problem auf Basis einer inneren periodischen Struktur erlernt, werden mehrere Testsätze (Größenordnung 100 Stück) mit zufälligen Startpositionen und Geschwindigkeiten generiert. Das Netz wird über alle trainiert. So haben die verschiedenen Fälle ihre Ereignisgrenzen in verschiedenen Abständen und das Netz wird dazu gezwungen das Bounce Verhalten an den Eventgrenzen zu erlernen. Das Problem hierbei war aber das die Definition des Bounceverhalten unklar war. Gewünscht ist natürlich, dass z.B. die Eventgrenze x=1 erreicht, aber nicht überschritten wird, da an ihr abgeprallt werden soll. Durch die Diskretisierung unser Zeitschritte

und die zufällige Wahl von Position und Geschwindigkeit, wird die Grenze jedesmal überschritten und jedesmal um einen anderen Wert. Hat der Ball zum Beispiel die Position 0.95 und Geschwindigkeit 0.15, wird er er im nächsten Schritt die Position 1.1 haben, beim überschreiten der Grenze die Ereignisgrenze erreichen und umkehren. Den Bounce dann zu definieren, wenn im nächsten Schritt die Grenze überschritten werden würde, scheint noch willkürlicher, da dies manchmal bei 0.8, manchmal bei 0.99 ist. Die korrekte Berechnung, welche Position der Ball im kontinuierlichen Fall bis zum nächsten Zeitschritt erreicht hat, wäre die sinnvollste Definition. Also im obigen Beispiel mit der Position 0.95 und der Geschwindigkeit 0.15, bewegt der Ball sich 0.05 bis zur Grenze und die verbleibenden 0.10 wieder zurück, ist beim nächsten Zeitschritt also bei 0.9. Wurde getestet, konnte aber vom Netz nicht ohne weiteres erlernt werden. Es bleibt also bei der Definition, dass ein Bounce ausgeführt wird, wenn eine Grenze überschritten wird. Vermutlich leidet aber die Genauigkeit des Netzes beim Simulieren des Bounces darunter, dass der Bounce nicht an der Grene x=1, sondern in einem kompletten Bereich, abhängig von der Geschwindigkeit Varianz, zum Beispiel zwischen 1.0 und 1.999, ausgeführt wird. Daher wurde folgender Ansatz weiter verfolgt:

- Zufällige Auswahl aus vorgegebenen Startpositionen und Geschwindigkeiten: Ein Kompromiss aus den beiden anderen Varianten. Es werden geeignete Startpositionen und Geschwindigkeiten vorgegeben, sodass auch bei der diskreten Berechnung der Positionen zu den Zeitschritten, die Ereignisgrenzen immer exakt erreicht werden, anstatt sie zu überschreiten. Eine der Möglichkeiten und auch von uns verwendete ist hierzu, die Startposition $x_0 \in \{-0.9, -0.8, ..., 0.8, 0.9\}$ und die Startgeschwindigkeit $v_0 \in \{-0.1, -0.05, 0.05, 0.1\}$ zu wählen. Im 2D Fall natürlich jeweils 2 wählen, pro Koordinate eine wie z.B. in Abbildung 3.5. Außerdem wurde für den 2D Fall noch genau die Teilmenge betrachtet, deren Flugbahn wieder genau auf dem Standard Quadrat wie in Abbildung 3.1 rechts liegt, nur eben in beide Richtungen und in 2 verschiedenen Geschwindigkeiten durchlaufen.

  Nun hat man beim erstellen der Testsätze zwar, wenn man auf die Position des Balles 100 mal eine Geschwindigkeit auffadiert um jeweils die neue Position zu erhalten, ein Computernumerisches Problem, dass eventuell anstatt genau die Eventgrenze x=1 zu erreichen, ein Wert knapp darunter, z.B 0.9999 berechnet wird und man die Grenze wieder überschreiten würde. Ist man sich dessen aber bewusst geworden, kann man es leicht, durch das einbauen eines kleinen Thresholdes um die Ereignisgrenzen, umgehen.

Feedback:

Die Rück-Einspeisung (engl. Feedback) des Outputs y_t des Netzes als Input im nächsten Zeitschritt x_{t+1} hat mehrere Effekte auf Training und Verhalten des Netzes. Sinnvoll ist dies natürlich nur, wenn der Output des Netzes die Informationen für einen geeigneten Input enthält. Genutzt haben wir dies für

die Position des Balles als Input (bzw. auch Noisy Position) und für den Output entweder die nächste Position des Balles, dann kann der Output direkt als Input für den nächsten Zeitschritt verwendet werden $x_{t+1} = y_t$, oder mit Geschwindigkeit im nächsten Zeitschritt als Output, dann wird die bestimmte Geschwindigkeit auf die letzte Position aufaddiert und als Input genommen $x_{t+1} = x_t + y_t$. Hierzu werden in den ersten 100 Zeitschritten wie gewohnt der Input aus dem jeweiligen Testsatz genommen, sodass sich das Netz gut auf den jeweiligen Fall einstellen kann. Nach dieser Lernphase beginnt man mit der Rück-Einspeisung des Outputs. Die Feedback-Methode wurde wie folgt mit den beschriebenen Effekten eingesetzt:

- Genaueres Training:
  Der Effekt der in Abbildung 3.2 beschrieben ist, das wenn das Netz die Position des Balles selbst lenkt kleine Abweichungen große Fehler erzeugen, kann sowohl ein Nachteil als auch für uns jetzt ein bewusst genutzter Vorteil sein. Wird Feedback schon beim Training verwendet, wird das Netz direkt gezwungen genaue Ausgaben zu erzeugen. Das Trainingsziel kann so deutlich schneller erreicht werden.
- Überprüfen eines bereits trainierten Netzes:
  Hat man ein Netz trainiert und wendet dann im Testfall die Feedback Methode an, sieht man, wie z.B. in Abbildung 3.3, recht schnell wie gut das Training ist und wie sehr das Problem wirklich erlernt oder nur eine passende Annäherung gefunden wurde.
- Konsistenz mit innerer Repräsentation:
  Hat man ein Netz welches intern eine Repräsentation des Modells hat, z.B. die Position des Balles selbst speichert und diese abhängig vom Input aktuallisiert, kann es zu einem Zitter-Effekt kommen, wie man ihn in Abbildung 3.5 sieht. Eine Abweichung in der gespeicherten Position des Balles im Gegensatz zur tatsächlichen Position die per Input reinkommt, wurde erlernt wieder korrigiert zu werden. Gehen die Abweichungen mal in beide Richtungen, sieht man dieses Zittern. Nutzt man hingegen die Feedback Methode werden diese Korrekturen nicht vorgenommen, da der Input selbst auch vom Netz generiert wurde. Dies ist je nach Anwendung natürlich ein Nachteil, da Abweichungen sich aufaddieren und über die Zeit das Endergebnis ein ganz anderes sein wird. Andererseits, wenn es einem wie in unserem Fall nicht um die Bestimmung der Endposition des Balles nach 300 Schritten, sondern um das Verhalten auf dem Weg dorthin geht, erhält man mit der Feedback Methode zueinander konsistente Zwischenergebnisse, wie z.B. in Abbildung 3.5 zu sehen.

Feedback funktioniert gut für die 1D Fälle und für die 2D Fälle, wo die Laufbahn des Balles auf dem Hauptquadrat bleibt, leider aber nicht für die Fälle wo der Ball sich frei bewegt. Es wurden mit verschieden langen Findungsphasen, also Phasen in denen der Input noch aus dem Testset kommt experimentiert, auch zum Beispiel 90% der gesammten Simulationslänge. Es wurde getestet wenn man das Feedback erst nach einer hohen Epochenzahl an das Netz lang-

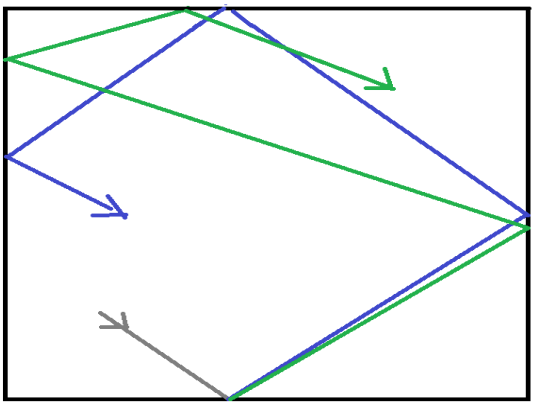

Abbildung 3.2: Chaostheorie im Bouncing Ball Szenario: Kleine Abweichungen im Bounce Verhalten haben große Auswirkungen auf den weiteren Flug des Balls. Dies ist der Grund für Probleme beim Offline Training von Fällen, wo das Netz die Position des Balles selbst lenkt, da sehr gute Annäherungen des korrekten Bounce Verhaltens trotzdem große Fehler über Zeit erzeugen, werden also nicht als gute Annäherungen erkannt.

sam heranführt, also das Netz erst ohne Feedback trainiert und erst wenn das Netz gute Ausgaben liefert, dann erst nur die letzten Zeitschritte mit Feedback simuliert und so das Netz langsam daran heranführt, auch ohne Erfolg. Es wurde mit einem Mittelwert aus dem vorgegeben Input und dem Input aus Feedback, also dem dem eigenen vergangenen Output, experimentiert, und auch mit verschiedenen Anteilen hier, auch einem der sich über höhere Epochenzahl anpasst, auch ohne Erfolg. Der Effekt aus Abbildung 3.2 ist wohl zu stark, das auch gute Annäherungen des Bounce-Verhaltens doch über die Zeit zu großen Abweichungen führen und diese dann nicht als gute Annäherungen erkannt werden.

Das Netz Online zu trainieren wäre für diesen speziellen Fall wohl die Lösung gewesen, d.h. den Input und vorallem das Trainingsziel nicht in Testsätzen im vorneherein zu berechnen und zu definieren, sondern auch das Trainingsziel in jedem Zeitschritt agil zu berechnen, also die aktuelle Geschwindigkeit aus den beiden vorherigen Positionen zu bestimmen, sodass Abweichungen in der Richtung auch vom Trainingsziel berücksichtigt werden. Ein solches Online-Training hätte aber eine komplette Umstrukturierung der bestehenden Implementierung vorausgesetzt, was an dieser Stelle nicht durchgeführt wurde, da die Tatsache ob das LSTM Netz mit Feedback trainiert wurde oder nicht, wohl keinen Einfluss darauf hat wie sich die LSTM Zellen im Bezug auf die Eventgrenzen verhalten, was der eigentliche Kern dieser Arbeit ist.

Abbildung 3.3: TODO 1D Fall trainiert mit einem 1-1-1 Netz, also nur einem Neuron im Hiddenlayer. Trainiert ohne Feedback, zur Überprüfung des Trainings dann hier im Bild dann verwendet. Man sieht, dass das Netz das Bounce Verhalten garnicht erlernt hat, sondern nur die kontinuierliche Bewegung. Außerdem ist hier der Nutzen von Feedback als Testmethode des Trainings gut veranschaulicht.

Abbildung 3.4: TODO 1D Fall genau gleich trainiert wie in Abbildung 3.3, nur mit mehr Epochen. Das Netz hat seine Methode komplett gewechselt. eine periodische Lösung gefunden, anhand der aktivierung sieht man mit Events nix am Hut. Forgetgate und Outputgate passiert nix, infrastruktur der lstm zelle wird nicht genutzt. lediglich im cellstate werden die werte aufaddiert und bei überschreiten eines thresholdes wird der sprung gemacht. sieht nach event aus ist aber nur eine periodische verkettung.

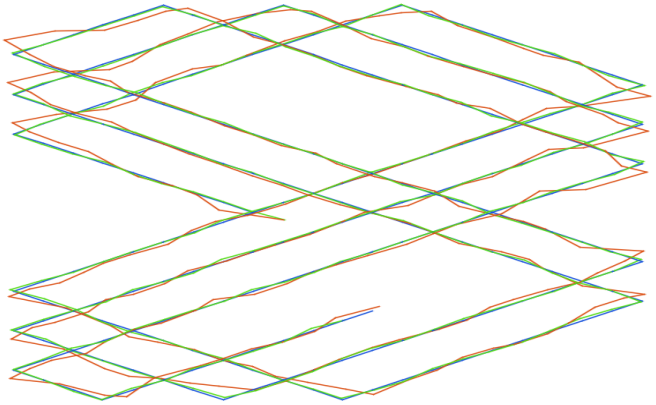

Abbildung 3.5: 2D Fall mit zufällig gewählter Startposition und Geschwindigkeit. Auch wenn das Ergebnis sehr nah am Trainingsziel ist, sieht man hier sehr deutlich das beschriebene Zittern, das sich durch anwenden der Feedback-Methode vermeiden lässt.

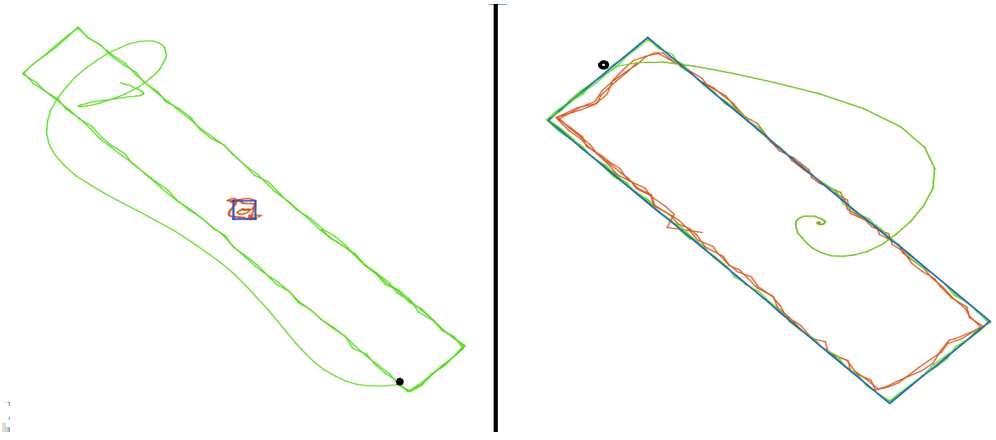

Abbildung 3.6: 2 Fälle trainiert mit je einem 2-16-2 Netz mit Feedback. Links als Trainingsziel die Geschwindigkeit, Rechts als Trainingsziel die nächste Position. Feedback setzt nach der halben Simulationslänge ein, hier mit schwarzem Punkt markiert. Auch wenn das Ergebnis offensichtlich weit vom Trainingsziel entfernt ist, sieht man die glättende Wirkung der Feedback Methode.

Literaturverzeichnis

[1] en.wikipedia.org/wiki/List_of_animals_by_number_of_neurons. abgerufen am 13.06.2017.

[2] S. U. Fernando Sancho Caparrini. Artificial neural networks. cs.us.es/~fsancho/?e=135, 2017. abgerufen am 13.06.2017.

[3] S. Haykin. *Neural Networks: A Comprehensive Foundation*. 1994.

[4] S. Otte, D. Krechel, and M. Liwicki. Jannlab neural network framework for java. In *Poster Proceedings Conference MLDM 2013*, pages 39–46, New York, USA, 2013. ibai-publishing.

[5] P. S. University. Neurons. online.science.psu.edu/bisc004_activewd001/node/1907, 2017. abgerufen am 10.05.2017.