Notes

- ☒ check uni requirements - with fontsize always talk with gutachter about it (like in my expose is okay)
- ☒ check if I can convert notebook to markdown and markdown than to pdf like i did in expose
- ☒ create header grundlagen

```
"""
# 1. Abstract
# 2. Einführung
    2.1 Motivation (bspw. Anwendungen von NER),
    2.2 Problem
    2.3 Ziele (bspw. Überblick über bestehende Ansätze, Analyse und Vergleich der drei gängi
    2.4 Struktur der Arbeit
# 3. Grundlagen
    3.1 Neuronale Netzwerke -
    3.2 Backpropagation -
    3.3 Deep Learning -
    3.3 NLP (Erklärung des gesamten Prozess von Wort zu Vektor und die verschiedenen Bereich
    3.4 NER -
    3.5 Labelling -
    3.7 RNN
    3.8 LSTM
    3.8 Transformer
    3.6 Seq2Seq
    3.9 Fine tuning
# 4. Tools
    4.2 PyTorch
    4.1 Huggingface
    ...
    ...
# 5. Methode
    5.1 Training mit schwach gelabelten Datensatz
    5.2 Training mit selbst gelabelten Datensatz
    5.3 Generatives Modell
# 6. Design der Experimente (inkl. Modell/e, Daten)
    6.1 Training mit schwach gelabelten Datensatz
    6.2 Training mit selbst gelabelten Datensatz
    6.3 Generatives Modell
# 7. Ergebnisse der Experimente
    7.1 Training mit schwach gelabelten Datensatz
    7.2 Training mit selbst gelabelten Datensatz
    7.3 Generatives Modell
# 8. Diskussion (inkl. wichtiger Ergebnisse, Einschränkungen und zukünftiger Arbeiten, Impli
# 9. Zusammenfassung
```

```
Notes:
- todo change "Entitäten" to "Eigennamen"

"""
```

Einführung

Named Entity Recognition (NER) ist ein weitverbreiteter Ansatz für die Analyse von Textblöcken und das Auffinden und Klassifizieren vordefinierter Eigennamen wie Standort, Unternehmen oder Personennamen. Da die Arbeit im zusammenarbeit mit ML6 angefertigt wurde - ein Unternehmen, welches sich darauf spezialisiert hat spezifische Lösungen für machine learning Probleme zu entwickeln - konnte der Anwendungsfall von einen der im Unternehmen ablaufenden Projekte abgeleitet werden. Das Unternehmen hat die Aufgabe Such...

Grundlagen

Die Arbeit basiert vor allem auf Transformer Modelle. Für das erstellen von Transformer Modelle bedarf es wissen verschiedener Grundlagen. Folgend werden Grundlagen Technologien beschrieben.

Neuronale Netze

Neurale Netzwerke sind Algorithmen die dafür entwickelt wurden die Funktionen eines Gehirns nachzubilden. Der erste Algorithmus der dazu entwickelt wurde ist das McCulloch-Pitts Neuronen Model. Dabei handelt sich um folgende Funktion:

$$y = 1 \text{ wenn } w_1 \cdot x_1 + w_2 \cdot x_2 + \cdots + w_n \cdot x_n \geq \theta$$

$$y = 0 \text{ sonst}$$

y = Ausgabe des Neurons

x_n = Eingabe des Neurons

w_n = Gewichte des Neurons

θ = Schwellenwert

Die Funktion lässt sich bildlich darstellen:

Der nächste Schritt in der Historie neuronaler Netzwerke war die Entwicklung des Perceptrons [todo wer und wann]. Der Aufbau ist ähnlich zu dem des MP Neuronen Model unterscheidet sich aber darin, dass der Schellenwert kontinuierlich ist. Das heißt ein nach dem Input von Eingaben Werten kann der Output zum Beispiel 0.7 annehmen wohin bei einem linearen Schwellenwert das Ergebnis

entweder 1 oder 0 sein kann. Anstatt nur die Schwellenwert Funktion kann das Perzeptron auch andere Aktivierungs funktionen annehmen [figure].

Folgend eine Tabelle, welche die Unterschiede aufzeigt:

Beschreibung	MP Neuron	Perzeptron
Typ des Modells	Binär	Linear
Schwellenwert	Statisch	Anpassbar während des Trainings
Ausgabe	Binär (1 oder 0)	Kontinuierliche Werte

[todo multi layer perzeptron]

Backpropagation

Backpropagation ist eine Methode welche in künstlichen Neuronalen netzwerken benutzt wird um die Gewichte von einem neuronalen Netzwerk anzupassen um vorhersagen basierend auf einem Datenset liefern zu können. Der Algorithmus betrachtet zuerst die Gewichte an der Ausgabeschicht und geht von dort aus bis hin zur Eingabeschicht.

Angenommen man hat zwei Neuronen x_1 und x_2 in der Eingabeschicht, zwei Neuronen h_1 und h_2 in der versteckten Schicht, y als Ausgangsneuron und die Gewichte $w1, w2, w3, w4, w5$ und $w6$ zwischen den Neuronen.

Dabei berechnet man die Ausgabe $\hat{y}$ folgendermaßen:

$\hat{y} = f(w1 * x1 + w2 * x2 + w3 * h1 + w4 * h2)$

f ist die Aktivierungsfunktion.

Um die Werte der einzelnen Gewichte so anzupassen genauere Ergebnisse zu erhalten verwendet man Backpropagation. Bei einem Durchlauf kommt es zu dem Ergebnis $\hat{y}$. Das ist die tatsächliche Ausgabe eines neuronalen Netzwerkes nach einem Durchlauf der Daten. Die tatsächliche Ausgabe wird mit dem Wert der erwarteten Ausgabe verglichen. Man berechnet den Fehler δ dieser zwei Werte.

$\delta = (y - \hat{y})^2$

Für den Fehler ist es möglich verschiedene mathematische Funktionen zu benutzen. Eine übliche Funktion ist die der mittleren quadratischen Abweichung.

$MSE = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y_i})^2$

Die Funktion quadriert jedes einzelne Fehlersignal und summt am Ende alles zusammen und teilt es durch die Anzahl der Datenpunkte. Das Ergebnis ist der durchschnittliche Fehler aller quadrierten Datenpunkte. Es wird quadriert, weil die Ergebnisse sich dadurch leichter weiterverarbeiten lassen.

Nachdem das Fehler signal berechnet wurde besteht der nächste Schritt darin diesen Fehler rückwärts durch das Netzwerk zurück zu propagieren, um damit die Gewichte $w1, w2, w3, w4, w5$ und $w6$ zu erneuern.

Zuerst berechnet man den Gradient ∇ von der Verlustfunktion [todo define Verlustfunktion is the loss function == error signal?] mit respekt zu jedem einzelnen Gewicht im Netzwerk. Das passiert mit der Kettenregel von der Infinitesimalrechnung. Diese macht es möglich die Ableitung von der Verlustfunktion mit Respekt zu dem Gewicht zu berechnen.

Mit der Verlustfunktion MSE würde der Gradient für das erste Gewicht so aussehen:

$$\frac{\partial MSE}{\partial w1} = \frac{2}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i) \cdot \frac{\partial \hat{y}_i}{\partial w1}$$

$$\frac{\partial \hat{y}_i}{\partial w1} = \frac{\partial f(w1*x1+w2*x2+w3*h1+w4*h2)}{\partial w1}$$

$$\frac{\partial f(w1*x1+w2*x2+w3*h1+w4*h2)}{\partial w1} = x_1 \cdot \frac{\partial f}{\partial w1}$$

$$\frac{\partial MSE}{\partial w1} = \frac{2}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i) \cdot x_1 \cdot \frac{\partial f}{\partial w1}$$

Dieser Gradient wird nun für jedes Gewicht berechnet. Eine Formel, die die Gewichte erneuert nennt man optimierer. Es gibt verschiedene Optimierer. Der folgende ist der SGD [todo deutsch ganz]. Der SGD angewandt auf das erste Gewicht sieht so aus:

$$w1' = w1 - \alpha \cdot \frac{\partial MSE}{\partial w1}$$

dabei ist α die Lernrate. Die Lernrate ist ein Hyperparameter der die Schrittweite bestimmt in welcher Größe Gewichte erneurte werden. Bei einem kleineren α sind die erneuerungen kleiner, aber genauer und bei einem größeren α Wert sind die erneuerungen größer aber ungenauer. Hyperparameter nennt man die Parameter die vor einem Training eingestellt werden und nicht lernbar sind.

Der SGD Optimierer wird für jedes Gewicht angewandt.

$$w2' = w2 - \alpha \cdot \frac{\partial MSE}{\partial w2}$$

$$w3' = w3 - \alpha \cdot \frac{\partial MSE}{\partial w3}$$

$$w4' = w4 - \alpha \cdot \frac{\partial MSE}{\partial w4}$$

$$w5' = w5 - \alpha \cdot \frac{\partial MSE}{\partial w5}$$

$$w6' = w6 - \alpha \cdot \frac{\partial MSE}{\partial w6}$$

Sobald jedes Gewicht neu berechnet wurde das Netzwerk hat eine Epoche des Trainings vervollständigt. Dieser Prozess kann dann in weiteren Epochen wiederholt werden bis die Leistung des Netzwerkes zufriedenstellend ist.

Deep Learning

Deep Learning ist die weiterentwicklung der vorherig beschriebenen einfachen neuralen netzwerken und ein Teilbereich des maschinellen lernens.

[todo bild mit einordnung verschiedener Bereiche]

Deep Learning wurde durch die 2010er bekannt. Zu dieser Zeit kam es zur Verfügbarkeit von großen Datensets und Fortschritte in der Rechenleistung. In 2012 ein Deep Learning Model, welches von Google entwickelt wurde [todo referenz] erzielte einen Durchbruch in Bilderkennung und outperformte alle vorherigen Models auf dem ImageNet Datenset. Diese Leistung förderte ein breiteres Feld an Interesse für Deep Learning und sorgte für Fortschritt in verschiedensten Bereichen von Anwendungen. Deep Learning Algorithmen können unter anderem für Bild und Spracherkennung, NLP und das spielen von Spielen wie Schach und Go benutzt werden.

Es gibt verschiedene Architekturen, die für verschiedene Arten des lernens von Daten eingsetzt werden. Beispiele für Architekturen neuronaler Netzwerke sind auf folgender Grafik gut einsehbar.

[link to image from https://www.asimovinstitute.org/neural-network-zoo/]

Anstatt von einer Schicht, welche die Eingabewerte von Neuronen verarbeitet gibt es bei einem Deep Learning Modell mehrere Schichten. Die genaue Anzahl an Layern, die benötigt werden um ein Modell ein Deep Learning werden zu lassen ist nicht definiert. Der Begriff Deep Learning ist also ein eher allgemeiner Begriff der Modelle beschreibt die mehrere Schichten enthalten. Eine Schicht besteht jeweils aus Eingabewerten, einer Funktion welche die Eingabewerte verarbeitet und Ausgabewerte erzeugt.

[todo bild mit deep learning model mit mehreren Schichten]

NLP

Natural language processsing (NLP) ist ein Teilbereich der Linguistik und des maschinellen lernens. Das verarbeiten von menschlicher Sprache findet eine breite Anwendung in verschiedensten Bereichen. Beispiele sind Entitäten Erkennung, Maschinenübersetzung, Spracherkennung, Sentiment Analyse. Dadurch, die sinnvolle weiterverarbeitung von Sprache eine komplexe Aufgabe ist, weil Wörter in verschiedene Kontexten zum Beispiel unterschiedlich Deutung besitzen können muss ein Deep Learning Modell mit viel Daten trainiert werden um den verschiedenen Kontext zu erkennen.

Beispiel für unterschiedliche Wortbedeutung in unterschiedlichen Zusammenhängen:

Neben der Kirche befindet sich eine Bank.

Sprache muss zuerst auf ein für den Computer verständliches Medium reduziert werden, um damit Algorithmen entwickeln zu können.

Tokenization Tokenisierung ist der Prozess einen Text in kleinere Einheiten die man "Tokens" nennt zu teilen. Diese Tokens können Wörter, Satzzeichen oder andere Stücke von Text sein. Das hängt von der Aufgabe ab.

Tokenisierung ist ein wichtiger Schritt im natural language processing und ist oft der erste Schritt in einer NLP pipeline. Es wird benutzt um Text in kleinere Einheiten aufzuteilen, um somit das analysieren und weiterverarbeiten des Textes zu vereinfachen. Mit den kleineren Einheiten wird es einfacher Muster zu erkennen, Bedeutung zu extrahieren oder andere Operationen, die man auf dem Text ausführen möchte.

Es gibt verschiedene Arten der Tokenisierung. Zum Beispiel gibt es Wort tokenisierung, Satz tokenisierung und das tokenisierung von einzelnen Satzzeichen. Welche der Beispiele sich am besten eignet hängt von der jeweiligen Aufgabe ab. Word tokenisierung eignet sich zum Beispiel für Aufgaben wie Textklassifizierung oder Übersetzung. Satzzeichen tokenisierung könnte für das erkennen von handgeschriebener Schrift besser sein.

Beispiele:

"Der Hund bellt."

Tokenisiert mit Wort tokenisierung:

"Der"

"Hund"

"bellt."

Satz tokenisierung:

"Der Hund bellt."

Satzzeichen tokenisierung:

"D"

"e"

"r"

" "

"H"

"u"

"n"

"d"

" "

"b"

"e"

“l”

“l”

“t”

“.”

Vectorization Vektorisierung beschreibt den Prozess Text in numerische Vektoren umzuwandeln. Diese Umwandlung kann dann als Input für Algorithmen des maschinellen lernens benutzt werden. Es gibt verschiedene Wege für das vektorisieren von Daten und hängt davon ab in welchem Kontext man vektorisierung benutzen möchte.

One-Hot-Codierung ist eine Methode in der jedes Wort mit einem binären Vektor dargestellt wird. Mit einer “1”, wenn die Position dem Wort entspricht und sonst “0”. Diese Methode kann großes Vokabular entstehen lassen. Ein Vokabular ist die Anzahl der einzigartigen Worte in einem Text.

Beispiel:

“Der Hund bellt.”

“Der” –> [1, 0, 0]

“Hund” –> [0, 1, 0]

“bellt.” –> [0, 0, 1]

Worteinbettungen werden auf vielen Textdaten trainiert und die Vektoren für jedes Wort werden so gelernt, dass der Vektor den Kontext wiederspiegelt in welchem das Wort auftritt. Wörter die in einem ähnlichen Kontext auftreten tendieren dazu ähnliche Vektoren zu haben. Wörter mit unterschiedlichen Kontexten dagegen haben unterschiedlichere Vektoren.

[todo MAYBE ADD: There are several different methods for learning word embeddings, including word2vec and GloVe. These methods typically involve training a neural network to predict the context of a target word based on the context of surrounding words, using a large corpus of text as input. The resulting vectors for each word can then be used in downstream NLP tasks.]

[todo take graphic from matthias?/add graphic like this https://towardsdatascience.com/creating-word-embeddings-coding-the-word2vec-algorithm-in-python-using-deep-learning-b337d0ba17a8]

NER

Named entity recognition (NER) ist eine NLP Aufgabe mit der man Entitäten in einem Text, wie Personen, Organisationen und Orte identifizieren kann. Das macht NER nützlich für Applikationen wie das beantworten von Fragen, Informationsextrahierung und Dokument Zusammenfassungen.

[todo image of ner demonstration]

Es gibt verschiedene Ansätze für die Durchführung von NER. Es gibt Regelbasierte Systeme, Systeme basierend auf maschinellem lernen und Hybrid Systeme. Regelbasierte Systeme benutzen definierte Regeln für das identifizieren von Entitäten in Text. Systeme basierend auf maschinellem lernen das identifizieren von Entitäten anhand von gelabelten Trainingsdaten. Hybride Systeme kombinieren beide Vorgehensweisen.

Um NER durchzuführen tokenisiert man den Text zuerst zu individuellen Wörtern und dann benutzt man Techniken wie POS Tags (Wortart tagging [todo find out whats correct]) oder dependency parsing [todo find out if Abhängigkeitsanalyse is correct word]. Die Ausgabe von NER ist eine Liste mit Entitäten und den dazugehörigen Attributen.

Wortart tagging

Mit Wortart tagging identifiziert man die grammatische Rolle eines Wortes in einem Satz. Die Labels geben die Wortart an wie Verb, Adjektiv und Subjektiv.

Part-of-speech (POS) tagging is a natural language processing task that involves identifying the grammatical role of each word in a sentence. POS tags are labels that indicate the part of speech of a word, such as noun, verb, adjective, and so on.

Mit dem Satz “Der Hund bellte in Venedig.” ergibt das Wortart tagging folgendes Resultat:

“Der” (ART) für Artikel “Hund” (SUB) für Substantiv “bellte” (VER) für Verb “in” (PRÄ) für Präposition “Venedig” (SUB) für Substantiv

Mit den Wortart Tags kann man “Hund” und “Venedig” als Eigennamen identifizieren. Die Methode ist hilfreich für NER, weil Eigennamen oft auch Substantive sind und daher kann man mit Wortart tagging helfen Eigennamen genauer zu identifizieren.

Abhängigkeitsanalyse

Bei der Abhängigkeitsanalyse analysiert man die grammatikalische Struktur von einem Satz und bestimmt seine Abhängigkeiten zwischen seinen Worten. Zusammen mit NER kann es helfen besser Eigennamen in einem Text zu erkennen.

Jedes Wort von einem Satz wird als Knoten in einer Baumstruktur repräsentiert. Die Baumstruktur nennt man Abhängigkeitsanalysebaum. Die Abhängigkeiten zwischen den Wörtern sind repräsentiert als Kanten welche die Knoten verbinden. Die Wurzel des Baumes stellt das Hauptverb da. Die anderen Knoten sind unter andere Verben, Adjektive und Substantive.

Mit dem Satz “Der Hund bellte in Venedig.” ergibt die Abhängigkeitsanalyse folgendes Resultat:

```
bellte
```

Dabei ist “bellte” das Hauptverb, “Der Hund” das Subjekt und “in Venedig” das Adverb. Beim analysieren von den Abhängigkeiten kann man “Hund” und “Venedig” als Eigennamen identifizieren.

Es gibt verschiedene Bibliotheken, welche man für NER benutzen kann - Stanford NER, spaCy, NLTK und HuggingFace [todo links].

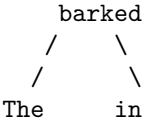
[todo MAYBE ADD: Named entity recognition (NER) is a task in natural language processing (NLP) that involves identifying and classifying named entities in text. NER has a long history, with early research dating back to the 1960s and 1970s.

One of the earliest NER systems was developed by the US Defense Advanced Research Projects Agency (DARPA) in the late 1970s, as part of the Message Understanding Conference (MUC). MUC was a series of evaluations that aimed to develop NLP systems that could understand and extract information from written text. The MUC evaluations focused on named entity recognition, along with other tasks such as coreference resolution and document summarization.

In the 1990s, NER became more widely used in a variety of applications, including information extraction and question answering. A number of open-source NER tools and libraries were developed during this time, including the Natural Language Toolkit (NLTK) and the Stanford Named Entity Recognizer (Stanford NER).

In the 2000s, NER continued to be an active area of research, with a focus on developing machine learning-based approaches to the task. One of the main challenges in NER is the large number of named entity types and the difficulty of accurately classifying them. In recent years, there has been a trend towards developing hybrid NER systems that combine rule-based and machine learning-based approaches.

Today, NER is widely used in a variety of applications, including information extraction, question answering, and document summarization. It is a key component of many natural language processing systems and is an active area of research and development.]



```
\
  Venice
```

Labelling

Labelling wird der Prozess genannt bei dem man einem Datenpunkt eine Klasse zuordnet. Labels sind da einen Datenpunkt mit einer eindeutigen Klasse zu verweisen. Bei dem erkennen von Eigennamen ist zu jedem Wort ein Label dazugehörig, welches eine Aussage über das Attribut, um welches es sich handelt macht. Labels werden benutzt um eine Model des maschinellen lernens zu trainieren, um dann Vorhersagen über neue nicht vorher gesehenen Daten treffen zu können.

Es gibt verschieden Möglichkeiten Daten zu labeln. Man kann Menschen manuell labeln lassen. ImageNet ist ein Beispiel für ein Datenset was zum Teil von Menschen gelabelt wurde. Der Vorteil ist, dass man eine hohe Qualität erwarten kann, aber im Vergleich zu anderen Techniken sehr zeitintensiv sein kann.

Eine andere Methode ist die des auto labeling. Beispiele sind Regel basierte Systeme oder aktiv lernende Algorithmen. Die Vorteile dabei sind, dass das labeln schneller geht aber dafür nicht so genau sein kann.

Regel basierte Systeme

Zuerst muss man sich manuell ein Datenset mit Labeln erzeugen. Zum Beispiel würde "Angela Merkel wurde 1954 in Hamburg geboren." die gelabelten Eigennamen "Angela Merkel" und "Hamburg" enthalten. Danach trainiert man das Datenset mit einem überwachten Algorithmus für maschinelles Lernen. Das Ziel des Models dabei ist die Muster der Entitäten im Text zu erkennen.
Das Model kann als eine Funktion f dargestellt werden, welche eine Eingabe x (ein Text Dokument) nimmt und eine Ausgabe y (die Labels für die bestimmten Eigennamen) erzeugt. Die Funktion f lernt mit einem Trainingsalgorithmus von dem gelabelten Datenset. Wenn das Model trainiert ist kann es für das bestimmen von neuen Labels bei eingebenen Text Dokumenten mit Eigennamen bestimmen. Zum Beispiel sollte das Model bei der Eingabe von "Wolfgang Schäuble wurde 1942 in Freiburg geboren." die Eigennamen "Wolfang Schäuble" und "Freiburg" vorhersagen. Die erstellten Labels werden benutzt um neue Regeln zu erzeugen oder bestehende zu erweitern.

Aktiv lernende Algorithmen

Gestarted wird mit einem überwachten Algorithmus für maschinelles Lernen, um von einem kleinen gelabelten Datenset zu lernen. Dann benutzt man das trainierte Model um vorhersagen für ungelabelte Daten zu treffen. Als nächstes werden die Beispiele genommen bei denen das Model die größte Unsicherheit besitzt bei der Vorhersage. Das sind zum Beispiel Eigennamen die im Model nicht vorkommen. Die gewählten Beispiele werden menschlich gelabelt und zu dem gelabelten Datenset hinzugefügt. Nach diesem Schritt wird das Model neu trainiert. Diesen Schritt wiederholt so oft bis die gewünschte Leistung erreicht

ist. Es ist auch möglich die Labels zufällig zu wählen, anstatt die Labels zu wählen, bei denen das Model die größte Unsicherheit besitzt. Der Nachteil beim zufälligen wählen ist, dass das Model länger braucht bessere Leistung zu erzielen.

Eine gute Qualität der Labels ist signifikant, um schlechte Trainingsergebnisse vermeiden zu können und eine Algorithmus effektiv lernen lassen zu können.

RNN

Rekurrente neuronale Netze (RNNs) ist ein Typ von neuronales Netzwerk welches designt wurde um nacheinander ablaufende Daten zu verarbeiten.

[todo image of sequential data]

In RNN's ist jeder Zeitpunkt in der Eingabe vom Netzwerk verarbeitet. Die Ausgabe an jedem Zeitpunkt ist Abhängig von der Ausgabe vom vorherigen Zeitschritt. Das erlaubt dem Netzwerk Abhängigkeiten, welche sich über mehrer Zeitschritte erstrecken zu erkennen.

Mathmatisch kann ein RNN wiefolgt beschrieben werden:

$h(t) = f(h(t-1), x(t))$

Dabei ist $h(t)$ die Ausgabe an jedem Zeitpunkt t, $h(t-1)$ is die Ausgabe am vorherigen Zeitpunkt und $x(t)$ ist die Eingabe am Zeitpunkt t. Die Funktion f repräsentiert das innere eines RNN.

LSTM

Long Short-Term Memory (LSTM) Netzwerke sind entwickelt worden um das Problem der verschwindenden Gradienten zu lösen, welches bei der Trainierung von traditionellen rekurrenten neuronalen Netzen (RNNs) auftreten kann.

In traditionellen RNNs sind die verborgenen Zustände zu jedem Zeitpunkt erneuert, indem der vorherige verborgene Zustand und die gegenwärtige Eingabe benutzt wird.

Bei dieser Regel kann es zu Problemen kommen beim lernen von langfristigen Abhängigkeiten, weil der Gradient des Fehlers mit Respekt zum verborgenen Zustand und den Gewichten dazu tendieren zu verschwinden mit fortschreitenden Zeitpunkten. Das ist als das Problem der verschwindenden Gradienten bekannt.

LSTMs sind entwickelt wurden das Problem zu lösen. Dabei wurden weitere Netzwerk Strukturen eingeführt, welche "Gedächtniszellen" und "Tore" genannt werden. Die Tore können von den Gedächtniszellen länger Informationen speichern und abrufen.

Damit sind LSTMs in der Lage langfristige Abhängigkeiten in den Daten zu behalten und die Leistung von RNNs zu übersteigen.

An jedem Zeitpunkt t, bekommt das LSTM als Eingabe die gegenwärtige Eingabe x_t und den vorherigen verborgenen Zustand h_{t-1} und erzeugt einen neuen verborgenen Zustand h_t und eine Ausgabe y_t.

Der verborgene Zustand h_t ist eine Funktion von der gegenwärtigen Eingabe x_t dem vorherigen verborgenen Zustand h_{t-1} und den vorherigen Zell Zustand c_{t-1}. Der Zellenzustand c_t ist eine "Erinnerung" an vergangene Eingaben und verborgene Zustände, die über die Zeit aufrechterhalten wird.

[todo update of hidden and cell state?]

To update the hidden state and cell state, the LSTM applies the following equations at each time step:
Calculate the forget gate f_t, which controls what information is discarded from the previous cell state: f_t = sigmoid(W_f * x_t + U_f * h_{t-1} + b_f)
where W_f and U_f are the weights for the input-to-hidden and hidden-to-hidden connections, respectively, and b_f is the bias.
Calculate the input gate i_t, which controls what information is stored in the current cell state: i_t = sigmoid(W_i * x_t + U_i * h_{t-1} + b_i)
where W_i and U_i are the weights for the input-to-hidden and hidden-to-hidden connections, respectively, and b_i is the bias.
Calculate the cell state candidate c_tˆ: c_tˆ = tanh(W_c * x_t + U_c * h_{t-1} + b_c)
where W_c and U_c are the weights for the input-to-hidden and hidden-to-hidden connections, respectively, and b_c is the bias.
Calculate the cell state c_t, which is a combination of the previous cell state and the cell state candidate, weighted by the forget and input gates: c_t = f_t * c_{t-1} + i_t * c_tˆ
Calculate the output gate o_t, which controls what information is output from the current hidden state: o_t = sigmoid(W_o * x_t + U_o * h_{t-1} + b_o)
where W_o and U_o are the weights for the input-to-hidden and hidden-to-hidden connections, respectively, and b_o is the bias.
Calculate the hidden state h_t, which is a combination of the cell state and the output gate, weighted

---

```
#RNN

class RNN:
    def __init__(self, input_size, hidden_size, output_size):
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size

        self.U = np.random.randn(hidden_size, input_size) * 0.01
        self.W = np.random.randn(hidden_size, hidden_size) * 0.01
        self.V = np.random.randn(output_size, hidden_size) * 0.01
```

```
        self.b = np.zeros((hidden_size, 1))
        self.c = np.zeros((output_size, 1))

    def forward(self, x):
        T = len(x)
        h = np.zeros((self.hidden_size, T + 1))
        h[-1] = np.ones(T + 1)
        o = np.zeros((self.output_size, T))

        for t in np.arange(T):
            h[:, t] = np.tanh(self.U @ x[t] + self.W @ h[:, t - 1] + self.b)
            o[:, t] = softmax(self.V @ h[:, t] + self.c)

        return o, h

    def bptt(self, x, y):
        T = len(y)
        o, h = self.forward(x)

        dLdU = np.zeros(self.U.shape)
        dLdV = np.zeros(self.V.shape)
        dLdW = np.zeros(self.W.shape)
        dLdb = np.zeros(self.b.shape)
        dLdc = np.zeros(self.c.shape)

        delta_o = o
        delta_o[y, np.arange(T)] -= 1.

        for t in np.arange(T)[::-1]:
            dLdV += delta_o[:, t].reshape(self.output_size, 1) @ h[:, t].reshape(1, self.hic
            dLdc += delta_o[:, t].reshape(self.output_size, 1)

            delta_t = (self.V.T @ delta_o[:, t] + self.c) * (1 - h[:, t] ** 2)

            for bptt_step in np.arange(max(0, t - self.bptt_truncate), t + 1)[::-1]:
                dLdW += delta_t.reshape(self.hidden_size, 1) @ h[:, bptt_step - 1].reshape(1
```