Professor Dr.
Jörn Fischer
Hochschule Mannheim, Fakultät für Informatik
j.fischer@hs-mannheim.de

Willkommen zur Vorlesung

Grundlagen Neuronale Netze

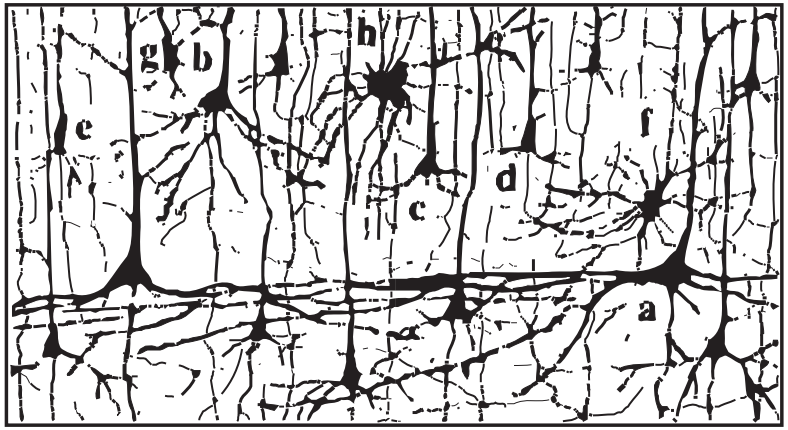


Abbildung aus »Maschinelles Lernen für Dummies«

Inhalt

- Organisatorisches
- Einführung
- Das Neuron
- Backpropagation
- Training und Test
- Autoencoder
- Faltungsnetze
- Generative Adversarial Networks
- Rekurrente Neuronale Netze
- Transformer
- Anwendungen

Organisatorisches

Fragen

- Fragen können in Vorlesung und Übung oder aber per Email an mich gerichtet werden.
- Rückkopplung ist ausdrücklich erwünscht!

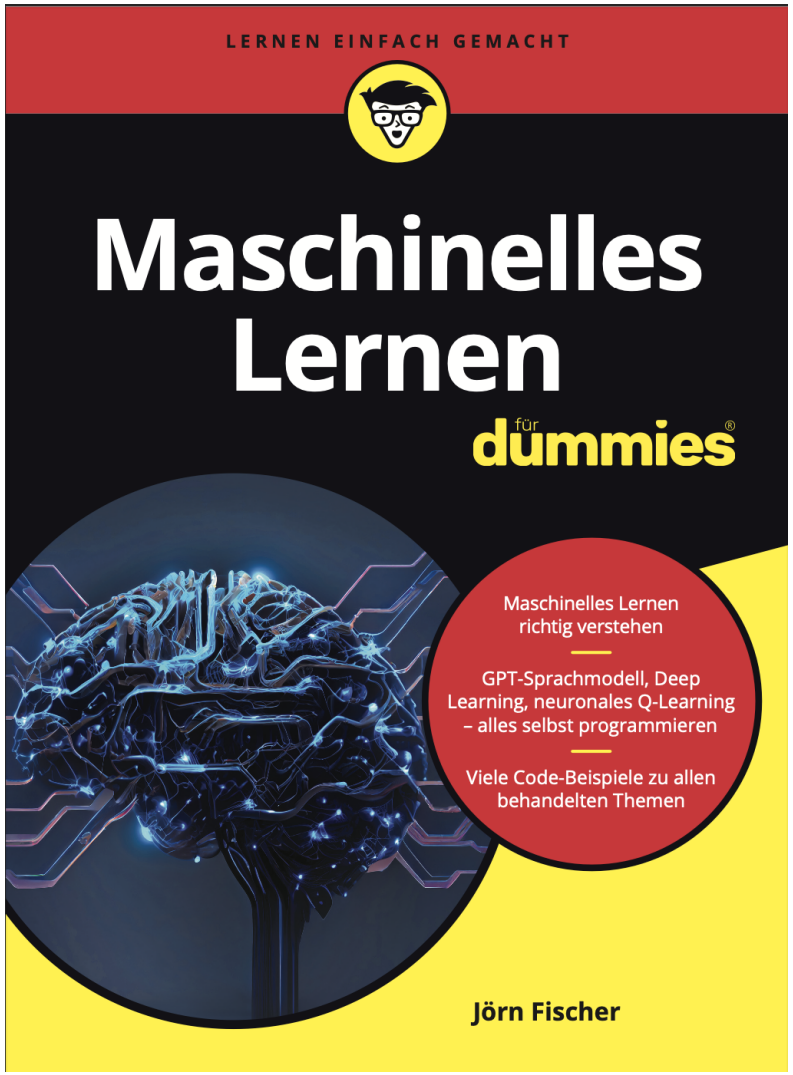
Ziel der Vorlesung

- ist, dass ihr maximal dabei lernt.
- ..., dass ihr die Algorithmen versteht.
- ..., dass ihr sie selbst programmieren könnt.
- ..., dass ihr sie selbst erklären könnt.

Mein Ansatz

- ich erkläre Euch ausgesuchte Algorithmen und deren Grenzen.
- Ihr programmiert die Algorithmen in Python (...auch zu Hause).
- Die Algorithmen werden testiert und gelten als Prüfungsleistung!

Literatur aus eigener Feder...



Natürlich gibt es auch andere Werke...

- I. Goodfellow, Y. Bengio, A. Courville: Deep Learning, MIT Press, ISBN: 978-0262035613, 2016
- https://www.cs.toronto.edu/~hinton/absps/NatureDeepReview.pdf
- https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

Fragen zu organisatorischen Dingen?

Fragen an Sie

- Wer hat schon MLE gehört?
- Wer hat schon mal Python programmiert?
- Wer hat sich schon mit neuronalen Netzen beschäftigt?

Einführung

Gedankenexperiment

Wenn man das Menschliche Gehirn in seiner technischen Leistungsfähigkeit mit einem Computer vergleicht...

technisch (Computer)	**biologisch (Gehirn)**
10 TByte Speicher	10^{13} Synapsen
100 bis 1000 TFlops	10^{14} bis 10^{15} Mult. pro Sekunde

Was gibt es schon? Wie weit sind wir?

- Bestes Programm für Gensequenzierung ist ein NN.
- Bestes Programm zur Wettervorhersage ist ein NN.
- Bestes Programm zur Proteinstrukturvorhersage ist ein NN.
- KI's sind in allen Brettspielen, selbst im GO besser als der beste Mensch.
- KI's können auch komplizierte Computerspiele erlernen und auf hohem Niveau spielen.
- Große Sprachmodelle können nicht nur Texte generieren oder vervollständigen, sondern Fragen beantworten, Probleme lösen, Texte übersetzen oder zusammenfassen.
- Mithilfe von Diffusionsmodellen können Bilder aus Sprache generiert werden.

Was für Arten des Lernens gibt es?

- **Überwachtes Lernen** (Supervised Learning)
  Eine Funktion wird anhand von Ein-Ausgabe-Paaren gelernt.
- **Unüberwachtes Lernen** (Unsupervised Learning)
  Der Algorithmus findet anhand von Eingabedaten eine Modell, welches die Eingabedaten beschreibt und vorhersagen ermöglicht.
- **Bestärkendes Lernen** (Reinforcement Learning)
  Der Algorithmus erhält nur ab und zu positives oder negatives Feedback.

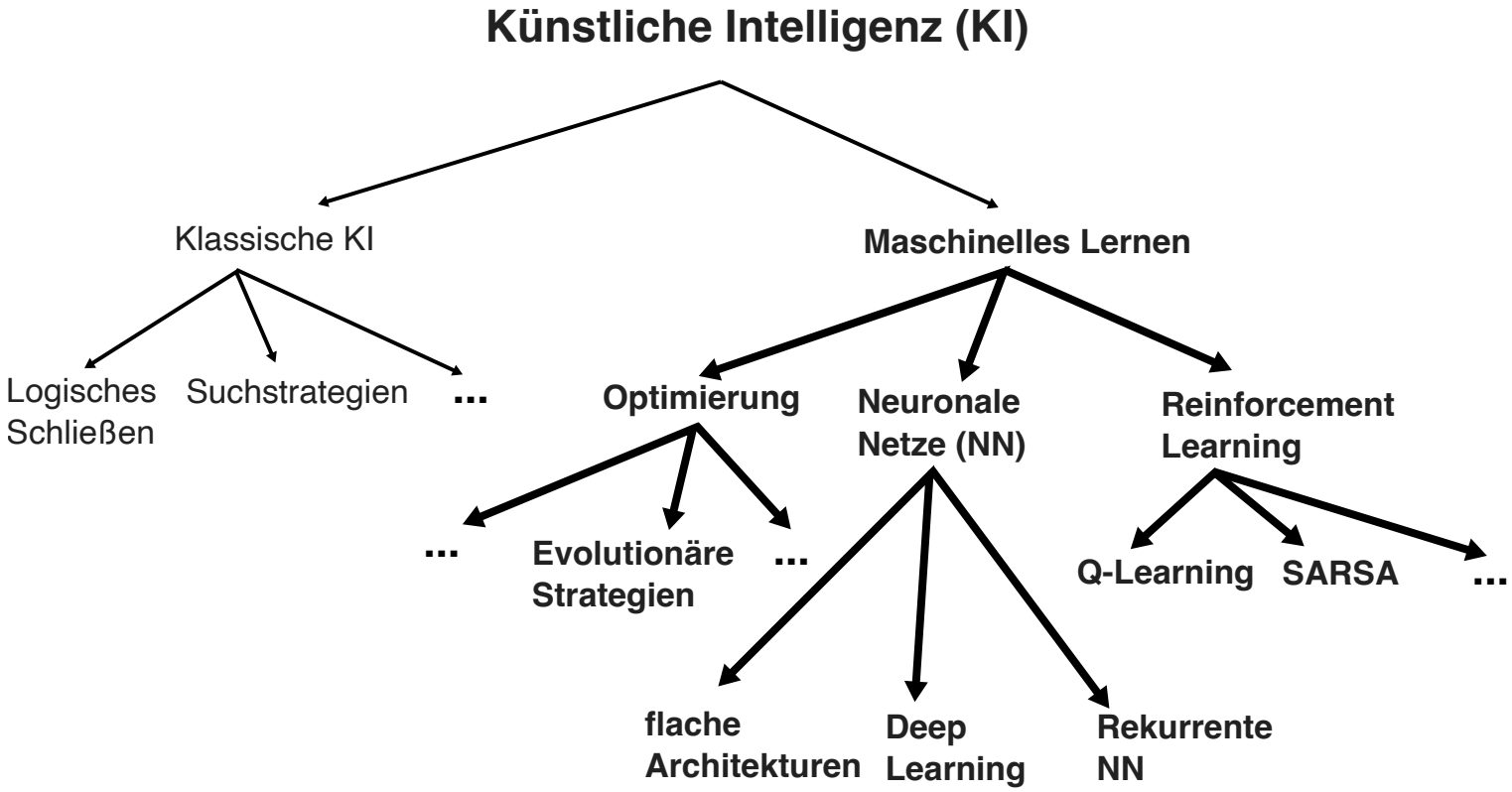


Abbildung: Die Abbildung zeigt die verschiedenen Bereiche der Künstlichen Intelligenz (KI). Neuronale Netze sind ein Teilgebiet des maschinellen Lernens mit seinen flachen, tiefen und rekurrenten Architekturen.

Das Neuron

Überblick

- Biologische Neuronen
- Künstliche Neuronen, Bias und Transferfunktionen
- Das Perzeptron
- Die Perzeptron Lernregel

Das Neuron

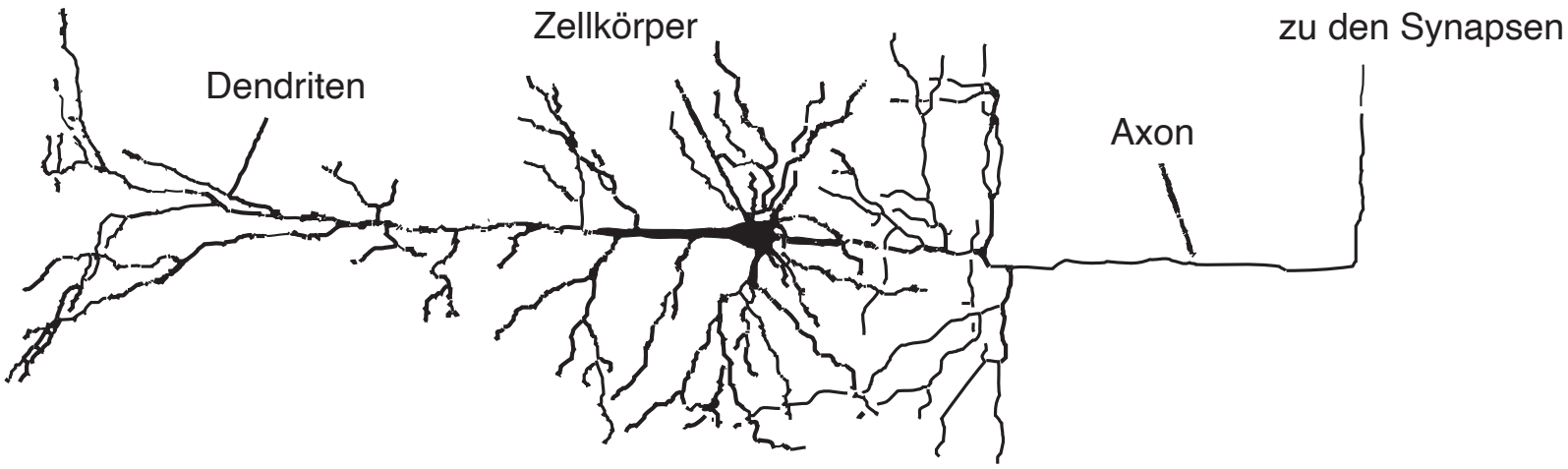


Abbildung: Eine Pyramidenzelle nach (Cajal, 1911). Die Information fließt von den Dendriten über den Zellkörper (Soma) zum Axon und zu den Synapsen, welche das Neuron mit anderen Neuronen verbindet.
(Bild aus: Maschinelles Lernen für Dummies, Wiley Verlag)

Aktionspotential

- Synapsen sind die Kopplungsstellen zwischen den Zellen, die hemmend oder erregend wirken können.
- Die Größe der Synapsen sagt etwas über ihre Effektivität bzw. über ihren Einfluss aus.
- Wird die Zelle über die Synapsen erregt, so werden an den Synapsen Transmitterstoffe ausgeschüttet und die nachfolgende Zelle depolarisiert.
- Überschreitet diese Depolarisation einen Schwellenwert, so beginnt das Neuron am Axon Aktionspotentiale zu erzeugen.

Lernen (Long Term Potentiation)

Folgende Abbildung zeigt die Prozesse, die stattfinden, wenn eine Synapse häufig Transmitterstoffe ausschüttet und damit ihren Einfluss vergrößert.

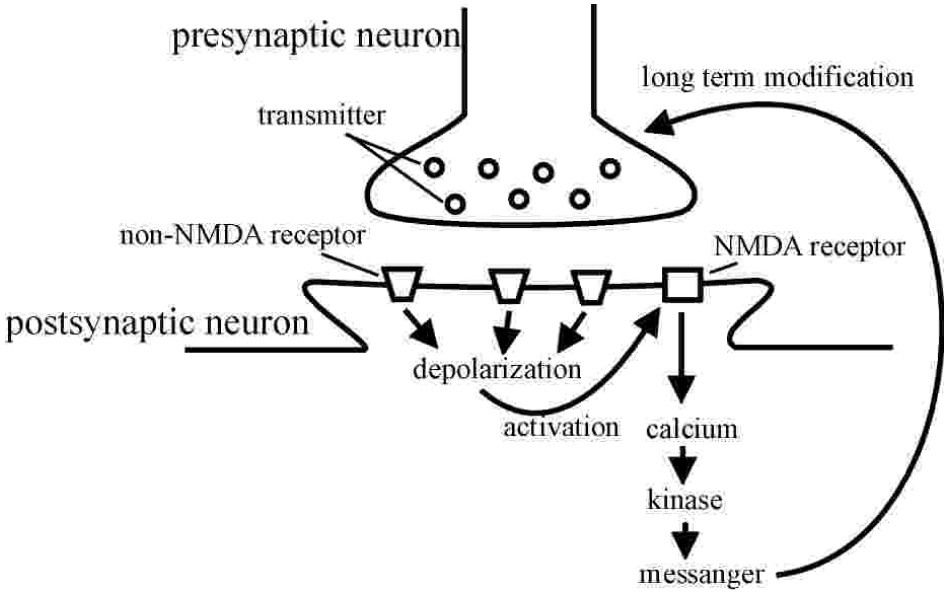


Abbildung: Die Synapsengröße (Gewicht) kann durch LTP verändert werden. Diese Gewichtsänderung nennen wir Lernen.

Das künstliche Neuron

Die Aktivität φ_j (phi) eines Neurons j wird in künstlichen Neuronen als Summe der gewichteten Eingänge geschrieben:

$$\varphi_j = \sum_i o_i w_{ji}$$

Die Ausgabe des Neurons ist die über eine Transferfunktion abgebildete Aktivität:

$$o_j = F(\varphi_j)$$

Die Ausgabe (englisch: output) des künstlichen Neurons entspricht dem, was die Frequenz der Aktionspotentiale eines biologischen Neurons ist.

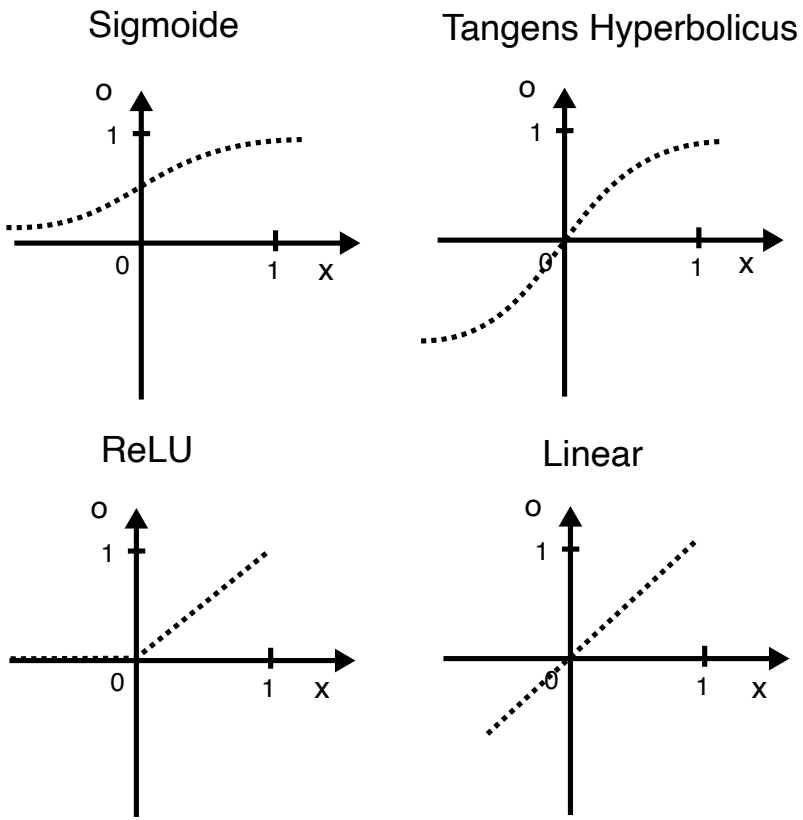


Abbildung: Transferfunktionen wie die Sigmoid- und Tangens-Hyperbolicus-Funktion sind beschränkt, während lineare und Rectified Linear Units (ReLU) durch ihre einfache Funktion und Ableitung effizienter sind.
(Bild aus: Maschinelles Lernen für Dummies, Wiley Verlag)

Die Transferfunktionen

Die Sigmoidfunktion

$$o_j = F(\varphi_j) = \frac{1}{1 + e^{-\varphi}}$$

Die ReLU-Funktion

$$o_j = F(\varphi_j) = \begin{cases} 0, & \text{if } \varphi \leq 0 \\ \varphi, & \text{if } \varphi > 0 \end{cases}$$

Die lineare Transferfunktion

$$o_j = F(\varphi_j) = \varphi$$

Das Perzeptron

- 1943: W. Mc Culloch und W. Pittsführen Neuron als Schwellwertelement ein.
- 1958: F. Rosenblattbeschreibt das mehrlagige Perzeptron als Grundlage für alle heutigen Feed Forward Netze.
- F. Rosenblatt zeigte, dass ein Neuron AND, OR und NOT Operationen ausführen und erlernen kann.
- M. Minsky und S. Papert zeigten, dass mit einer Schicht von Neuronen jedoch kein XOR erlernt werden kann.

Das Perzeptron - Aktivierung

Um ein Perzeptron Neuron zu aktivieren berechnet man die Summe der Gewichteten Eingaben minus dem Schwellwert θ und prüft, ob sie größer als null sind.
Die Ausgabe des Neurons lautet dann:

$$o_j = \begin{cases} 1 & \text{für } \sum_i w_{ji} o_i - \theta_j > 0 \\ 0 & \text{sonst} \end{cases}$$

θ wird dabei üblicherweise durch ein negatives w_θ und ein Bias-Neuron $o_\theta = 1$ ersetzt und in die Summe gezogen.

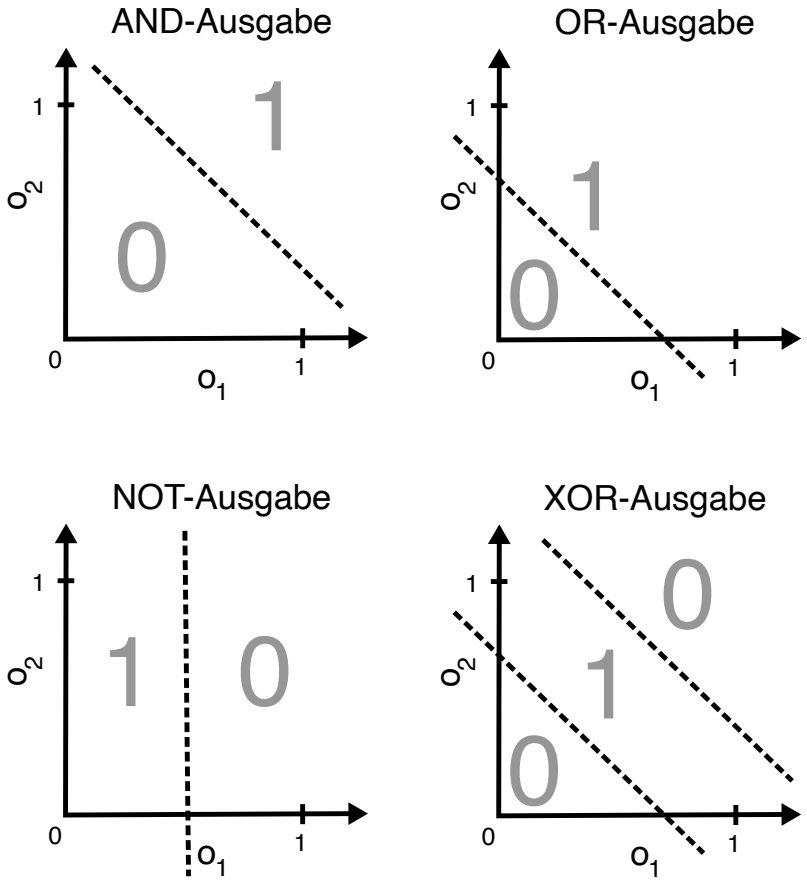


Abbildung: Um ein Neuron zu betrachten, kann man für jeden Punkt im Eingaberaum seine Ausgabe anzeigen. Die gestrichelte Linie zeigt an, wo der Schwellenwert einer Funktion liegt, um die AND-, OR-, NOT- oder XOR-Funktion abzubilden. Die grauen Nullen und Einsen deuten die Ausgaben in den Bereichen an, die durch die Stufenfunktion getrennt werden. (Bild aus: Maschinelles Lernen für Dummies, Wiley Verlag)

Zweischichtiges Perzeptron

- Erst durch eine zweite, versteckte Schicht ist es möglich beliebige Gebiete zu klassifizieren.
- Dabei definieren die Grenzlinien der Neuronen der versteckten Schicht die Grenze des zu klassifizierenden Gebietes.
- Diese Hidden-Neuronen kann man dann durch eine AND-Operation in der Ausgabeschicht zur Gebietserkennung heranziehen.

Die Perzeptron-Lernregel

Um ein Perzeptron-Neuron lernen zu lassen wird die Perzeptronlernregel angewendet:

- Bei einer null am Eingang ($o_i = 0$) hilft eine Änderung der Gewichte nicht...
  Bei einer eins am Eingang ($o_i = 1$) gilt:
- Ist die Ausgabe kleiner als das Target ($o_j < target_j$), dann vergrößere das Gewicht
- Ist die Ausgabe größer als das Target ($o_j > target_j$), dann verkleinere das Gewicht

Die Perzeptron Lernregel

Mathematisch drückt man das folgendermaßen aus:

Wenn $o_i = 1$ und $o_j = 0$ und $target_j = 1$, dann

$$w_{ji} = w_{ji} + o_i$$

Wenn $o_i = 1$ und $o_j = 1$ und $target_j = 0$, dann

$$w_{ji} = w_{ji} - o_i$$

Aufgabe 1

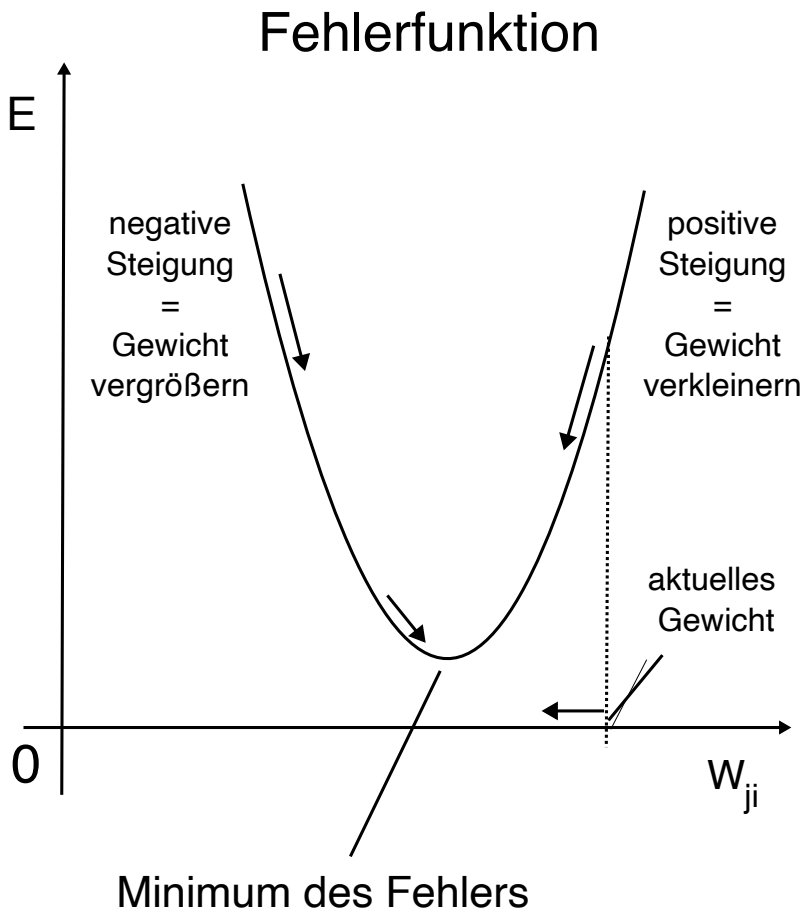
Unter den Beispielen aus meinem Buch finden Sie auch das Neuron.py Beispiel. Verändern Sie die Lernregel so, dass es die Perzeptron Lernregel ist.

Backpropagation

Überblick

- Delta-Lernregel
- Fehlerrückführung
- Backpropagation
- Batch, mini-Batch
- Warum ist die Handbremse angezogen?
- RProp, Stochastischer Gradientenabstieg
- Momentum Term, RMS-Prop, ADAM
- Kreuzentropie

Die Delta-Lernregel



Die Delta-Lernregel

Minimiere quadratischen Fehler zwischen Output und Target:

$$E = \frac{1}{2}(o_j - t_j)^2 \quad \text{mit} \quad o_j = \sum_i w_{ji} o_i$$

Ableiten nach zu optimierender Variablen (w_{ji}):

$$\frac{dE}{dw_{ji}} = \frac{\partial E}{\partial o_j} \cdot \frac{\partial o_j}{\partial w_{ji}}$$

$$= (o_j - t_j) \cdot o_i$$

Die Änderung des Gewichtes folgt dem negativen Gradienten!

$$w_{ji} \leftarrow w_{ji} - \eta(o_j - t_j)o_i$$

Dabei ist $\eta \in [0..1]$ (eta) die Lernrate.

Fehlerrückführung

Möchte man des Fehler vom Ausgabeneuron zum Eingabeneuron zurückführen, so muss man folgende Schritte ausführen:

- Vorwärtsaktivierung des Netzes vom Eingang zum Ausgang
- Fehlerpropagation rückwärts vom Ausgang zum Eingang
- Anpassung der Gewichte

Backpropagation

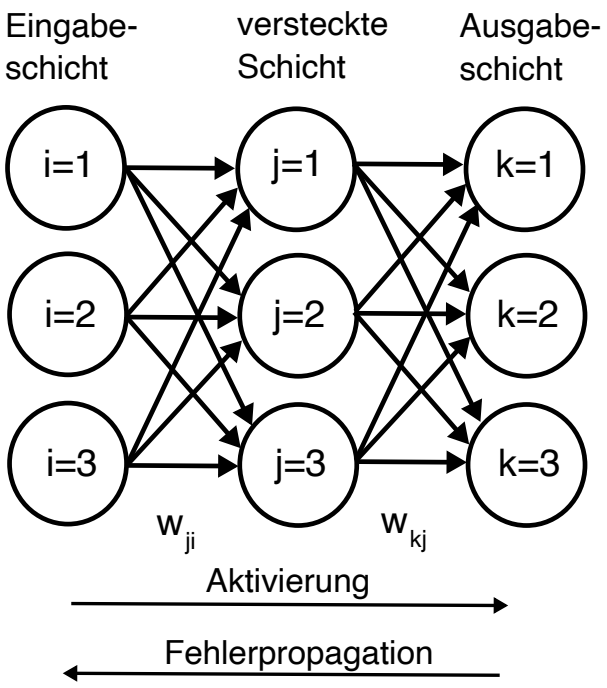


Abbildung: Aktivierung und Fehlerrückführung
(Bild aus: Maschinelles Lernen für Dummies, Wiley Verlag)

Gewichtsänderung Ausgabeschicht

Gradientenabstieg auf quadratischen Fehler des nichtlinearen Ausgabeneurons

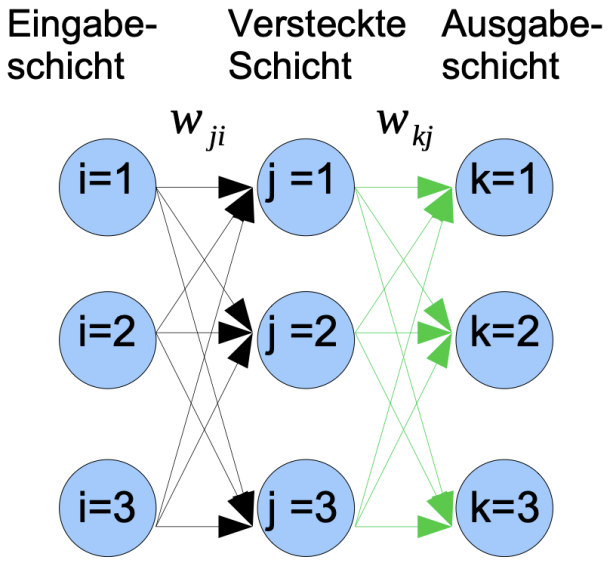
$$\Delta w_{kj} = -\eta \frac{\partial E_k}{\partial w_{kj}} = -\eta \frac{\partial E_k}{\partial o_k} \cdot \frac{\partial o_k}{\partial w_{kj}}$$

$$= -\eta (o_k - t_k) F'\left(\sum_j w_{kj} o_j\right) o_j$$

$$= -\eta e_k o_j$$

mit $E_k = \frac{1}{2}(o_k - t_k)^2$

mit $o_k = F\left(\sum_j w_{kj} o_j\right)$



- Wobei e_k der Fehlergradient der Ausgabeschicht ist
- Für eine Standardsigmoide F gilt:

$$F'(x) = \frac{\partial F(x)}{\partial x} = F(x)(1 - F(x))$$

Gewichtsänderung versteckte Schicht

Gradientenabstieg nichtlineares verstecktes Neuron

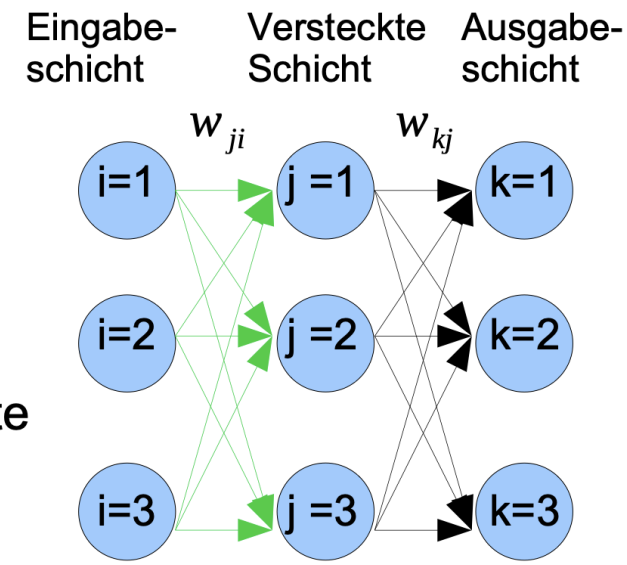
$$\Delta w_{ji} = -\eta \frac{\partial \sum_k E_k}{\partial w_{ji}} = -\eta \sum_k \frac{\partial E_k}{\partial o_k} \frac{\partial o_k}{\partial o_j} \frac{\partial o_j}{\partial w_{ji}}$$

$$= -\eta \sum_k (o_k - t_k) \frac{\partial o_k}{\partial o_j} \frac{\partial o_j}{\partial w_{ji}}$$

$$= -\eta \sum_k (o_k - t_k) F'\left(\sum_j w_{kj} o_j\right) w_{kj} \frac{\partial o_j}{\partial w_{ji}}$$

$$= -\eta \sum_k (o_k - t_k) F'\left(\sum_j w_{kj} o_j\right) w_{kj} F'\left(\sum_i w_{ji} o_i\right) o_i$$

$$\Delta w_{ji} = -\eta e_j o_i \qquad e_j = \sum_k e_k w_{kj} F'\left(\sum_i w_{ji} o_i\right)$$



Der Fehlergradient e_k wird rückwärts über die Gewichte w_{kj} verteilt!

Batch, mini-Batch

Batch-Processing

Batch processing berechnet den Gradienten aus allen Trainingsmustern und geht erst einen Gradientenschritt nach Anlegen aller Trainigsdaten. (z.B. RProp-Optimizer)

mini-Batch

Beim mini-Batch wird nur ein Teil der Trainingsmuster ans Netz angelegt, um den Gradienten zu berechnen (z.B. 32,64,128,256). Wird üblicherweise verwendet bei großer Anzahl von Trainigsdaten. (z.B. ADAM-Optimizer)

Warum ist die Handbremse angezogen?

- **Initialisierung**:Sind die Gewichte gleich initialisiert, kann das Netz nicht lernen.
- **Flat Plateau**: Auf flachen Plateaus ist die Schrittweite zu klein. Man kommt kaum in lokale Minima.
- **Steep Gaps**: In Steilen Schluchten ist sie zu groß. Das kann zu Oszillationen führen, bei denen gute Minima sogar verlassen werden.
- **Steepness Diversity**:Unterschiedliche Steilheit in unterschiedliche Richtung führt dazu, dass man sich nicht direkt in Richtung Minimum bewegt.

iRprop$^-$

Martin Riedmiller et. al. stellte in den 90ger Jahren den *Rprop* Algorithmus vor, der dann von von Christian Igel und Michael Hüsken u.a. zu *iRprop*$^-$ verfeinert wurde:

- Der alte Gradient wird zwischengespeichert, der neue errechnet.
- Falls das Vorzeichen des Gradienten für ein Neuronengewicht gleich bleibt, wird dessen Lernrate erhöht (mit 1.2 multipliziert).
- Falls sich das Vorzeichen ändert, wird die Lernrate verringert (mit 0.5 multipliziert).
- Dann wird die Lernrate mit dem Vorzeichen des Gradienten vom Gewicht abgezogen. (nicht der Gradient selbst!)

- *iRprop* ist für Neuronale Netze im Full-Batch (man legt alle Muster an bevor man den Gradientenschritt macht) eines der Besten Lernalgorithmen!
- Leider kann *iRprop* in der beschriebenen Form nicht als Mini-Batch-Verfahren (man legt nur einen Teil der Muster an bevor der Gradientenschritt gemacht wird) eingesetzt werden, da die auf einen Teil der Daten optimierten Lernraten meist für den nächsten Teil nicht gut funktionieren.

Stochastischer Gradientenabstieg

Den normale Backpropagation-Algorithmus kann man auch nur auf einen Teil der Trainingsdaten anwenden. Das nennt man dann **stochastischer Gradientenabstieg**. Es ist ein häufig verwendeter Algorithmus, der normalerweise im Mini-Batch ausgeführt wird. Dabei wird der Gradient aus mehreren zufällig zusammengestellten Trainingsmustern erst gemittelt, bevor Gewichtsänderungen vorgenommen werden.

$$w_{ji} \leftarrow w_{ji} - \eta \left(\frac{1}{m}\right) \sum_{b=1}^{m} \frac{dE_b}{dw_{ji}}$$

Dabei ist m die Anzahl der Trainingsbeispiele im Mini-Batch.

Momentum Term

Um die Konvergenz des Algorithmus zu erhöhen führt man den Momentum Term ein.

$$m_{ji} \leftarrow \beta \cdot m_{ji} + (1 - \beta) \frac{dE}{dw_{ji}}$$

Dabei wird die Variable m_{ji} am Anfang mit $m_{ji} = 0$ initialisiert. Die Gewichtsänderung sieht dann wie folgt aus:

$$w_{ji} \leftarrow w_{ji} - \eta \cdot m_{ji}$$

Der Koeffizient $\beta \in [0, 1]$ gibt an, wie stark der Einfluss des neuen und der alten Gradienten ist.

RMS-Prop

Im RMSprop-Algorithmus wird für jedes Gewicht eine zusätzliche Variable v_{ji} gespeichert, die nach jedem Mini-Batch als gleitender Mittelwert des quadrierten Gradienten aktualisiert wird. Dabei wird v_{ji} am Anfang mit $v_{ji} = 0$ initialisiert.

$$v_{ji} \leftarrow \alpha \cdot v_{ji} + (1 - \alpha) \cdot \left(\frac{dE}{dw_{ji}} \right)^2$$

Danach werden auch die Gewichte aktualisiert:

$$w_{ji} \leftarrow w_{ji} - \eta * \frac{\frac{dE}{dw_{ji}}}{(\sqrt{v_{ji}} + \epsilon)}$$

ADAM

ADAM kombiniert RMS-Prop mit dem Momentum-Term. Die Gleichungen für den Momentum-Teil schreibt man als:

$$m_{ji} \leftarrow \beta_1 \cdot m_{ji} + (1 - \beta_1) \cdot \left(\frac{dE}{dw_{ji}} \right)$$

Die Gleichung für den RMSprop-Teil schreibt man als:

$$v_{ji} \leftarrow \beta_2 \cdot v_{ji} + (1 - \beta_2) \cdot \left(\frac{dE}{dw_{ji}} \right)^2$$

Zusätzlich gibt es zwei Bias-Korrekturterme, die den Bias im Momentum- und im RMSprop-Teil korrigieren:

$$\hat{m}_{ji} \leftarrow \frac{m_{ji}}{1 - \beta_1} \qquad \hat{v}_{ji} \leftarrow \frac{v_{ji}}{1 - \beta_2}$$

Die daraus resultierende Gewichtsänderung für den ADAM-Optimierer lautet:

$$w_{ji} \leftarrow w_{ji} - \eta * \frac{\hat{m}_{ji}}{(\sqrt{\hat{v}_{ji}} + \epsilon)} \tag{1}$$

Der ADAM-Algorithmus ist eines der besten Optimierungsmethoden für neuronale Netze. Durch die adaptive Schrittweite und den Momentum-Term konvergiert er schneller und robuster als die meisten anderen Trainingsalgorithmen.

Kreuzentropie

Der quadratische Fehler wird bei Regressionsaufgaben verwendet, während bei Klassifikationsaufgaben die Kreuzentropie $H(p, q)$ zwischen einer wahren Verteilung p und einer vorhergesagten Verteilung q gemessen wird.

$$H(p, q) = -\sum_i p(i) \log(q(i)) \tag{2}$$

Dabei läuft der Summationsindex i über alle möglichen Klassen oder Ereignisse. Ist die Kreuzentropie gleich null, dann sind beide Verteilungen gleich. Je größer die Kreuzentropie, desto unterschiedlicher sind die Verteilungen.

Kreuzentropie (Beispiel)

Ein Bild soll als Hund, Katze oder Maus klassifiziert werden. Unser Modell gibt für ein bestimmtes Bild die folgenden vorhergesagten Wahrscheinlichkeiten aus:

$$P(\text{Hund}) = 0.7$$
$$P(\text{Katze}) = 0.2$$
$$P(\text{Maus}) = 0.1$$

Das tatsächliche Label für dieses Bild ist "Hund". Die wahre Wahrscheinlichkeitsverteilung für dieses Bild ist daher:

$$P(\text{Hund}) = 1$$
$$P(\text{Katze}) = 0$$
$$P(\text{Maus}) = 0$$

Die Kreuzentropie zwischen der wahren Verteilung p und der vorhergesagten Verteilung q wird durch die Formel berechnet:

$$H(p,q) = -\sum_i p(i)\log(q(i)) \tag{3}$$

Setzen wir die gegebenen Werte in die Formel ein:

$$\begin{aligned} H(p,q) &= -[1 \cdot \log(0.7) + 0 \cdot \log(0.2) + 0 \cdot \log(0.1)] \\ &= -\log(0.7) \approx 0.3567 \end{aligned}$$

Daher beträgt die Kreuzentropie für dieses Beispiel ungefähr 0.3567. Dieser Wert gibt an, wie gut die vorhergesagte Wahrscheinlichkeitsverteilung die tatsächliche Verteilung repräsentiert. Ein Wert von 0 würde bedeuten, dass die Vorhersage perfekt ist, während höhere Werte auf eine schlechtere Übereinstimmung hinweisen. (Zur Erinnerung: $log(1) = 0$; wäre der Hund mit Wahrscheinlichkeit $P(\text{Hund}) = 1$ klassifiziert worden, so wäre die Kreuzentropie $H(p,q) = 0$.)

Aufgabe 2

Unter den Beispielen aus meinem Buch finden Sie sowohl einen Backpropagation Algorithmus, als auch einen Beispielcode für den $iRProp^-$ Algorithmus. Verändern Sie den Backpropagation Algorithmus so, dass er den Gradientenabstieg vollständig mit $iRProp^-$ macht.

Training und Test

Überblick

- Gewichte initialisieren
- Kreuzvalidierung
- Overfitting
- Early Stopping
- Batch-Normalisierung
- Layer-Normalisierung
- Regularisierung
- Dropout
- Rauschen

Überblick zur Gewichtsinitialisierung

- Kleine zufällige Werte für Symmetriebrechung und effiziente Konvergenz.
- Xavier/Glorot für angepasste Varianz, verhindert Gradientenprobleme.
- Orthogonale Initialisierung stabilisiert Gradientendynamik.
- Sparse Initialisierung reduziert Überanpassung, verbessert Generalisierung.

kleine zufällige Werte

Ziel

Symmetriebrechung und effiziente Konvergenz.

Methode

- Gewichte aus Normalverteilung
- Kleine Varianz z.B.`var=0.2`

Quelltext

```
w=np.random.normal(loc=0,
scale=var,size=(n_in,n_out))
```

Xavier/Glorot-Initialisierung

Ziel

Vermeidung des verschwindenden/explodierenden Gradientenproblems in tiefen Netzwerken.

Methode

- Gewichte aus Gleichverteilung
- Basierend auf Anzahl der Neuronen-Ein-/Ausgänge

Xavier/Glorot-Initialisierung

Varianzberechnung

- Varianz: $\frac{2}{n_{\text{in}}+n_{\text{out}}}$
- n_{in}: Anzahl Eingänge
- n_{out}: Anzahl Ausgänge

Quelltext

```
l=np.sqrt(6/(n_in + n_out)) # Xavier Glorot
w=np.random.uniform(low=-l,high=l,size=(n_in,n_out)
```

Nutzen

Stabile Aktivierung über Schichten hinweg $\rightarrow$ Effektives Lernen

Orthogonale Initialisierung

Ziel

Stabilisieren der Gradientendynamik

Methoden

- Orthogonalisierung einer zufälligen Matrix (z.B. QR-Zerlegung)
- Wie bei Deep Belief Netzen mit Contrastive Divergence vortrainieren

Kreuzvalidierung

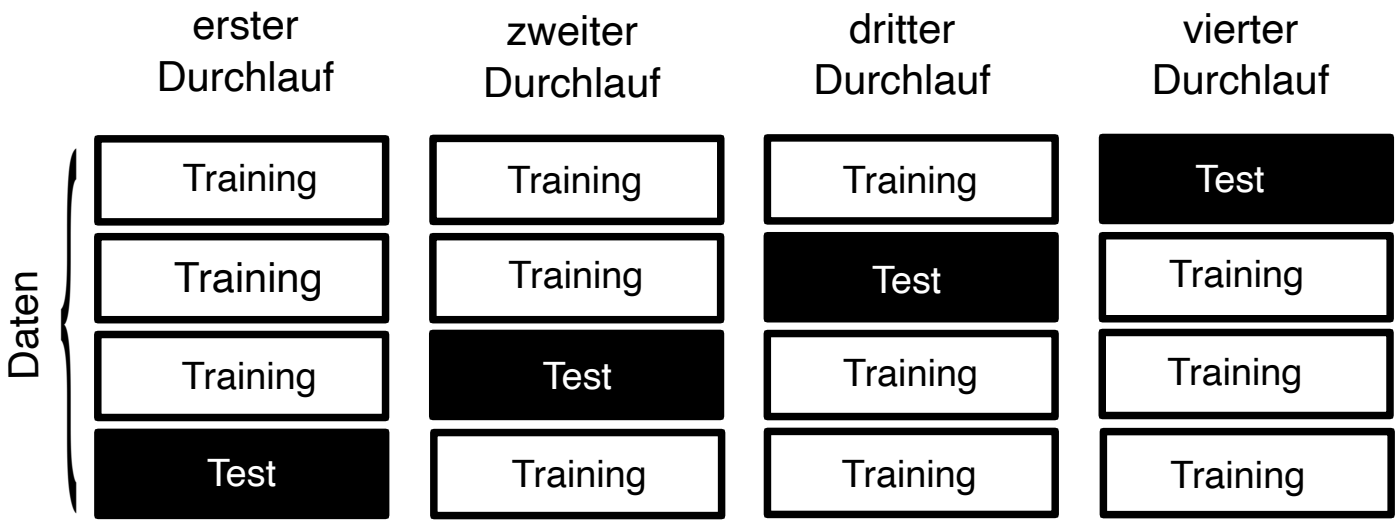


Abbildung: Bei k-facher Kreuzvalidierung unterteilt man die Daten in k Datenpakete (hier k=4). Ein Paket wird zum Testen, der Rest zum Training des neuronalen Netzes verwendet. Der Fehler wird ermittelt und dann mit dem nächsten Datenpaket fortgesetzt. Schließlich berechnet sich der Gesamtfehler als Mittelwert aus den Fehlern aller Testmengen. (Bild aus: Maschinelles Lernen für Dummies, Wiley Verlag)

Overfitting

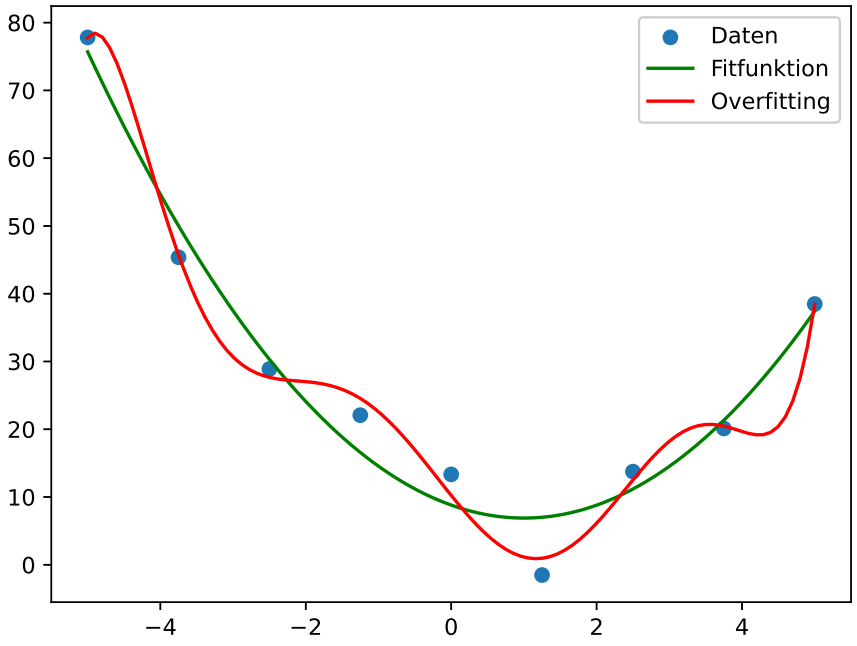


Abbildung: Overfitting bedeutet, dass die zu fittende Funktion die Trainingsdaten zu genau abbildet (rote Kurve) und z.B. auf Testdaten nicht gut generalisiert.

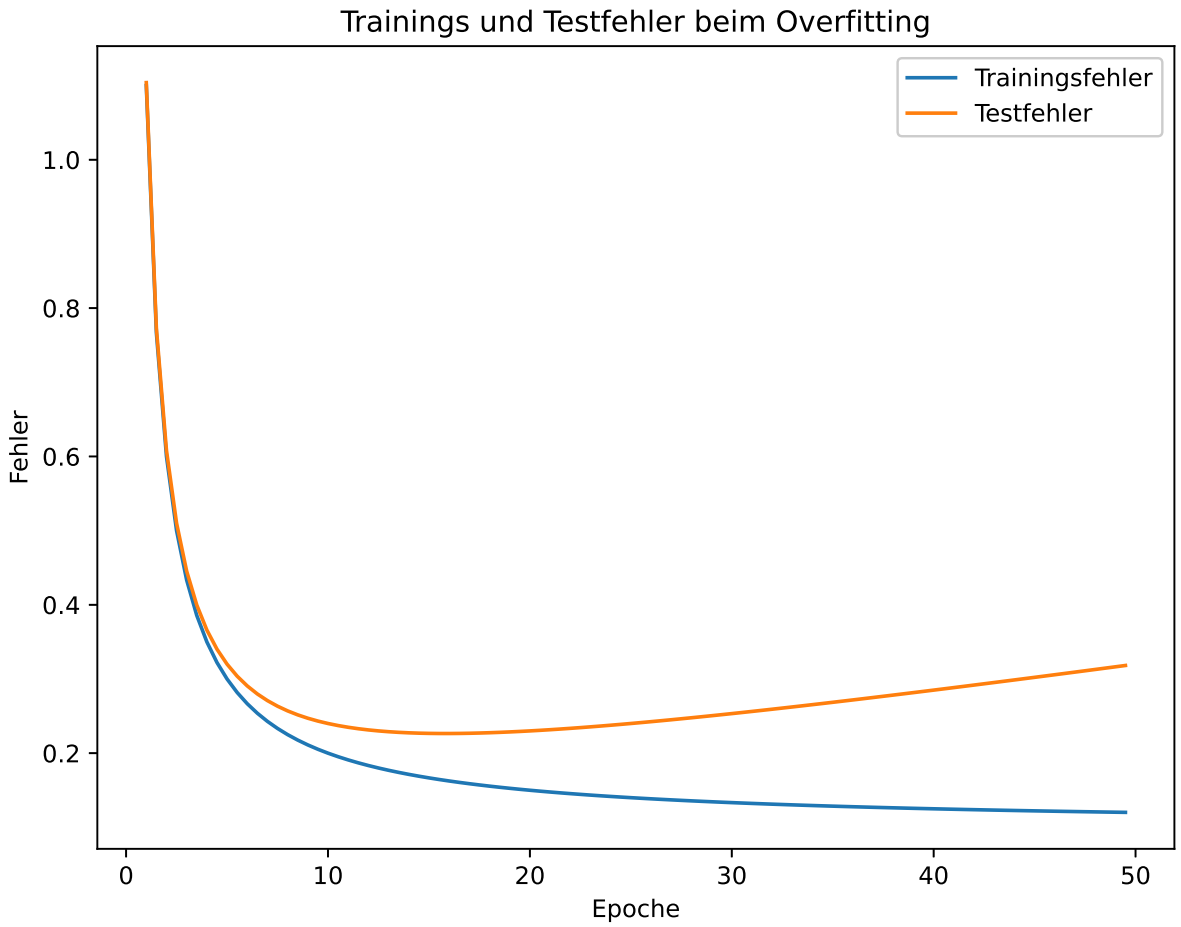


Abbildung: Während der Trainingsfehler stetig sinkt, kann der Testfehler durch Overfitting irgendwann wieder steigen.

Overfitting – Was hilft?

Es gibt unterschiedliche Techniken, um Overfitting zu vermeiden:

- Early stopping
- Batch-Normalisierung
- Layer-Normalisierung
- L1-/ L2-Regularisierung
- Dropout
- Hinzufügen von Rauschen

Early Stopping

- Überwacht die Leistung des Modells auf einer Validierungsdatenmenge während des Trainings.
- Das Training wird beendet, wenn die Leistung auf der Validierungsdatenmenge nicht mehr verbessert wird, um Overfitting zu vermeiden.
- Das Modell mit der besten Validierungsleistung wird ausgewählt.

Vorteile

- Verhindert Overfitting effektiv ohne das Netz zu verändern.
- Optimiert die Trainingszeit
- Einfach zu implementieren und universell anwendbar.

Was ist Batch-Normalisierung?

Ein Verfahren zur Normalisierung der Aktivität (vor der Aktivierungsfkt.) von Neuronen in einem Netzwerk, angewandt pro Mini-Batch, um Training zu beschleunigen und die Stabilität zu verbessern.

Warum Batch-Normalisierung?

Reduziert den internen kovarianten Shift, erlaubt höhere Lernraten und macht das Modell weniger empfindlich gegenüber der Initialisierung.

Implementierung der Batch-Normalisierung

Mittelwert des Mini-Batch:

$$\mu_B \leftarrow \frac{1}{m} \sum_{i=1}^{m} x_i \tag{4}$$

Varianz des Mini-Batch:

$$\sigma_B^2 \leftarrow \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu_B)^2 \tag{5}$$

Normalisierung:

$$\hat{x}_i \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \tag{6}$$

Implementierung der Batch-Normalisierung

Skalieren und verschieben: Zwei lernbare Parameter, γ (Gamma) und β (Beta), werden eingeführt, um die Skalierung und Verschiebung der normalisierten Aktivierungen anzupassen. Die normalisierte Aktivierung wird mit γ skaliert und um β verschoben:

$$y_i \leftarrow \gamma \hat{x}_i + \beta \tag{7}$$

Die Einführung von γ und β ist entscheidend, da sie dem Netzwerk die Flexibilität geben, die ursprüngliche Verteilung der Aktivierungen wiederherzustellen, falls dies für das Lernen vorteilhaft ist.

Vorteile der Batch-Normalisierung

- Verbessert die Konvergenzgeschwindigkeit. (bis zu 14 x schneller)
- Ermöglicht die Verwendung höherer Lernraten.
- Kann zur Regularisierung beitragen und die Notwendigkeit von Dropout reduzieren.

Was ist Layer-Normalisierung?

Im Gegensatz zur Batch-Normalisierung, die Mini-Batch-Eingaben normalisiert, führt die Layer-Normalisierung eine Normalisierung über die n Aktivierungen $x = [x_1, x_2, \ldots, x_n]$ eines Layers durch:

$$\mu = \frac{1}{n}\sum_{i=1}^{n} x_i, \quad \text{(Mittelwert)} \tag{8}$$

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n} (x_i - \mu)^2, \quad \text{(Varianz)} \tag{9}$$

$$\hat{x}_i = \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}}, \quad \text{(normalisierte Aktivierungen)} \tag{10}$$

Auch hier gibt es eine erlernbare Skalierung und Verschiebung $y_i \leftarrow \gamma \hat{x}_i + \beta$, die eine Anpassung der Normalisierung ermöglicht, so dass sie für's Lernen optimal funktioniert.

Vorteile der Layer-Normalisierung

- **Unabhängigkeit von der Batch-Größe:** Ermöglicht effizientes Training mit beliebigen Batch-Größen.
- **Verbesserte Trainingsdynamik:** Stabilisiert das Training durch Reduzierung der internen Kovariate-Verschiebung.
- **Anwendbar auf rekurrente Netzwerke:** Effektiv in LSTMs und GRUs ohne Beeinträchtigung der Datenabhängigkeit.

Regularisierung

L1-Regularisierung

Bei L_1-Regularisierung (auch Lasso-Regularisierung genannt) wird ein Strafwert hinzugefügt, der proportional zum absoluten Wert der Gewichte des neuronalen Netzes ist. Dieser Strafterm führt zu einer spärlichen Gewichtsmatrix.

Implementierung

- $L = L_0 + L_1 = L_0 + \lambda * \sum_i^n |w_i|$
- Fügt dem Loss L_0 einen L1-Regulierungs-Term hinzu.
- Minimiert die Summe der Beträge der Gewichte (nur für lineare Schichten).
- Reduziert das Risiko von Overfitting und kann zur Verbesserung der Generalisierung beitragen.

Regularisierung

L2-Regularisierung

L_2-Regularisierung (Ridge-Regularisierung) fügt einen Strafterm hinzu, der proportional zur Summe der quadrierten Gewichte des neuronalen Netzes ist. Dieser Strafterm reduziert die Gewichte und verhindert Überanpassung bei Klassifikation.

Implementierung

- $L = L_0 + L_2 = L_0 + \lambda * \sum_i^n w_i^2$
- Fügt dem Loss L_0 einen L2-Regulierungs-Term hinzu.
- Minimiert die Summe der Quadrate der Gewichte (nur für lineare Schichten).
- Reduziert das Risiko von Overfitting und kann zur Verbesserung der Generalisierung beitragen.

Dropout

- Deaktiviert zufällig einige Neuronen während des Trainings.
- Reduziert das Risiko von Overfitting und kann zur Verbesserung der Generalisierung beitragen.
- Anwendbar in allen Schichten, jedoch häufig in versteckten Schichten.

Vorteile

- Ermöglicht eine bessere Generalisierung.
- Reduziert das Risiko von Overfitting.
- Kann zur Verbesserung der Stabilität des Netzes beitragen.

Rauschen

- Durch hinzufügen von Rauschen zu den Trainingsmustern können Trainingsmuster schlechter „auswendig“gelernt werden.
- Das kann auch auf die Ausgänge von hidden Schichten angewendet werden.

Vorteile

- Fördert Generalisierungsleistung.
- Reduziert das Risiko von Overfitting.

Autoencoder

Überblick

- Restricted Bolzman Machines
- Autoencoder

Restricted Bolzman Machines

- RBMs können verwendet werden, um verschiedene Arten von Daten zu modellieren
- Sie bestehen aus zwei Schichten von Neuronen, der sichtbaren Schicht und der verborgenen Schicht
- Die sichtbaren Neuronen repräsentieren die Eingabe, während verborgene Neuronen Features oder Muster in den Daten erkennen
- Alle Verbindungen zwischen den Neuronen sind bidirektional. Die Aktivierung der verborgene Schicht erfolgt mit den selben Gewichten, wie die Aktivierung der Eingabeschicht.

Restricted Bolzman Machines

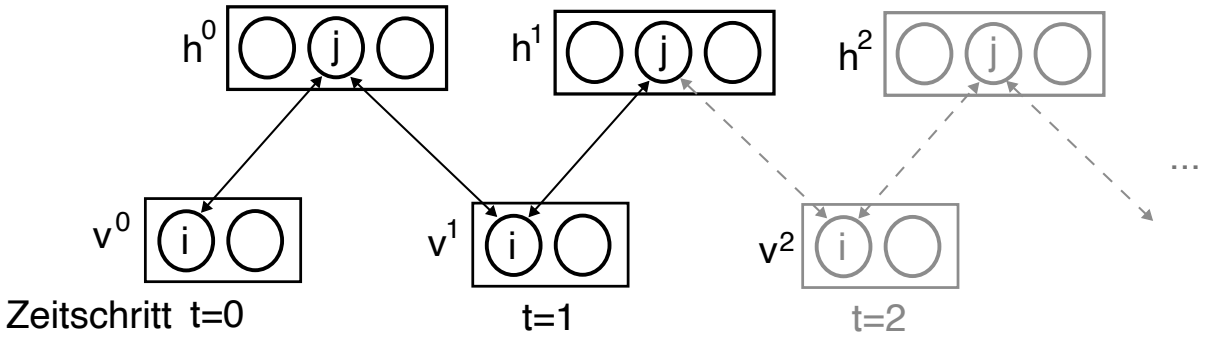


Abbildung: Beim Contrastive Divergence nutzt man eine sichtbare (unten) und eine versteckte Schicht (oben). Ein Input v^0 aktiviert h^0, wird im nächsten Zeitschritt zu v^1 rekonstruiert und aktiviert h^1. Die Gewichtsänderung für CD^1 ist $\Delta w_{ji} = \eta(< v_i h_j >^0 - < v_i h_j >^1)$, wobei $<>$ der Erwartungswert ist.
(Bild aus: Maschinelles Lernen für Dummies, Wiley Verlag)

Restricted Bolzman Machines

- **Positive Phase:** In der positiven Phase wird der Eingabevektor v^0 in das Netzwerk eingespeist, um die verborgenen Einheiten h^0 mittels der Formel $P(h|v) = \sigma(Wv + c)$ zu aktivieren. Diese Aktivierung basiert auf der Wahrscheinlichkeit, der Sigmoidfunktion σ, der Gewichtsmatrix W, dem Eingabevektor v und dem Bias-Vektor c der verborgenen Einheiten.
- **Negative Phase:** In der negativen Phase wird Gibbs-Sampling mit den aktivierten verborgenen Einheiten angewandt, um den rekonstruierten Eingabevektor v^1 zu berechnen. Sichtbare Einheiten werden mit $P(v^1|h) = \sigma(W^T h + b)$ reaktiviert und verborgene Einheiten anschließend mit $P(h^1|v^1) = \sigma(Wv + c)$.

Restricted Bolzman Machines

- **Update der Gewichte:** Für die Gewichtsaktualisierung, nachdem mindestens eine Wiederholung der Phasen ausreicht, wird die Contrastive-Divergence-Lernregel verwendet: $\Delta W = \eta(v^0(h^0)^T - v^n(h^n)^T)$, $\Delta c = \eta(h^0 - h^n)$, und $\Delta b = \eta(v^0 - v^n)$, wobei η die Lernrate und n die Anzahl der Iterationen repräsentiert.

Autoencoder

Ein Autoencoder ist ein Netzwerk, das trainiert wird, um Eingaben am Ausgang des Netzes wiederherzustellen, wie bei den Restricted Boltzmann Machines. Es verwendet jedoch übliche gerichtete Neuronen und komprimiert die Daten in mehreren Schichten zu niedrigdimensionalen Daten, bevor sie über mehrere Schichten dekomprimiert und rekonstruiert werden.

Autoencoder

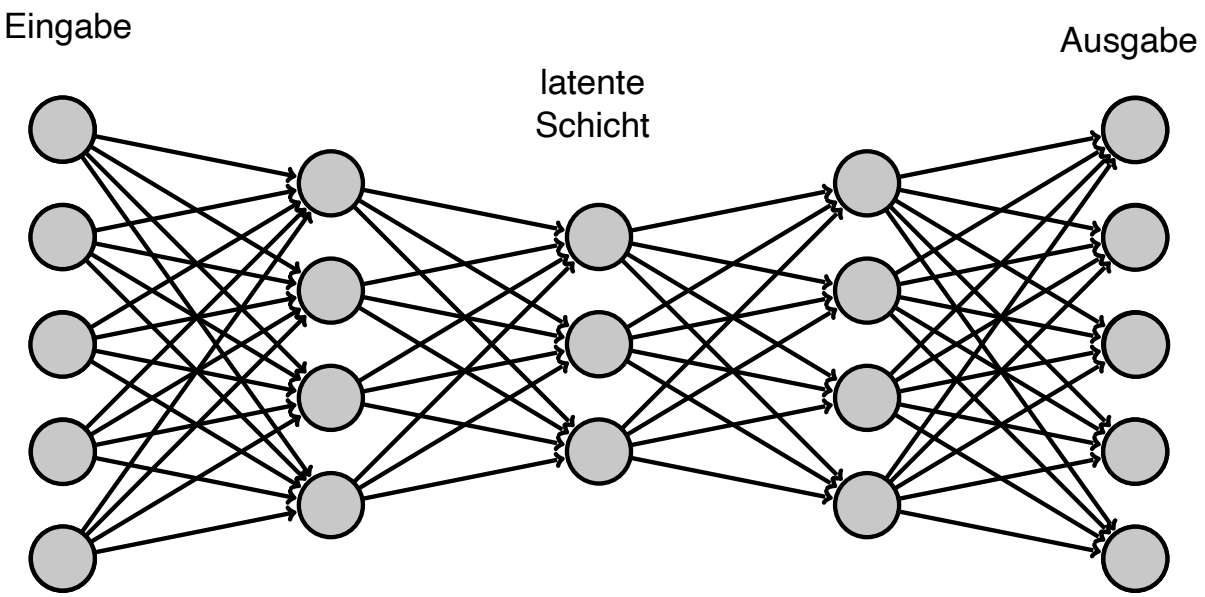


Abbildung: Der Autoencoder besteht aus fünf Schichten: Eingang, drei versteckte (einschließlich der latenten Schicht) und Ausgabe.
(Bild aus: Maschinelles Lernen für Dummies, Wiley Verlag)

Aufgabe 3

Implementieren Sie eine Restricted Bolzman Machine als einfachste Form eines Autoencoders. Am Eingang sollen die MNIST Ziffern angelegt werden. Durch Aktivierung der Hidden Schicht und Rückaktivierung sollten die MNIST Ziffern wieder rekonstruiert werden können. Nun reduzieren Sie die Anzahl der Hidden Neuronen auf 100. Können die Ziffern trotzdem so rekonstruiert werden, dass man sie noch erkennen kann?

Faltungsnetze

Überblick

- Faltung
- Pooling
- Convolutional Neural Networks

Faltung

In Faltungsschichten werden Filter, auf Eingabedaten angewendet, um Merkmale aus Bildern zu extrahieren. Filter sind kleine Matrizen oder Tensoren mit Filterkoeffizienten. Das Filter wird über das gesamte Eingabebild geschoben und jedes Pixel des Eingabebildes wird mit den darüberliegenden Filterkoeffizienten multipliziert. Die Produkte werden summiert und als Pixelwert in der Merkmalskarte gespeichert.

Faltung

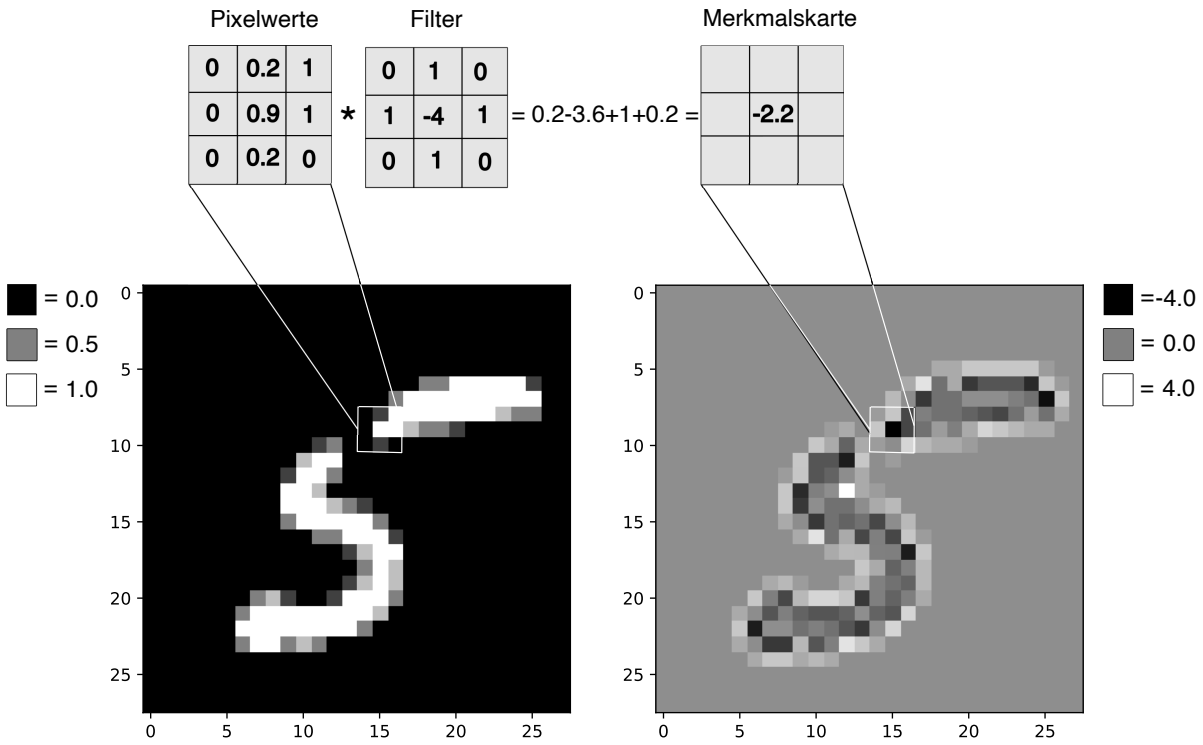


Abbildung: Eine 3x3-Filtermaske wird über das Bild bewegt und jedes Filterkoeffizient multipliziert mit dem darunterliegenden Pixelwert, wobei das Ergebnis an der mittleren Position der Maske in der Merkmalskarte eingetragen wird. Für Rand-Pixel können Spiegelungen des Originalbildes verwendet werden, um Artefakte zu vermeiden.
(Bild aus: Maschinelles Lernen für Dummies, Wiley Verlag)

Pooling

In Faltungsnetzwerken folgen Pooling-Schichten nach einer oder mehreren Faltungsschichten, um die Größe der Merkmalskarten und die Parameteranzahl zu reduzieren, was die Rechenlast für nachfolgende Schichten verringert. Wie bei der Faltung bewegt sich eine Maske (Pooling-Fenster) über die Merkmalskarte, aber im Gegensatz zur Faltung werden andere Operationen durchgeführt, wie das Auswählen des Maximalwerts (Max-Pooling) oder das Berechnen des Durchschnitts (Average-Pooling).

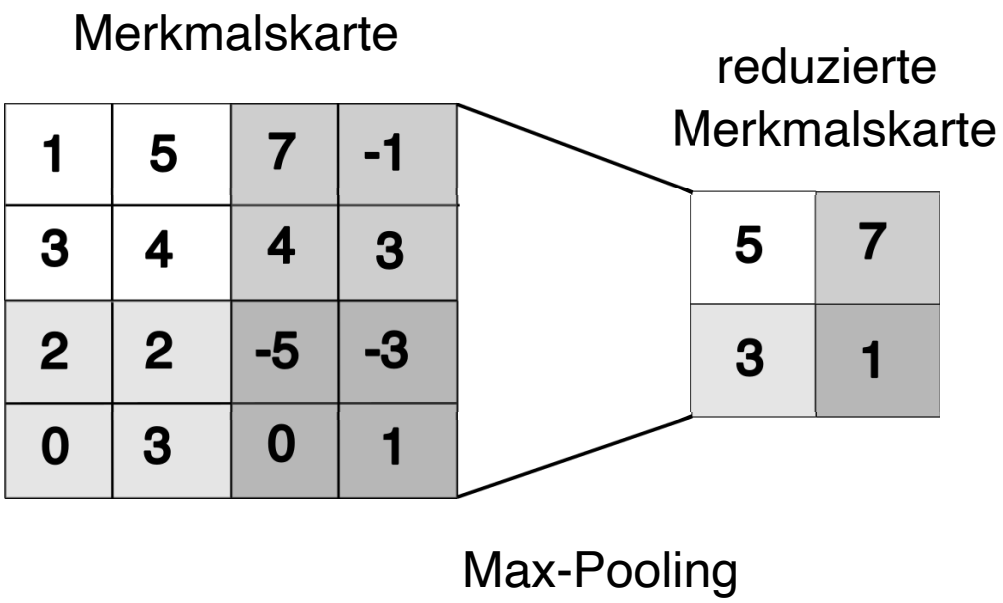


Abbildung: Beim Pooling wird eine Maske über den Merkmalskarten verschoben, wobei der Stride vorgegeben ist. In der Abbildung zeigt sich, dass mehrere Werte zu einem Wert zusammengefasst werden, oft durch Max-Pooling mit dem höchsten Wert der Daten. Alternativ können auch Durchschnittswert oder Summe verwendet werden, um in die nächste Merkmalskarte übertragen zu können.
(Bild aus: Maschinelles Lernen für Dummies, Wiley Verlag)

Letzte Schichten eines CNN

Die abschließenden Schichten eines neuronalen Netzes sind typischerweise vollvernetzte Schichten, die die abgeflachte Ausgabe von Pooling-Schichten aufnehmen und verarbeiten, um die endgültige Vorhersage zu generieren.

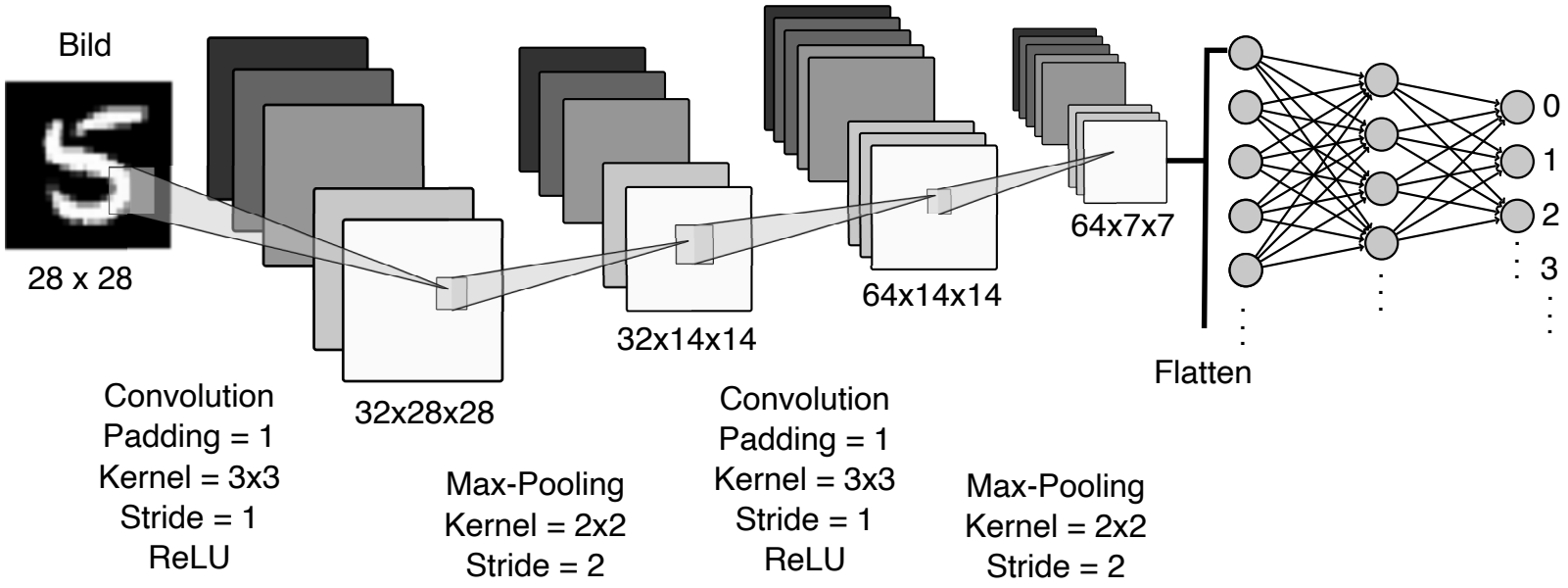


Abbildung: Das Faltungsnetz besteht aus zwei Faltungsschichten und zwei Max-Pooling-Schichten, die abwechselnd durchlaufen.
(Bild aus: Maschinelles Lernen für Dummies, Wiley Verlag)

Aufgabe 4

Implementieren Sie den zweidimensionalen Pooling Layer eines Convolutional Networks. Laden Sie dazu ein Schwarz-Weiß Bild und erzeugen Sie ein kleineres Bild indem Sie die Pooling Maske mit Stride 2 (heißt in Schrittweite 2) über das Bild schieben.

Generative Adversarial Networks

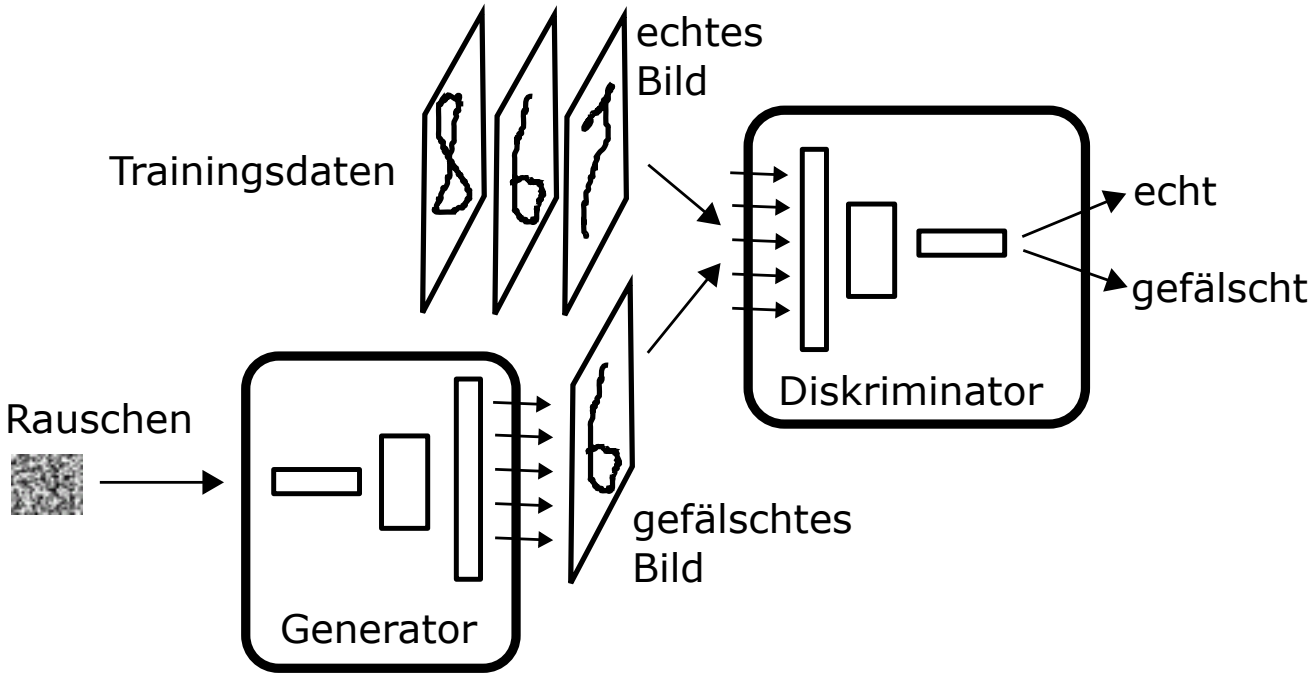


Abbildung: Die Abbildung zeigt eine GAN-Architektur mit Diskriminator und Generator, bei denen der Diskriminator echte von gefälschten Bildern unterscheiden lernt und der Generator trainiert wird, um das erzeugte Bild möglichst als echt zu klassifizieren.
(Bild aus: Maschinelles Lernen für Dummies, Wiley Verlag)

Generator

Der Hauptzweck eines Generatornetzes besteht darin, synthetische Daten zu erzeugen, indem es Zufallsrauschen in ein Muster transformiert, das der Zieldatenverteilung ähnelt. Der Generator lernt, den latenten Raum auf den Datenraum abzubilden und verbessert im Laufe der Zeit seine Fähigkeit, den Diskriminator zu täuschen, indem er realistischere Stichproben erzeugt.

Diskriminator

Das Diskriminatornetz zielt darauf ab, zwischen realen und synthetischen Daten zu unterscheiden und wird mit Stichproben aus beiden Quellen gefüttert. Es lernt, diese als »echt« oder »gefälscht« einzuordnen, um echte Proben korrekt zu identifizieren und die generierten Proben zurückzweisen zu können.

Training

GANs werden in einem iterativen Prozess trainiert. In jeder Iteration versucht der Generator realistischere Proben zu erzeugen, um den Diskriminator zu täuschen, während der Diskriminator seine Fähigkeit verbessert, echte und gefälschte Daten korrekt zu klassifizieren. Der Trainingsprozess basiert auf Backpropagation zur Gewichtsaktualisierung beider Netze.

Training

Wenn dem Diskriminator eine echte Probe vorgelegt wird, werden seine Gewichte so angepasst, dass die Probe als »echt« klassifiziert wird. Bei einer generierten Probe werden die Diskriminatorgewichte so angepasst, dass er sie als »gefälscht« einordnet. Gleichzeitig berechnet man einen Fehler zwischen der realen und der »echten« Ausgabe des Diskriminators, der durch das Diskriminatornetzwerk propagiert wird, um die Generatorgewichte anzupassen und eher »echt« wirkende Proben zu erzeugen.

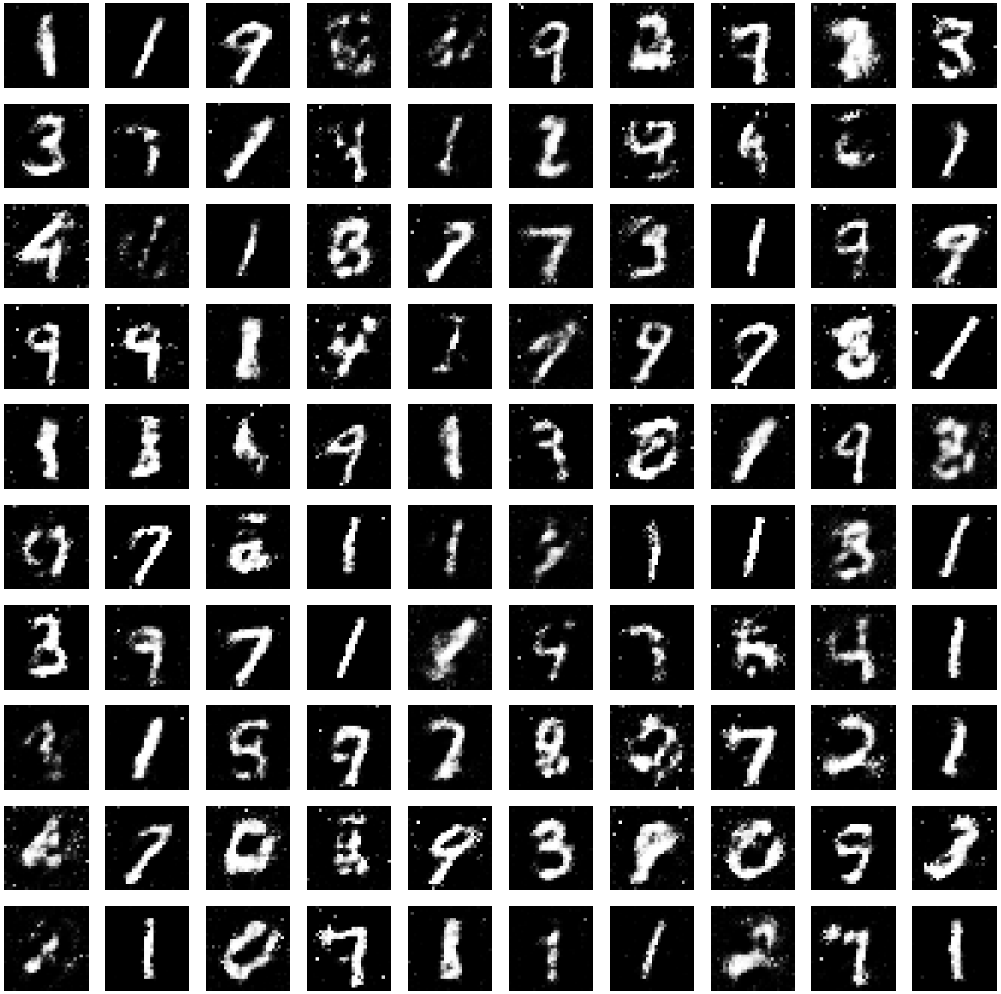


Abbildung: Das GAN-Netzwerk erzeugt MNIST-Ziffern mit lediglich Rauschen am Eingang des Generators.
(Bild aus: Maschinelles Lernen für Dummies, Wiley Verlag)

Rekurrente neuronale Netze

Echo-State-Netzwerk

Echo-State-Netze (ESNs) sind eine Variante von rekurrenten neuronalen Netzen, die eine interessante Methode bietet, um einige der Herausforderungen beim Trainieren von Rückkoppelungsnetzwerken zu überwinden. ESNs bestehen aus einem Reservoir von Neuronen, deren Verbindungen und Gewichte bei der Initialisierung zufällig und dünn besetzt festgelegt und während des Trainings nicht verändert werden.

ESN - Reservoir

- Um in dem Reservoir ein Gedächtnis zu erzeugen, muss es zu einer reichenden Dynamik kommen. Eine vollständige Kopplung der Neuronen würde dies jedoch im Allgemeinen verhindern.
- Daher sind im Reservoir normalerweise nur etwa 5% der Neuronen miteinander verbunden.
- Zusätzlich gibt es die Echo-State Property, die besagt, dass der spektrale Radius der Jakobi-Matrix (das heißt der größte Eigenwert) <1 sein sollte, damit das Netz immer wieder konvergiert (zur Ruhe kommt). Die Gewichtsmatrix sollte jedoch auch so skaliert werden, dass die Eigenwerte möglichst nahe bei eins sind, damit das Netz nicht sofort zur Ruhe kommt.

ESN - Training

Das Trainieren eines Echo-State-Netzes (ESN) zielt darauf ab, die Verbindungswegte zwischen dem Reservoir und den Ausgangsneuronen zu lernen. Da die innerreservoarischen Gewichte konstant bleiben, reduziert sich das Training im Grunde auf eine einfache lineare Regression, häufig mit dem Least-Squares-Algorithmus. Dies führt dazu, dass das Trainieren von ESNs verglichen mit anderen rekurrenten Netzwerken, bei denen alle Gewichte gelernt werden müssen, wesentlich einfacher und schneller ist.

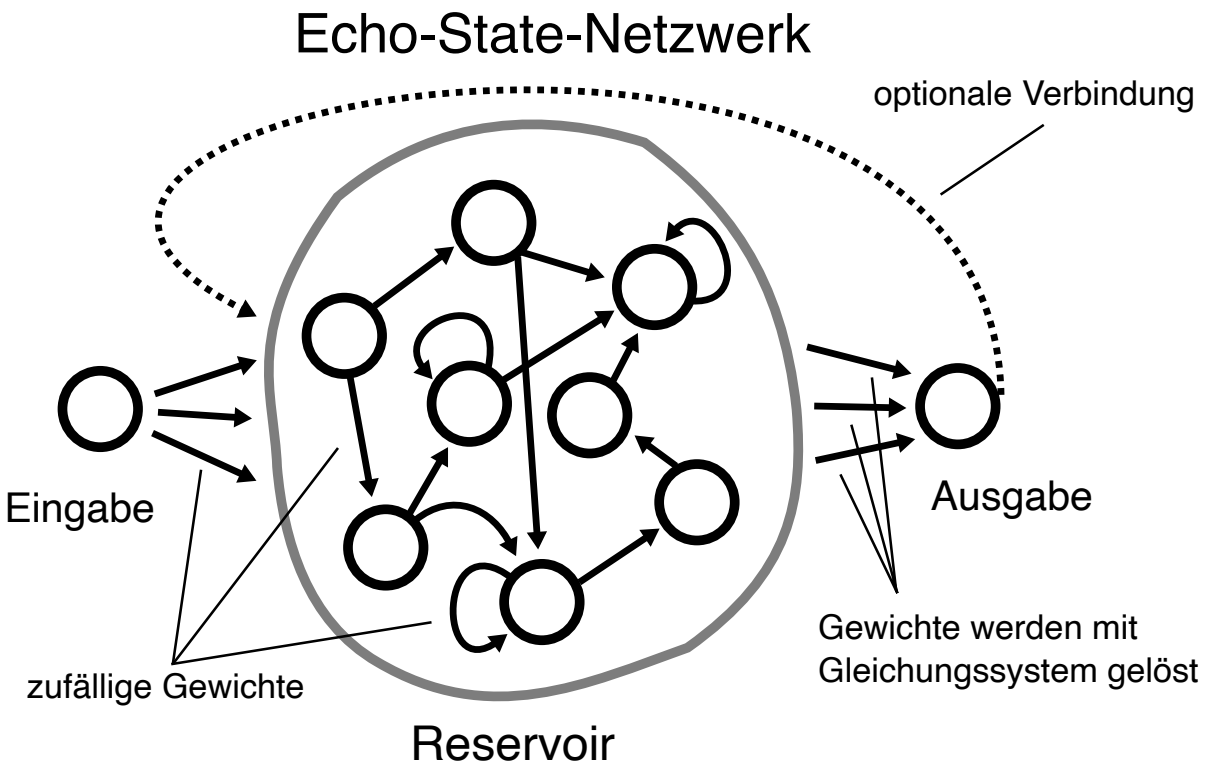


Abbildung: Dies ist ein Echo-State-Netzwerk mit Reservoir, bei dem alle Gewichte zufällig initialisiert sind und die Echo-State-Bedingung erfüllt ist.
(Bild aus: Maschinelles Lernen für Dummies, Wiley Verlag)

ESN - Wofür geeignet?

ESNs haben sich als effektive Lösung für verschiedene Aufgaben erwiesen, insbesondere bei der Modellierung von Zeitreihen und der genauen Vorhersage auf Basis historischer Daten kürzerer Zeiträume. ESNs können besonders Attraktoren mit chaotischem Verhalten herausfordern, die mit anderen Netzwerkarchitekturen schwer zu modellieren sind. Bei gut eingestellten Reservoirgewichten und Hyperparametern liefern ESNs oft viel genauere Vorhersagen als andere neuronale Netze.

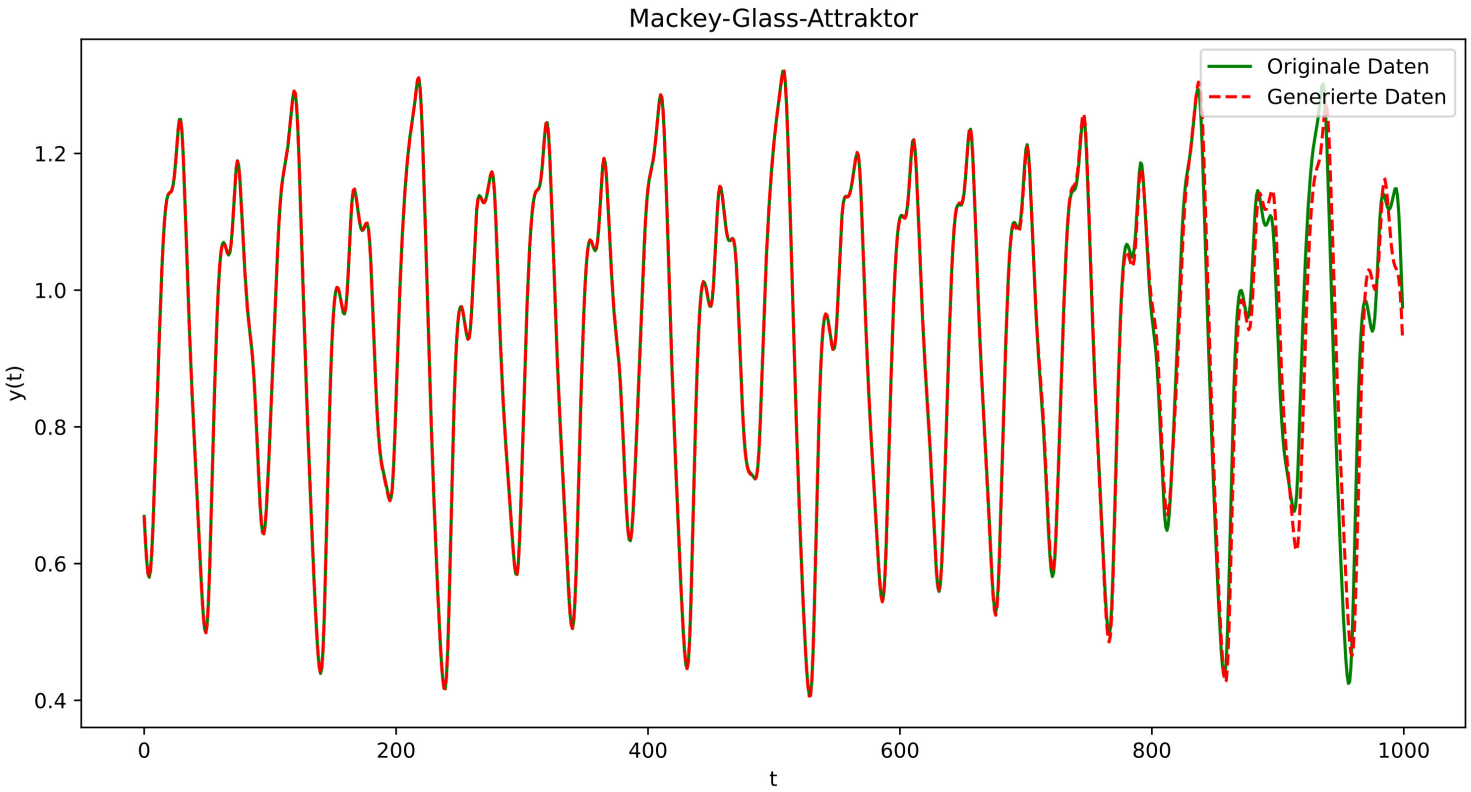


Abbildung: Der Mackey-Glass-Attraktor ist ein chaotischer Attraktor, der für rekurrente Netze eine große Herausforderung darstellt und erst nach einer bestimmten Zeit vom originalen Attraktor abweicht.
(Bild aus: Maschinelles Lernen für Dummies, Wiley Verlag)

Long Short Term Memory

Ein LSTM ist ein rekurrentes neuronales Netzwerk, das gut geeignet ist, um Probleme zu lösen, bei denen der Kontext über längere Zeiträume berücksichtigt werden muss. Im Gegensatz zu herkömmlichen RNNs können LSTM-Netze effektiv lange Zeiträume von Informationen speichern und darauf zugreifen.

LSTM - Memory Zelle

Der Schlüssel zum Verständnis von LSTM-Netzen liegt in der LSTM-Memory-Zelle, die aus vier Hauptkomponenten besteht (Zellzustand, Eingangstor, Vergesstor und Ausgangstor). Diese Komponenten arbeiten zusammen, um Informationen durch das Netzwerk zu schieben und zu modifizieren, was LSTM-Netze geeignet macht, den Kontext über längere Zeiträume zu berücksichtigen.

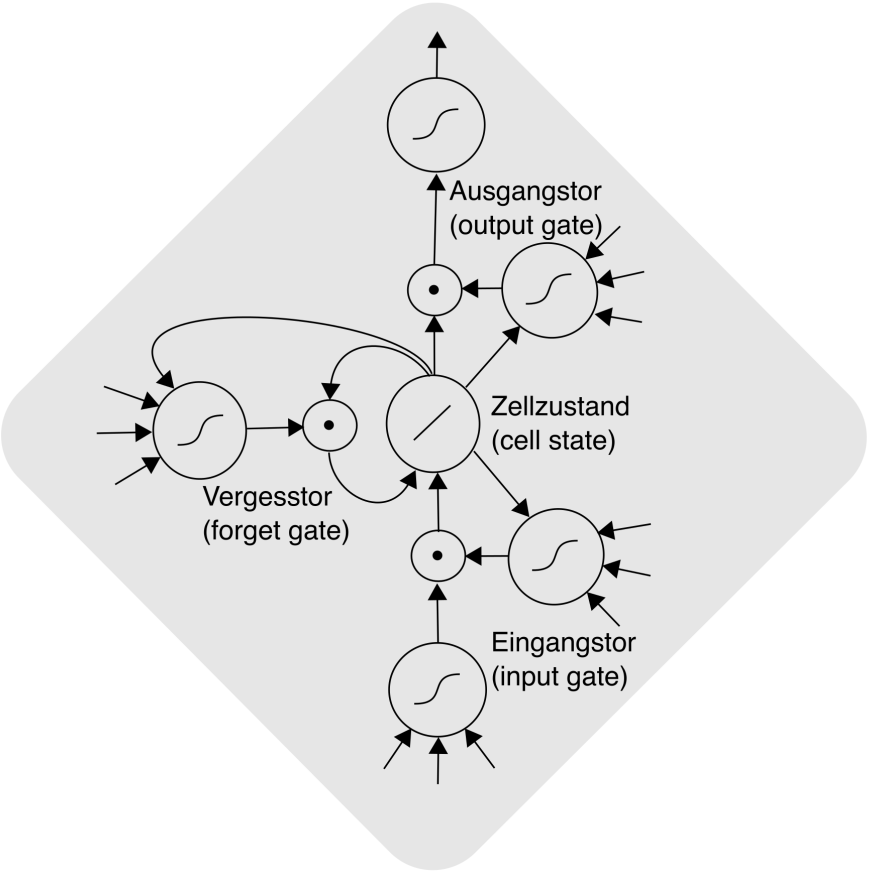


Abbildung: Eine LSTM-Memory-Zelle enthält ein lineares rückgekoppeltes Neuron (cell state), welches einen Fließkommawert dauerhaft speichern kann, solange das »forget-gate« keine Null liefert. Eingabe und Ausgabe können durch das »input gate« und das »output gate« gesteuert werden. (Bild aus: Maschinelles Lernen für Dummies, Wiley Verlag)

LSTM - Lineares Neuron

Im Zentrum der LSTM-Zelle befindet sich ein lineares Neuron, das dazu beiträgt, dass der Fehlergradient beim Backpropagation Through Time ungehindert fließt. Dadurch wird das Problem des verschwindenden oder explodierenden Gradienten, das bei herkömmlichen RNNs schnell zu Problemen führt, weitgehend vermieden.

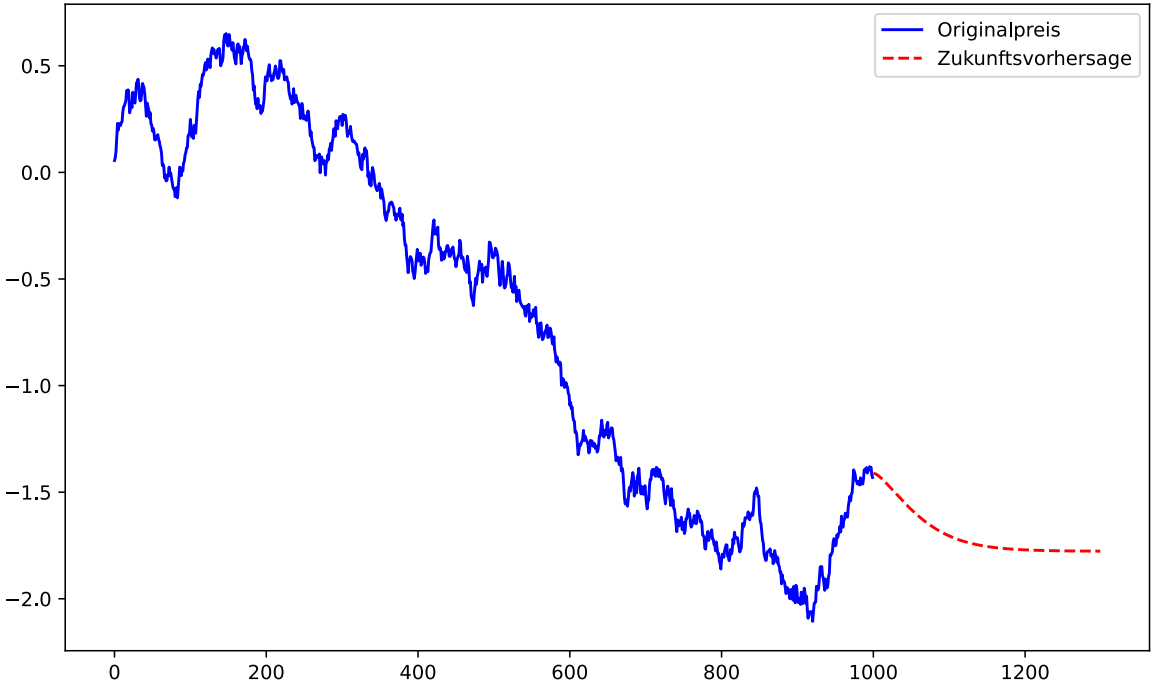


Abbildung: Das Beispiel zeigt einen erfundenen Börsenkurs, der von fünf LSTM-Zellen gelernt wird und eine Vorhersage (gestrichelte Linie) über den weiteren Verlauf gibt. Reale Börsenkurse sind von vielen Faktoren abhängig und die einfache Strategie, einen LSTM-Netz anhand bisheriger Kursverläufe zu vorhersagen, funktioniert nicht wirklich gut.
(Bild aus: Maschinelles Lernen für Dummies, Wiley Verlag)

Aufgabe 5

Programmieren Sie ein rekurrentes Neuronales Netz aus zwei Neuronen mit Tangens hyperbolicus Transferfunktion:

$$o_j = tanh(\sum_i W_{ji} o_i) = \frac{2}{1 + e^{-2\sum_i w_{ji} o_i}} - 1$$

und folgenden Gewichten:
$w_{bias1} = -3.37$, $w_{bias2} = 0.125$,
$w_{11} = -4$, $w_{12} = 1.5$,
$w_{21} = -1.5$, $w_{22} = 0$
und den Anfangswerte: $o_1 = 0.0$, $o_2 = 0.0$
Zeichnen Sie den Output o_1 und o_2 der beiden Neuronen mit Hilfe von Matplotlib in ein zweidimensionales Diagramm.

Transformer

Transformer

Das Transformer-Modell wurde von Google-Mitarbeitern im Jahr 2017 vorgestellt und ist für sequenzielle Datenverarbeitung konzipiert. Er hat sich in Anwendungen wie maschinelle Übersetzung und Sprachmodellierung als äußerst effektiv erwiesen.

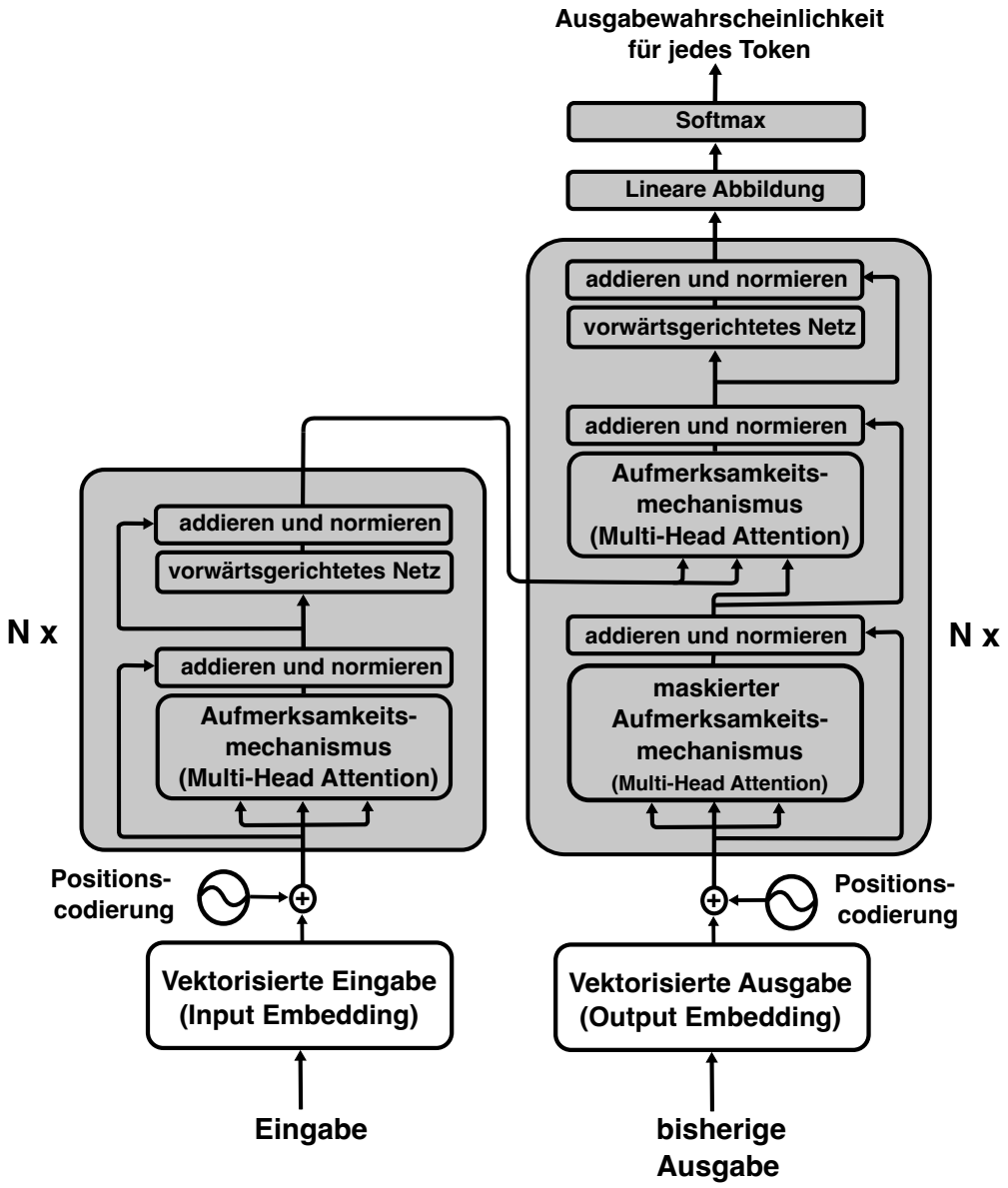


Abbildung: Dies ist die Transformer-Modell-Architektur [Vasvani et. al.]. (Bild aus: Maschinelles Lernen für Dummies, Wiley Verlag)

Attention-Mechanismus

Die Kernidee hinter Transformer-Modellen ist der sogenannte Attention-Mechanismus, ein Aufmerksamkeitsmechanismus, der dem Netzwerk ermöglicht, sich auf Teilbereiche der Eingabesequenz zu konzentrieren. Im Gegensatz zu herkömmlichen rekurrenten neuronalen Netzen (RNN), die die Eingabesequenz Element für Element in einer linearen Reihenfolge verarbeiten, kann der Transformer parallel über die gesamte Sequenz iterieren.

Self-Attention Mechanismus

- Query (Q), Key (K), Value (V):
  $Q = W^Q X, K = W^K X, V = W^V X$
- Aufmerksamkeitsgewichte: $A = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)$
- Ausgabe: $O = AV$

Schlüsselkonzepte

- W^Q, W^K, W^V sind lernbare Gewichtsmatrizen.
- d_k ist die Dimension der Key-Vektoren, dient der Skalierung.
- Der Self-Attention-Mechanismus ermöglicht es, die Relevanz aller Wörter der Sequenz für jedes Wort zu bewerten.

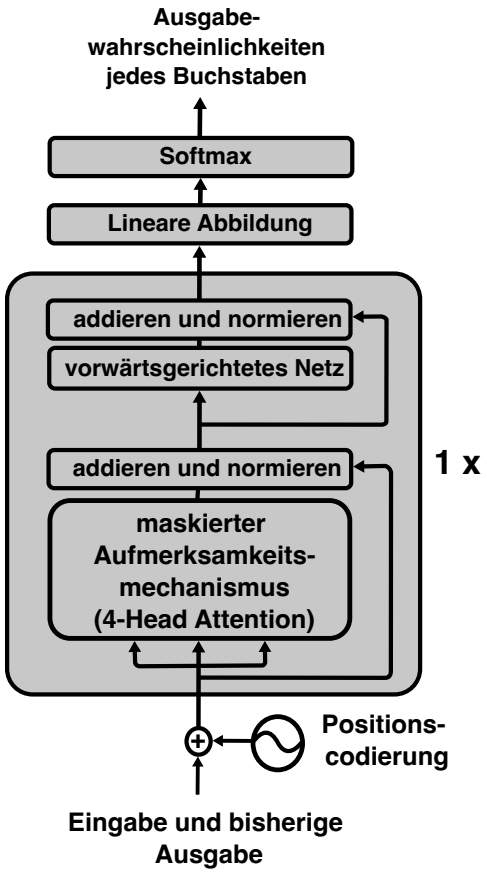


Abbildung: Ein kleines Transformer-Modell verwendet Positional Encoding, maskiertes Multi-Head Attention, Residual-Verbindungen, Normierungen und ein zweischichtiges Feed-Forward-Netz mit Softmax-Ausgabe zur Wahrscheinlichkeitsberechnung des nächsten Tokens (hier Buchstaben) für Textgenerierung wie bei ChatGPT. (Bild aus: Maschinelles Lernen für Dummies, Wiley Verlag)

Anwendungen

Einführung

- Um Neuronale Netze erfolgreich anwenden zu können reicht es nicht, die Theorien dazu zu kennen.
- Vielmehr sollte man eine Vorgehensweise anwenden, die möglichst zuverlässig zum Ziel führt.
- Zusätzlich zum theoretischen Verständnis der neuronalen Netze sind beispielsweise Zielsetzung und Performance-Metriken wichtig.
- Man sollte eine Idee zu einem ersten Prototypen entwickeln.
- Hat man Hyperparameter, die man optimieren möchte, sollte man Randbedingungen beachten.

Performance Metriken

- Zunächst ist die Frage, welches Ziel man in einer Anwendung verfolgt und welche Performance Metriken man verwendet.
- In typischen AI-Anwendungen ist es unmöglich einen Fehler von Null zu erreichen. Der Bayes Fehler gibt dann das theoretische Minimum an, selbst wenn man unendlich viele Trainingsdaten hat.
- Meist ist die Anzahl der Daten begrenzt. Daten sammeln kostet Zeit, Geld und Aufwand. Ist Datengenerierung möglich? (z.B. Bild spiegeln, verschieben, drehen).
- Man muss herausfinden, wann ein Fehler akzeptabel ist? Welche Metrik für die Performanz nutzt man? z.B. Likelihood Maximierung für ein statistisches Modell, Minimierung des quadratischen Fehlers, Varianzminimierung usw.

Erstes Modell

- Ist das Problem (1) statisch oder (2) dynamisch (zeitabhängig)?
- Im Fall 2: Sind ein Zeitfenster von fester Größe gegeben? Ist es ein
  - Klassifikationsproblem
  - eine Rekonstruktion (Vervollständigung) eine Vorhersage
  - ein Übersetzungsproblem
  - ein Reinforcement Problem

Anwendungen

	Klassifikation	Rekonstruktion/ Vorhersage	Übersetzung	Reinforcement Learning
statisch	DNN (z.B. CNN) oder SVM bei unkorellierten Daten	RBM, DNN	GAN, DNN	DNN, RF
Dynamisch mit Zeitfenster	ESN, LSTM, DNN	ESN, LSTM	GAN, DNN, LSTM,ESN	ESN,LSTM, DNN,RF
Dynamisch	LSTM	LSTM	LSTM	LSTM, RF

DNN = Deep Neural Network,
RBM = Restricted Bolzmann Machine,
RF = Reinforcement Learning,
LSTM = Long Short Term Memory
SVM = Support Vector Machines (Kernel Method),
GAN = Generative Adversarial Networks,
ESN = Echo State Network,
CNN = Convolutional Neural Network

Hyperparameter

- Oft lassen sich Hyperparameter separat (einer nach dem anderen optimieren).
- Sind die Parameter aber abhängig, so kann eine Optimierung des einen Parameters auch eine Verschlechterung des anderen bedeuten.
- Lässt man die Hyperparameter automatisch optimieren z.B. mit CMA-ES, dann sollte man eine Zeitabschätzung machen, wie lange ein Satz von Hyperparametern für eine Evaluation braucht. Resultiert eine Zeit von mehreren Tagen für einen Parametersatz, so kommt man mit händischer Optimierung wahrscheinlich schneller zum Ziel.
- Sind es wenige Parameter, so kann man auch systematisch suchen. Dabei sollte man beachten, dass Grid-Search im Allg. schlechtere Ergebnisse liefert, als Zufallsbasiertes suchen.

Debugging

- Zunächst sollte man, wenn man einen Algorithmus nicht zum reinen Verstehen selbst implementiert, Bibliotheken wie z.B. Tensorflow, Keras, Caffe etc. nutzen. Die sind oft hoch optimiert und nutzen z.B. auch die Rechenpower von Grafikkarten.
- Tensorboard liefert eine Vielzahl an Visualisierungsmöglichkeiten um die Entwicklung der Gewichte, um die Fehlerentwicklung und um die zu klassifizierenden Daten zu beobachten.
- Allgemein gilt: Baue dein Programm in kleinen überprüfbaren Methoden auf, verkleinere das Problem (z.B. die Dimensionalität, die Anzahl der Neuronenschichten etc.), Visualisiere alles, was man interpretieren kann!