Architektur Neuronaler Netze für Generative KI

Trainieren Neuronaler Netze

Trainingsdaten, Overfitting,
Regularisierung, Dropouts, etc.

Hochschule Worms • Fachbereich Informatik

Prof. Dr. Stephan Kurpjuweit

Training, Validierung und Testdaten

Typische Aufteilung der Daten beim Training

1) Trainingsdaten (klassisch: 80%)

- Zur Anpassung der Modellparameter

2) Validierungsdaten (klassisch: 10%)

- Zur Überprüfung der Generalisierungsfähigkeit während des Trainings, um eine Überoptimierung (Overfitting) zu vermeiden

3) Testdaten (klassisch: 10%)

- Zur Evaluation des finalen Modells

Quelle: DALL-E

- Bei den aktuell sehr großen Trainingsdatensätzen haben Test- und Validierungsdaten oft einen deutlich kleineren Anteil (< 1%).
- **Wichtig:** Alle drei Datensätze sollten unabhängig voneinander sein, um sicherzustellen, dass das Modell nicht nur die Trainingsbeispiele auswendig lernt, sondern wirklich generalisiert.

K-fache Kreuzvalidierung (k-fold cross validation)

- Der Datensatz wird in k gleiche Teile unterteilt.
- Das Modell wird dann k mal trainiert und validiert, wobei bei jedem Durchgang ein anderer Teil als Validierungsdatensatz, als Testdatensatz und die restlichen als Trainingsdatensatz verwendet werden.

➔ Dies bietet eine gründliche Einschätzung der Modellleistung, da jeder Datenteil für Training, Validierung und Test genutzt wird.

Underfitting und Overfitting

Problemtypen

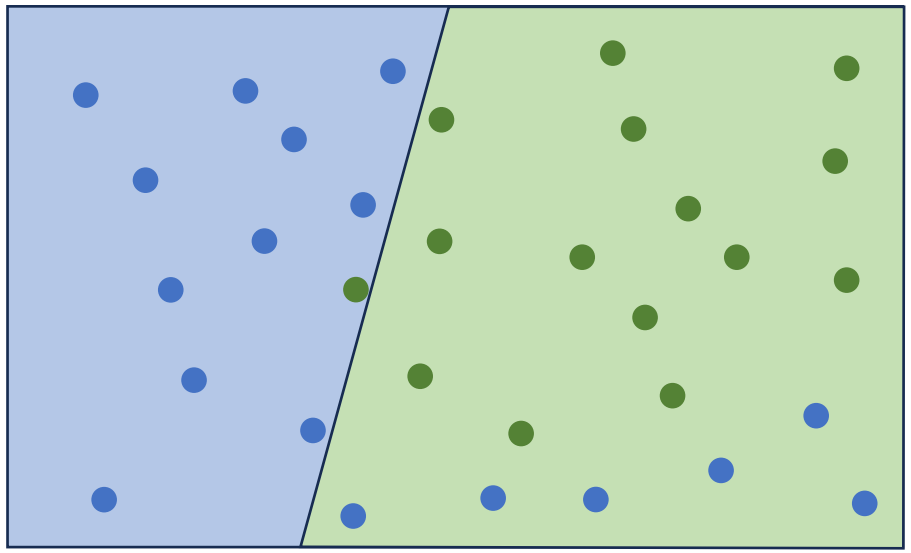
Beim Training der Modelle kann es passieren, dass diese bei den Trainingsdaten sehr gut performen, bei den Testdaten jedoch deutlich schlechter. Zwei häufige Gründe dafür sind:

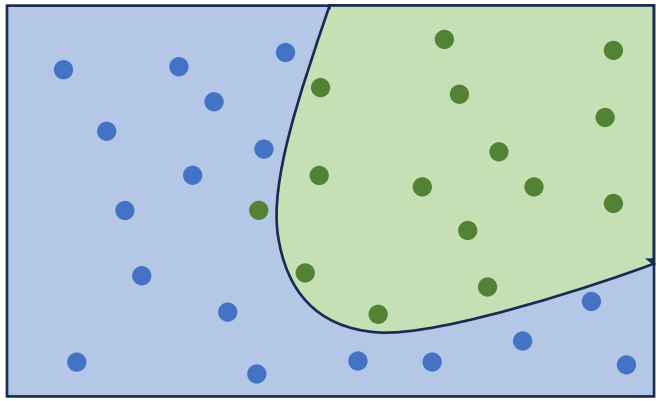
Underfitting (Unteranpassung)

Quelle: DALL-E

Overfitting (Überanpassung)

Underfitting (Unteranpassung)





Ideales Modell

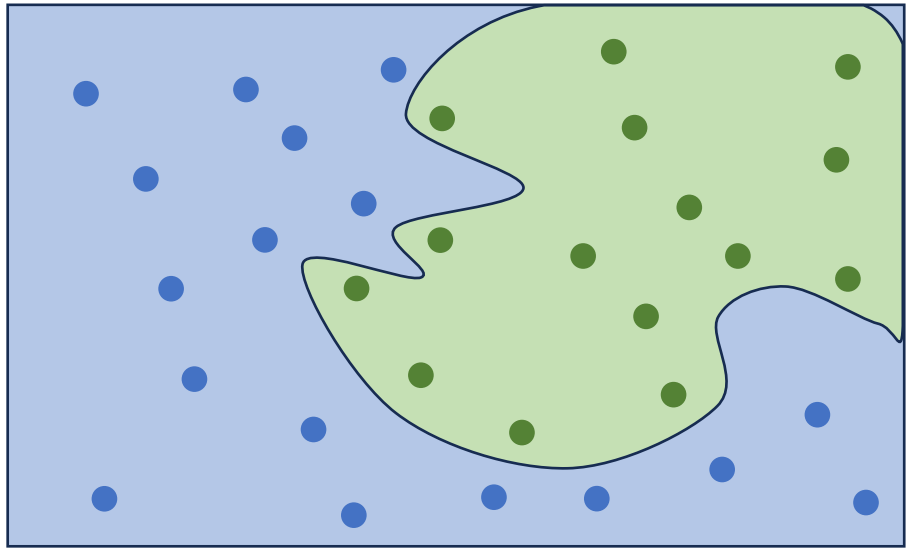
- Das Modell ist zu einfach (hat zu wenige Parameter), um der Komplexität der Trainingsdaten gerecht zu werden.
- Analogie: Das Modell hat nicht genug gelernt.

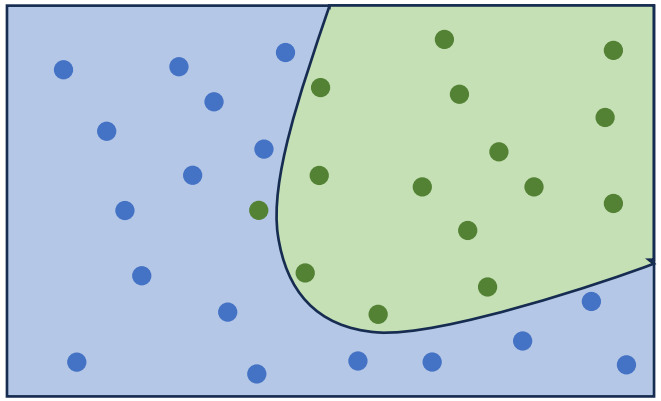
Variante: Fehler durch Verzerrung (error due to bias)

- **Beispiel:** Wir trainieren einen Bildklassifikator, um Katzen zu erkennen.
- **Trainingsdaten:** Als einzige Tierart in den Trainingsdaten kommen Katzen vor.
- **Fehlverhalten:**
  - Vielleicht lernt das Modell eine allzu einfache Regel wie "wenn es vier Beine hat, ist es eine Katze".
  - Wenn wir dann unser Modell mit Daten testen, die andere Tiere enthalten, kann es einen Hund fälschlicherweise als Katze klassifizieren.

Problem: Die Trainingsdaten enthalten einen „Bias“ (Verzerrung) , welcher zu einem zu stark vereinfachten Modell führt.

Overfitting (Überanpassung)





Ideales Modell

- Das Modell lernt die Trainingsdaten zu genau einschließlich des Rauschens und der Ausreißer. Alternative Bezeichnung: Fehler durch Varianz (error due to variance).
- Es passt dann sehr spezifisch für die vorliegenden Trainingsdaten und hat Probleme, neue oder nicht gesehene Daten korrekt vorherzusagen.
- Analogie: Das Modell hat alle Datenpunkte auswendig gelernt, anstatt die zugrundeliegenden Muster zu erkennen.

Fehler durch Varianz (error due to variance)

- **Beispiel:** Wir trainieren einen Bildklassifikator, um Katzen zu erkennen.
- **Trainingsdaten:** Die Trainingsdaten enthalten braune, graue und schwarze Katzen.
- **Fehlverhalten:**
  - Das Modell lernt, braune, graue und schwarze Katzen zu erkennen (das nur diese in den Trainingsdaten vorkommen)
  - Wenn wir dann unser Modell mit Daten testen, die andersfarbige Katzen (z.B. weiße Katzen) enthalten, kann das Modell diese fälschlicherweise als „Nicht-Katze“ klassifizieren.

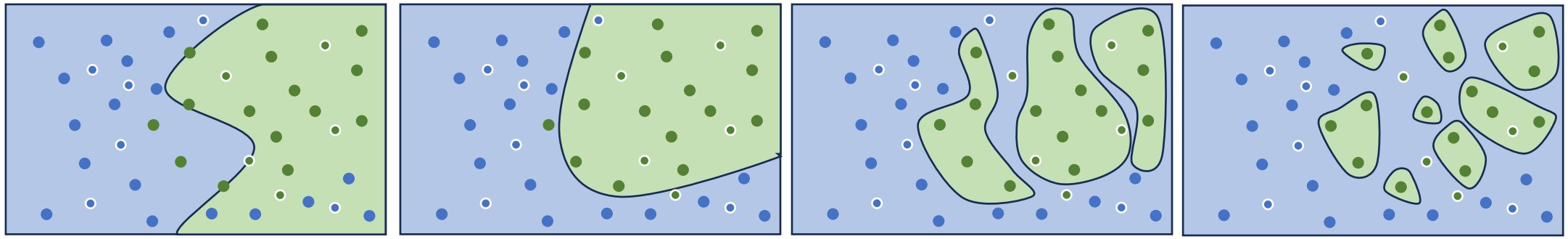
Wahl der Modellgröße

- **Grundregel:** Wir wählen ein (leicht) zu großes Modell und treffen Vorkehrungen, um ein Overfitting zu vermeiden.
- Analogie: Anstatt einer zu kleinen Hose nehmen wir lieber eine leicht zu große Hose und benutzen einen Gürtel um sicherzustellen, dass sie passt.
- Aktuelle KI-Modelle werden zum Teil deutlich überdimensioniert.

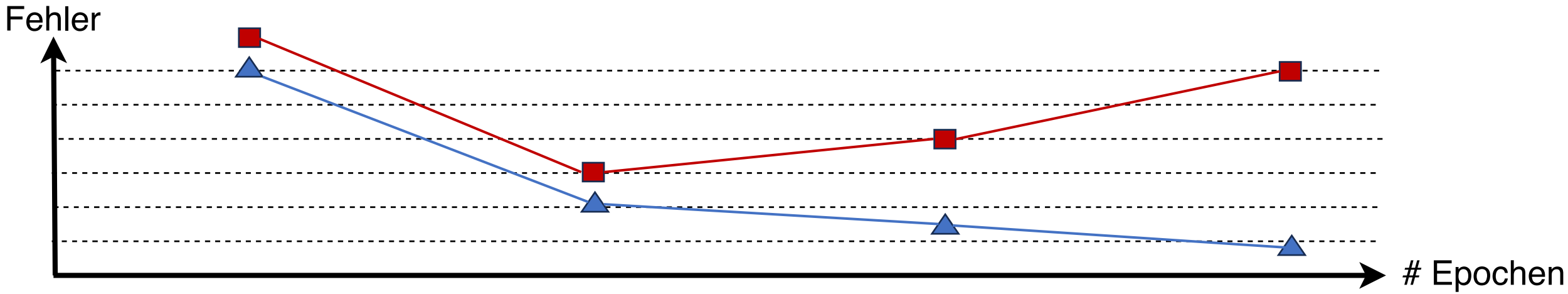
Frühzeitiger Abbuch (Early Stopping)

Entscheidungsgrenze im Laufe des Trainings

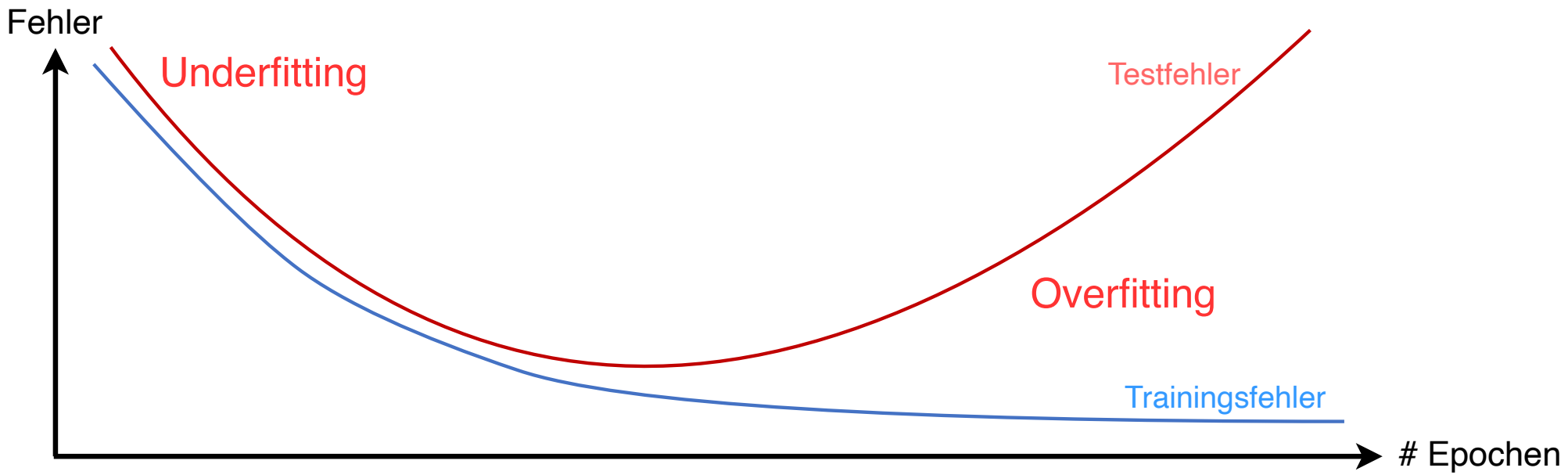
Epoche (Epoch): Ein Durchlauf des Trainingsalgorithmus über den gesamten Datensatz



Epoche 1	Epoche 100	Epoche 500	Epoche 1000
• Testfehler: sehr groß	• Testfehler: klein	• Testfehler: winzig	• Testfehler: winzig
• Trainingsfehler: sehr groß	• Trainingsfehler: klein	• Trainingsfehler: mittel	• Trainingsfehler: groß



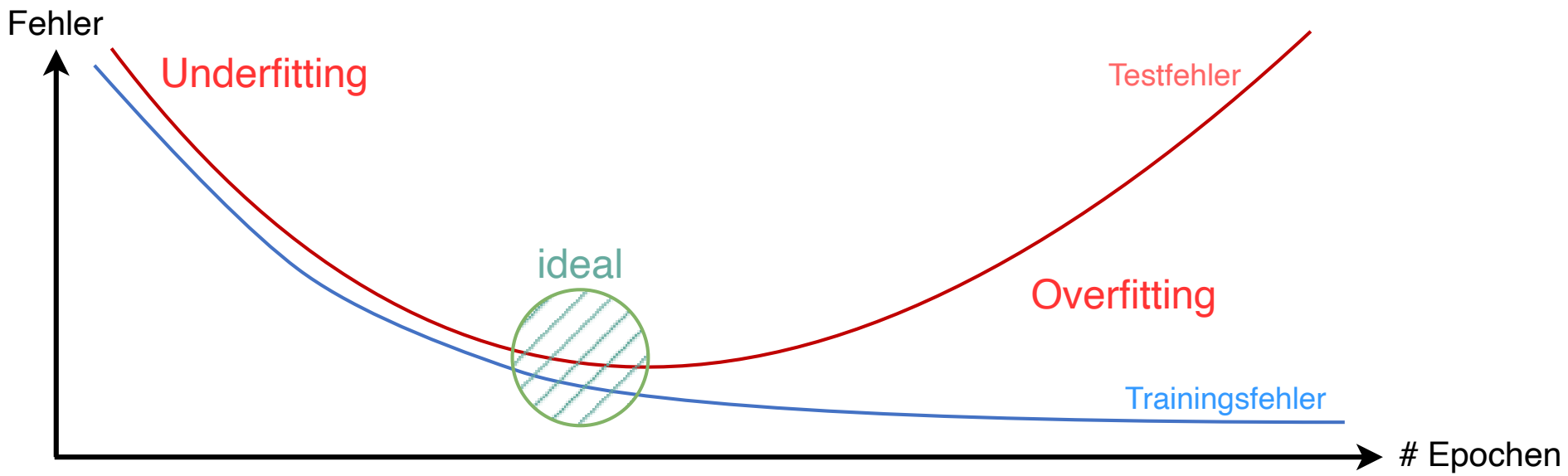
Model Complexity Graph



- Auf der Y-Achse befindet sich ein Maß für den Fehler und auf der X-Achse die Anzahl der Epochen (Trainingsdurchläufe).
- Auf der linken Seite haben wir einen hohen Test- und Trainingsfehler, also eine Unteranpassung (Underfitting).
- Auf der rechten Seite haben wir einen hohen Testfehler und einen niedrigen Trainingsfehler, also eine Überanpassung (Overfitting).

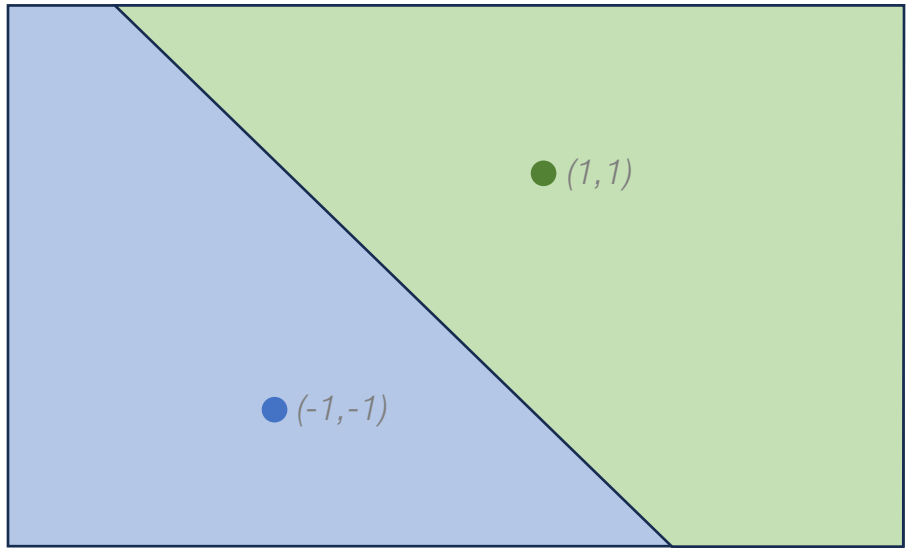
Frühzeitiger Abbruch (early stopping)

Idee: Wir führen den Gradientenabstieg durch, bis der Testfehler nicht mehr abnimmt, sondern wieder zunimmt. In diesem Moment brechen wir das Training ab (early stopping).



Regularisierung (Regularization)

Überlegung: Klassifikation



Welche der beiden Funktionen führt zu besseren Vorhersagen?

$$\hat{y} = \sigma(w_1 x_1 + w_2 x_2 + b)$$

1) $x_1 + x_2$

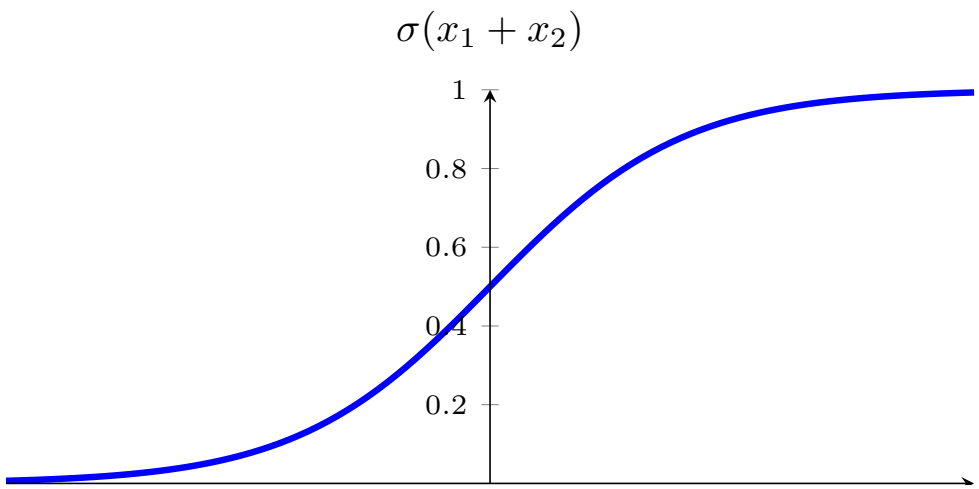
$\sigma(1+1) = 0.88$
$\sigma(-1-1) = 0.12$

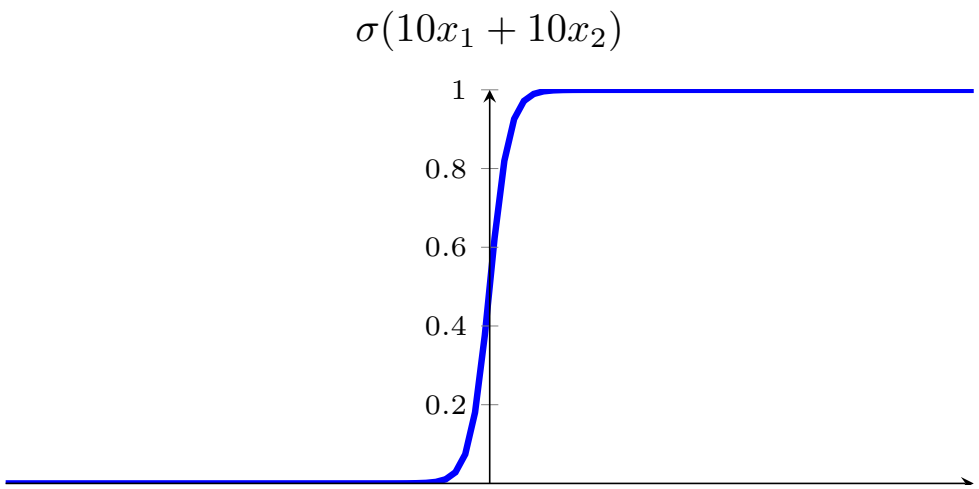
2) $10x_1 + 10x_2$

$\sigma(10+10) = 0.9999999979$
$\sigma(-10-10) = 0.0000000021$

➔ Die zweite Funktion führt zu besseren Vorhersagen.

Aber…





Aber: Die zweite Funktion „overfittet“ und führt zu einer sehr steilen Kurve. Dies erschwert den Gradientenabstieg während des Trainings. Daher ist die erste Kurve zu bevorzugen.

➔ Große Gewichte sollten vermieden werden.

Regulierte Fehlerfunktionen

Idee: Große Gewichte werden mit einer „Penalty“ bestraft.

$$L1 = -\frac{1}{m}\sum_{i=1}^{m}[(1-y_i)\log(1-\hat{y}_i) + y_i\log(\hat{y}_i)] + \lambda\sum_{j=1}^{n}|w_j|$$

$$L2 = -\frac{1}{m}\sum_{i=1}^{m}[(1-y_i)\log(1-\hat{y}_i) + y_i\log(\hat{y}_i)] + \lambda\sum_{j=1}^{n}w_j^2$$

Unterschied zwischen L1 und L2

- L1 hat die Eigenschaft, dass kleine Gewichte gegen Null gehen. L1 kann somit genutzt werden, um die Anzahl der Features zu reduzieren (Feature Selection).

  z.B. (1,0,0,1,0)

- L2 behält alle Gewichte homogen klein. L2 ist daher besser für das Training geeignet und wird meistens verwendet.

  z.B. (0.3,0.2,-0.3,0.6)

Dropouts

Analogie: Mannschaftstraining

- Wenn man bei jedem Spiel und Training immer die stärksten Spieler einsetzt, können diese zwar sehr gut werden, aber die Mannschaft als Ganzes könnte sich zu sehr auf diese Schlüsselspieler verlassen.
  ➔ Darunter kann die Leistung der Mannschaft erheblich leiden, weil die anderen Spieler nicht so viel trainieren oder sich weiterentwickeln können.
- Stellen wir uns vor, dass der Trainer während der Trainingseinheiten nach dem Zufallsprinzip eine andere Gruppe von Spielern auswählt, die bei jeder Einheit aussetzen.
  ➔ Dadurch werden alle Spieler gezwungen, mitzumachen und sich mit der Zeit zu verbessern, nicht nur die stärksten. Dadurch wird jeder Spieler ermutigt, seinen Beitrag zu leisten, was die allgemeine Widerstandsfähigkeit der Mannschaft und ihre Fähigkeit, mit unerwarteten Herausforderungen wie Verletzungen umzugehen, erhöht.

Quelle: DALL-E

Wie passt das zum Training neuronaler Netze?

- Durch die zufällige Initialisierung der Parameter können sich Situationen entwickeln, in denen manche Bereiche des neuronalen Netzes große Parameterwerte haben und das Verhalten dominieren, während andere Bereiche sehr kleine Parameterwerte aufweisen und nicht richtig mittrainiert werden.
- Dropout hilft dabei, indem es zufällig Neuronen während des Trainings "ausschaltet", was das Netzwerk dazu zwingt, redundante Pfade für die gleichen Ausgaben zu bilden.
  ➔ Dies führt zu einem ausgeglicheneren Lernen und einer besseren Generalisierung.

Umsetzung

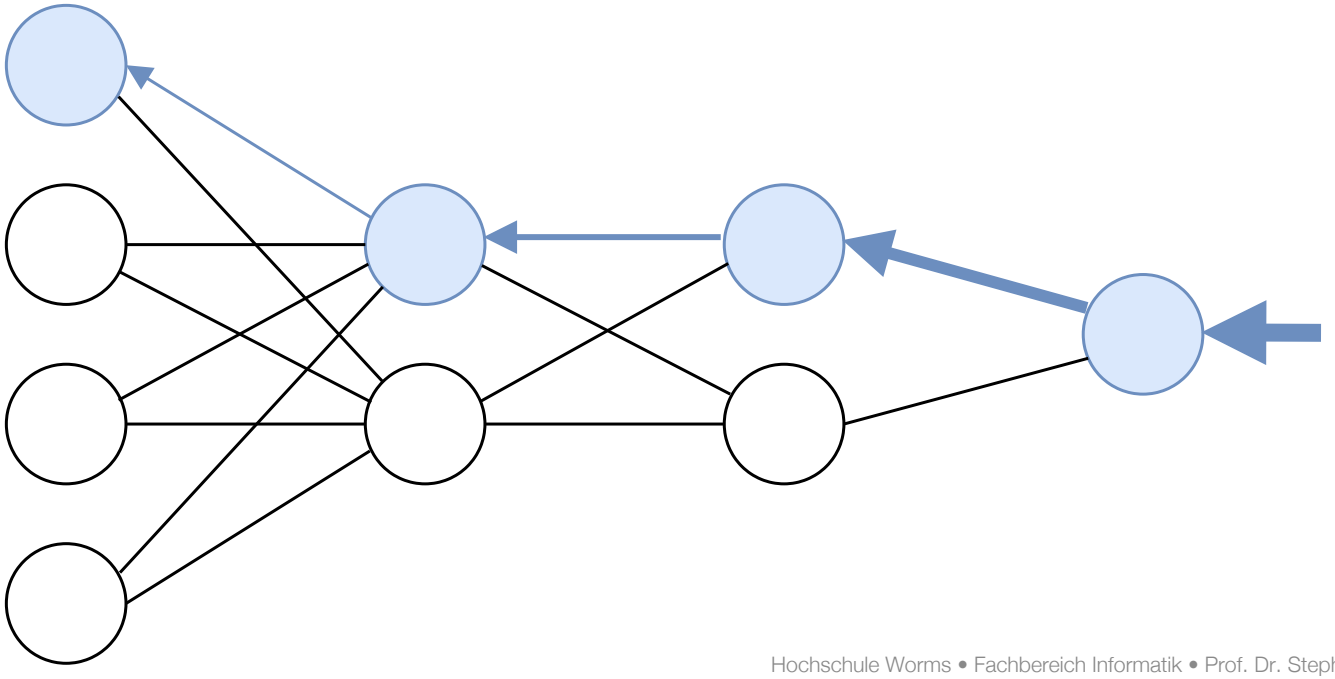
- Wir schalten während des Trainingsprozesses in jeder Epoche zufällig einige der Knoten aus.
- Wir legen dazu einen Parameter fest, der die Wahrscheinlichkeit angibt, mit der ein Knoten während einer Epoche ausgeschaltet wird. Wenn wir diesen Parameter z. B. auf 0,2 setzen, bedeutet dies, dass während jeder Epoche jeder Knoten mit einer Wahrscheinlichkeit von 20 % ausgeschaltet wird.
- Im Durchschnitt sind dann zu jeder Zeit 80% der Knoten aktiv und jeder Knoten ist in 80% der Epochen aktiv.

Verschwindende und explodierende Gradienten (vanishing and exploding gradients)

Verschwindender Gradient (Vanishing Gradient)

Das Phänomen des verschwindenden Gradienten (Vanishing Gradient) kann vor allem in neuronalen Netzwerken mit vielen Schichten (Deep Neural Networks) auftreten.

- **Grund:** Verwendung von „sättigenden "Aktivierungsfunktionen (wie sigmoid oder tanh) in den versteckten Schichten. Eine sättigende Aktivierungsfunktion (saturating activation function) hat einen begrenzen Wertebereich.
- Sättigende Aktivierungsfunktionen haben Bereiche, in denen ihre Ableitungen sehr klein sind (nahe 0). Wenn der Eingabewert einer solchen Funktion sehr groß oder sehr klein ist, wird die Ableitung der Funktion fast null.

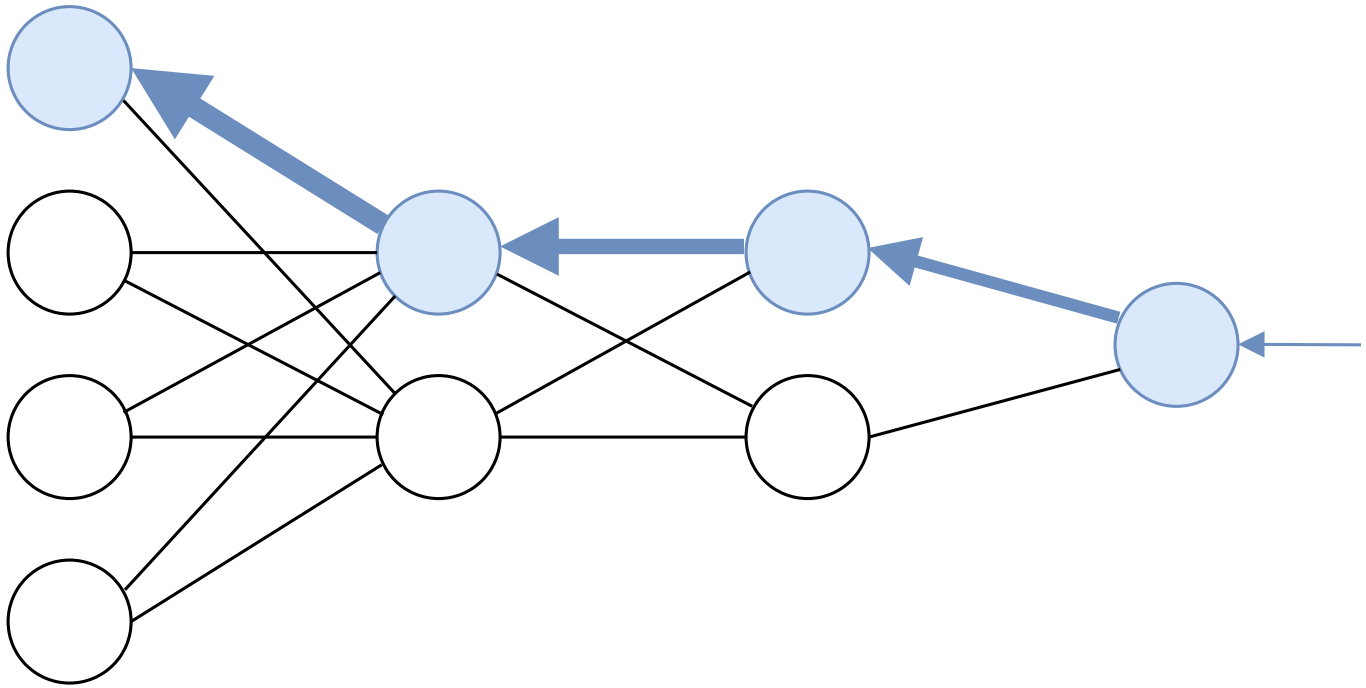


- Die wiederholte Multiplikation dieser kleinen Werte im Rahmen der Kettenregel bei der Backpropagation führt dazu, dass der Gradient für die vorderen Schichten des Netzwerks sehr klein werden.
- ➔Dies führt wiederum dazu, dass die Gewichte in den vorderen Schichten des Netzwerks kaum aktualisiert werden, was das Lernen effektiv verlangsamt oder sogar stoppt.

Explodierender Gradient (Exploding Gradient)

Das Phänomen des explodierenden Gradienten (Exploding Gradient) kann ebenfalls vor allem in neuronalen Netzwerken mit vielen Schichten (Deep Neural Networks) auftreten.

- **Grund:** Verwendung von „nicht-sättigenden “Aktivierungsfunktionen (wie Relu) in den versteckten Schichten. Eine nicht-sättigende Aktivierungsfunktion (non-saturating activation function) hat einen unbegrenzen Wertebereich.
- Nicht-sättigende Aktivierungsfunktionen können Bereiche haben, in denen ihre Ableitungen sehr groß oder zumindest größer 1 ist.



- Die wiederholte Multiplikation dieser Werte im Rahmen der Kettenregel bei der Backpropagation führt dazu, dass der Gradient über die Schichten hinweg exponentiell wächst und für die vorderen Schichten sehr groß werden kann.
- ➔Dies kann zu instabilem Lernverhalten führen.

Wie erkennt und löst man diese Probleme?

Erkennung durch Überwachung der Modellparameter während des Trainings (in Tools wie TensorBoard). Beachte dabei folgende Indizien:

1) Verschwindende Gradienten

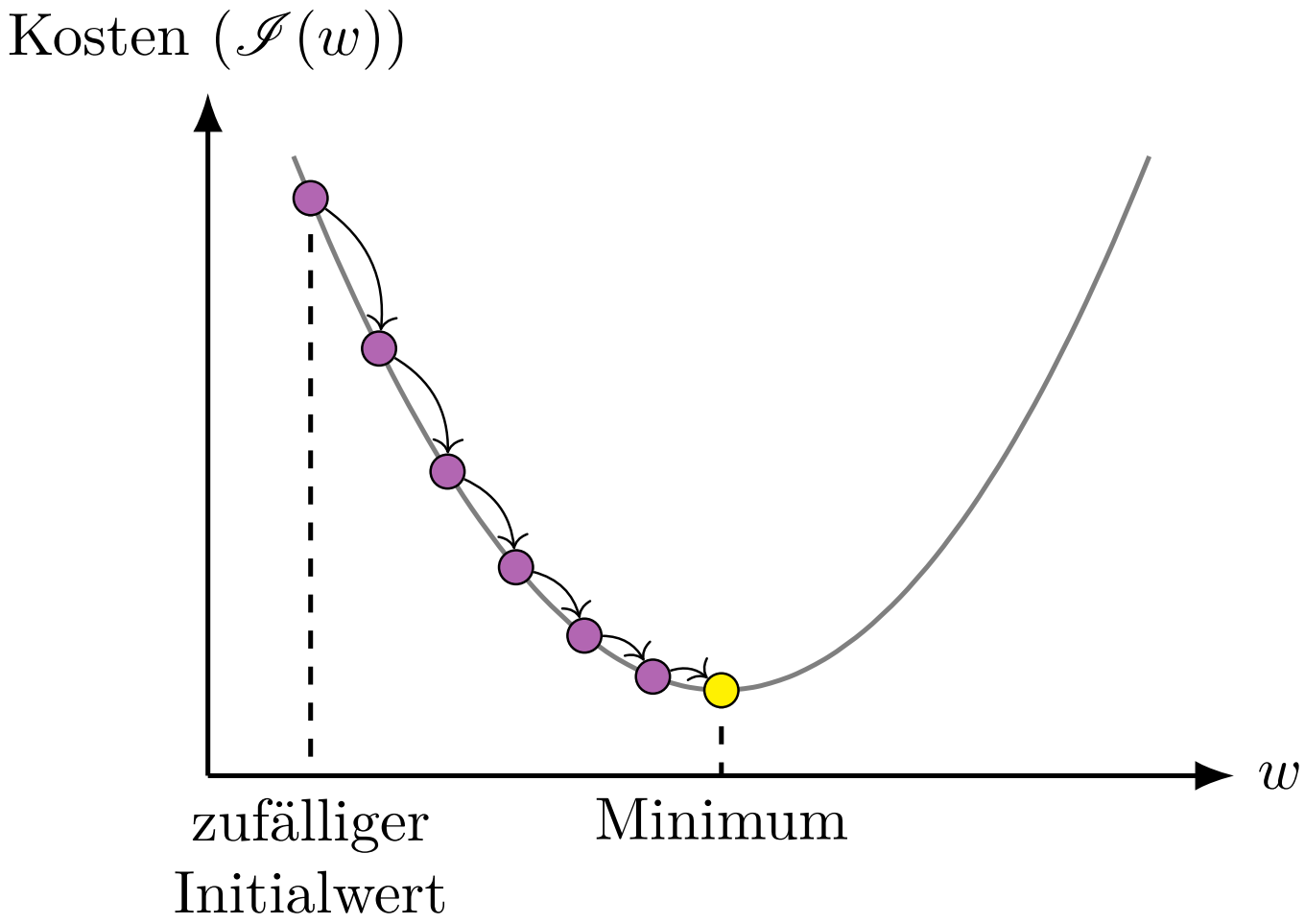
- Die Gewichte der Neuronen fallen und gehen gegen Null.
- Der Verlustwert ist noch hoch, sinkt aber nicht mehr weiter
- Mögliche Abhilfe: Verwende eine andere Aktivierungsfunktion

2) Explodierende Gradienten

- Die Gewichte der Neuronen steigen und werden immer größer
- Instabile / springende Verlustwerte während des Trainings
- Mögliche Abhilfe:
  - Gradientenbeschneidung (Gradient Clipping): Die Größe des Gradienten wird auf einen festgelegten Maximalwert beschränkt.
  - Regularisierung der Gewichte
  - ➔ Diese werden von den Optimierern (z.B. in PyTorch) umgesetzt.

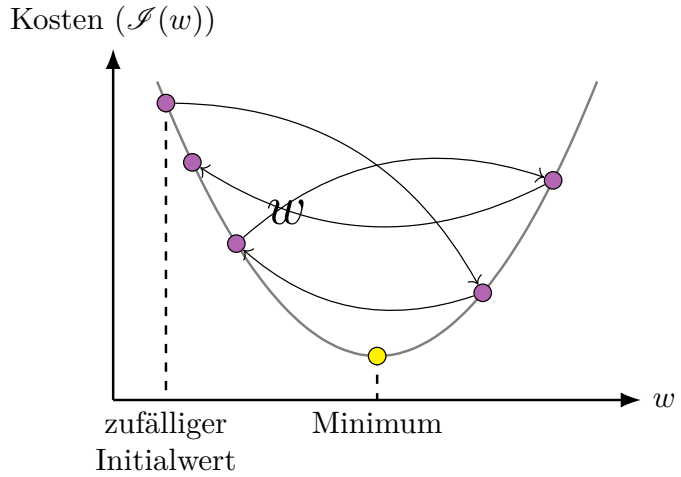
Weitere Optimierungen

Grundprinzip von Gradient Descent



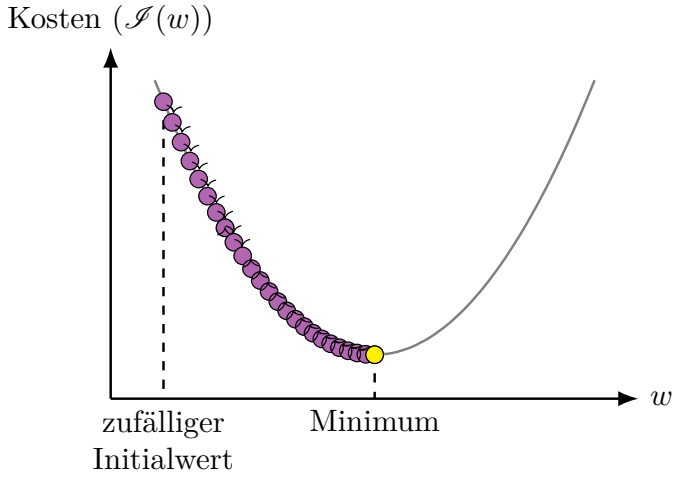
Learning Rate Decay

zu große Lernrate



- Der Algorithmus macht große Schritte und ist daher schnell.
- Er kann jedoch überschießen und das Minimum verfehlen.

zu kleine Lernrate

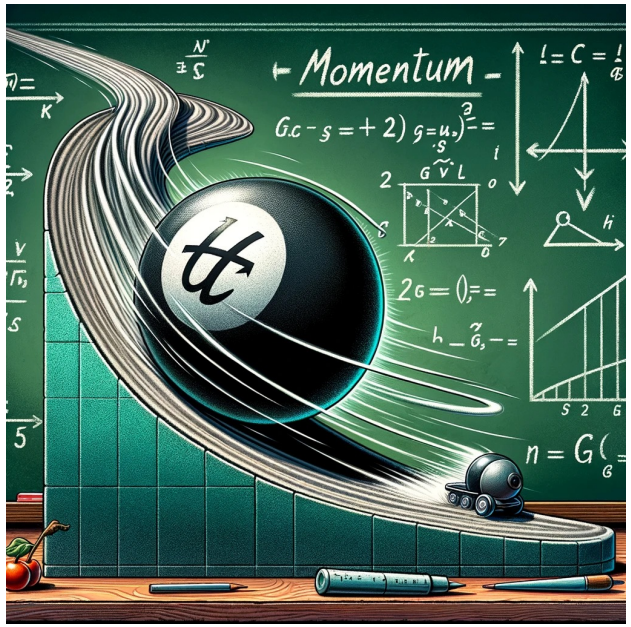


- Der Algorithmus macht kleine Schritte und ich daher langsamer.
- Er macht kontinuierliche Fortschritte und kann das Minimum mit höherer Wahrscheinlichkeit erreichen.

- Faustregel: Wenn das Modell nicht funktioniert, sollte man die Lernrate verringern.
- Learning Rate Decay: Schrittweise Reduktion der Lernrate, während der Algorithmus sich der Lösung annähert.

Momentum

- In der Praxis hat sich ein Ansatz bewährt, um
  1) die Richtung des Gradienten zu stabilisieren,
  2) die Konvergenz zu einem Optimum hin zu beschleunigen und
  3) zu verhindern, dass der Algorithmus in einem lokalen Minimum stecken bleibt.
- Hierzu wird der aktuelle Gradient mit einem Teil der vorherigen Gradienten kombiniert. (Annahme dabei: Die Richtung des Gradienten in der Vergangenheit könnte eine gute Vorhersage für die zukünftige Richtung sein.)



Quelle: DALL-E

β ist eine Konstante zwischen 0 und 1.

Zum aktuellen Schritt wird das gewichtete Mittel der vorherigen Schritte addiert:

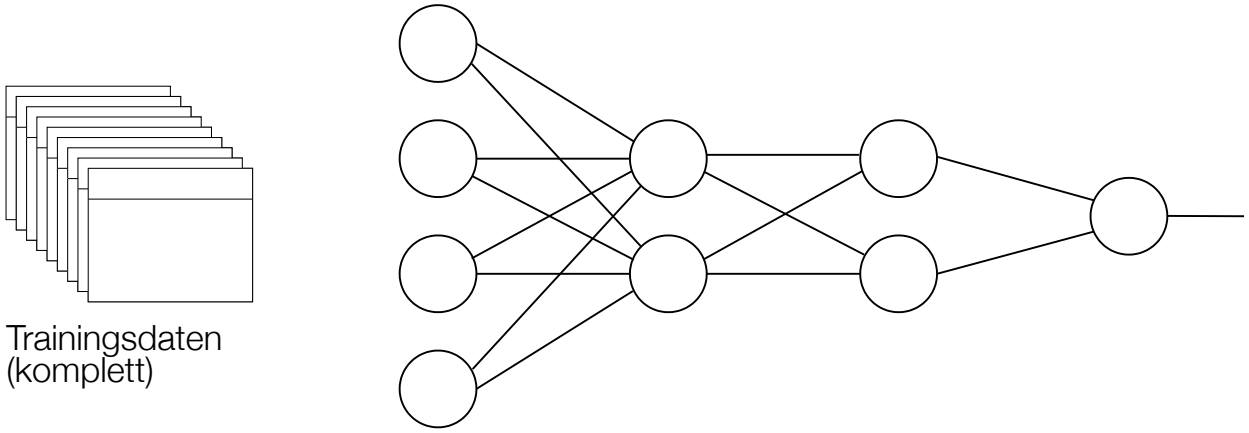
$$Schritt_n + \beta \cdot Schritt_{n-1} + \beta^2 \cdot Schritt_{n-2} + \beta^3 \cdot Schritt_{n-3} + \ldots$$

Optimierer

- Gradient Descent in Reinform ist meist nicht der ideale Optimierungsalgorithmus. Beispielsweise setzt er nicht die Abnahme der Lernrate (Learning Rate Decay) oder Momentum um.
- In Deep Learning-Bibliotheken gibt es vordefinierte Optimierungsalgorithmen (Optimizers), welche diverse Verbesserungen umsetzen (siehe https://pytorch.org/docs/stable/optim.html).
- Die Default-Wahl ist meist Adam - Adaptive Moment Estimation (siehe https://pytorch.org/docs/stable/generated/torch.optim.Adam.html#torch.optim.Adam).

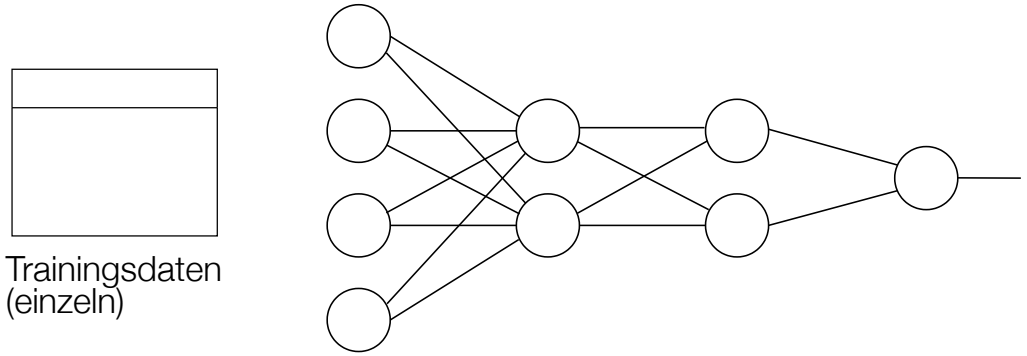
Batch Gradient Descent

- In jedem Schritt (jeder Epoche) verwenden wir **alle Daten des Traingsdatensatzes** und lassen sie durch das gesamte neuronale Netz laufen.
- Dann ermitteln wir unsere Vorhersagen und berechnen den Fehler (wie weit die Vorhersagen von den tatsächlichen Labels entfernt sind).
- Dann werden die Gewichte im neuronalen Netz über Backpropagation aktualisiert. Dadurch erhalten wir eine bessere Entscheidungsgrenze für die Vorhersage unserer Daten.
- Bei einer großen Anzahl von Datenpunkten erfordert dieser Prozess umfangreiche Matrixberechnungen, die viel Speicherplatz beanspruchen würden.



Stochastic Gradient Descent (SGD)

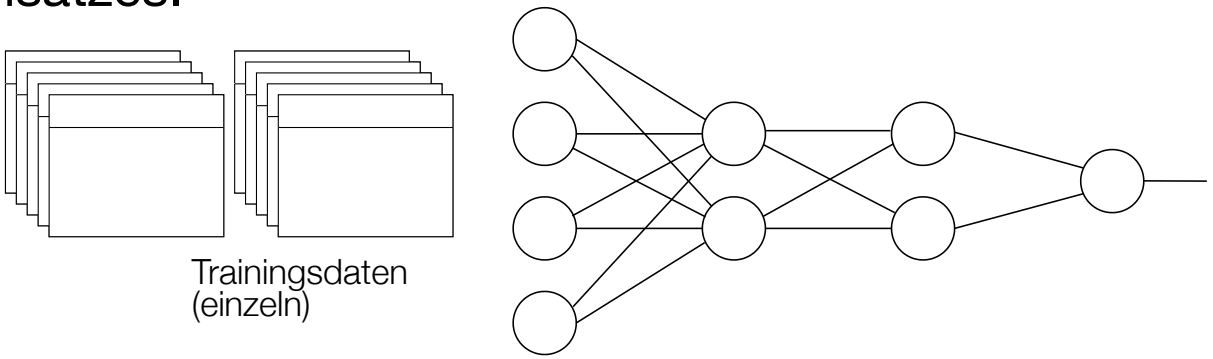
- In jedem Schritt verwenden wir ein einzelnes Datum des Trainingsdatensatzes.



- **Vorteile**
  - **Geschwindigkeit:** Schnellere Parameteraktualisierung, da nur ein Datenpunkt betrachtet wird.
  - **Randomisierung:** Hilft, aus lokalen Minima herauszukommen, weil zufällige Störungen durch einzelne Datenpunkte eingeführt werden.
- **Nachteile**
  - **Instabilität:** Kann zu starken Schwankungen im Konvergenzprozess führen, da die Richtung durch individuelle Datenpunkte beeinflusst wird.
  - **Genauigkeit:** Das finale Minimum kann weniger präzise sein.

Mini-Batch Gradient Descent

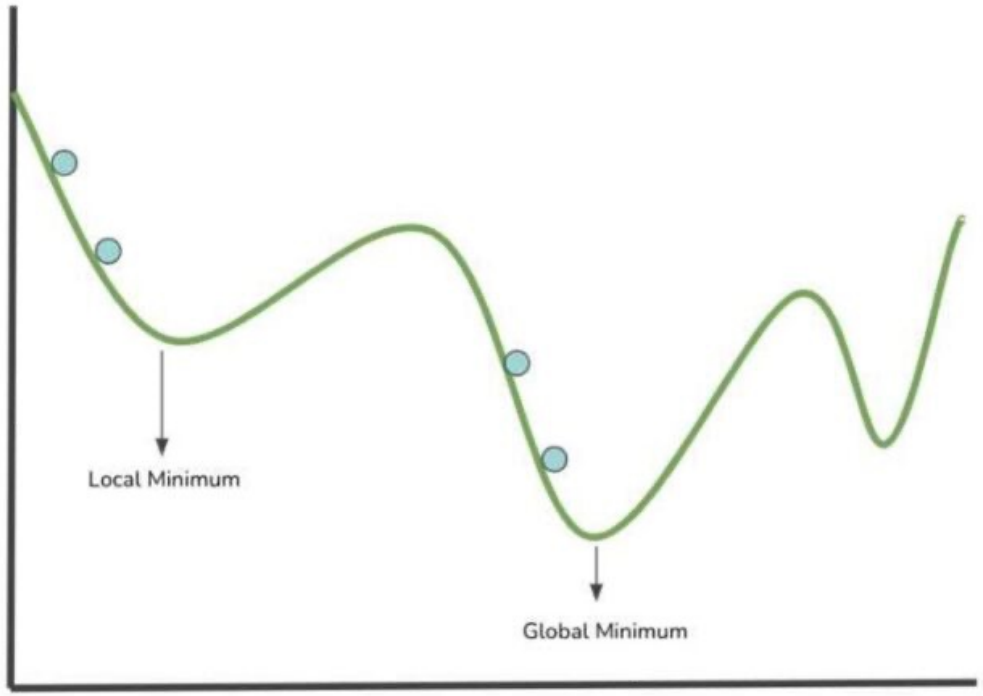
- In jedem Schritt verwenden wir eine Teilmenge (einen Mini-Batch) des Trainingsdatensatzes.



- **Vorteile**
  - **Kompromiss:** Kombiniert Vorteile von Batch Gradient Descent und SGD, indem es die Geschwindigkeit von SGD mit der Stabilität und Effizienz von Batch Gradient Descent verbindet.
  - **Parallelisierung:** Erlaubt eine effiziente Nutzung moderner Hardware, z.B. GPUs.
- **Nachteile**
  - **Parametereinstellung:** Die optimale Größe des Mini-Batches hängt vom Datensatz und Problem ab und erfordert möglicherweise Experimente.

Zufälliger Neustart (Random Restart)

Um die Wahrscheinlichkeit zu erhöhen, das globale Minimum oder zumindest ein ziemlich gutes lokales Minimum zu finden, starten wir den Gradientenabstieg von mehreren zufälligen Stellen aus.



https://developer.nvidia.com/blog/a-data-scientists-guide-to-gradient-descent-and-backpropagation-algorithms/