

Grundlagen des maschinellen Lernens

Neuronale Netze II

Ole Meyer

Formel

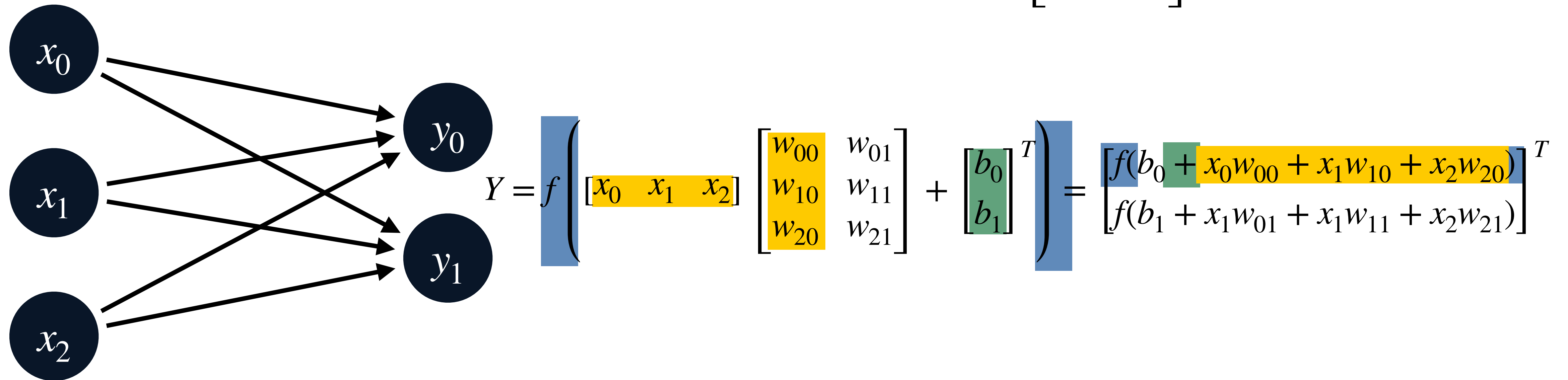
$$Y = f(X^T W + B)$$

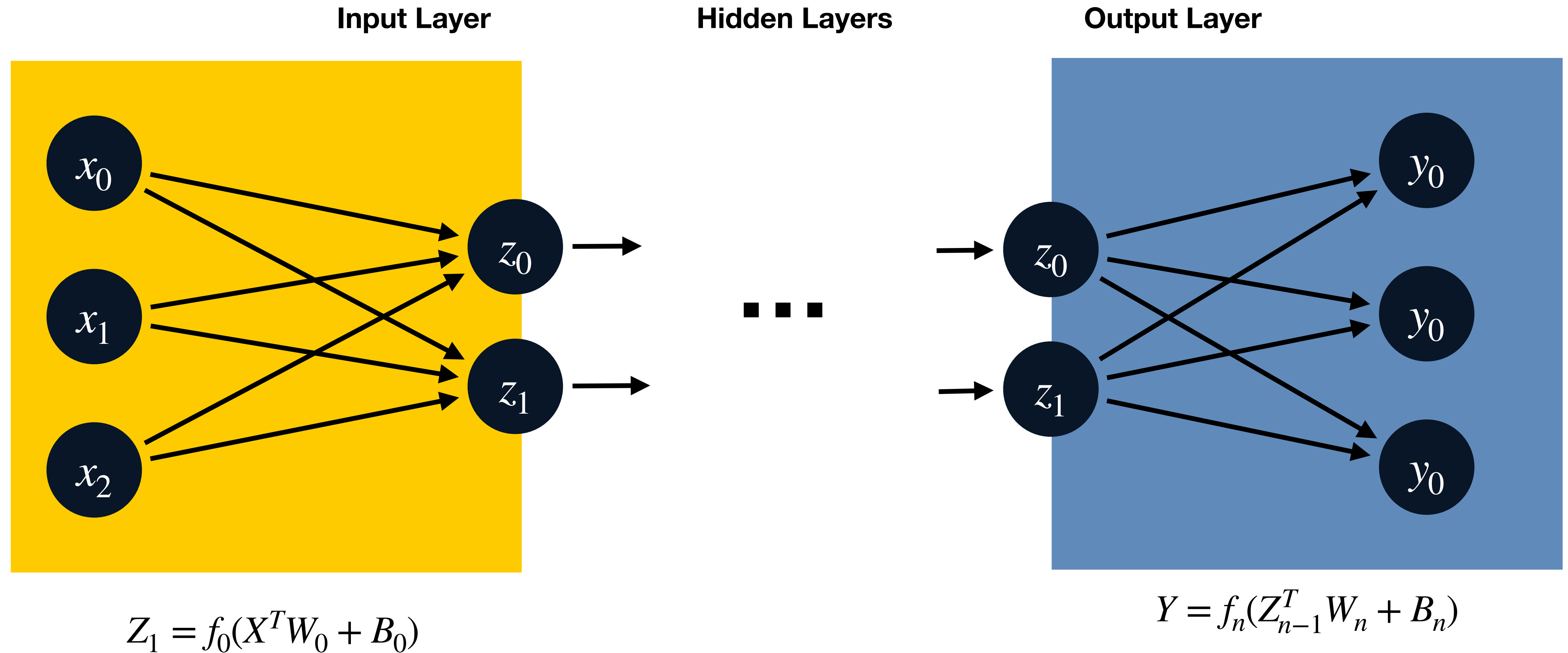
Eingabe & Parameter

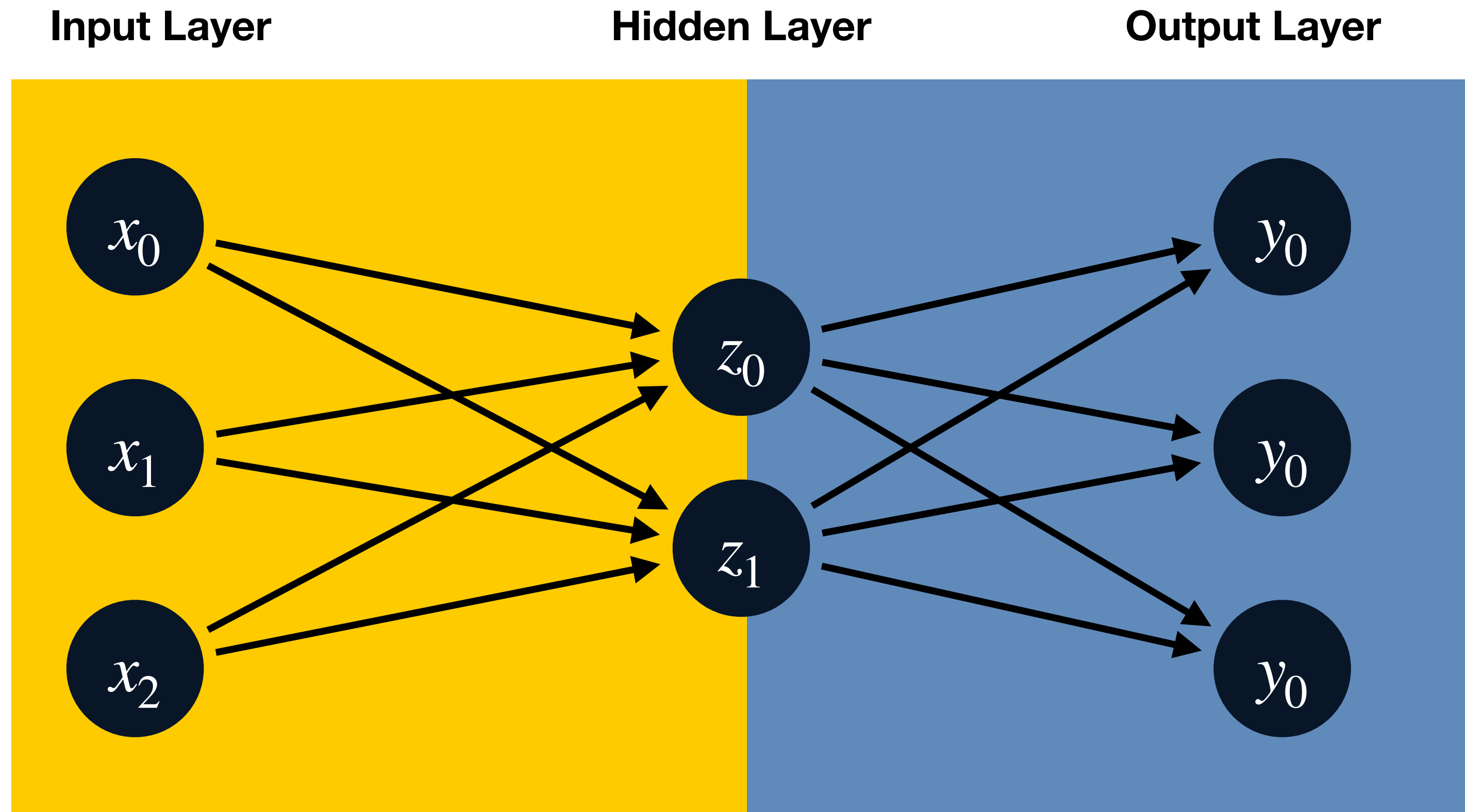
$$X^T = [x_0 \quad x_1 \quad x_2]$$

$$W = \begin{bmatrix} w_{00} & w_{01} \\ w_{10} & w_{11} \\ w_{20} & w_{21} \end{bmatrix}$$

$$B = \begin{bmatrix} b_0 \\ b_1 \end{bmatrix}$$

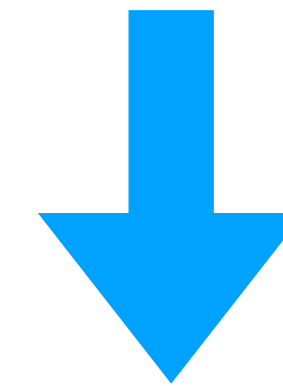






Bis jetzt:

$$Y = f_1((f_0(X^T W_0 + B_0))^T W_1 + B_1)$$

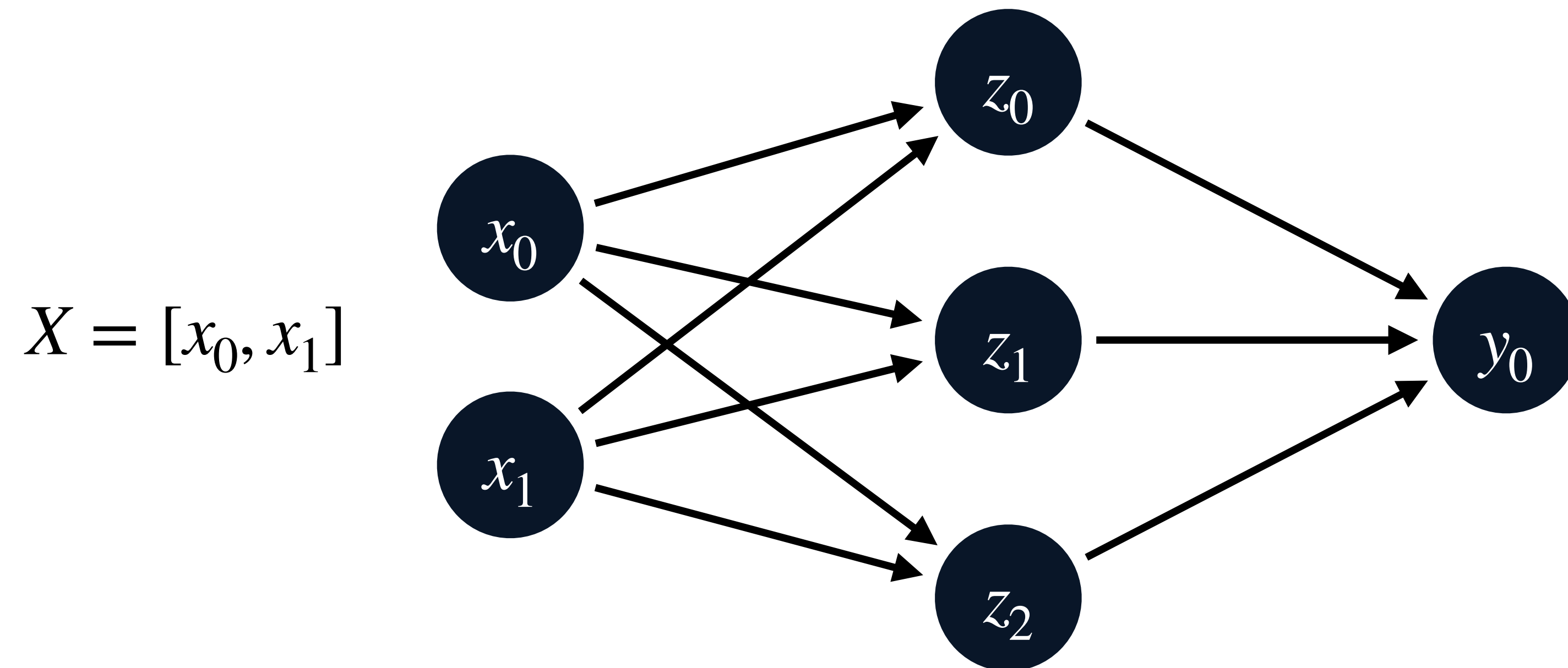


Einfacher:

$$Y = f(X; \theta)$$

- Ermöglicht einfacher über komplexe Netzwerke zu sprechen
- Interpretation:
 - $f(\cdot)$ = Die Funktion des gesamten Netzwerkes: Alle bis jetzt besprochenen Vorgänge.
 - θ = Alle Parameter die benötigt werden, also alle W und B
 - X = Die Eingabe

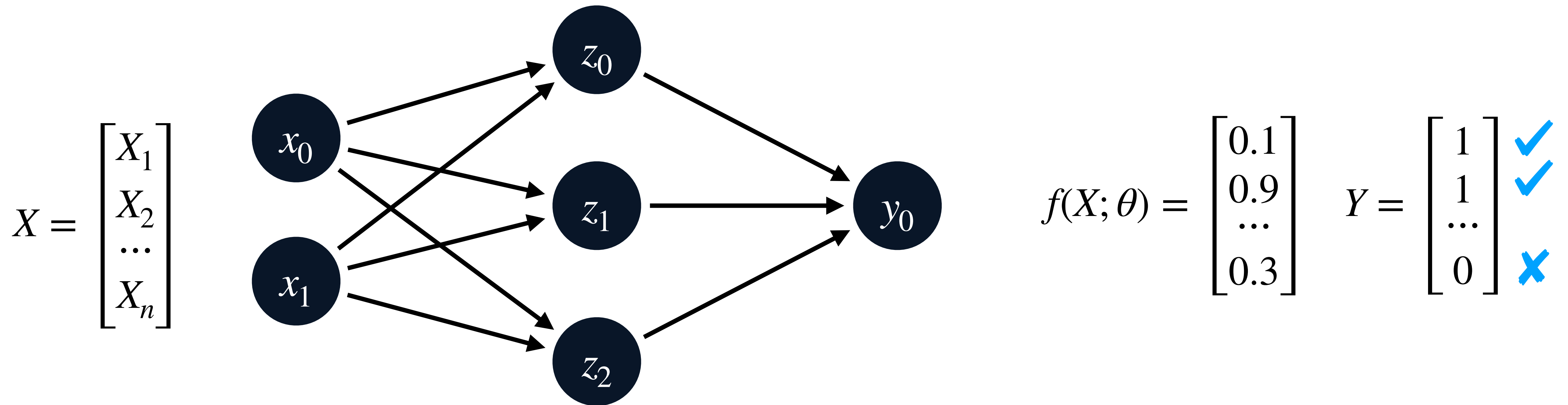
Fehler Quantifizieren



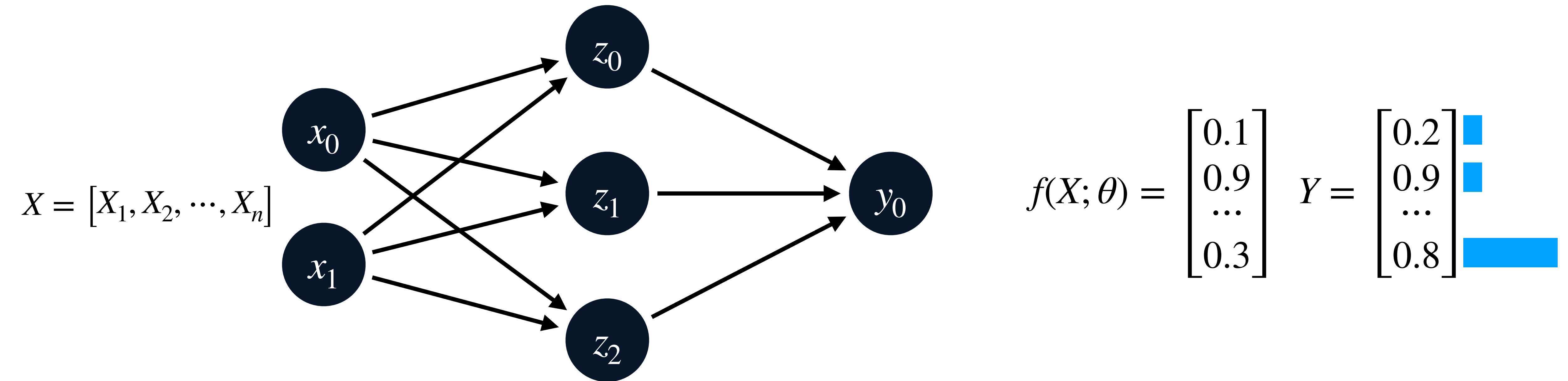
- Vorhergesagter Wert = $Y = [y_0]$

- Zielwert = $Y^* = [y_0^*]$

$$L(f(X; \theta), Y^*)$$

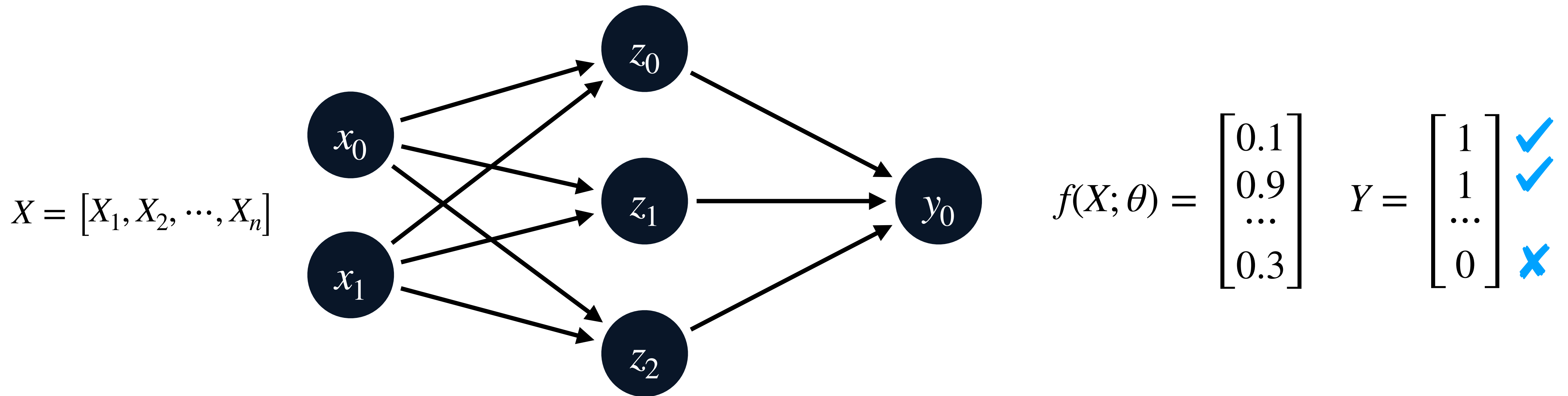


$$J(\theta) = \frac{1}{n} \sum_{i=1}^n L(f(X_i; \theta), Y_i^*)$$



$$J(\theta) = \frac{1}{n} \sum_{i=1}^n (Y_i - f(X_i; \theta))^2$$

Binary Cross Entropy Loss



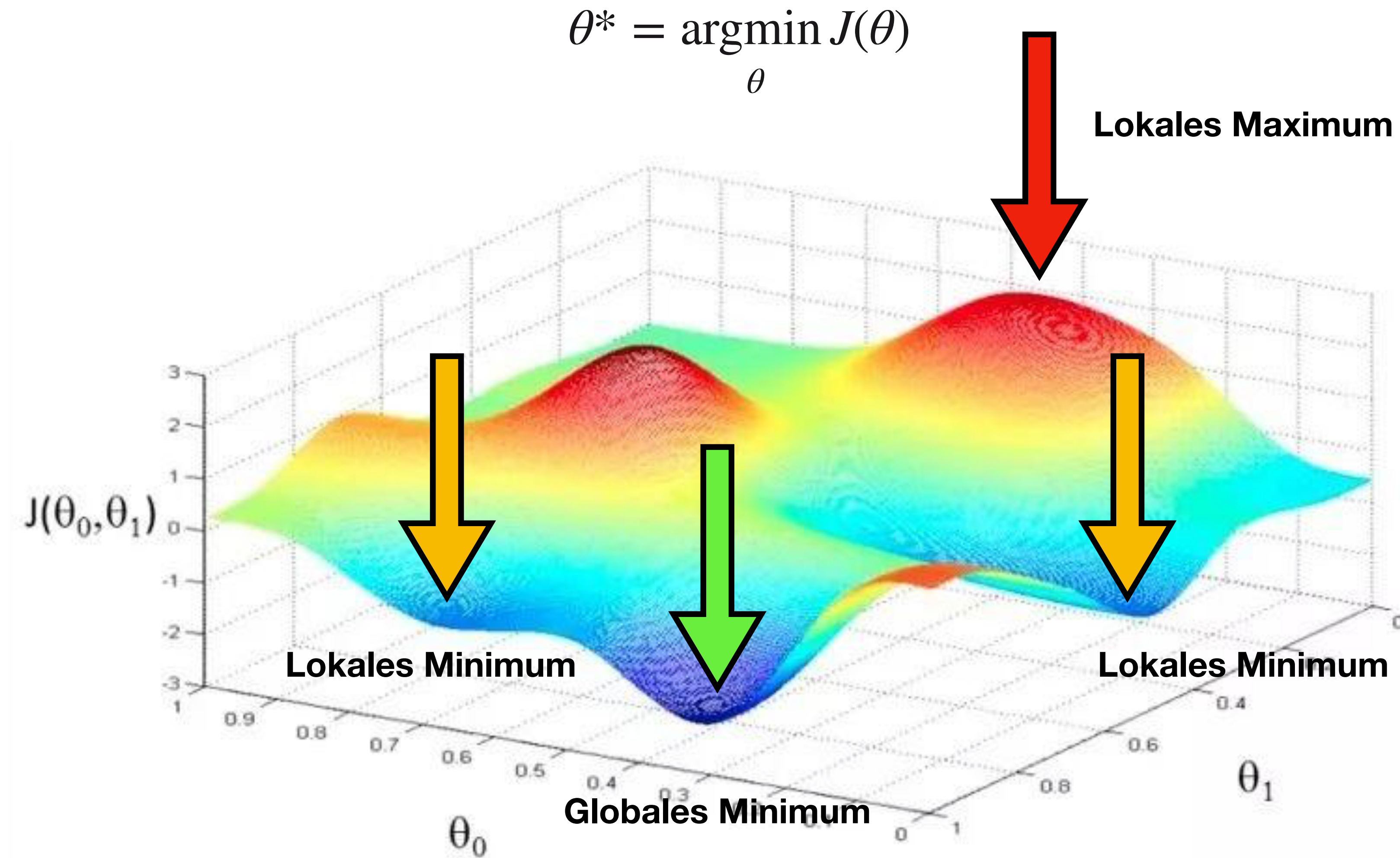
$$J(\theta) = -\frac{1}{n} \sum_{i=1}^n Y_i \log (f(X_i; \theta)) + (1 - Y_i) \log (1 - f(X_i; \theta))$$

Training von Neuronalen Netzen

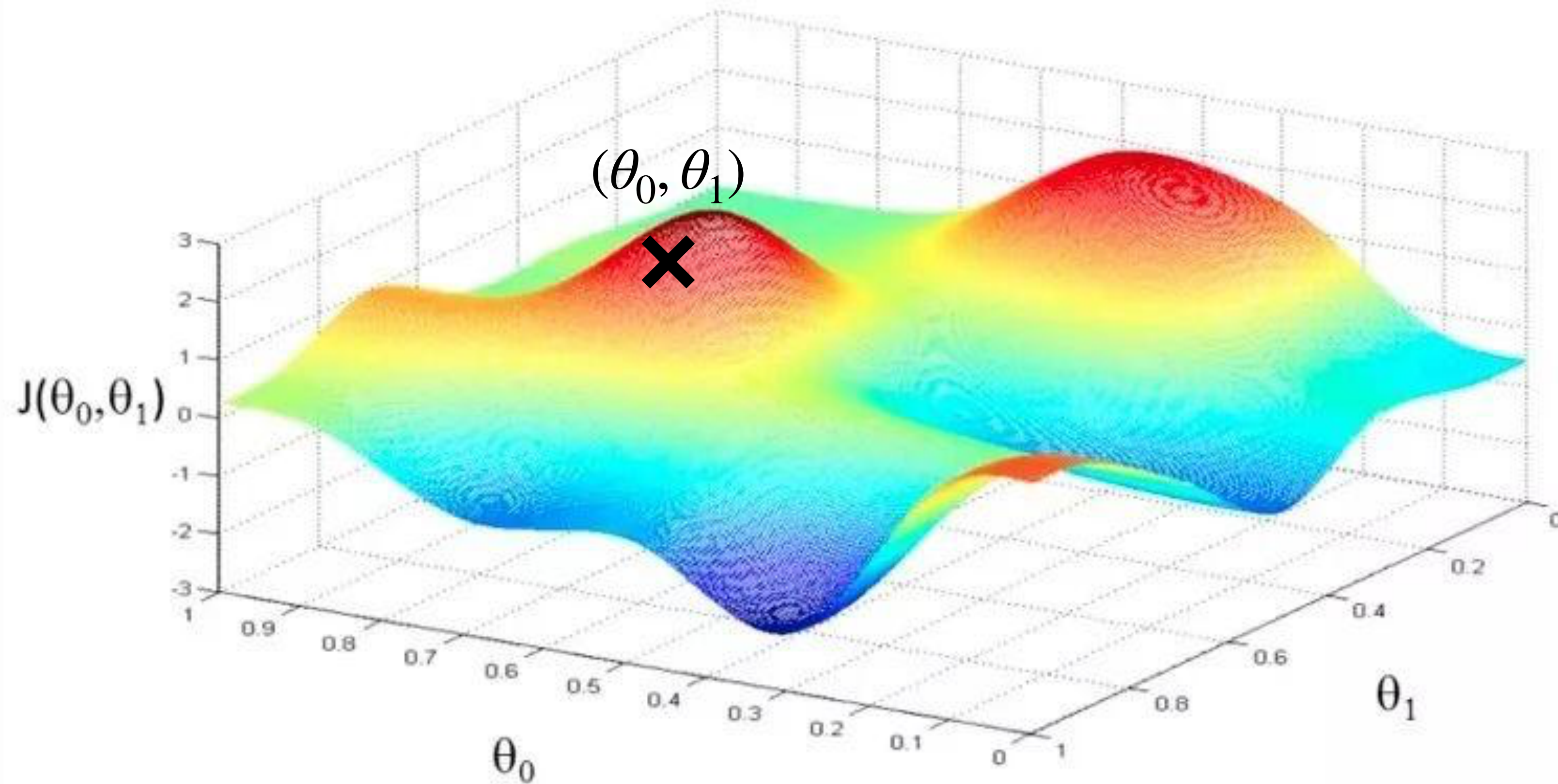
Ziel: Gesucht werden die Parameter für das Netzwerk, welche den **geringsten Fehler** erzeugen.

$$\theta^* = \operatorname{argmin}_{\theta} \frac{1}{n} \sum_{i=1}^n L(f(X_i; \theta), Y_i)$$

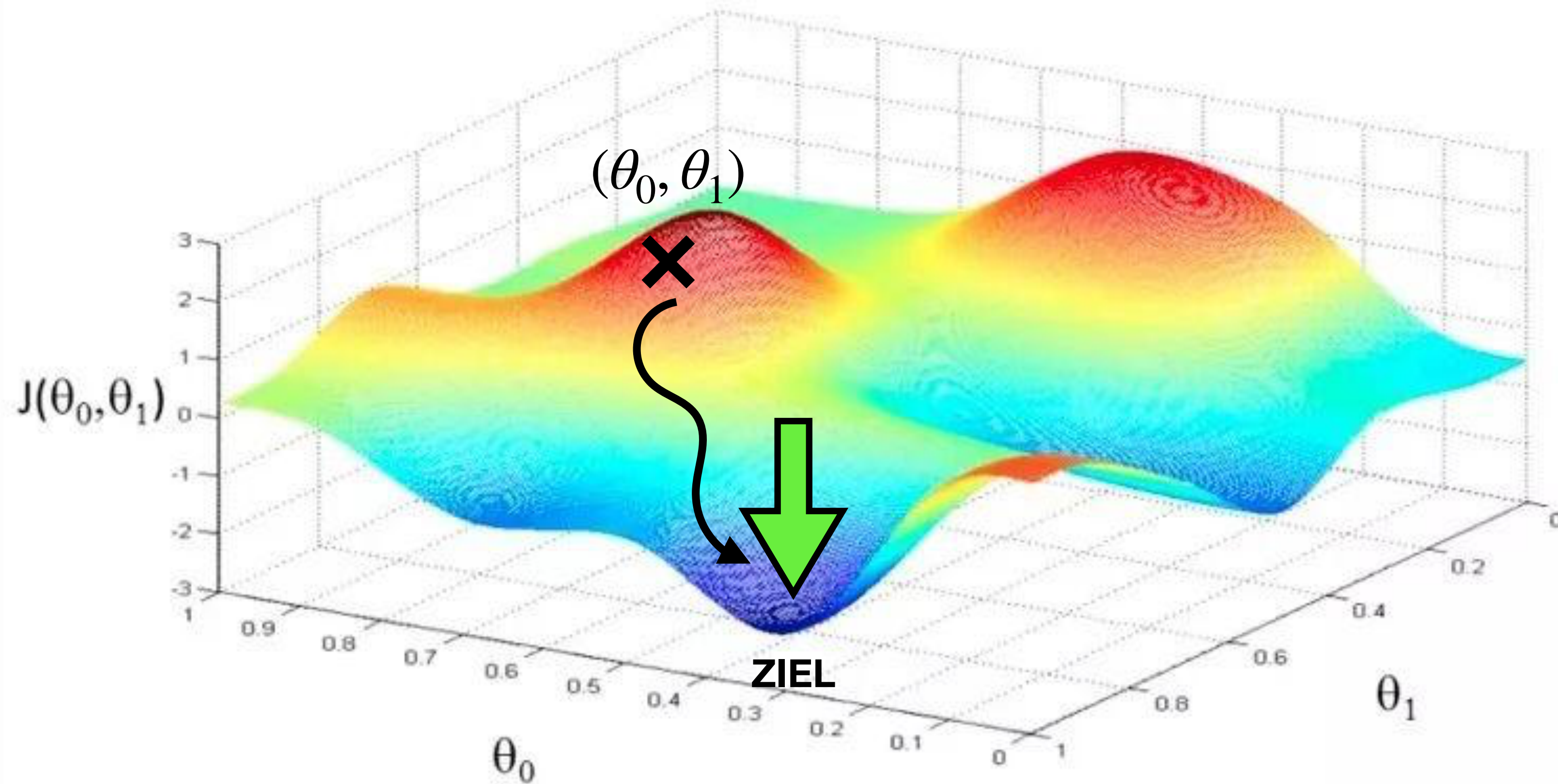
$$\theta^* = \operatorname{argmin}_{\theta} J(\theta)$$



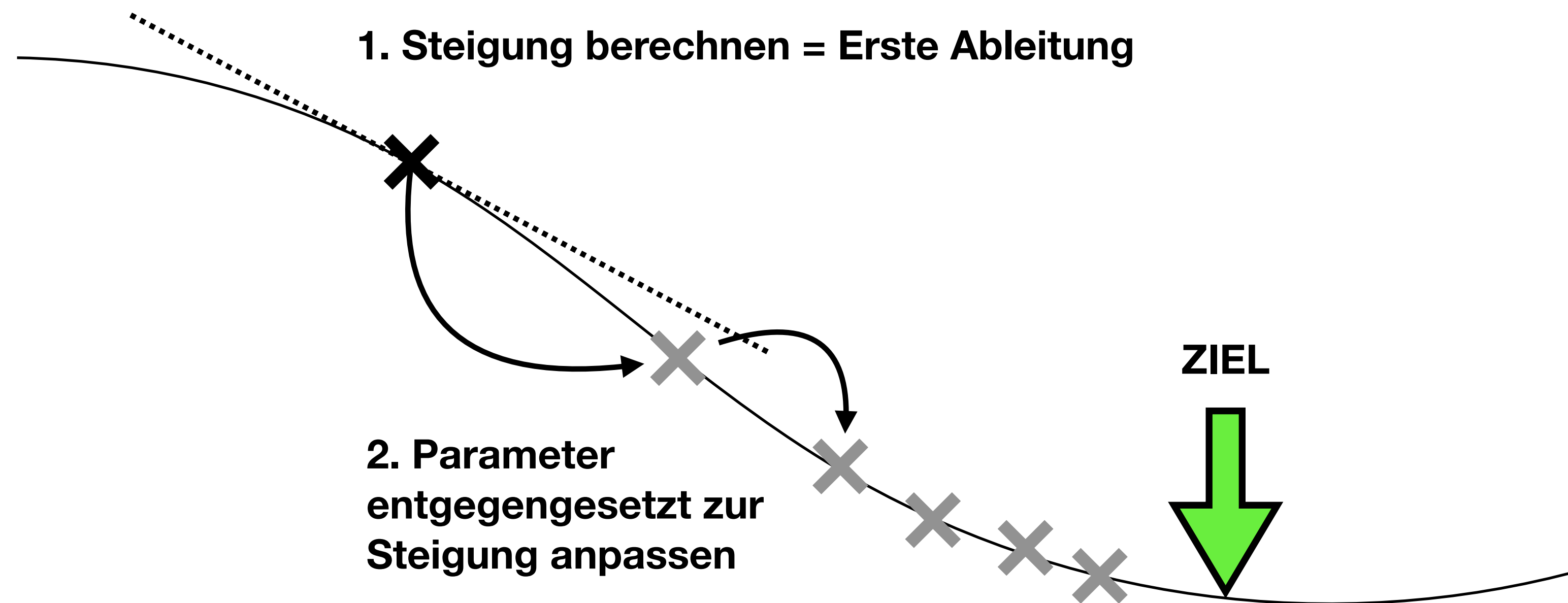
Die zufällige Initialisierung der Parameter führt zu einer zufälligen Position im Wertebereich der Fehlerfunktion.



Herausforderung: Wie kann der Weg zum Ziel (globales Minimum = θ^*) gefunden werden?



Herausforderung: Wie kann der Weg zum Ziel (globales Minimum = θ^*) gefunden werden?



Eine partielle Ableitung ist die Ableitung einer Funktion mit mehreren Variablen nach einer dieser Variablen.

Beispiel: $f(\theta)$ mit $\theta \in \mathbb{R}^2 = f(\theta_0, \theta_1) = 2\theta_0 + \theta_1$

Partielle Ableitung nach θ_0 ist $\frac{\partial f}{\partial \theta_0} = 2$

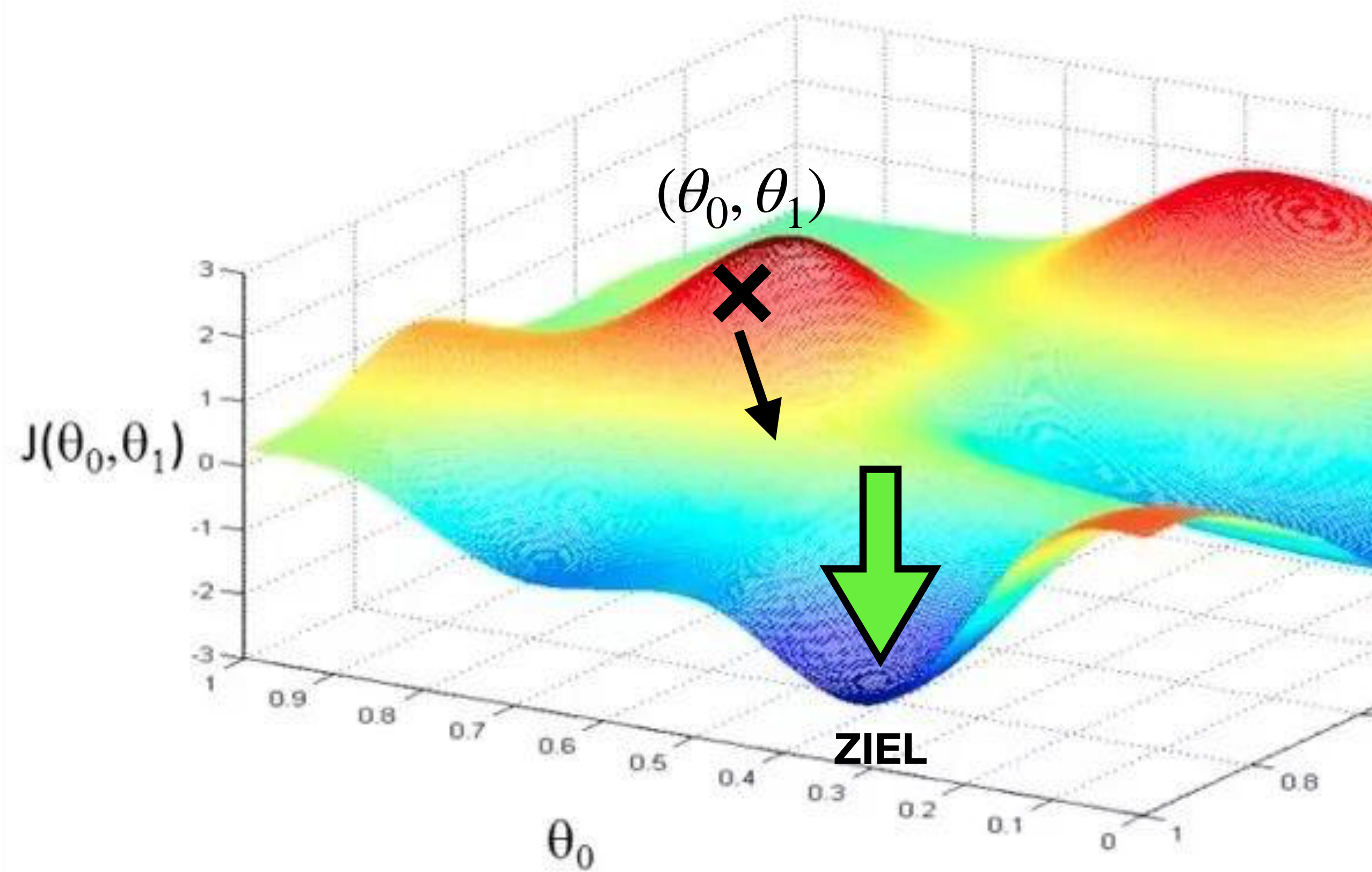
Partielle Ableitung nach θ_1 ist $\frac{\partial f}{\partial \theta_1} = 1$

Alle bekannten Ableitungsregeln gelten auch für partielle Ableitungen!

Mit $\frac{\partial f}{\partial \theta}$ werden alle partiellen Ableitungen von θ gemeint. Also $\begin{bmatrix} \frac{\partial f}{\partial \theta_0} \\ \frac{\partial f}{\partial \theta_1} \\ \dots \\ \frac{\partial f}{\partial \theta_n} \end{bmatrix}$

Neuronale Netze lernen über den Gradienten, welcher über die partielle Ableitung von θ bestimmt werden kann.

Aus diesem Grund muss sichergestellt werden, dass alle verwendeten Funktionen differenzierbar sind!



Algorithmus

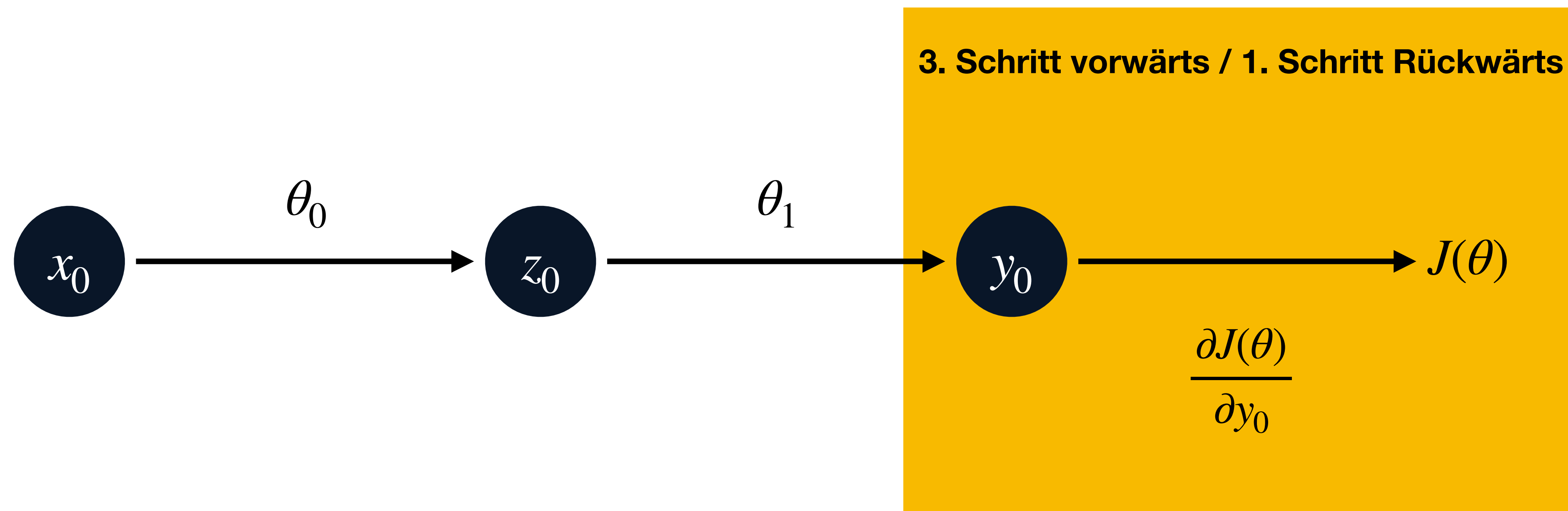
1. θ zufällig initialisieren mit $\sim N(0, \sigma^2)$
2. Bis das Netzwerk konvergiert:

1. Gradient der Fehlerfunktion bestimmen $\frac{\partial J(\theta)}{\partial \theta}$

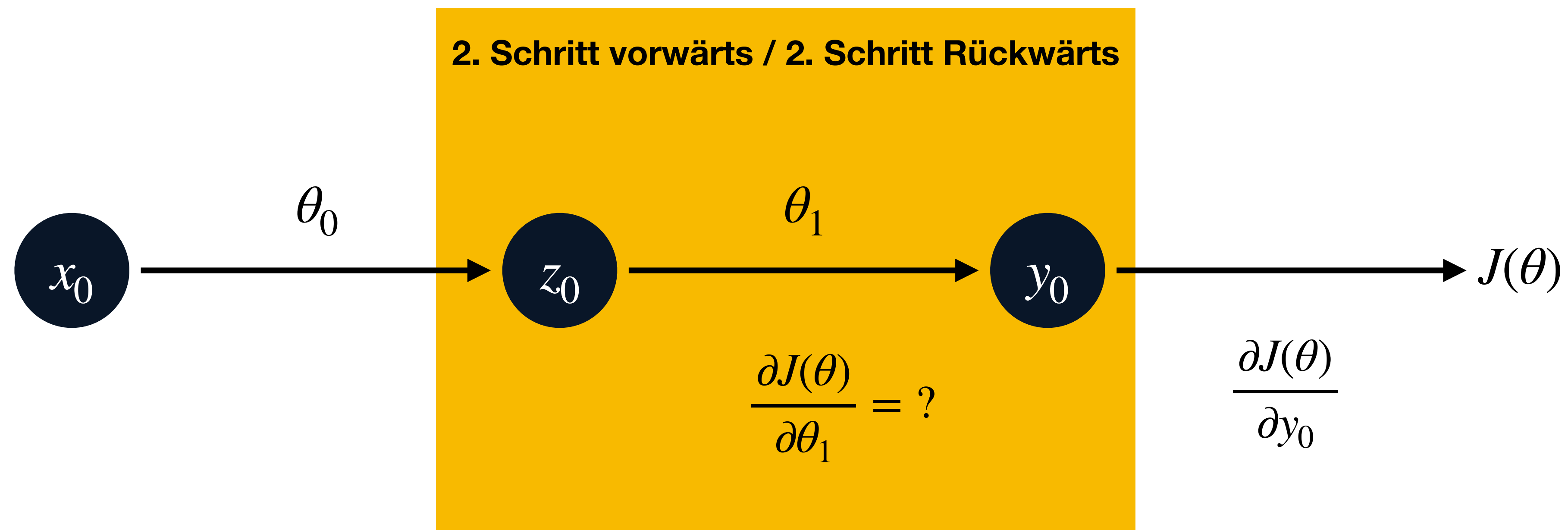
2. Gewichte updaten $\theta \leftarrow \theta - \eta \frac{\partial J(\theta)}{\partial \theta}$

Der Parameter η (Lernrate) bestimmt die Schrittgröße

Herausforderung: Gradient über beliebig viele Layer zurückführen.



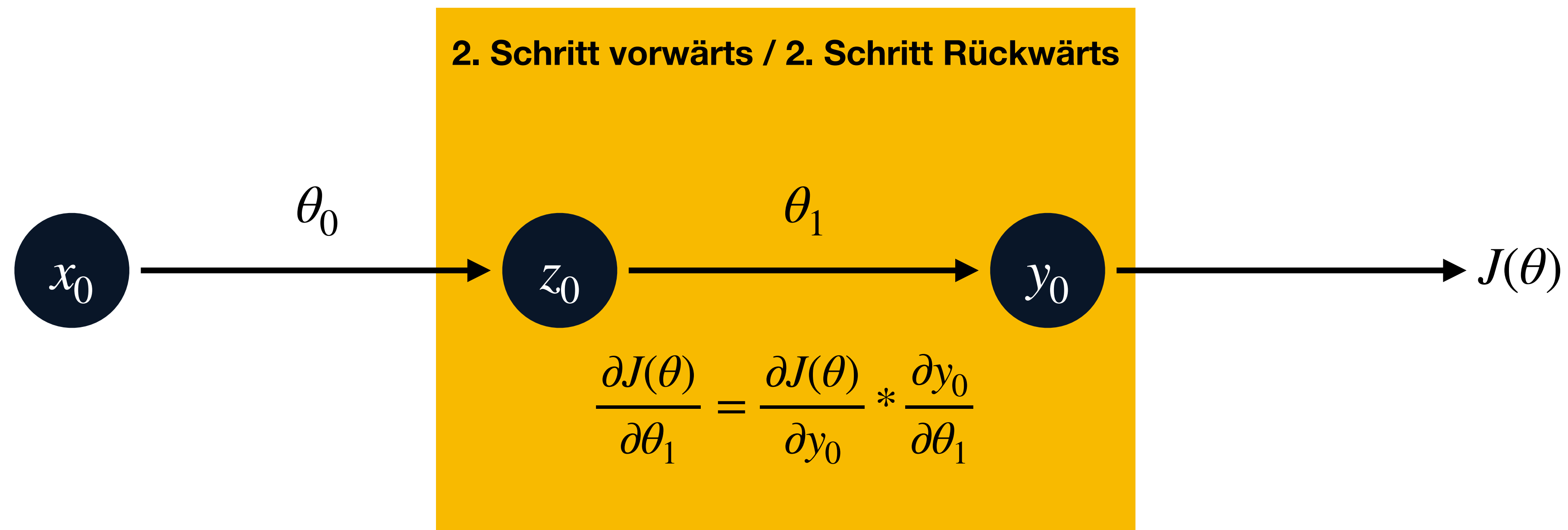
Herausforderung: Gradient über beliebig viele Layer zurückführen.



Wie stark ändert eine kleine Änderung von θ_1 den Fehler $J(\theta)$?

Lösung: Anwendung der Kettenregel

Herausforderung: Gradient über beliebig viele Layer zurückführen.

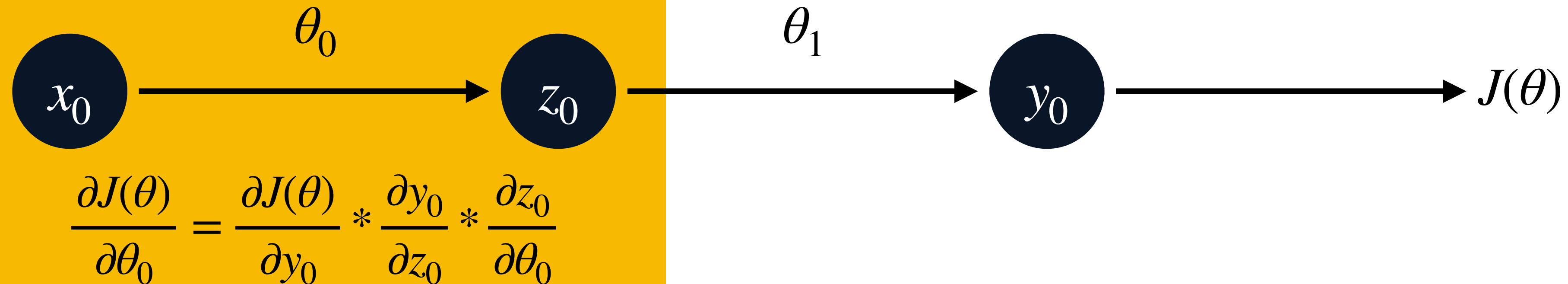


Wie stark ändert eine kleine Änderung von θ_1 den Fehler $J(\theta)$?

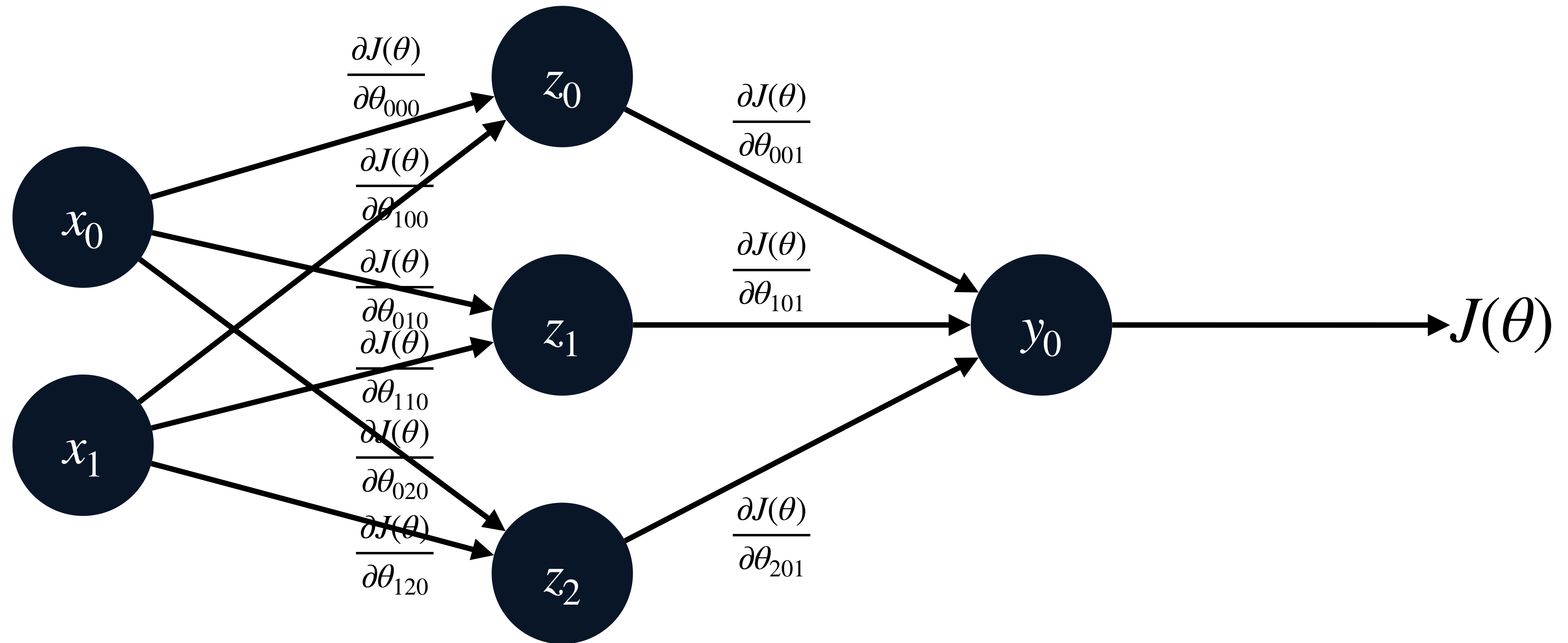
Lösung: Anwendung der Kettenregel

Herausforderung: Gradient über beliebig viele Layer zurückführen.

1. Schritt vorwärts / 3. Schritt Rückwärts



Wie stark ändert eine kleine Änderung von θ_0 den Fehler $J(\theta)$?



Dieser Prozess muss für jeden (lernbaren) Parameter des Netzwerkes wiederholt werden.

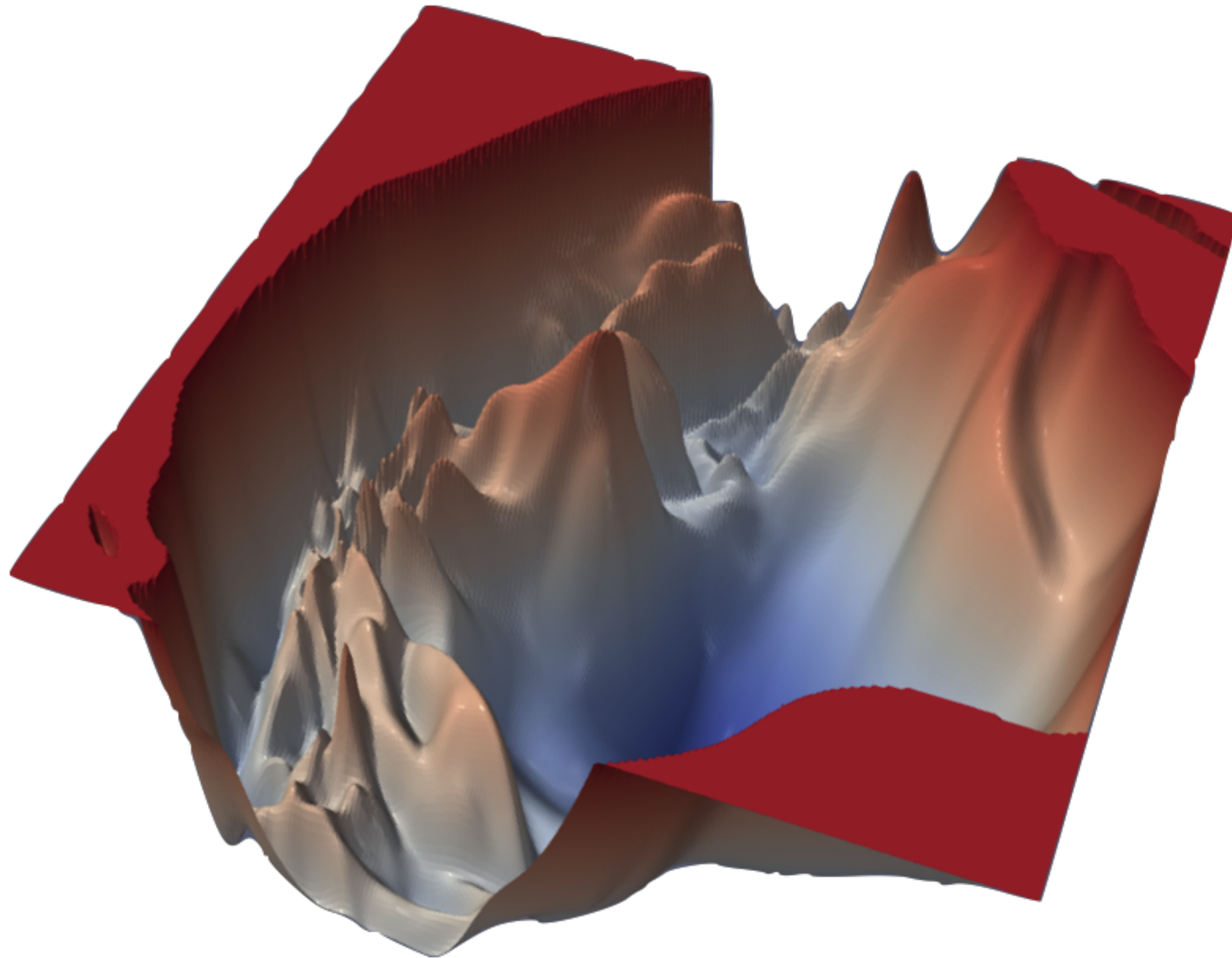
Danach können die Parameter angepasst werden $\theta \leftarrow \theta - \eta \frac{\partial J(\theta)}{\partial \theta}$

Training von Neuronalen Netzen

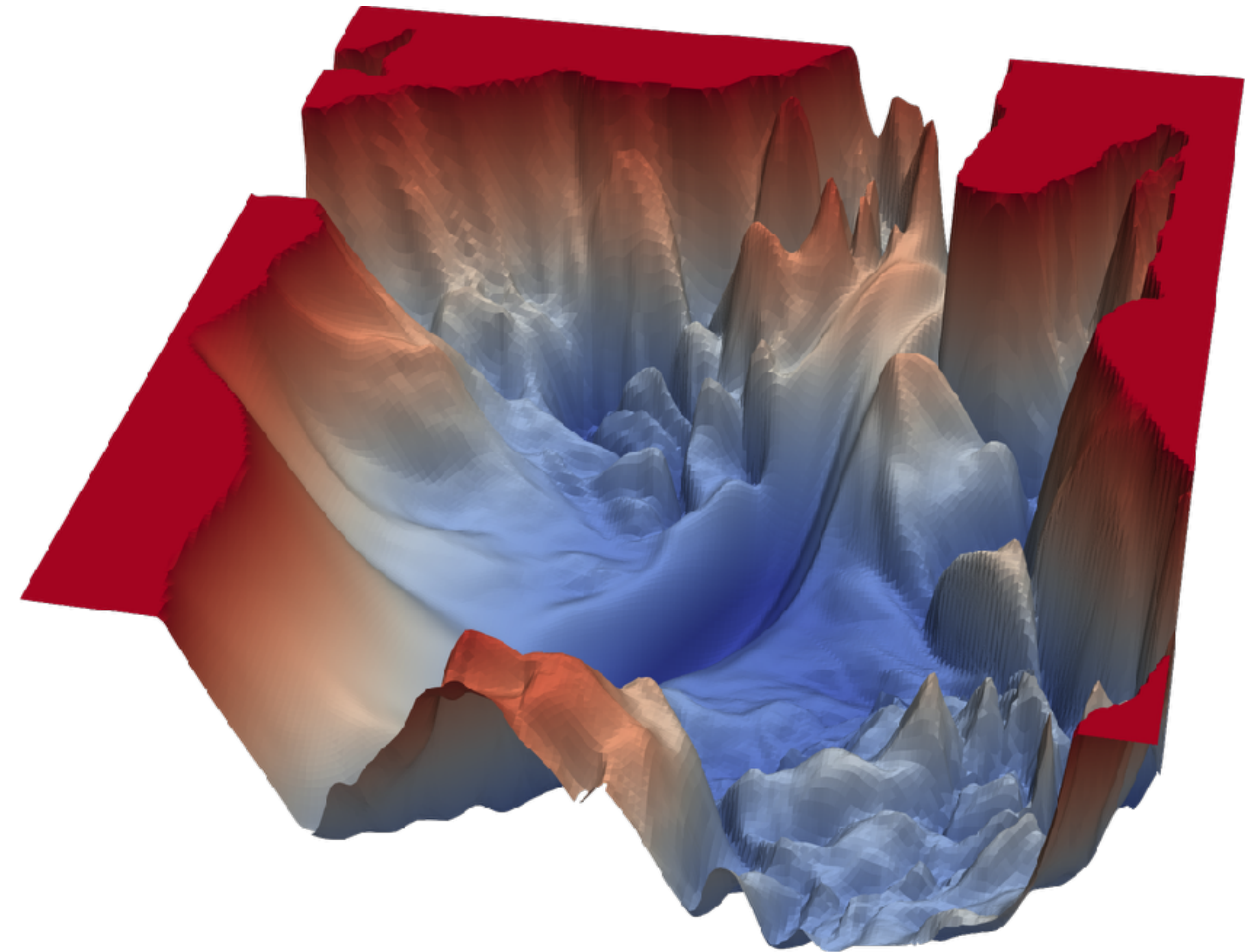
Praxis: Lernrate

Die Optimierung von Neuronalen Netzen in der Praxis ist voller Herausforderungen!

VGG-56

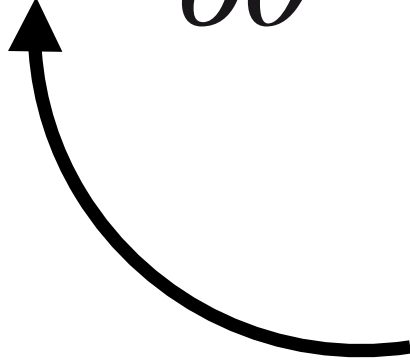


VGG-110



Li, H., Xu, Z., Taylor, G., Studer, C., & Goldstein, T. (2017). Visualizing the loss landscape of neural nets. *arXiv preprint arXiv:1712.09913*.

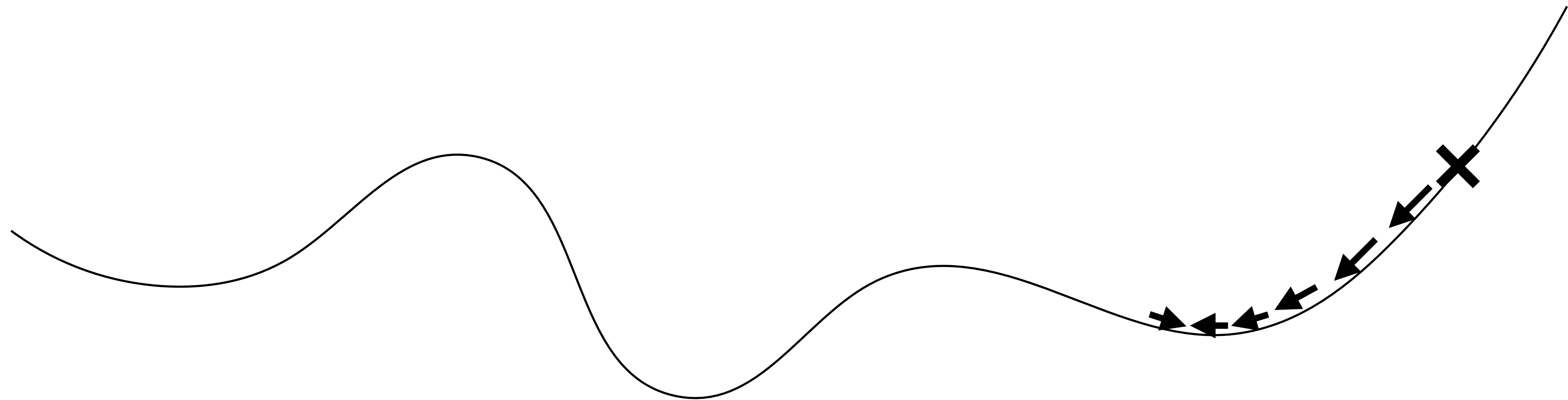
Die Lernrate beim Training ist ein wesentlicher Hyperparameter (von vielen).

$$\theta \leftarrow \theta - \eta \frac{\partial J(\theta)}{\partial \theta}$$


Wie wählen?

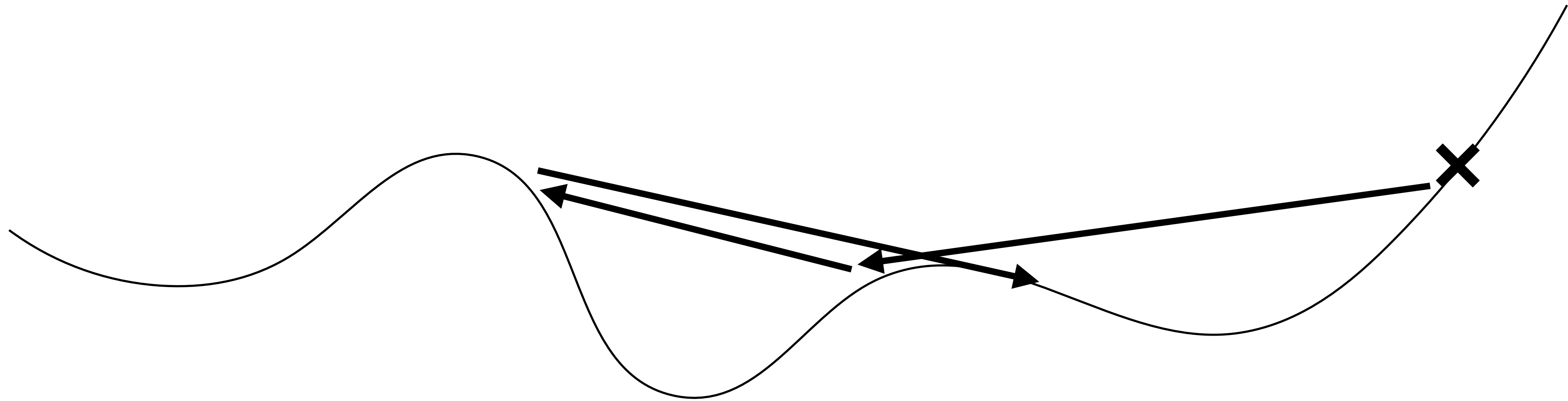
Die Lernrate beim Training ist ein wesentlicher Hyperparameter (von vielen).

Kleine Lernraten führen zur langsamen Konvergenz und bleiben möglicherweise in einem lokalen Minimum „hängen“.



Die Lernrate beim Training ist ein wesentlicher Hyperparameter (von vielen).

Große Lernraten führen können über das Ziel „hinausschießen“ und verhindern so möglicherweise eine ausreichende Konvergenz.



Die Lernrate beim Training ist ein wesentlicher Hyperparameter (von vielen).

- Möglichkeit 1: Viele verschiedene Werte ausprobieren.
 - Kann sehr lange dauern. Möglicherweise wird der optimale Wert nie gefunden.
- Möglichkeit 2: Adaptive Lernraten, welche sich den Gegebenheiten der Landschaft der Fehlerfunktion anpassen.

Der Lernrate ist nicht fixiert, sondern wird adaptive zur Oberfläche der Fehlerfunktion angepasst.

Es gibt verschiedene Möglichkeiten und Ansatzpunkte:

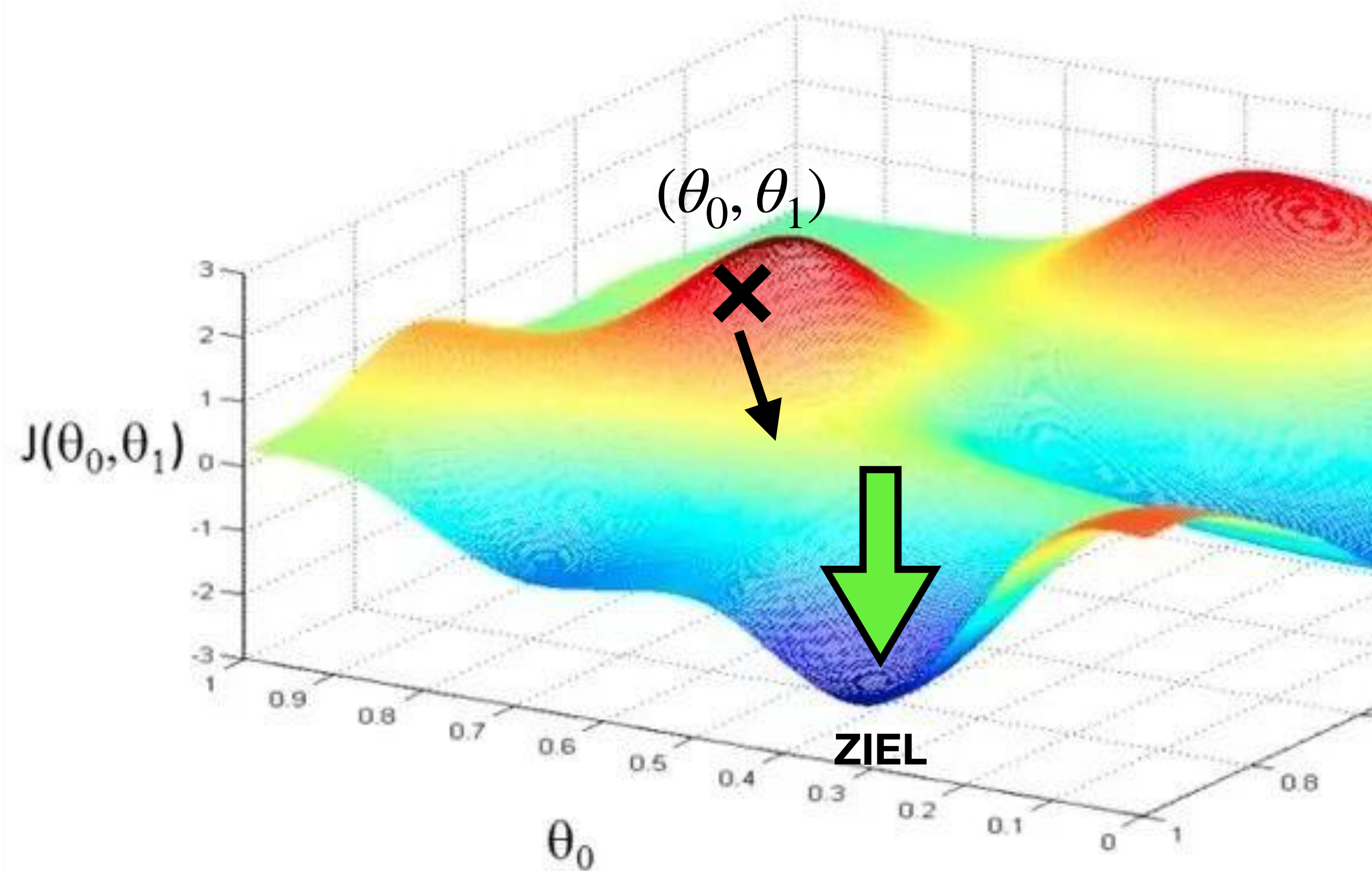
- Abhängig von der Größe des Gradienten
- Abhängig von der realen Verbesserung der Fehlerfunktion
- Die Größe von Gewichten oder deren Verteilung
- ...

Es gibt verschiedene Strategien und Implementierungen, welche sich durchgesetzt haben:

- Momentum
 - Fügt ein Momentum zur Optimierung hinzu
- RMSProp
 - Berechnung des gleitenden Durchschnitts der quadrierten Gradienten
 - Gradienten werden vor dem Update durch die Wurzel des Durchschnittes geteilt
- Adagrad
 - Die Lernrate wird pro Parameter abhängig von der Häufigkeit der Updates angepasst
 - Je mehr eine Anpassung stattfinden, desto kleiner die Lernrate
- Adam
 - Kombiniert die Ansätze von RMSProp und Adagrad

Training von Neuronalen Netzen

Praxis: Mini-Batches



Algorithmus

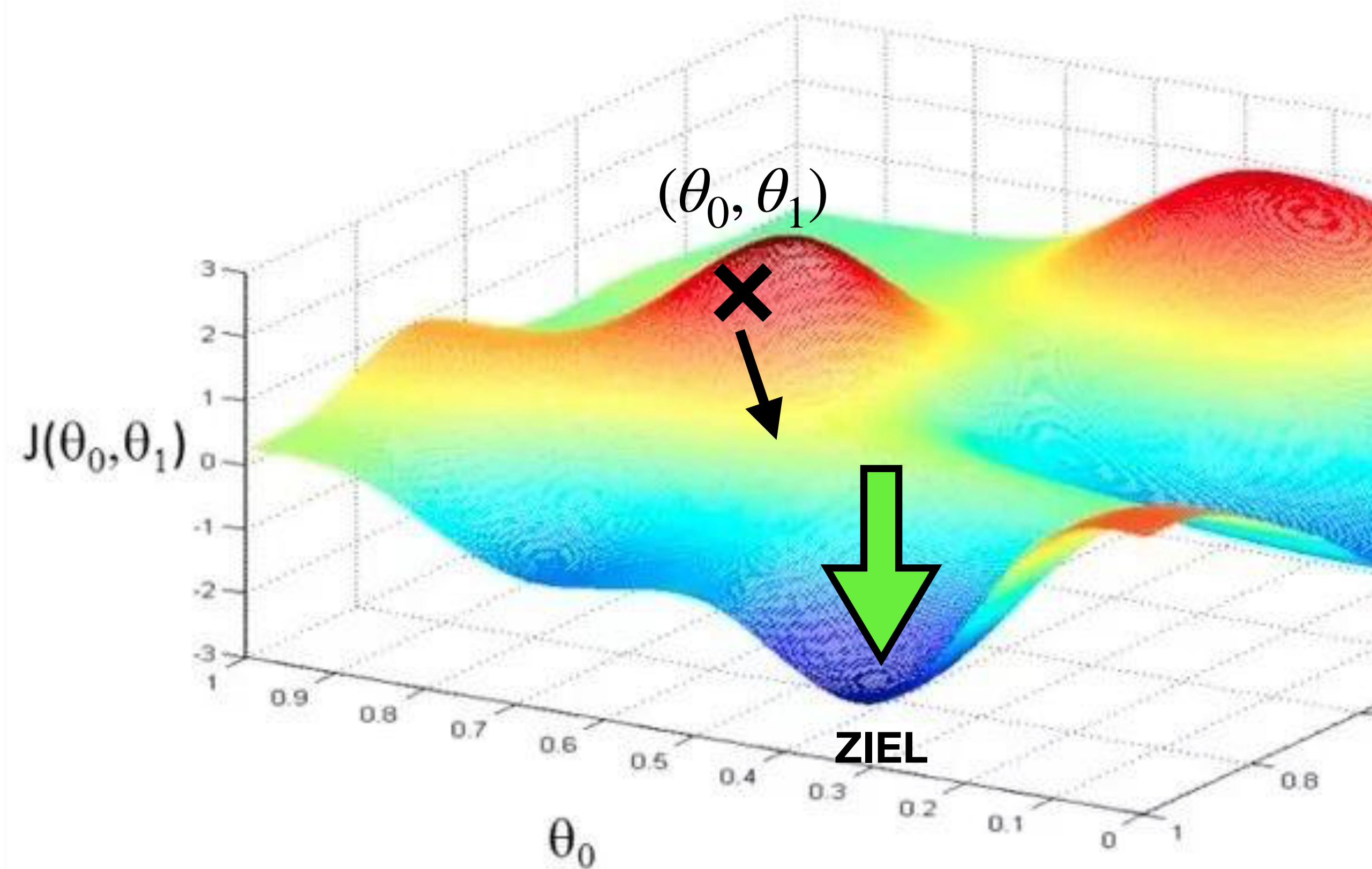
1. θ zufällig initialisieren mit $\sim N(0, \sigma^2)$

2. Bis das Netzwerk konvergiert:

1. Gradient der Fehlerfunktion bestimmen $\frac{\partial J(\theta)}{\partial \theta}$

2. Gewichte updaten $\theta \leftarrow \theta - \eta \frac{\partial J(\theta)}{\partial \theta}$

Die Berechnung von $J(\theta)$ kann sehr sehr teuer sein!



Algorithmus

1. θ zufällig initialisieren mit $\sim N(0, \sigma^2)$

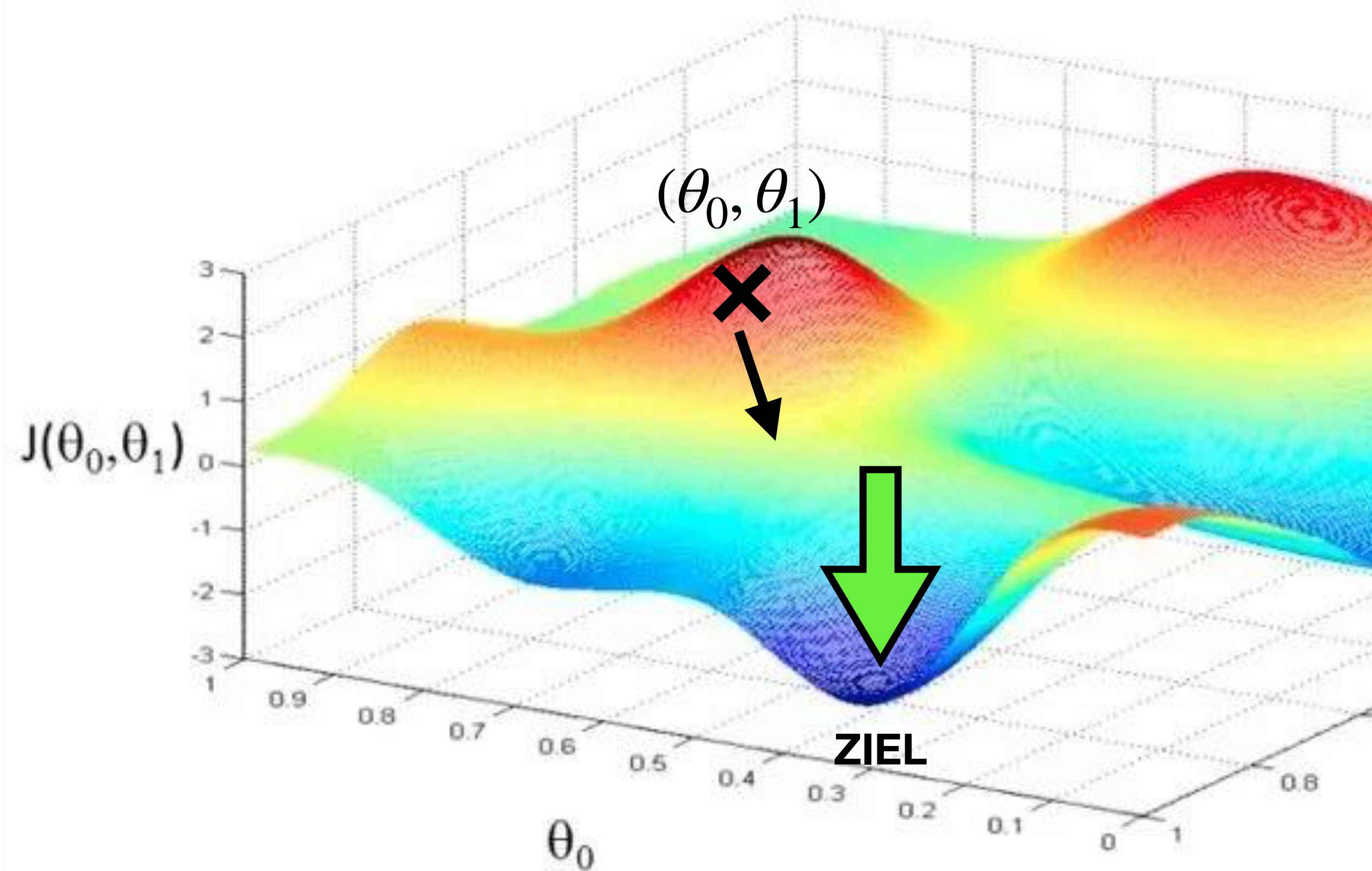
2. Bis das Netzwerk konvergiert:

1. Wähle einen Datenpunkt i

2. Gradient der Fehlerfunktion bestimmen $\frac{\partial J_i(\theta)}{\partial \theta}$

3. Gewichte updaten $\theta \leftarrow \theta - \eta \frac{\partial J_i(\theta)}{\partial \theta}$

Sehr einfach zu berechnen aber häufig mit sehr viel Rauschen verbunden (stochastisch).



Algorithmus

1. θ zufällig initialisieren mit $\sim N(0, \sigma^2)$

2. Bis das Netzwerk konvergiert:

1. Wähle einen Batch von Datenpunkten B

2. Gradient der Fehlerfunktion bestimmen $\frac{1}{B} \sum_{i=1}^B \frac{\partial J_i(\theta)}{\partial \theta}$

3. Gewichte updaten $\theta \leftarrow \theta - \eta \frac{1}{B} \sum_{i=1}^B \frac{\partial J_i(\theta)}{\partial \theta}$

Schnell zu berechnen.

Abhängig von der Batchgröße eine gute Approximation des echten Gradienten.

Effekte von Mini-Batches auf die Effektivität des Trainings

- Approximation des echten Gradienten $\frac{\partial J(\theta)}{\partial \theta}$
- Bessere (gleichmäßigere) Konvergenz des Trainings
- Bietet die Möglichkeit für größere Lernraten

Effekte von Mini-Batches auf die Geschwindigkeit des Trainings

- Parallelisieren möglich (von parallelen Berechnungen der einzelnen Datenpunkten in einem Batch)
- Signifikante Geschwindigkeitsvorteile auf spezialisierter Hardware (GPUs/TPUs)