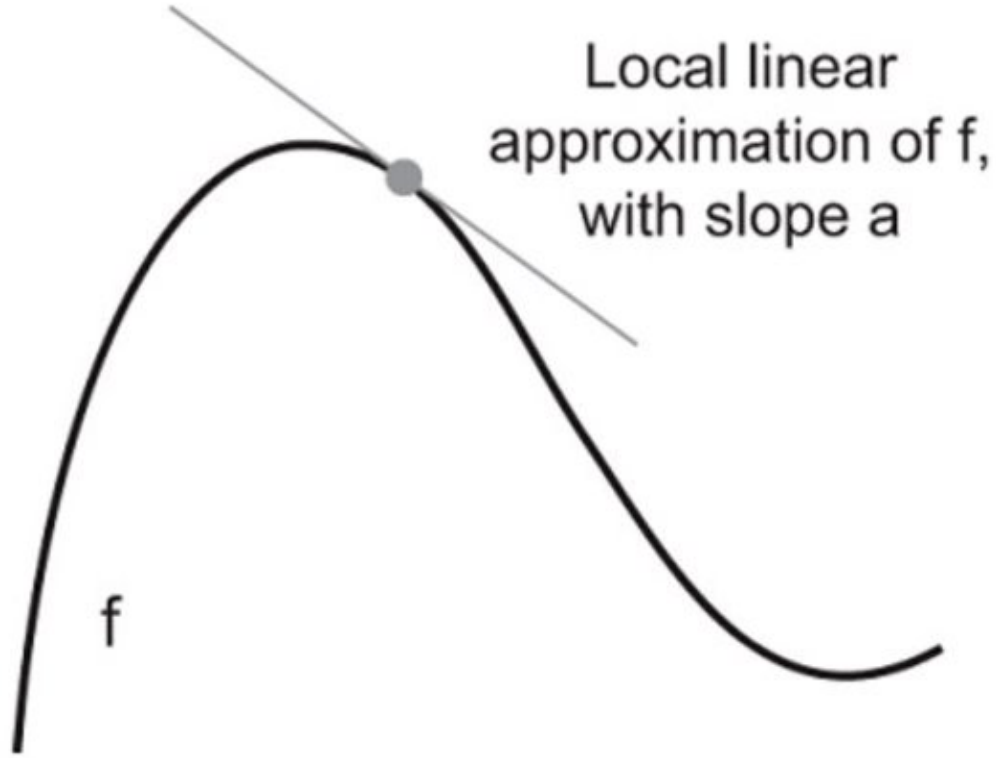
Gradient Descent und Backpropagation

Wie werden neuronale Netzwerke trainiert?

Gradient Descent

Die Ableitung einer Funktion sagt uns in welche Richtung sie lokal kleinere Werte hat.
Im mehrdimensionalen nennt man die Ableitung Gradient.
Der Gradient zeigt in Richtung der größten Steigung.

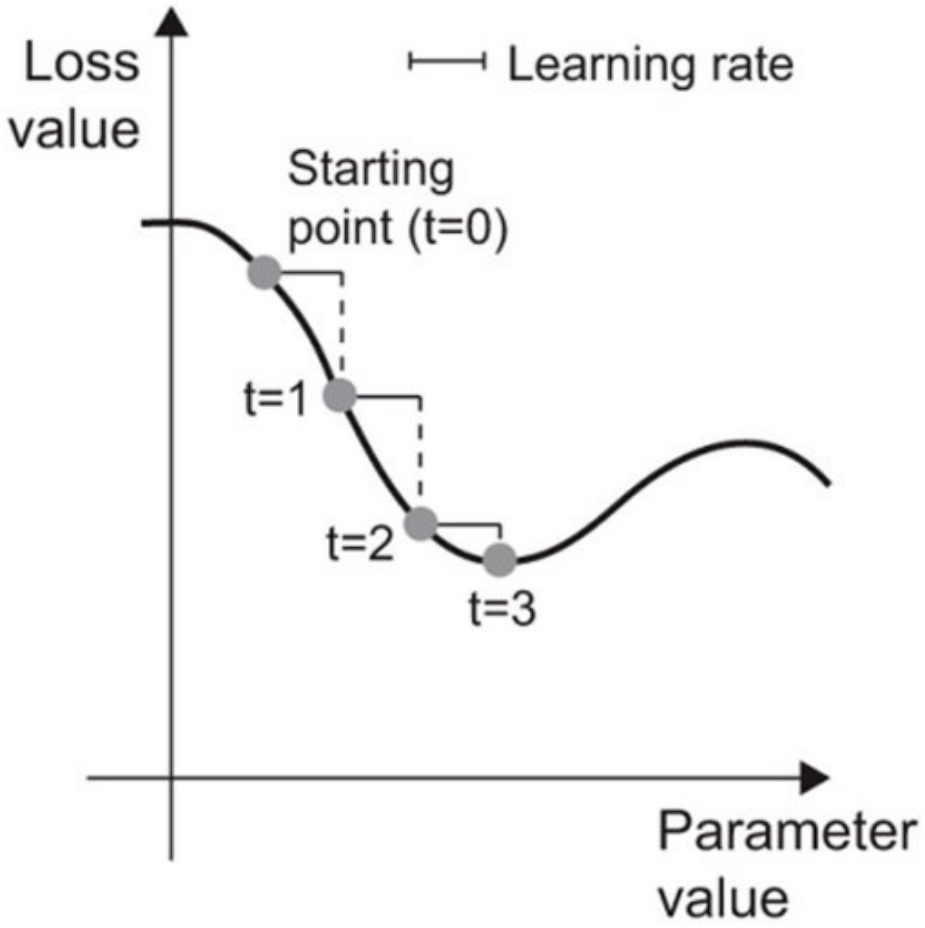


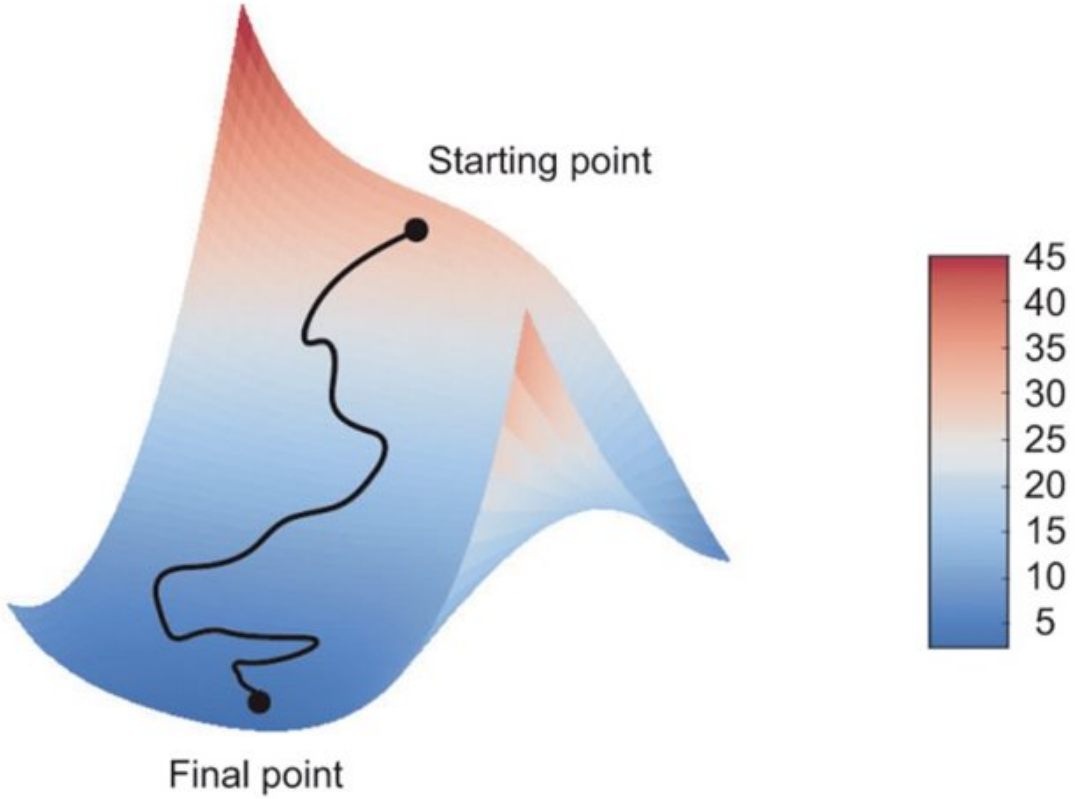
Gradient: $\mathbf{g} = \nabla f(\mathbf{x}) = \begin{pmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \dots \\ \frac{\partial f}{\partial x_n} \end{pmatrix}$

Abstieg: $\mathbf{x}_{\text{new}} = \mathbf{x} - \epsilon \mathbf{g}$

ϵ : Learning rate (Parameter)

Gradient Descent





Gradient Descent – Learning rate

- Die Learning rate ist ein wichtiger Parameter, welcher entscheidend für das Funktionieren und die Effizienz des Trainings ist

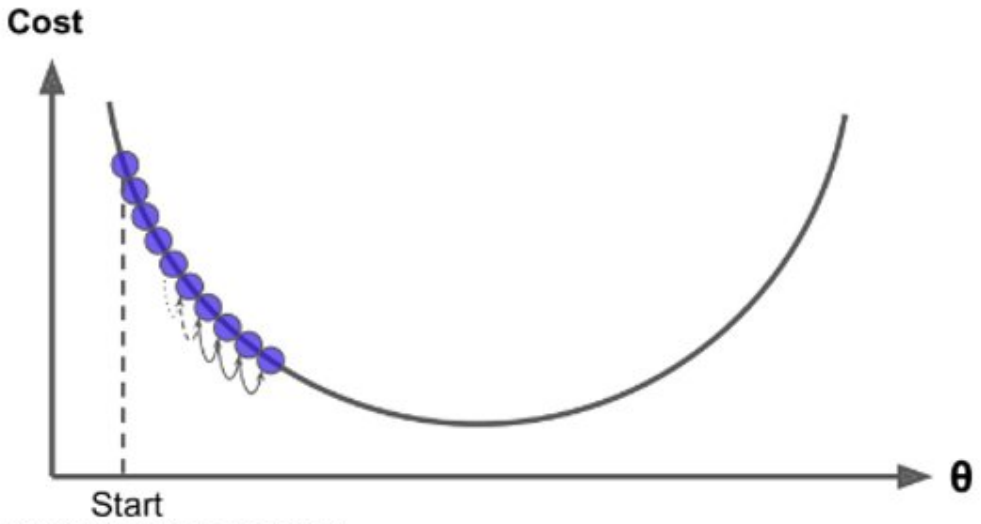


Figure 4-4. Learning rate too small

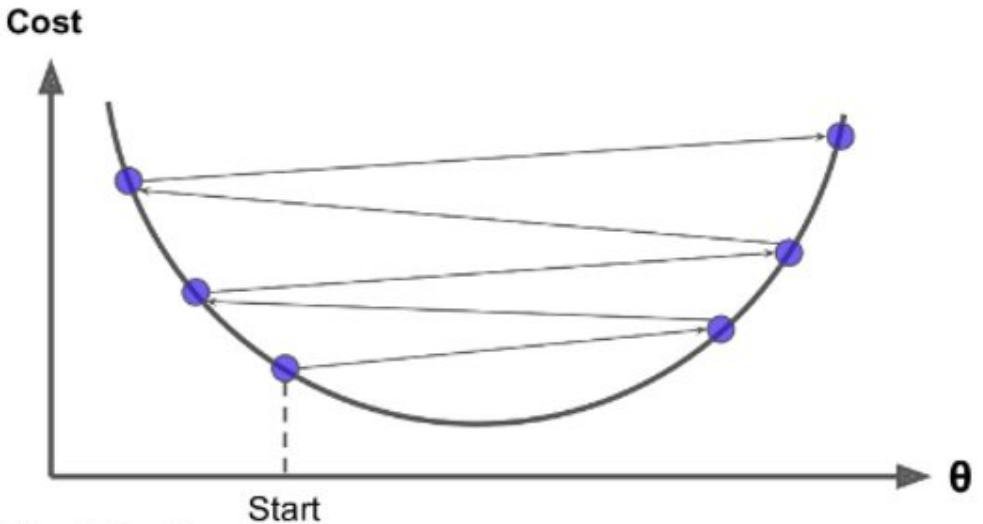
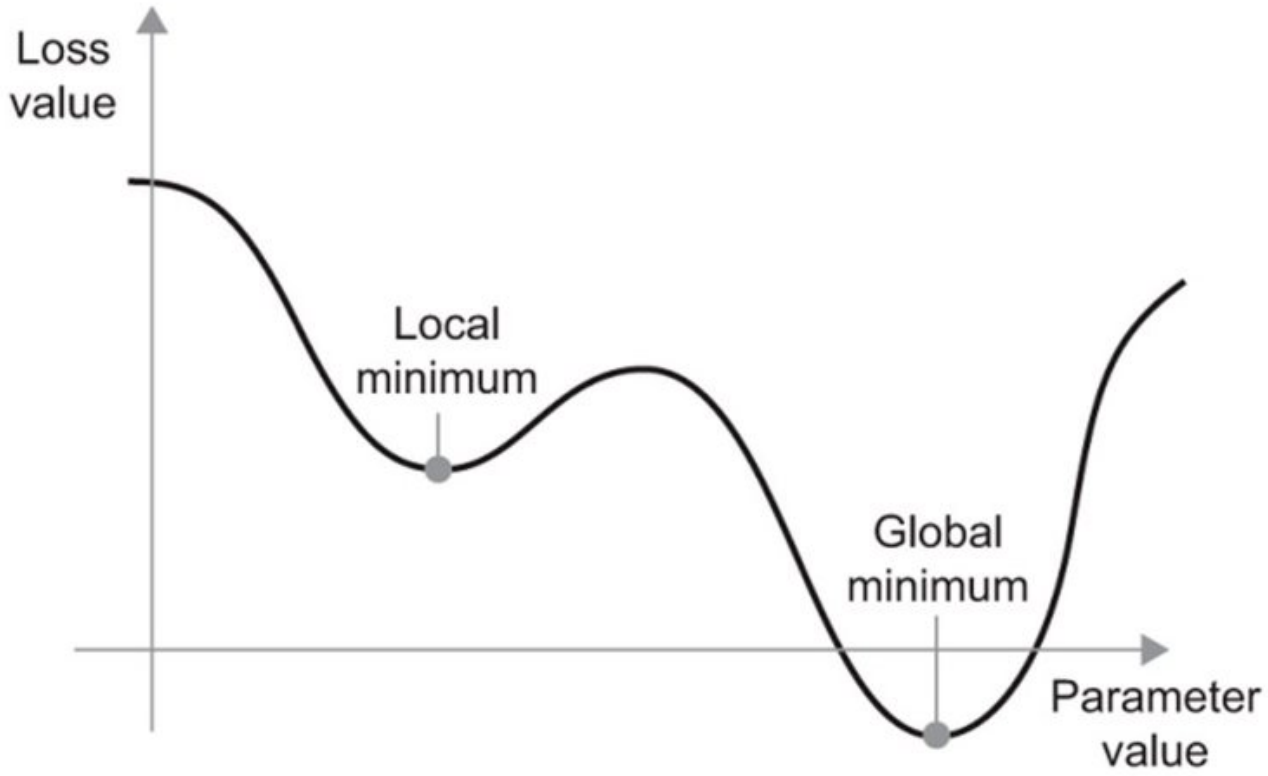


Figure 4-5. Learning rate too large

Gradient Descent

- Im Deep Learning gibt es in der Regel immer mehrere Minima
- Das Optimierungsproblem ist nicht konvex
- Es gibt viele Optimierer, die noch weitere Informationen ausnutzen, z.b. „Momentum“



Überblick über den Trainingsprozess – In Worten

Der Trainings-Loop:

- Nehme die Trainingsbeispiele
- Steck sie in das Model und berechne die Vorhersagen (Forward-Propagation)
- Vergleiche Vorhersagen mit dem Target also der Groundtruth (Berechnung des Losses)
- Berechne in welche Richtung sich die Parameter ändern müssen, sodass sich der Loss verringert (Backpropagation)
- Verändere die Parameter des Modells, so dass der Loss sich verringert. (Gradientenabstieg)

Überblick über den Trainingsprozess – In Formeln

- Netzwerk erzeugt Output $\hat{y}_i = f(\mathbf{x}_i; \theta)$
- Berechne den Loss $\mathcal{L}(\theta) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 = \frac{1}{n}\sum_{i=1}^{n}(y_i - f(\mathbf{x}_i; \theta))^2$
- Berechne Gradient (Backpropagation) $\mathbf{g} = \nabla_\theta \mathcal{L}(\theta)$
- Gradientenabstieg zum Parametervektor θ^*(Minima des Losses) $\theta_{\text{new}} = \theta - \epsilon\mathbf{g}$

Batch Training

Typischerweise wird nicht der ganze Trainingssatz zur Berechnung des Gradienten eingesetzt

- Stattdessen: Wähle einen zufälligen Batch (Teilmenge) von Trainingsbeispielen

In Formeln:

$$\mathbf{g} = \begin{cases} \frac{1}{|I|} \sum_{i \in I} \nabla_\theta \mathcal{L} & \text{mini-batch Gradient Descent}, \quad I \subseteq \{1, 2, ..., n\} \\ \frac{1}{n} \sum_{i=1}^{n} \nabla_\theta \mathcal{L} & \text{batch Gradient Descent} \end{cases}$$

Iteration : Einen (mini-) batch durchlaufen

Epoch : Einmal den ganzen Trainingsdatensatz durchlaufen

Notebooks

- **1.8.Tensorflow_Beispiel.ipynb**

Backpropagation ‘86

Learning representations by back-propagating errors

David E. Rumelhart*, Geoffrey E. Hinton†
& Ronald J. Williams*

* Institute for Cognitive Science, C-015, University of California, San Diego, La Jolla, California 92093, USA
† Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Philadelphia 15213, USA

We describe a new learning procedure, back-propagation, for networks of neurone-like units. The procedure repeatedly adjusts the weights of the connections in the network so as to minimize a measure of the difference between the actual output vector of the net and the desired output vector. As a result of the weight adjustments, internal 'hidden' units which are not part of the input or output come to represent important features of the task domain, and the regularities in the task are captured by the interactions of these units. The ability to create useful new features distinguishes back-propagation from earlier, simpler methods such as the perceptron-convergence procedure[1].

There have been many attempts to design self-organizing neural networks. The aim is to find a powerful synaptic modification rule that will allow an arbitrarily connected neural network to develop an internal structure that is appropriate for a particular task domain. The task is specified by giving the more difficult when we introduce hidden units whose actual or desired states are not specified by the task. (In perceptrons, there are 'feature analysers' between the input and output that are not true hidden units because their input connections are fixed by hand, so their states are completely determined by the input vector: they do not learn representations.) The learning procedure must decide under what circumstances the hidden units should be active in order to help achieve the desired input–output behaviour. This amounts to deciding what these units should represent. We demonstrate that a general purpose and relatively simple procedure is powerful enough to construct appropriate internal representations.

The simplest form of the learning procedure is for layered networks which have a layer of input units at the bottom; any number of intermediate layers; and a layer of output units at the top. Connections within a layer or from higher to lower layers are forbidden, but connections can skip intermediate layers. An input vector is presented to the network by setting the states of the input units. Then the states of the units in each layer are determined by applying equations (1) and (2) to the connections coming from lower layers. All units within a layer have their states set in parallel, but different layers have their states set sequentially, starting at the bottom and working upwards until the states of the output units are determined.

The total input, x_j, to unit j is a linear function of the outputs, y_i, of the units that are connected to j and of the weights, w_{ji}, on these connections

$$x_j = \sum_i y_i w_{ji} \qquad (1)$$

Die Ketten-Regel

- Neuronale Netze sind ineinander geschachtelte Funktionen
  - Affine Transformationen werden in Activation-Funktionen gesteckt
  - Ein Layer wird in das nächste gesteckt
- Solche Ketten von Funktionen werden nach der Kettenregel abgeleitet

  „Äußere Ableitung mal innere Ableitung“.
- In Code mit mehr Funktionen: $(u \circ v)'(x_0) = u'\big(v(x_0)\big) \cdot v'(x_0)$

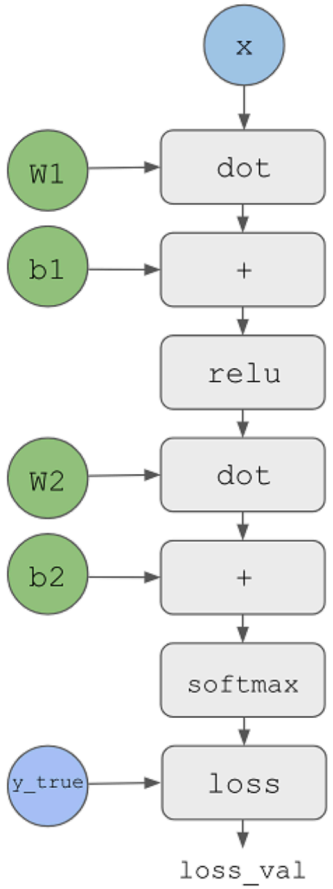
```
def fghj(x):
  x1 = j(x)
  x2 = h(x1)
  x3 = g(x2)
  y = f(x3)
  return y

grad(y, x) == grad(y, x3) * grad(x3, x2) * grad(x2, x1) * grad(x1, x)
```

Computation Graphs

Ein Computation Graph ist ein
DAG aus Operationen
also Tensoroperationen
oder Anwendungen von Funktionen.

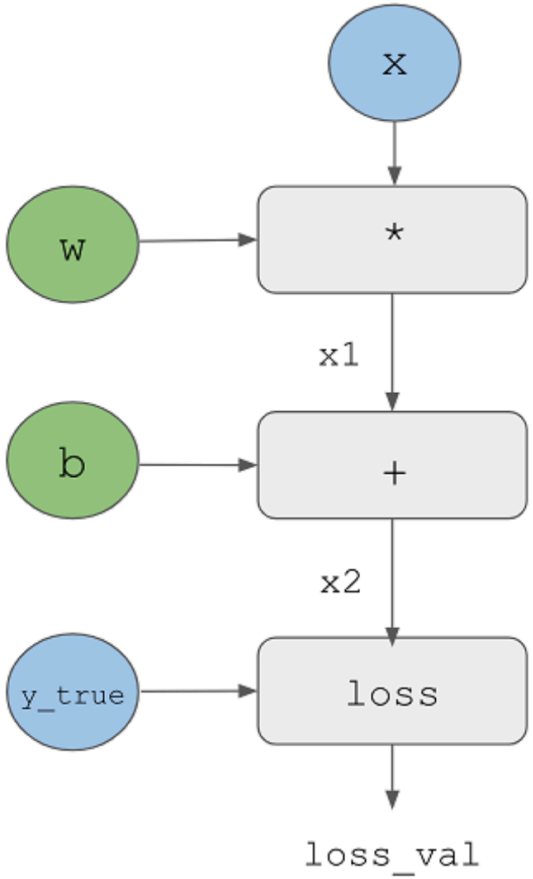
Unser erstes Model →



Backprop im Computation Graph

- Vorwärtsdurchgang: loss_val wird berechnet

```
loss_val = abs(y_true - y)
```

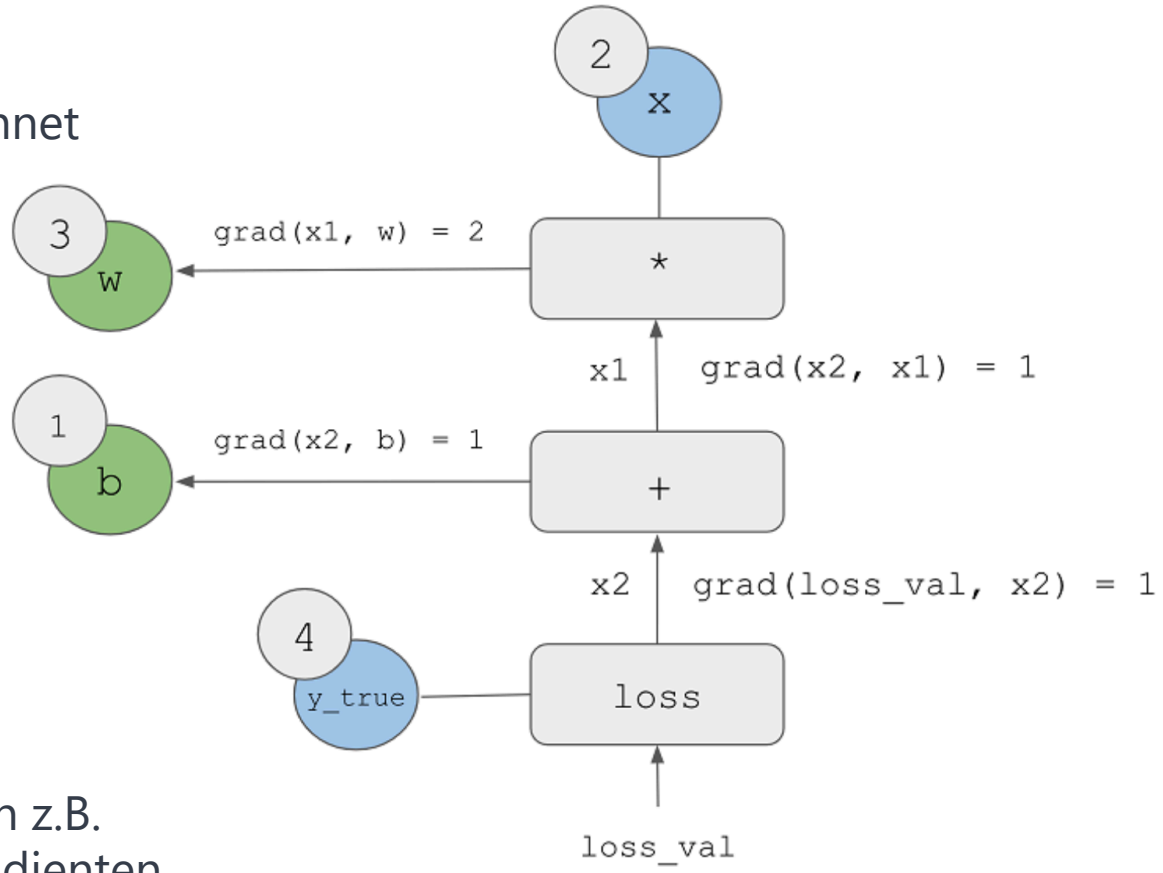


Backprop im Computation Graph

- Vorwärtsdurchgang: loss_val wird berechnet

```
loss_val = abs(y_true - y)
```

- Rückwärtsdurchgang:
  Gradienten für *W* und *b* werden berechnet



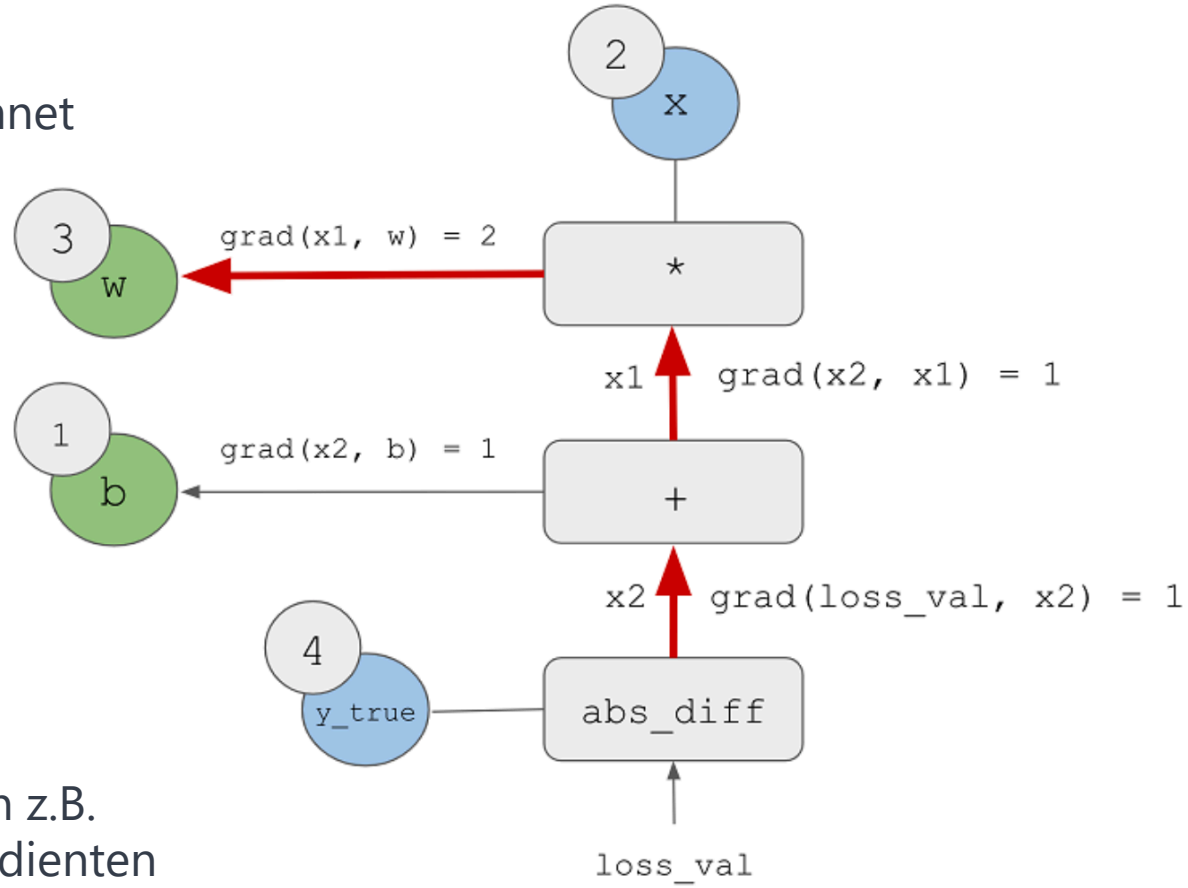
grad(loss_val, x2) = 1
grad(x2, x1) = 1
grad(x2, b) = 1
grad(x1, w) = 2
→ Kettenregel: Wir erhalten den Gradienten z.B. von *loss_val* bezüglich *W* indem wir die Gradienten der Kanten auf dem Weg von *loss_val* zu *W* multiplizieren.

Backprop im Computation Graph

- Vorwärtsdurchgang: loss_val wird berechnet

```
loss_val = abs(y_true - y)
```

- Rückwärtsdurchgang:
  Gradienten für *W* und *b* werden berechnet



grad(loss_val, x2) = 1
grad(x2, x1) = 1
grad(x2, b) = 1
grad(x1, w) = 2
→ Kettenregel: Wir erhalten den Gradienten z.B. von *loss_val* bezüglich *W* indem wir die Gradienten der Kanten auf dem Weg von *loss_val* zu *W* multiplizieren.

Backprop im Computation Graph

Warum ist das toll?

- Automatic Differentiation
  = die automatische Berechnung von Gradienten

- Backpropagation bedeutet, die Komplexität der Gradientenberechnung ist gleich der Komplexität des Vorwärtsdurchgangs.

Backprop im Computation Graph

Mathematische Details für eine Implementierung per Hand:
www.neuralnetworksanddeeplearning.com

Neural Networks and Deep Learning

Neural Networks and Deep Learning is a free online book. The book will teach you about:

- Neural networks, a beautiful biologically-inspired programming paradigm which enables a computer to learn from observational data
- Deep learning, a powerful set of techniques for learning in neural networks

Neural networks and deep learning currently provide the best solutions to many problems in image recognition, speech recognition, and natural language processing. This book will teach you many of the core concepts behind neural networks and deep learning.

Neural Networks and Deep Learning
What this book is about
On the exercises and problems
Using neural nets to recognize handwritten digits
How the backpropagation algorithm works
Improving the way neural networks learn
A visual proof that neural nets can compute any function
Why are deep neural networks hard to train?
Deep learning
Appendix: Is there a *simple* algorithm for intelligence?
Acknowledgements
Frequently Asked Questions

Optimierer

- SGD
- RMSprop
- Adam
- Adagrad
- Nesterov
- Ftrl
- Etc.

Optimierer - Recherche

- SGD
- RMSprop
- Adam
- Adagrad
- Nesterov
- Ftrl
- Etc.