Implementing Artificial Neural Networks (ANNs) with TensorFlow

Session 3: Gradient Descent and Backpropagation

University of Osnabrück
Institute of Cognitive Science

WS 2017 / 18

Lukas Braun, lbraun@uos.de

5 B BODIDDLY100 7 months ago

"Oh boy, I'm gonna learn how to implement neural networks OH GOD CALCULUS EVERYWHERE"

[5] youtube.com

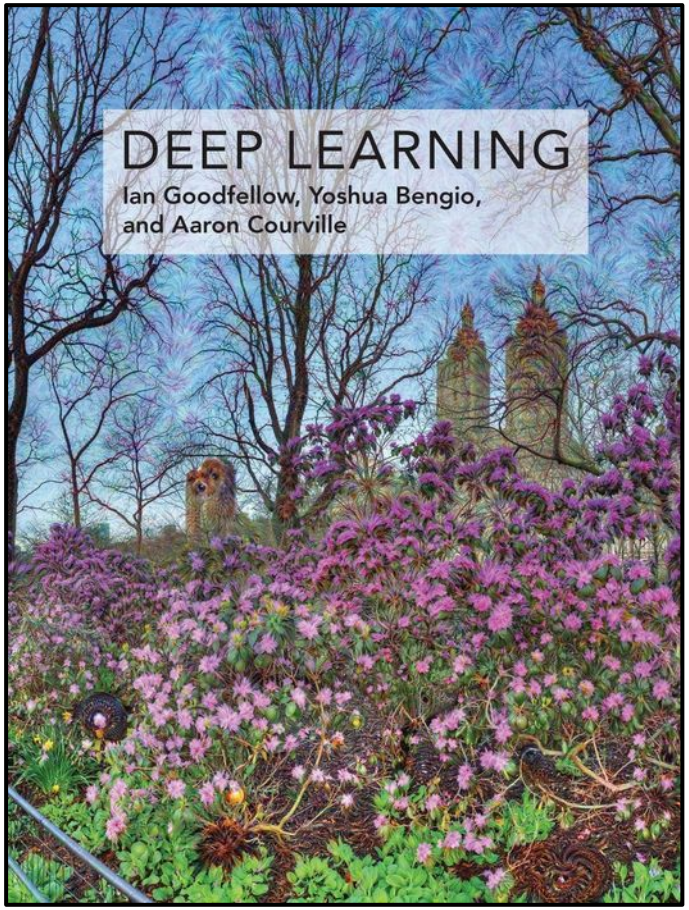
Resources

Deep Learning
by Ian Goodfellow, Yoshua Bengio and Aaron Courville

Available for free at:
deeplearningbook.org

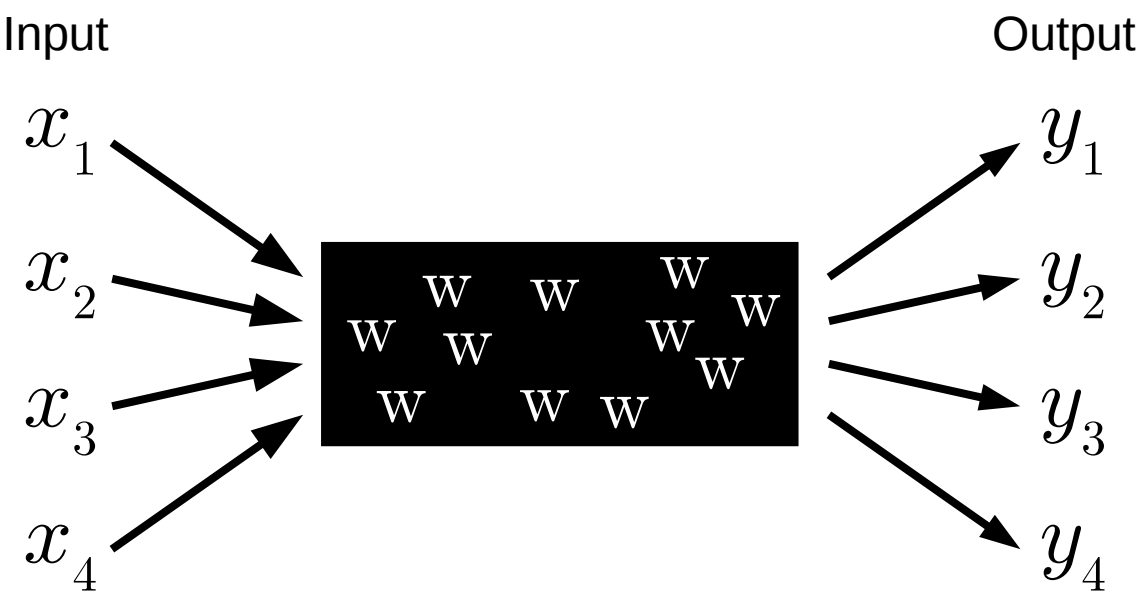
Very comprehensive but still compressed book which covers most topics of the class

Covers all necessary mathematical background knowledge



Training Artificial Neural Networks

Training



Weights and biases are adjustable parameters

Output as function of inputs and network parameters

$$y = f(x; \theta)$$

Desired output as a function of the input

$$\hat{y} = g(x)$$

Try to approximate $g(x)$ with $f(x; \theta)$

Loss function (a.k.a cost or error function)

How well does our ANN perform?

Output

$$y = f(x; \theta)$$

Desired output

$$\hat{y} = g(x)$$

Introduce a loss function which measures the difference between the output of the ANN and the desired output

$$\text{loss} = L(y, \hat{y})$$

Training the network means minimizing the loss

Crucial insight

The loss is a function of the network function

The network function is a function of the input and the parameters

The loss is a function of the parameters

$$\text{loss} = L(y, \hat{y})$$

$$\uparrow$$

$$y = f(x; \theta)$$

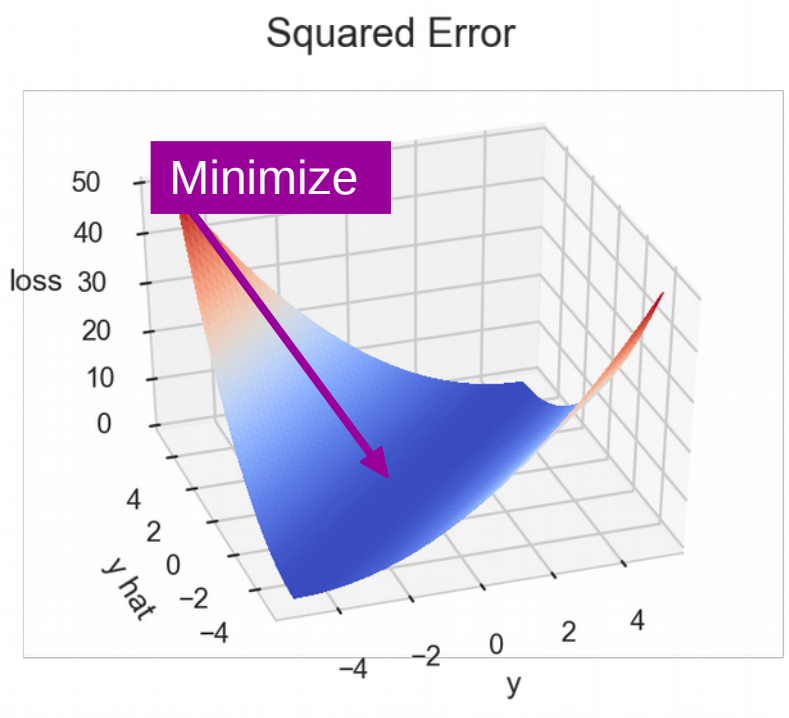
Training:
Minimize loss by modifying the parameters

Example

Sum Squared Error Function

$$loss = \sum_i \frac{1}{2}(y_i - \hat{y}_i)^2$$

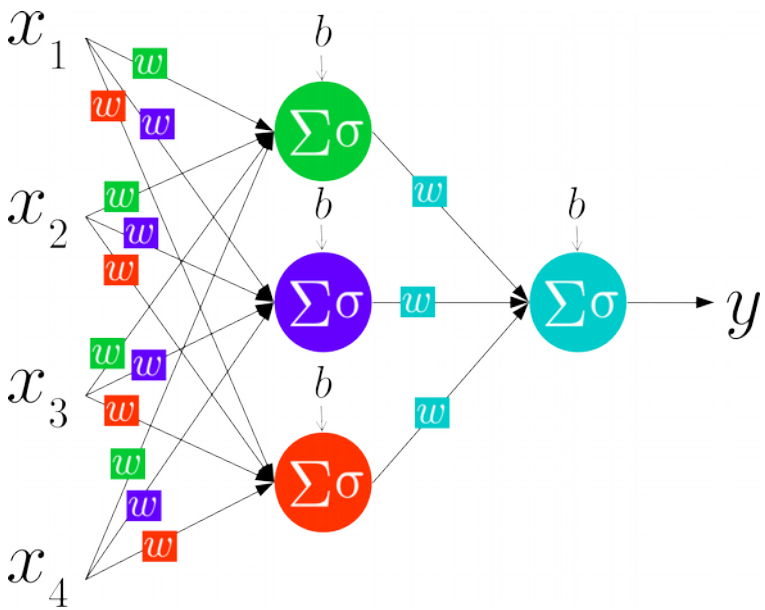
Sums over the squared difference of several samples



Example

The loss is determined by the network function

$$loss = \sum_i \frac{1}{2}(y_i - \hat{y}_i)^2$$



Insert

$$y = \sigma(\sigma(\bar{x}^T W_1)\bar{w}_2)$$

Get

The loss as a function of the network function

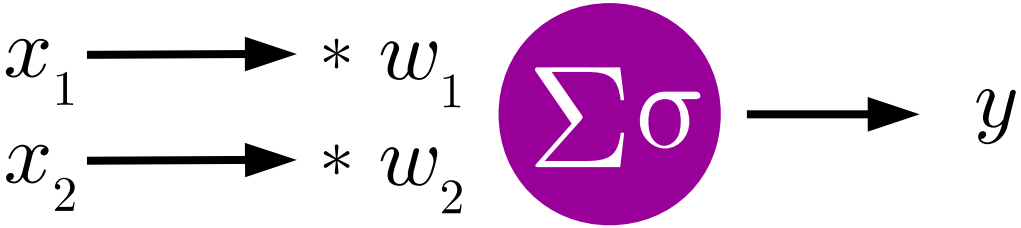
$$loss = \sum_i \frac{1}{2}\left(\sigma(\sigma({x_i}^T W_1)\bar{w}_2) - \hat{y}_i\right)^2$$

The Error Surface

Networks

One weight

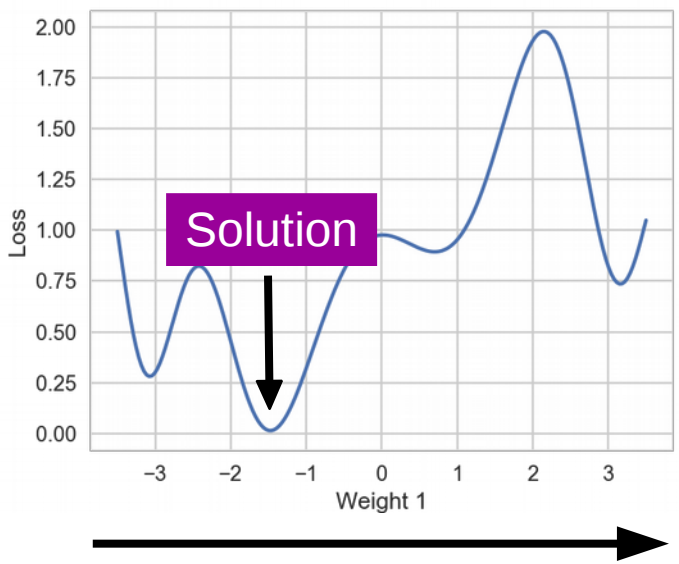
Two weights



Brute force

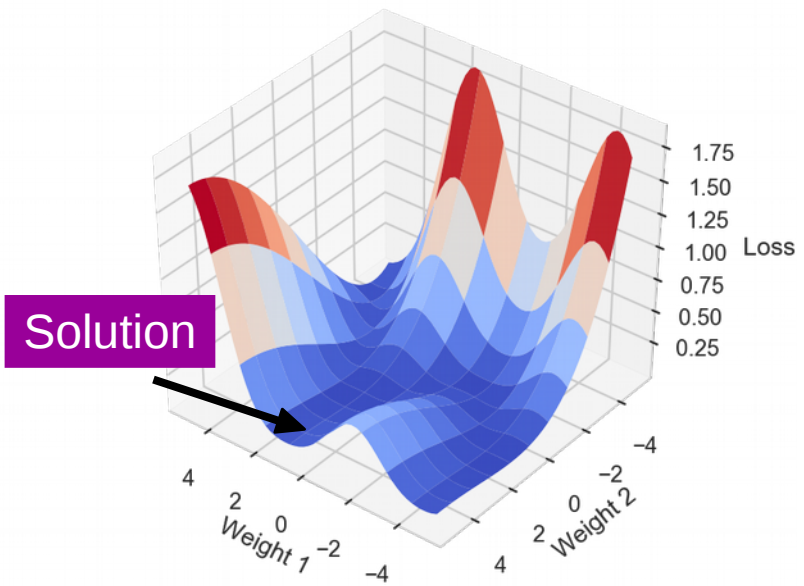
Training: Reduce loss by adjusting the weights

One weight



Calculate loss for different values of weight 1

Two weights



Brute force

Training: Reduce loss by adjusting the weights

Trying c values of n Weights

Exponential class: (c^n)

Network with 20 weights

$$10^{20} = 100\ 000\ 000\ 000\ 000\ 000\ 000$$

At 0.0001 seconds per test

$$\approx 317 \text{ million years}$$

AlexNet has 60 million weights

[1]

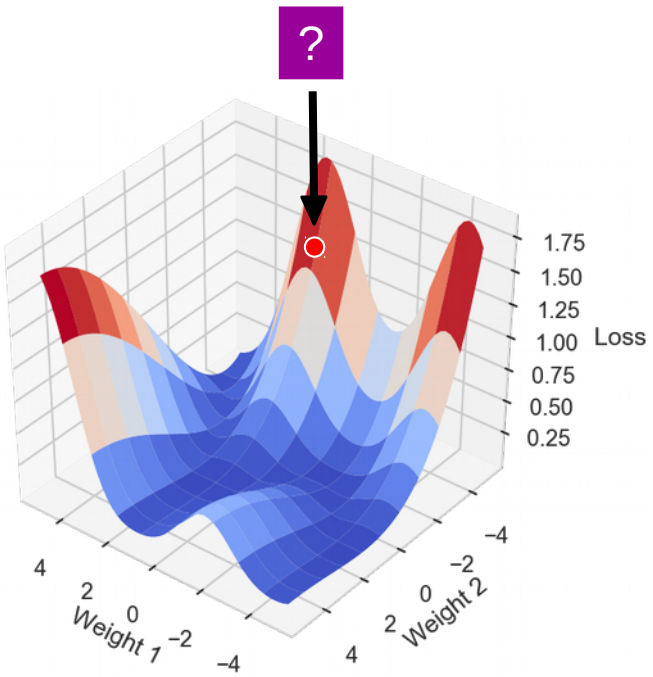
[1] Krizhevsky et al. 2012

Going downhill

Which way is downhill?

How does the loss change, with respect to the weights?

Use calculus: Calculate the partial derivatives!



Derivatives, Partial Derivatives, Gradients and the Jacobian

Derivative

The derivative (a.k.a rate of change) of a function f, is a measure of the rate at which the output of that function changes with respect to it's input variable(s)

For linear functions in slope intersect form, this is equivalent to the parameter m

$$f(x) = mx + b$$

Change of output

$$m = \frac{\Delta y}{\Delta x} = \frac{y_1 - y_2}{x_1 - x_2}$$

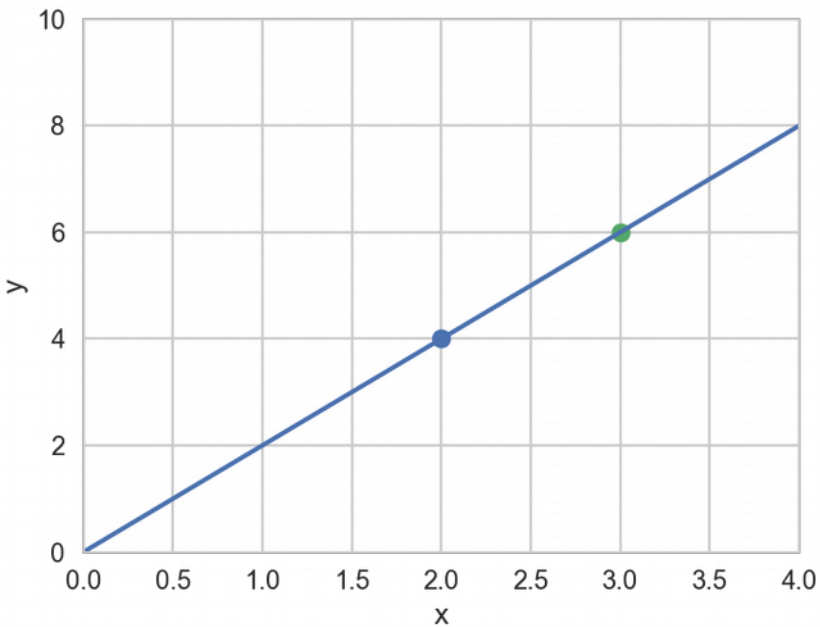
Change of input

Derivative example

The rate of change of a line function can be calculated by selecting two points on that line and dividing their rate of change in y by the rate of change in x

$$m = \frac{\Delta y}{\Delta x} = \frac{y_1 - y_2}{x_1 - x_2}$$

$$m = \frac{6 - 4}{3 - 2} = 2$$



Derivative

How to find the derivative for non-linear functions? We know how to calculate the slope of a line!

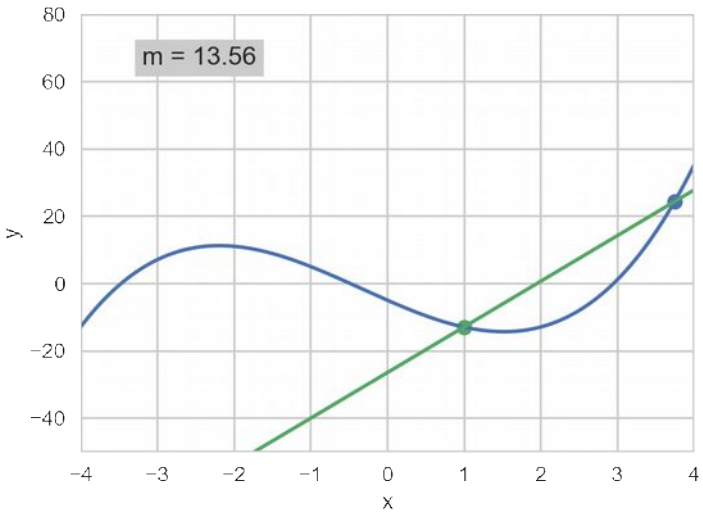
1) Select two points of that function
2) Make the distance between these two points indefinitely small
3) Calculate the slope of the line that connects these two points

Change of output

$$m = \frac{\Delta y}{\Delta x} = \frac{y_1 - y_2}{x_1 - x_2}$$

$$f'(x) = \frac{\Delta f(x)}{\Delta x} = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

Change of input



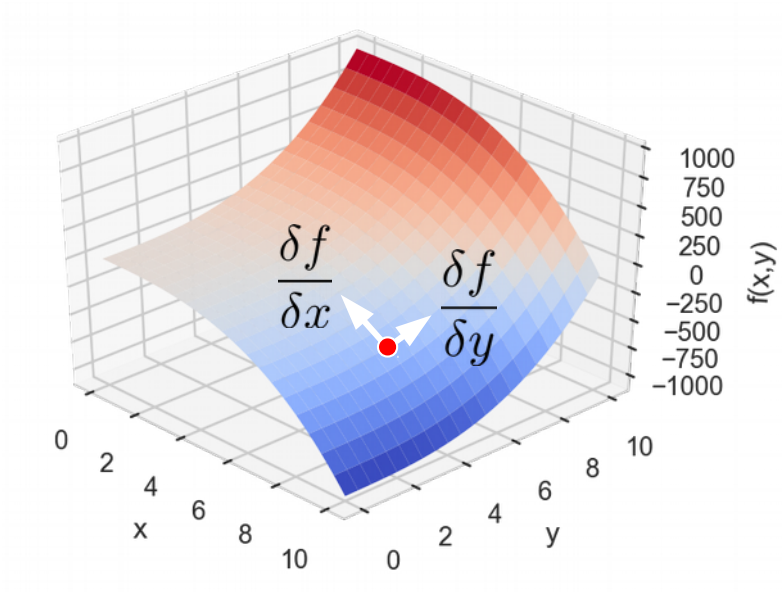
Derivatives of multi-variable functions

The derivative with respect to a single variable in a multi-variable function is called ***partial derivative***

$$f(x, y) = -x^3 - x^2 + y^3 + 3$$

Change of *f* → $\frac{\delta f}{\delta x} = -3x^2 - 2x$

With respect to *x* → $\frac{\delta f}{\delta y} = 3y^2$



The gradient

The ***gradient*** is the “derivative” of multi-variable functions, which map a vector onto a point. The gradient is the vector that contains all partial derivatives of that function and points in the direction of the slope of the function in that point.

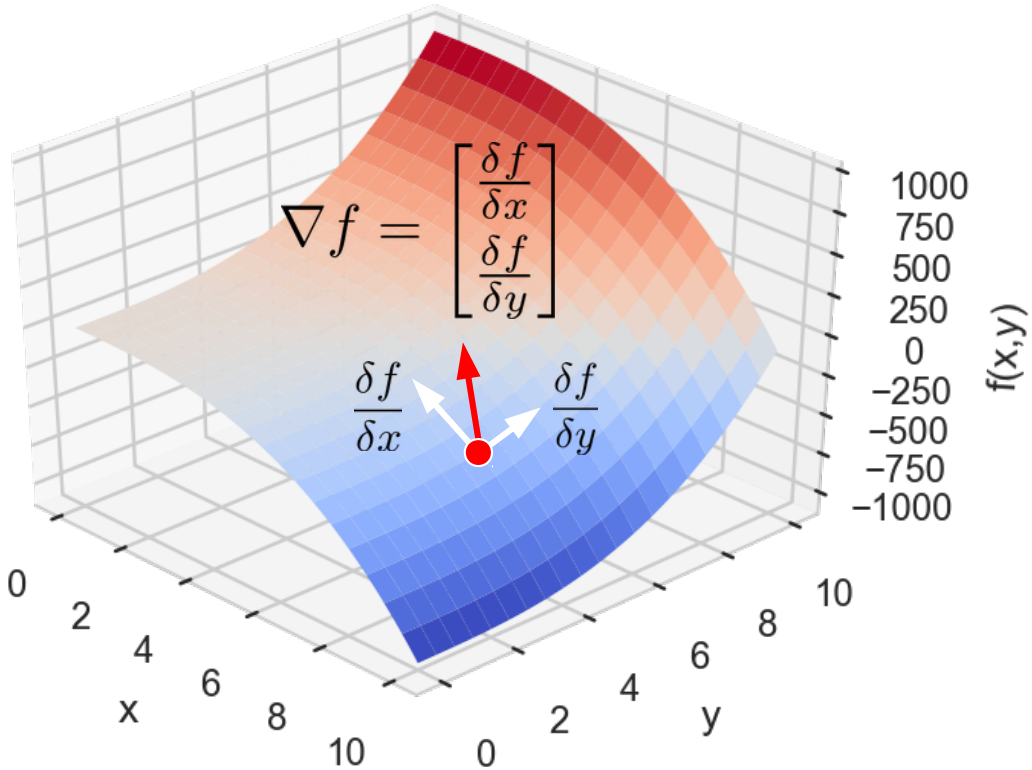
$$f : \mathbb{R}^n \rightarrow \mathbb{R}$$

$$\nabla f = \begin{bmatrix} \frac{\delta f}{\delta \theta_1} \\ \frac{\delta f}{\delta \theta_2} \\ \cdots \\ \frac{\delta f}{\delta \theta_n} \end{bmatrix}$$

Nabla

Example

The gradient (in red) is the vector that contains all partial derivatives (in white) and points in the direction of the slope of the function in that point



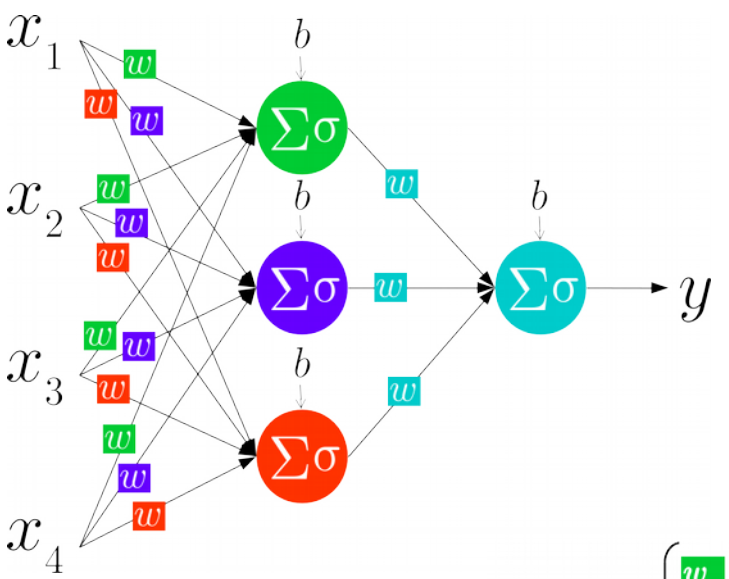
The Jacobian matrix

The ***Jacobian*** is the “derivative” of multi-variable functions, which map a vector onto a vector. The Jacobian is the matrix that contains all partial derivatives of that function and points in the direction of the slope of the function in that point.

$$f : \mathbb{R}^n \rightarrow \mathbb{R}^m \qquad J = \begin{bmatrix} \frac{\delta f_1}{\delta x_1} & \cdots & \frac{\delta f_n}{\delta x_n} \\ \vdots & \ddots & \vdots \\ \frac{\delta f_m}{\delta x_1} & \cdots & \frac{\delta f_m}{\delta x_n} \end{bmatrix}$$

Example

The Jacobian is the matrix that contains all partial derivatives of that function and points in the direction of the slope of the function in that point.



$$\bar{x} = \begin{bmatrix} x_1 & x_2 & x_3 & x_4 \end{bmatrix}^T \quad W_1 = \begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \\ w_{41} & w_{42} & w_{43} \end{pmatrix}$$

$$\bar{d}_1 = \bar{x}^T W_1$$

$$J_{W_1} = \begin{bmatrix} \frac{\delta d_{11}}{\delta W_{11}} & \frac{\delta d_{12}}{\delta W_{12}} & \frac{\delta d_{13}}{\delta W_{13}} \\ \frac{\delta d_{11}}{\delta W_{21}} & \frac{\delta d_{12}}{\delta W_{22}} & \frac{\delta d_{13}}{\delta W_{23}} \\ \frac{\delta d_{11}}{\delta W_{31}} & \frac{\delta d_{12}}{\delta W_{32}} & \frac{\delta d_{13}}{\delta W_{33}} \\ \frac{\delta d_{11}}{\delta W_{41}} & \frac{\delta d_{12}}{\delta W_{42}} & \frac{\delta d_{13}}{\delta W_{43}} \end{bmatrix}$$

Linear algebra calculus

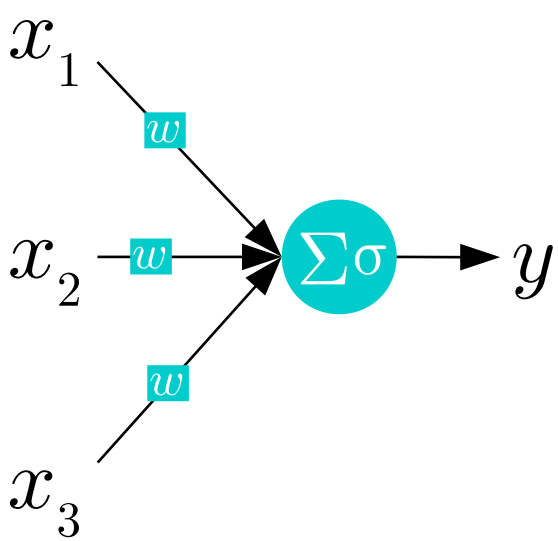
The gradient in vector multiplication

$$f(x,y) = x^T y$$
$$= \left[x_1, x_2, ..., x_n\right] \begin{bmatrix} y_1 \\ y_2 \\ ... \\ y_n \end{bmatrix} = x_1 y_1 + x_2 y_2 + ... + x_n y_n$$

$$\nabla_y f = \begin{bmatrix} \frac{\delta f}{y_1} \\ \frac{\delta f}{y_2} \\ ... \\ \frac{\delta f}{y_n} \end{bmatrix} = x \qquad \nabla_{x^T} f = \begin{bmatrix} \frac{\delta f}{x_1} & \frac{\delta f}{x_2} & ... & \frac{\delta f}{x_n} \end{bmatrix} = y^T$$

Example

Derivative of network function with respect to weight vector



$$\bar{x} = \begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix}^T \qquad \bar{w} = \begin{pmatrix} w_{11} \\ w_{21} \\ w_{31} \end{pmatrix}$$

$$y = \bar{x}^T \bar{w} = \begin{array}{l} \quad x_1 * w_{11} \\ + \; x_2 * w_{21} \\ + \; x_3 * w_{31} \end{array}$$

$$\boxed{\nabla_{\bar{w}} y = \bar{x}}$$

Linear algebra calculus

The gradient in vector matrix multiplication

$$f(x, Y) = x^T Y$$

$$= \left[x_1, x_2, ..., x_m\right] \begin{bmatrix} y_{11} & y_{12} & \cdots & y_{1n} \\ y_{21} & y_{22} & \cdots & y_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ y_{m1} & y_{m2} & \cdots & y_{mn} \end{bmatrix}$$

$$= \left[x_1 y_{11} + x_2 y_{21} + \cdots + x_m y_{m1}, \quad x_1 y_{12} + x_2 y_{22} + \cdots + x_m y_{m2}, \quad \cdots, \quad x_1 y_{1n} + x_2 y_{2n} + \cdots + x_m y_{mn}\right]$$

$$\nabla_x f = \begin{bmatrix} \frac{\delta f}{x_1} \\ \frac{\delta f}{x_2} \\ \cdots \\ \frac{\delta f}{x_n} \end{bmatrix} = Y$$

Linear algebra calculus

The Jacobian in vector matrix multiplication

$$f(x, Y) = x^T Y$$

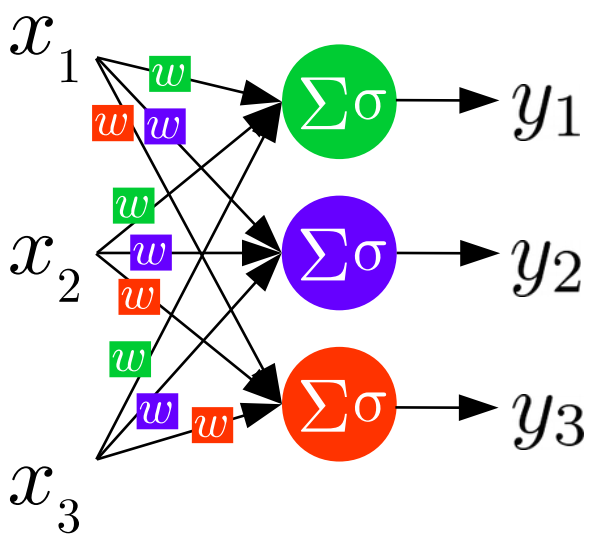
$$= \begin{bmatrix} x_1, x_2, ..., x_m \end{bmatrix} \begin{bmatrix} y_{11} & y_{12} & \cdots & y_{1n} \\ y_{21} & y_{22} & \cdots & y_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ y_{m1} & y_{m2} & \cdots & y_{mn} \end{bmatrix}$$

$$= \begin{bmatrix} x_1 y_{11} + x_2 y_{21} + \cdots + x_m y_{m1}, & x_1 y_{12} + x_2 y_{22} + \cdots + x_m y_{m2}, & \cdots, & x_1 y_{1n} + x_2 y_{2n} + \cdots + x_m y_{mn} \end{bmatrix}$$

$$J_Y f = \begin{bmatrix} \frac{\delta f}{y_{11}} & \frac{\delta f}{y_{12}} \cdots & \frac{\delta f}{y_{1n}} \\ \frac{\delta f}{y_{21}} & \frac{\delta f}{y_{22}} \cdots & \frac{\delta f}{y_{2n}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\delta f}{y_{m1}} & \frac{\delta f}{y_{m2}} \cdots & \frac{\delta f}{y_{mn}} \end{bmatrix} = \begin{bmatrix} x_1 & x_1 & \dots & x_1 \\ x_2 & x_2 & \dots & x_2 \\ \vdots & \vdots & \ddots & \vdots \\ x_3 & x_3 \dots & x_3 \end{bmatrix} = \bar{x} \begin{bmatrix} 1 & 1 & \dots & 1 \end{bmatrix} = \bar{x}\bar{1}$$

Example

Derivative of network function with respect to weight matrix



$$\bar{x} = \begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix}^T \qquad W = \begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{pmatrix}$$

$$\bar{y} = \bar{x}^T W = \begin{pmatrix} x_1 * w_{11} + x_2 * w_{21} + x_3 * w_{31} \\ x_1 * w_{12} + x_2 * w_{22} + x_3 * w_{32} \\ x_1 * w_{13} + x_2 * w_{23} + x_3 * w_{33} \end{pmatrix}^T$$

$$J_W y = \bar{x} \begin{bmatrix} 1 & 1 & 1 \end{bmatrix} = \bar{x}\bar{1} = \begin{pmatrix} x_1 & x_1 & x_1 \\ x_2 & x_2 & x_2 \\ x_3 & x_3 & x_3 \end{pmatrix}$$

The chain rule

Let $f(x)$ be the composition of the functions g and h

$$f(x) = g(h(x))$$

Then the derivative of f is equivalent to the product of the derivative of the outer and the derivative of the inner function

$$f'(x) = g'(h(x))h'(x)$$

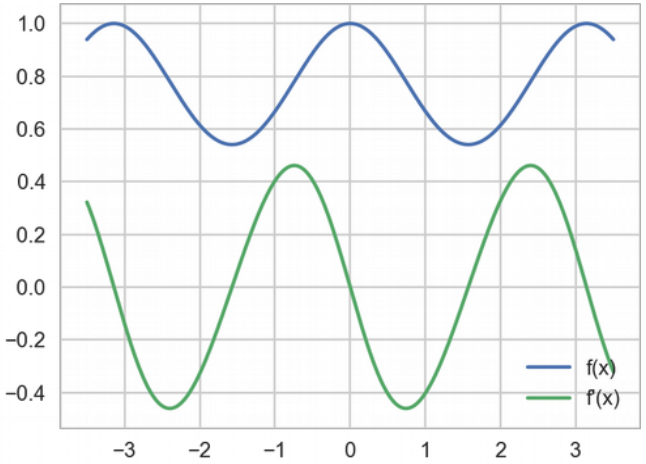
Example

Chain Rule

$$f'(x) = g'(h(x))h'(x)$$

$$f(x) = cos(sin(x))$$

$$f'(x) = cos'(sin(x))sin'(x)$$
$$= sin(sin(x))(-cos(x))$$



Leibniz's notation

Let $f(x)$ be the composition of the functions g and h

$$f(x) = g(h(x))$$
$$f(x) = z = g(y)$$
$$y = h(x)$$

Then we can write the chain rule $f'(x) = g'(h(x))h'(x)$ in Leibniz's notation as

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

The Backpropagation Algorithm

Backpropagation

"Learning representation by back-propagating errors"

David E. Rumelhart, Geoffrey E. Hinton & Ronald J. Williams (1986), Nature

1) Error is backpropagated to individual weights
2) Output of later layers depend on output of earlier layers
3) Derivatives of earlier layers depend on derivatives of later layers (can be **implemented efficiently**)

$$y = \sigma(\sigma(\bar{x}^T W_1)\bar{w}_2)$$

$$f'(x) = g'(h(x))h'(x)$$

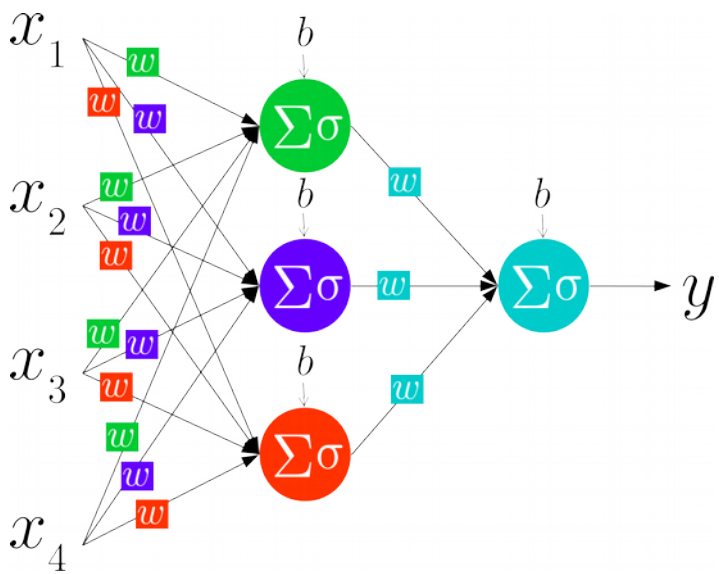
$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

Backpropagation in Multilayer Perceptrons

Multilayer perceptron

The loss is determined by the network function

$$loss = \sum_i \frac{1}{2}(y_i - \hat{y}_i)^2$$



Insert

$$y = \sigma(\sigma(\bar{x}^T W_1)\bar{w}_2)$$

Get

The loss as a function of the network function

$$loss = \sum_i \frac{1}{2}\left(\sigma(\sigma({x_i}^T W_1)\bar{w}_2) - \hat{y}_i\right)^2$$

Partial derivatives

How does the loss change, with respect to the weights?

$$loss = \sum_i \frac{1}{2}\left(\sigma(\sigma({x_i}^T W_1)\bar{w}_2) - \hat{y}_i\right)^2$$

Calculate the partial derivatives of the loss function with respect to the parameters. In other words, the gradient $\nabla_{w_2}\mathrm{loss}(y, \hat{y})$ and the Jacobian $J_{W_1}\mathrm{loss}(y, \hat{y})$.

Gradient for w2

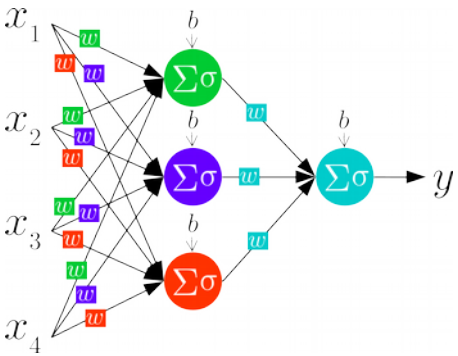
Sums over samples

$$\nabla_{w_2} \text{loss}(y, \hat{y}) = \nabla_{w_2} \sum_j \frac{1}{2}(y_j - \hat{y}_j)^2$$

Derivative of the sums equals the sum of the derivatives (sum rule of differentiation)

$$\nabla_{w_2} \text{loss}(y, \hat{y}) = \sum_j \nabla_{w_2} \frac{1}{2}(y_j - \hat{y}_j)^2$$

Gradient for w2



Inner

Outer

$$\nabla_{w_2}\text{loss}(y,\hat{y}) = \sum_j \nabla_{w_2}\frac{1}{2}(y_j - \hat{y}_j)^2$$

Apply the chain rule

$$= \sum_j (y_j - \hat{y}_j)\nabla_{w_2}(y_j - \hat{y}_j)$$

y hat

$$= \sum_j (y_j - \hat{y}_j)(\nabla_{w_2} - \sigma(\bar{a_1}\bar{w_2}))$$

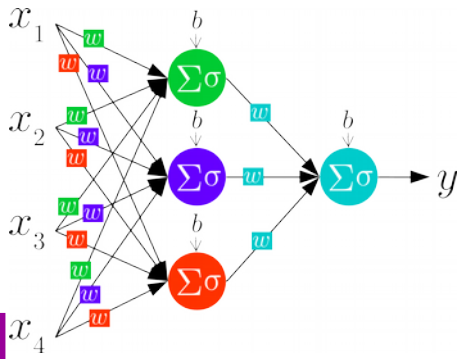
$$\bar{a_1} = \sigma(\bar{x}^T W_1)$$

Apply the chain rule

$$= \sum_j (y_j - \hat{y}_j)(-\sigma'(\bar{a_1}\bar{w_2}))\nabla_{w_2}\bar{a_1}\bar{w_2}$$

$$= \sum_j (y_j - \hat{y}_j)(-\sigma'(\bar{a_1}\bar{w_2}))\bar{a_1}$$

Jacobian for W1



Inner

Outer

$$J_{W_1}\text{loss}(y,\hat{y}) = \sum_j J_{W_1}\frac{1}{2}(\overbrace{y_j - \hat{y}_j})^2$$

y hat

Apply the chain rule

$$= \sum_j (y_j - \hat{y}_j) J_{W_1}(-\overbrace{\sigma(\bar{a_1}\bar{w_2})})$$

Apply the chain rule

$$= \sum_j (y_j - \hat{y}_j)(-\sigma'(\bar{a_1}\bar{w_2}))J_{W_1}\bar{a_1}\bar{w_2} \qquad \bar{a_1} = \sigma(\bar{x}^T W_1)$$

Apply the chain rule

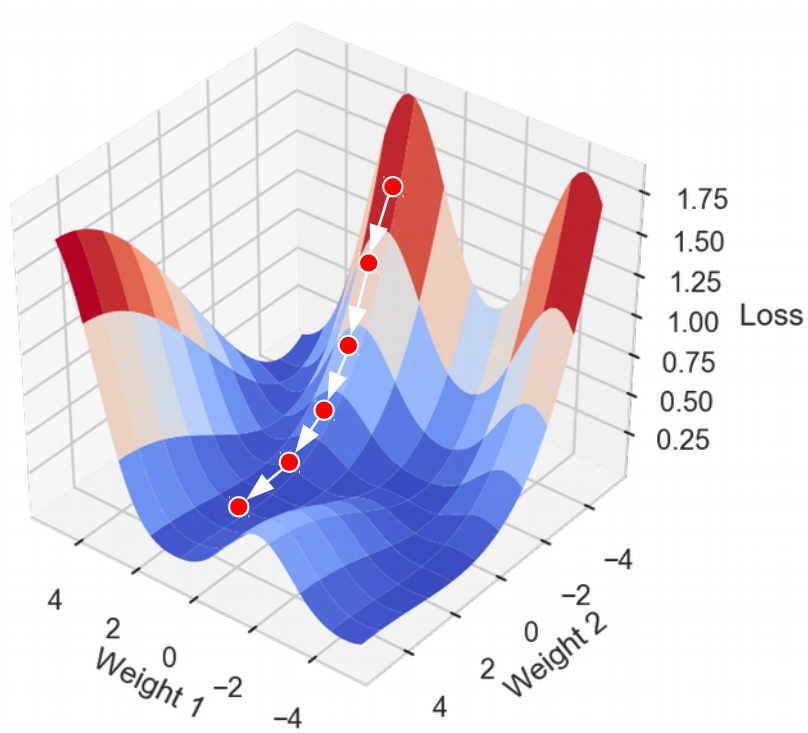
$$= \sum_j (y_j - \hat{y}_j)(-\sigma'(\bar{a_1}\bar{w_2}))\bar{w_2}^T J_{W_1}\sigma(x^T W_1)$$

Apply the chain rule

$$= \sum_j (y_j - \hat{y}_j)(-\sigma'(\bar{a_1}\bar{w_2}))\bar{w_2}^T \sigma'(x^T W_1)(x\bar{1})$$

Gradient Descent

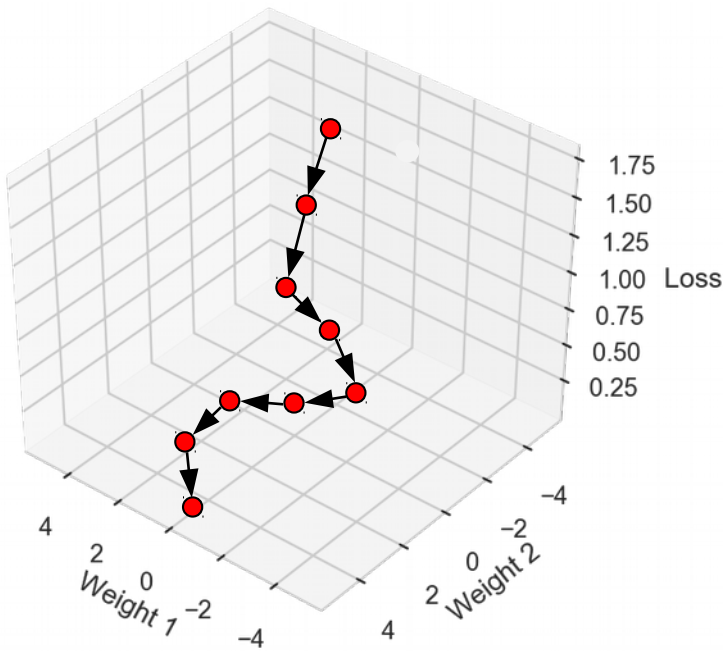
Algorithm



Walking downhill

1) Initialize weights
2) Determine which way is downhill by calculating the gradient
3) Take a step downhill by updating the weights
4) Continue with step 2

Algorithm



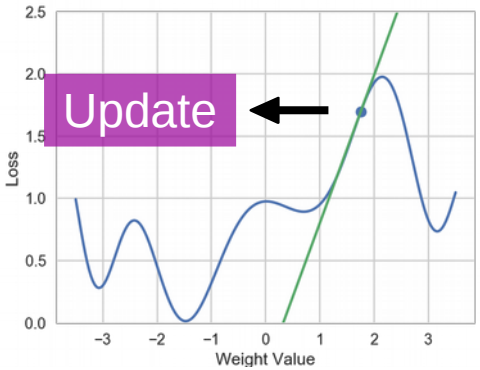
The real, underlying error surface is unknown to us!

Trying to get downhill

1) Initialize weights
2) Stand in the fog
3) Determine which way is downhill by calculating the gradient
4) Teleport yourself to a position in a certain direction
5) Continue with step 2

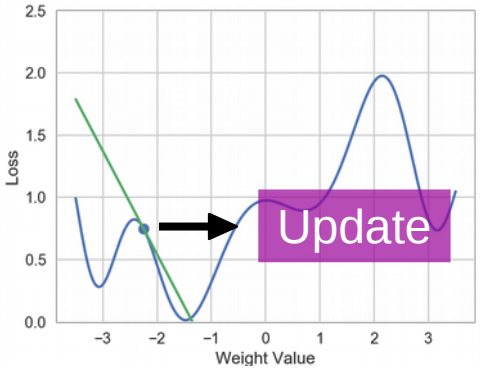
Updating the parameters

If $\frac{\delta f}{\delta \theta_i} > 0$, decrease θ_i to walk downhill



$$\frac{\delta f}{\delta w_1} \approx 1.188$$

If $\frac{\delta f}{\delta \theta_i} < 0$, increase the parameter to walk downhill



$$\frac{\delta f}{\delta w_1} \approx -0.831$$

Formalization

The gradient descent parameter update rule

Old parameters

Gradient with respect to parameters

$$\theta_{\text{new}} = \theta_{\text{old}} - \eta \nabla_\theta f$$

Updated parameters

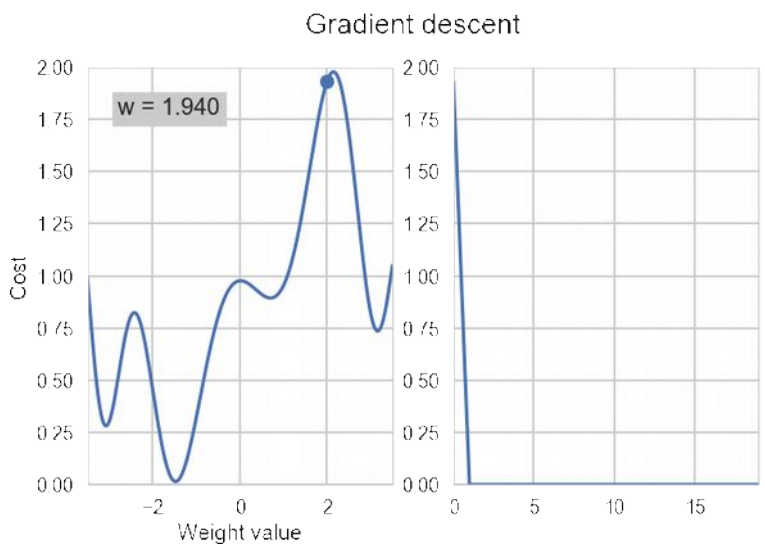
Go downhill

Step size (a.k.a. learning rate)

The same update rule can be applied to derivatives and Jacobians

Example

Stepwise movements against the gradient for different initial weight values. Learning rate is set to 0.1



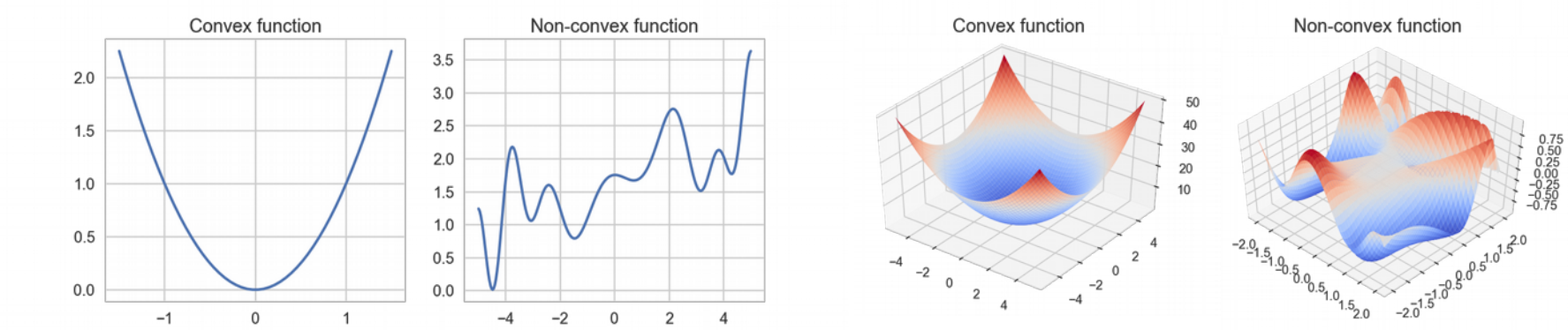
Gradient descent halts in flat regions, since the gradient is close or equal to 0

Gradient descent is not guaranteed to find an optimal solution for the optimization problem!

Gradient Descent and the Shape of the Error Surface

Convex functions

Convex functions are functions, with only one minimum

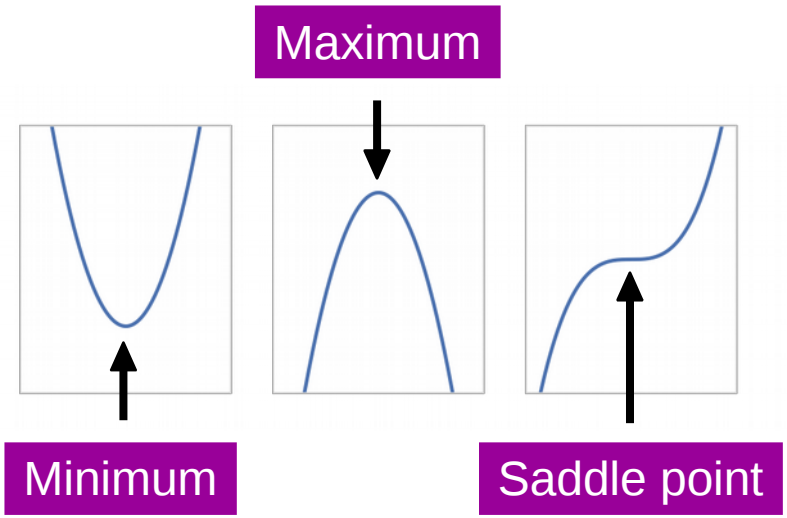


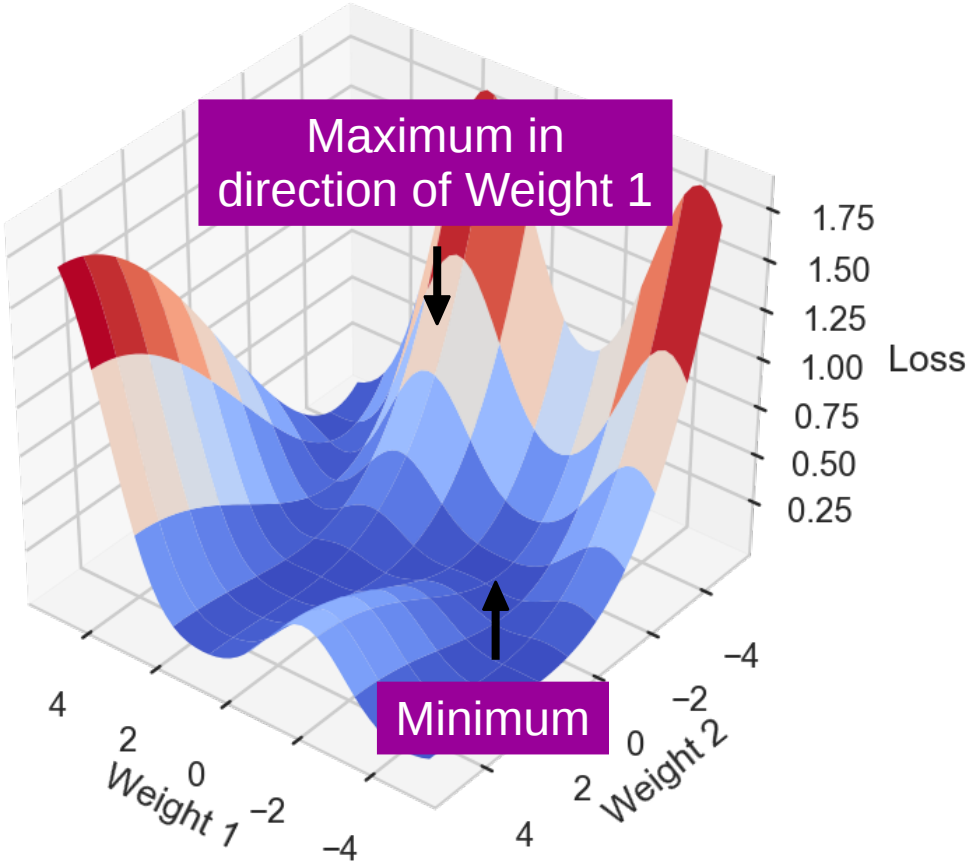
In many ML approaches (logistic regression, Support vector machines …) problems are reduced to a convex problem

The error surface of artificial neural networks is (in most of the cases) non-convex

Non-convex functions

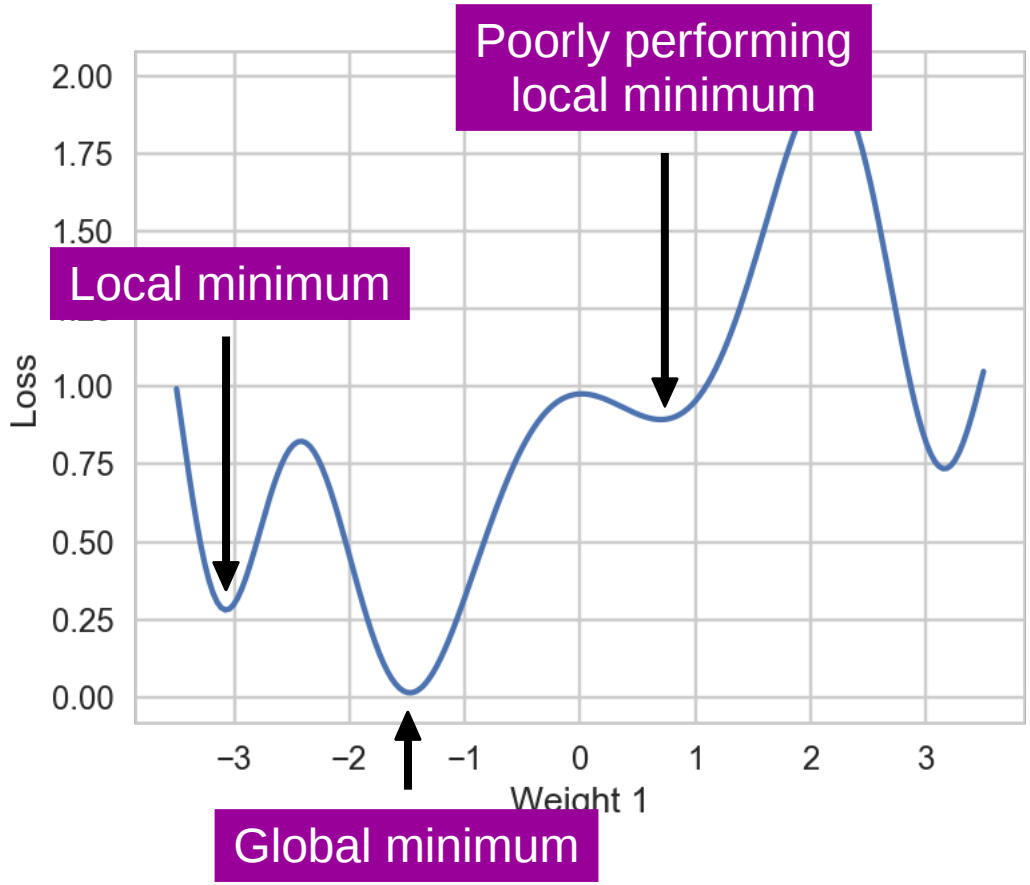
Critical points: points with zero slope





Non-convex functions

Different kinds of minima



Ideally, we want to reach the global minimum, however a local minimum might still serves as a good solution

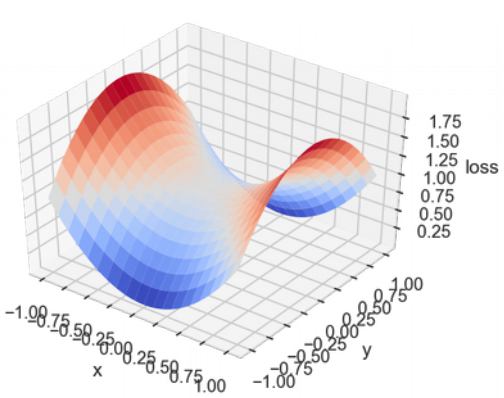
Non-convex functions

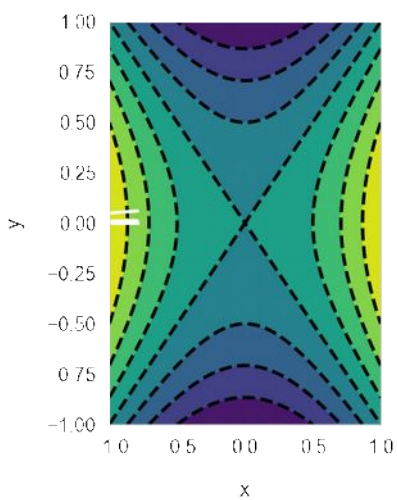
Is it a problem, that gradient descent gets stuck at critical points?

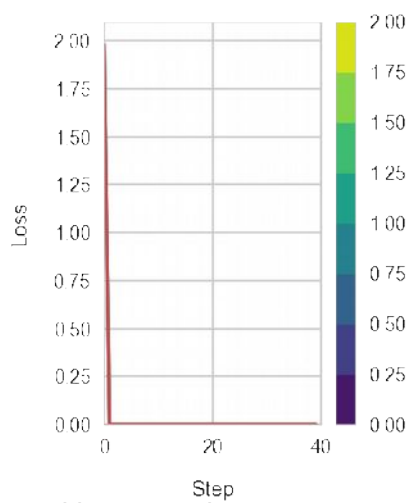
How does a N-Dimensional error surface look like?

In 2-D space it is already possible to walk around maxima

How likely is it, to get stuck on a saddle point?







*Only if the parameter is initialized **exactly** on the saddle, otherwise it will diverge from it*

Batch, Mini-Batch and Stochastic Learning

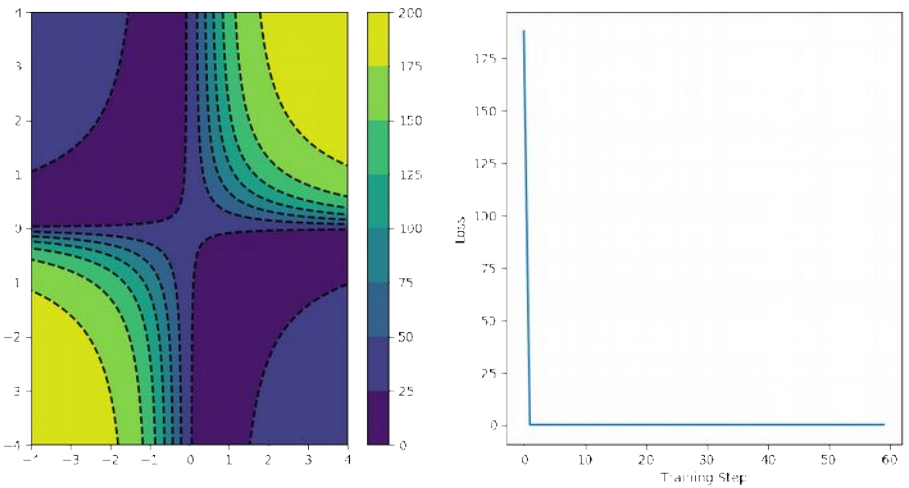
Batch learning

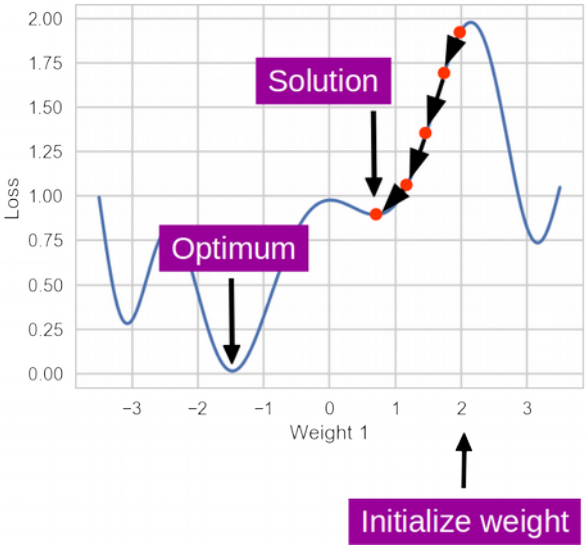
Steps of batch learning

1) Initialize weights
2) Calculate gradient for **whole** dataset
3) Update weights
4) Continue with 2

Reaches local minimum!

Often impractical, since datasets are to large



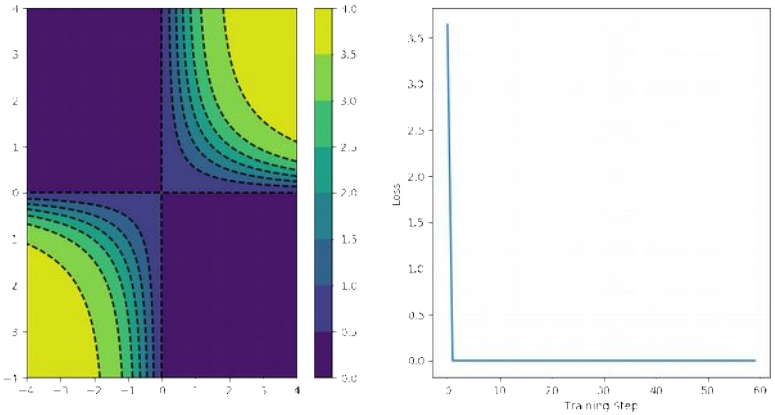


Stochastic learning

Steps of stochastic learning

1) Initialize weights
2) Calculate loss for a **single** sample
3) Calculate gradients
4) Update weights
5) Continue with 2

Reaches global minimum **(eventually)!**



Stochastic learning

Advantages of stochastic learning

1) Noisy, wobbles around and can hence escape from local minima
2) Less computations per learning step and hence much faster
3) Results often in better solutions

Disadvantages of stochastic learning

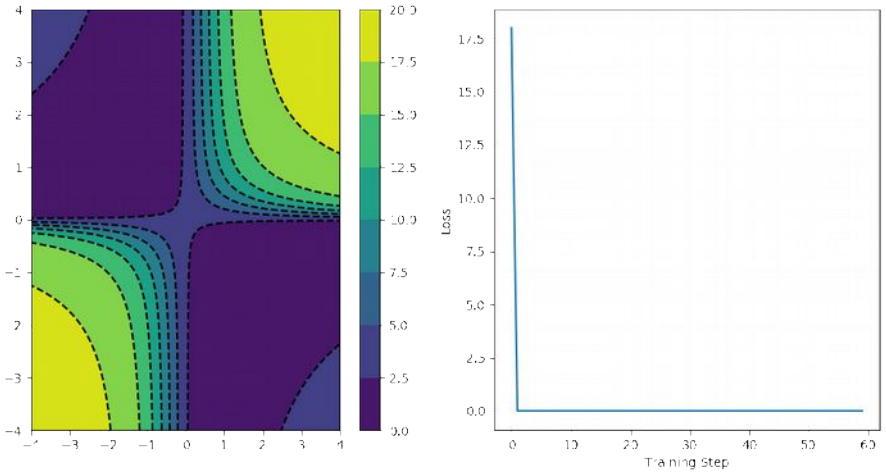
1) Noisy, wobbles around very strongly and hence might does not converge
2) Small computations do not use full capacity
3) Conditions of convergences are hidden

Mini-Batch learning

Steps of stochastic learning

1) Initialize weights
2) Calculate loss for a **subset** of the training samples
3) Calculate gradients
4) Update weights
5) Continue with 2

Reaches global minimum (eventually)!



Mini-batch learning

Advantages of mini-batch learning

1) Wobbles around slightly and can hence escape from local minima
2) Can fully exhaust computational capacity
3) Results in good solutions

Disadvantages of mini-batch learning

1) Slightly Wobbles around the solution

Mini-Batch Composition

Mini-batch composition

- Network learns fastest from unexpected and new examples
- Samples from same class contain similar information: Compose mini-batches from samples of different classes
- Same sample creates different error surfaces when it is contained in varying mini-batch combinations
- Shuffle your samples and do not reuse mini-batches

Weighting inputs

- Networks learn fastest from unexpected and new examples
- Present samples with larger errors more frequently
- Improves performance of rare inputs
- Be careful: Do not learn outliers!

Example: Categorize animals from images:

1000x Lions
1000x Gorilla
120x Slender Loris

Show Loris more often

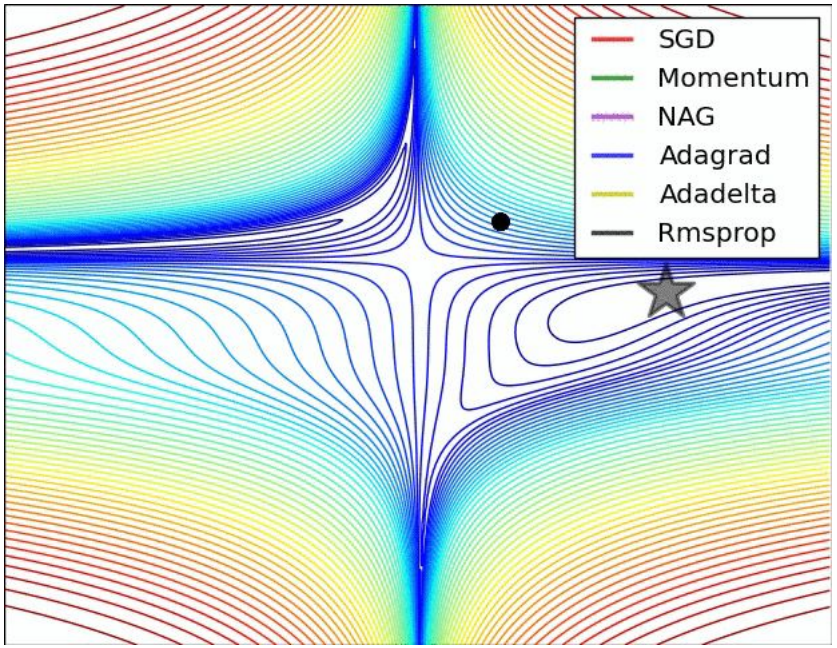
[2]

[2] wikipedia.org

Optimize Gradient Methods

Monday November 6th

Preview



[3]

[3] sebastianruder.com

Further resources

- YouTube playlist “Neural Networks Demystified” by Stephen C. Welch (especially part 3 and 4)
- MIT lecture on neural nets (part 1, part2)
- A book chapter about the derivation of the backpropagation algorithm
- The paper “Efficient BackProp” by Yann LeCun
- An overview of gradient descent algorythms by Sebastian Ruder
- Advanced: Ben Recht’s talks on optimization

This Week’s Task

Single variable gradient descent

```
import numpy as np

def function(x, derive = False):
    if derive:
        return 2*x
    return x**2

learning_rate = 0.1
weight = 3

for training_step in range(20):
    print(function(weight))
    weight = weight - learning_rate * function(weight, True)
```

Updating a plot in a loop

```
import numpy as np
import matplotlib.pyplot as plt
from time as sleep
%matplotlib notebook # activate inline plotting in ipython notebooks

fig, ax = plt.subplots()

x = np.linspace(0, 20, 200)
for step in range(20):
    ax.cla() # clear plot
    ax.plot(x, np.random.rand(x.shape[0]))
    fig.canvas.draw() # update plot
    sleep(0.5)
```

Creating contour plots

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib notebook # activate inline plotting in ipython notebooks

fig, ax = plt.subplots()
x = np.linspace(-1, 1, 40)
y = np.linspace(-1, 1, 40)
X, Y = np.meshgrid(x, y)
Z = X**1 - Y**2 + 1

# plot contour
cp = ax.contourf(X, Y, Z, cmap = "viridis")
# plot contour lines
ax.contour(X, Y, Z, colors = "black", linestyles = "dashed")
fig.colorbar(cp) # add color-bar to figure
```

THANK YOU!

6

[4] giphy.com

Sources

[1] *Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." Advances in neural information processing systems. 2012.*

[2] By Kalyan Varma (Kalyanvarma) - Own work, GFDL 1.2, *https://commons.wikimedia.org/w/index.php?curid=12692513*

[3] Retrieved October 29th, 2016 from http://sebastianruder.com/optimizing-gradient-descent/

[4] Retrieved October 30th, 2017 from https://giphy.com/gifs/banana-peel-EzPlK1kEUZQti

[5] Retrieved October 6 th , 2016 from

https://www.youtube.com/watch?v=S4ZUwgesjS8&lc=z131gh2j4mikihiqe04cizzhxr2bedrr3nw0k