Implementing ANNs with TensorFlow

Session 05 - More on Deep Neural Networks

Last Week

- Last lecture we learned how to implement a simple neural network in TensorFlow to solve a regression task.

- In the homework you programmed another neural network yourself to solve a classification task.

- But which tasks can you solve with a neural network?

- And how can you make the optimization more efficient?

Agenda

1. Tasks for DNNs
2. Output Functions
3. Loss Functions
4. Training and Test Data
5. Optimization Strategies
6. Optimizer

Tasks for DNNs

Tasks for DNNs

Which kinds of tasks can a neural network solve?

Tasks for DNNs

Which kinds of tasks can you solve with a neural network?

Not available due to copyright reasons.

George Cybenko

- Every task that you can formulate as a mapping from one space to another space.
- Neural networks are universal function approximators (Cybenko, 1989).
- Already a network with only one hidden layer can in theory approximate every function. (Note: It might require an extremely large number of neurons though.)

Tasks for You

If a neural network can approximate everything, what is left for us?

- You have to come up with the correct mapping!
- There actually has to be a mapping!
- You have to find “good” data.
- You have to use a clever architecture.
- You have to define the loss function and the optimizer.
- You have to supervise the training and make sure that the network learns what it is supposed to learn.

Tasks for You

If a neural network can approximate everything, what is left for us?

- You have to come up with the correct mapping!
- There actually has to be a mapping!
- You have to find “good” data.
- You have to use a clever architecture.
- You have to define the loss function and the optimizer.
- You have to supervise the training and make sure that the network does what it is supposed to do.

Today

Encoding

Encoding

- If you decide for a task that you want to solve, you first have to formulate it in a way that it is an “input-to-output” mapping.

- This requires that you think about how you can represent the input and the output in a mathematical way; you need to define the encoding.

Encoding - Example

Goal: Build a emotion detection algorithm for faces.

Mapping: Image of face → Emotion.

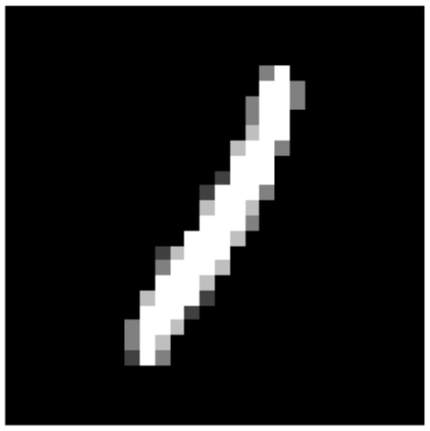
→ **“happy”**

But how to represent both as mathematical entities?

Image Space

- Images are defined in pixel values.
- Most common are gray scale or RGB scale.
- In gray scale each pixel has a value between 0 and 255, such that an image $\overrightarrow{x} \in [0,255]^{h \times w}$ (h is height, w is width).
- For RGB scale each pixel has three values (red, green, blue) between 0 and 255, such that here an image $\overrightarrow{x} \in [0,255]^{h \times w \times 3}$.

Example: Image Space



[ylc]

```
[ [0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0],
  [0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0],
  [0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0],
  [0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0],
  [0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0  12  255   0   0   0   0   0   0   0   0   0],
  [0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 255   255 128   0   0   0   0   0   0   0   0],
  [0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 128 255  255 128   0   0   0   0   0   0   0],
  [0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 128 255  255   0   0   0   0   0   0   0   0],
  [0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 191 255  255   0   0   0   0   0   0   0   0],
  [0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 191 255 255  128   0   0   0   0   0   0   0],
  [0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 255 255 255    0   0   0   0   0   0   0   0],
  [0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0  64 255 255 255    0   0   0   0   0   0   0   0],
  [0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0  64 255 255 255  128   0   0   0   0   0   0   0],
  [0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 191 255 255 191    0   0   0   0   0   0   0   0],
  [0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 255 255 255 128    0   0   0   0   0   0   0   0],
  [0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 255 255 255 191    0   0   0   0   0   0   0   0],
  [0   0   0   0   0   0   0   0   0   0   0   0   0   0   0  64 191 255 255 255    0   0   0   0   0   0   0   0],
  [0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 128 255 255 255 191    0   0   0   0   0   0   0   0],
  [0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 255 255 255 191    0   0   0   0   0   0   0   0   0],
  [0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 191 255 255 255  64   0   0   0   0   0   0   0   0],
  [0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 255 255 255  64   0   0   0   0   0   0   0   0   0],
  [0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 128 255 255 191    0   0   0   0   0   0   0   0   0],
  [0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 128 255 191    0   0   0   0   0   0   0   0   0   0],
  [0   0   0   0   0   0   0   0   0   0   0   0   0   0   0  64 255 128    0   0   0   0   0   0   0   0   0   0],
  [0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0],
  [0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0],
  [0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0],
  [0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0] ]
```

One-Hot Encoding

- But how to encode abstract categories (like emotions)?
- The simplest form is the so called one-hot encoding.
- Each category is simply represented by a n-dimensional array with one 1 and the rest 0.

- E.g. you want to classify four emotions:

“happy” $\widehat{=}$ $\begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$ **“sad”** $\widehat{=}$ $\begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}$ **“angry”** $\widehat{=}$ $\begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}$ **“suprise”** $\widehat{=}$ $\begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$

Mapping

- With this encoding the task of detecting emotions from images simply becomes the approximation of a function

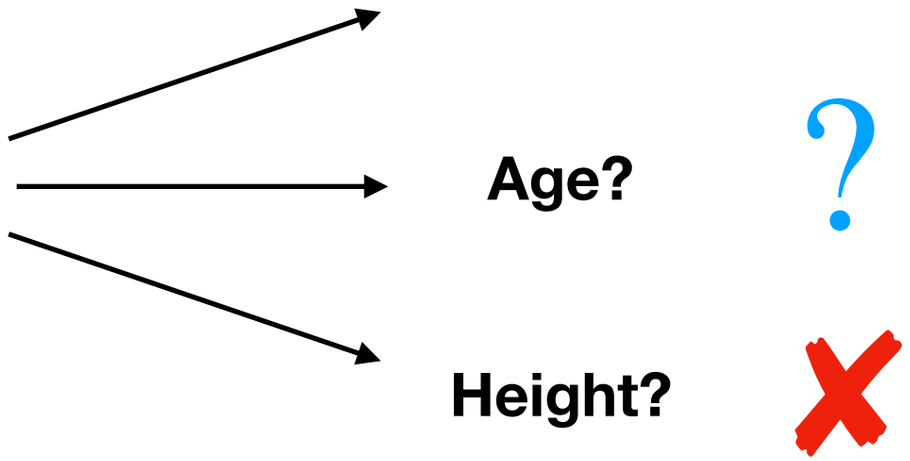
$$f : [0,255]^{h \times w \times 3} \rightarrow [0,1]^4$$

→ **“happy”**

- And as neural networks are universal function approximators a neural network should in general be able to approximate this function.

Sensible Mapping

- Lastly you should think about whether the mapping you want to approximate is actually sensible.



Output Functions

Output Functions

- Depending on the encoding of your output you have to choose a suitable output function (= activation function of last layer).

- There are many different ones.

- I will introduce you to the most common.

Linear Activation

- In the case of a regression task your network usually needs to be able to map to the whole space of $\mathbb{R}^n$.

- Here a linear activation (= no activation at all) makes sense.

$$y_i = d_i^{(N)}$$

i-th output neuron

Drive of i-th output neuron

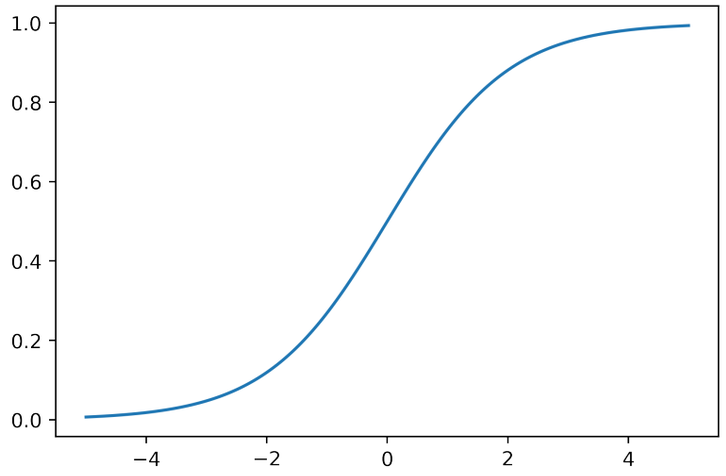
Logistic Activation Function

- If you solve a binary classification problem you want the network to return a number between 0 and 1.

- 0 is for the one class, 1 is for the other classs, values in between resemble uncertainty.

- Here the already introduced logistic activation function (also called sigmoid activation function) is a good choice.

$$y_i = \sigma(d_i^{(N)}) \quad \textbf{with} \quad \sigma(x) = \frac{1}{1 + e^{-x}}$$



Softmax Activation Function

- But how about a classification task with multiple categories?

- We already saw that we would encode such categories in one-hot: $\begin{pmatrix}1\\0\\0\\0\end{pmatrix}, \begin{pmatrix}0\\1\\0\\0\end{pmatrix}, \begin{pmatrix}0\\0\\1\\0\end{pmatrix}, \begin{pmatrix}0\\0\\0\\1\end{pmatrix}.$

- This offers a nice interpretation: Each vector encodes a <u>discrete probability distribution</u>.

$$\begin{pmatrix}0\\1\\0\\0\end{pmatrix} \longrightarrow$$ **100 % the second class!**

Softmax Activation Function

- So we need an activation function that computes a discrete probability distribution.

- The <u>softmax activation function</u>: $softmax(\vec{x})_i = \frac{exp(x_i)}{\sum_j exp(x_j)}$

$$\vec{y} = softmax(\vec{d}^{(N)})$$

Output ↑ ↑ **Logits (Drive of output layer)**

<u>Example</u>

$$softmax\left(\begin{pmatrix}-2\\3\\0.1\\2\end{pmatrix}\right) = \begin{pmatrix}\frac{e^{-2}}{e^{-2}+e^{3}+e^{0.1}+e^{2}}\\\vdots\\\vdots\end{pmatrix} \approx \begin{pmatrix}0\\0.7\\0.04\\0.26\end{pmatrix}$$

→ **0 % the first class!**
→ **70 % the second class!**
→ **4 % the third class!**
→ **26 % the fourth class!**

Loss Functions

Loss Functions

- Depending on your objective the chosen encoding and the chosen output function you have to decide which loss function you use.

- Again there are many different ones!

- We will have a look at two important ones.

Mean Squared Error

- In the case of a regression task the goal to approximate the function as close as possible/minimizing the error.

- The obvious choice is to use a loss function that simply measures how large the mean error is:

$$L_\theta = \frac{1}{N} \sum_{i=1}^{N} \frac{1}{2} (t_i - y_i)^2$$

Cross Entropy

- If the task is a classification task the objective is another.
- We saw that we can interpret the labels and the output of the network as discrete probability distributions over the possible classes.
- Cross entropy is a measurement to compare two probability distributions:

$$H(p, q) = -\sum_{x} p(x) log(q(x))$$

where $p(x)$ is the true and $q(x)$ the estimated distribution.

Cross Entropy - Example

Input

Label

$$\begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

“happy”

Output of DNN

$$\begin{pmatrix} 0.2 \\ 0.5 \\ 0.1 \\ 0.2 \end{pmatrix}$$

$$loss = -\sum_{x} p(x) log(q(x))$$

$$= -(1 \cdot log(0.2) + 0 \cdot log(0.5) + 0 \cdot log(0.1) + 0 \cdot log(0.2)$$

$$= -log(0.2)$$

$$\approx 0.699$$

Training and Test Data

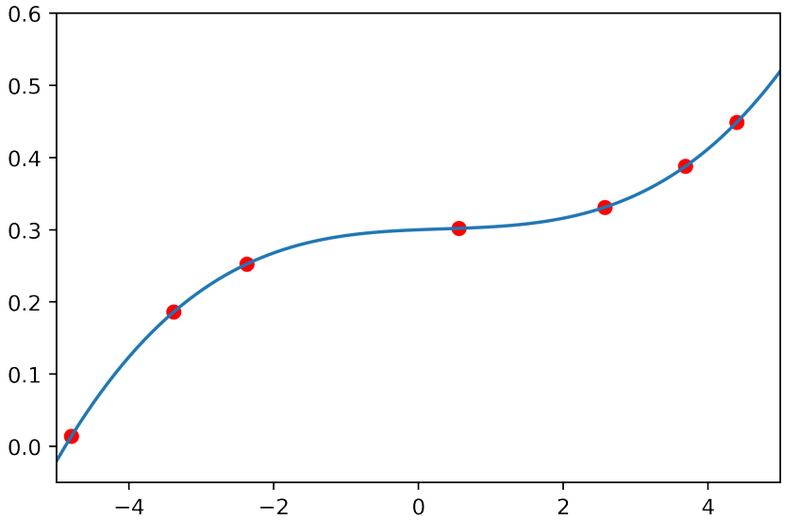
Generalization

- You supply the network with a dataset, which contains examples of the mapping that you want to approximate.
- We said the goal is to match this mapping as close as possible.
- But the actual goal is a different one:

We want the network to approximate the underlying function. Such that it generalizes to novel unseen inputs.

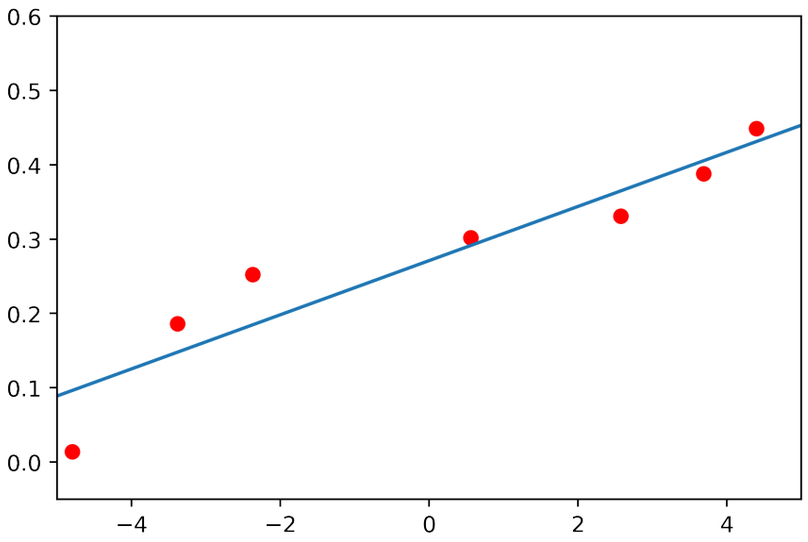
Under- & Overfitting

Original Function



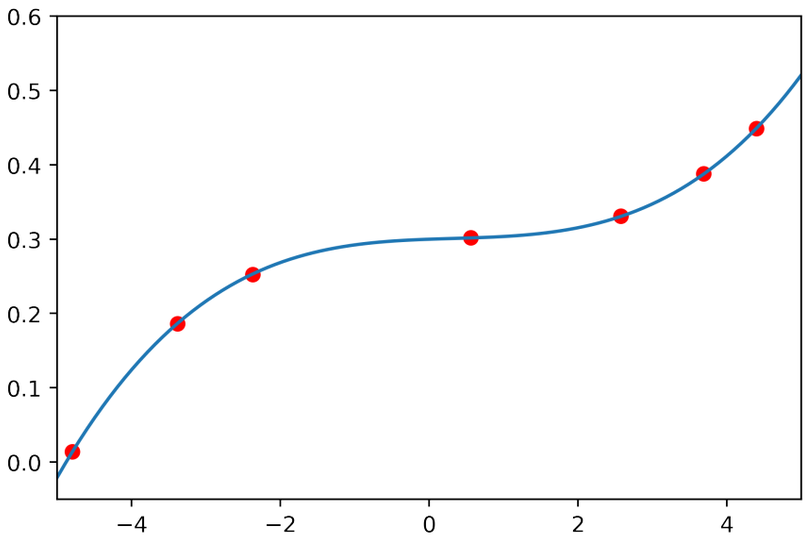
Approximation with polynomial linear regression with degree k:

$k = 1$



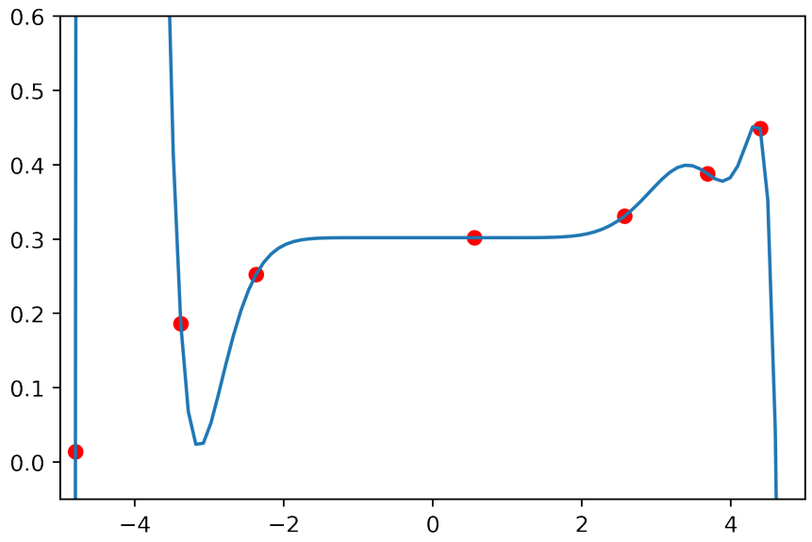
Underfitting

$k = 3$



Good generalization

$k = 10$



Overfitting

Test Data

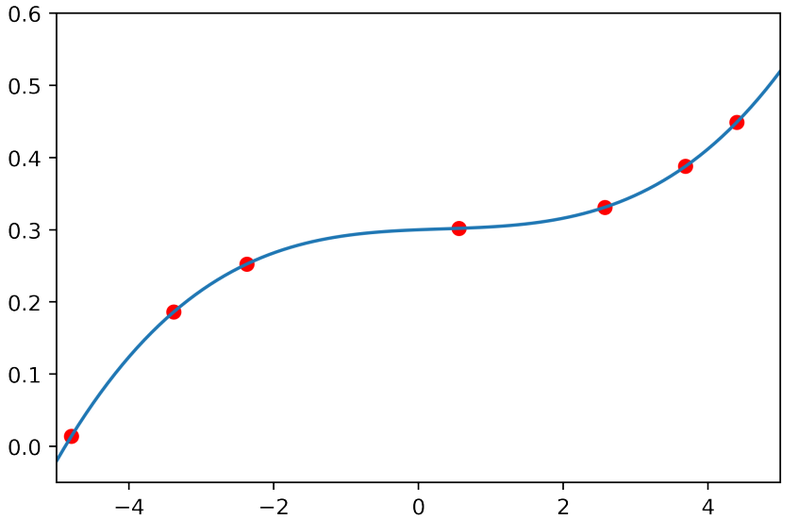
- In the real world we do not know the underlying function, which means we can’t check whether the network overfitted by comparing it to the true function.

- Instead we hold back a subset of our dataset to test whether the network generalizes to unseen data.

- We split the dataset in training and test data (also called validation data).

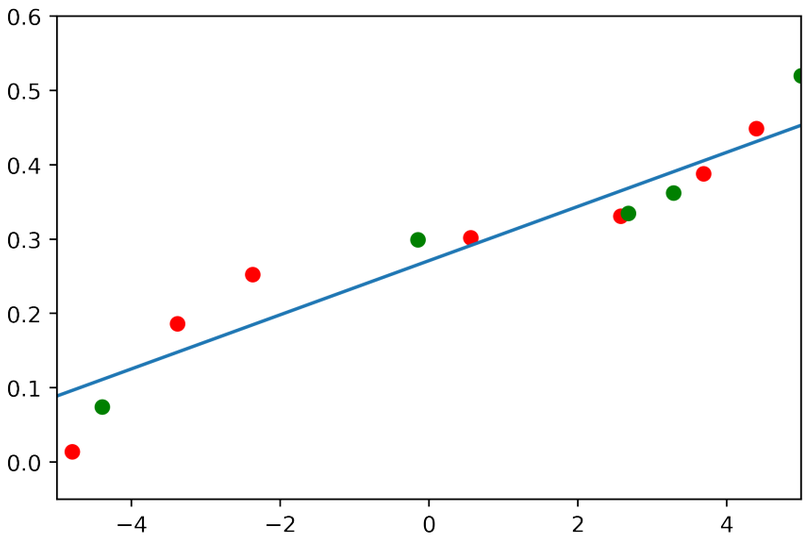
Overfitting

Original Function



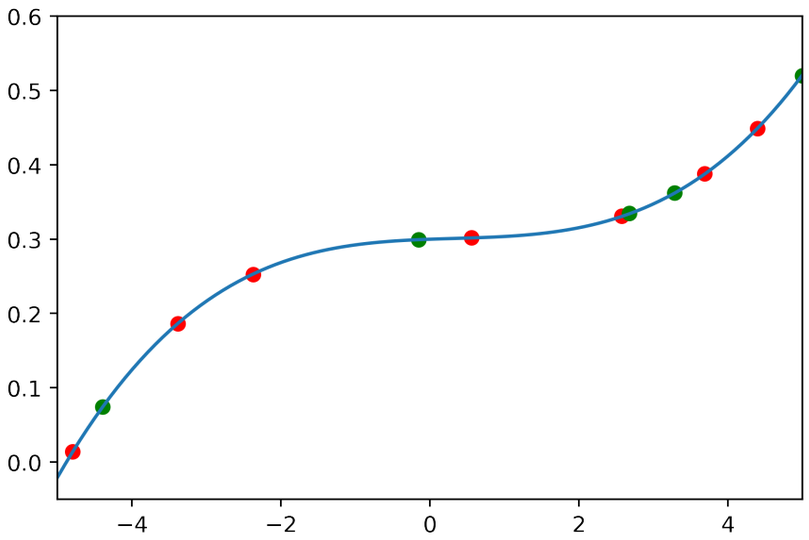
By comparing how much the training and the test error differ we can check whether the network overfitted the data!

$k = 1$



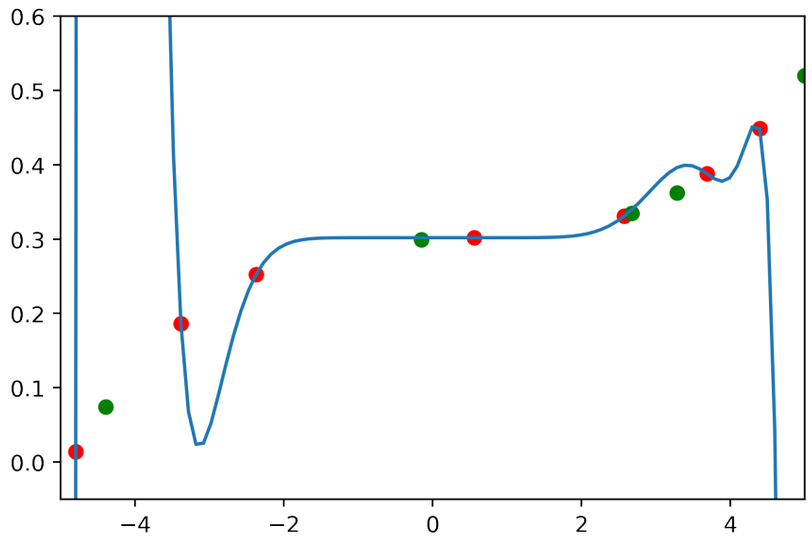
Underfitting

$k = 3$



Good generalization

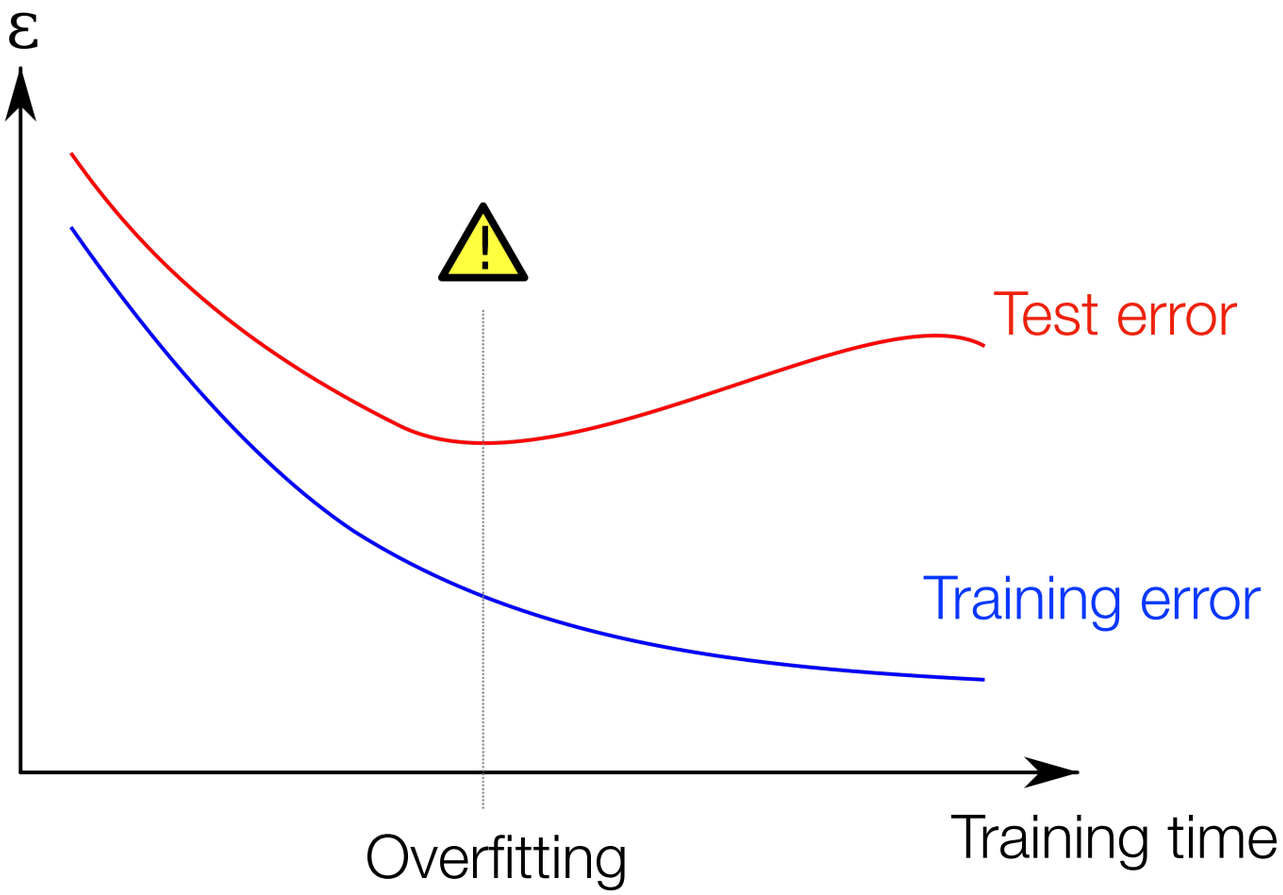
$k = 10$



Overfitting

Overfitting

- Observing how training and test error develop over time can reveal when the network starts to overfit.

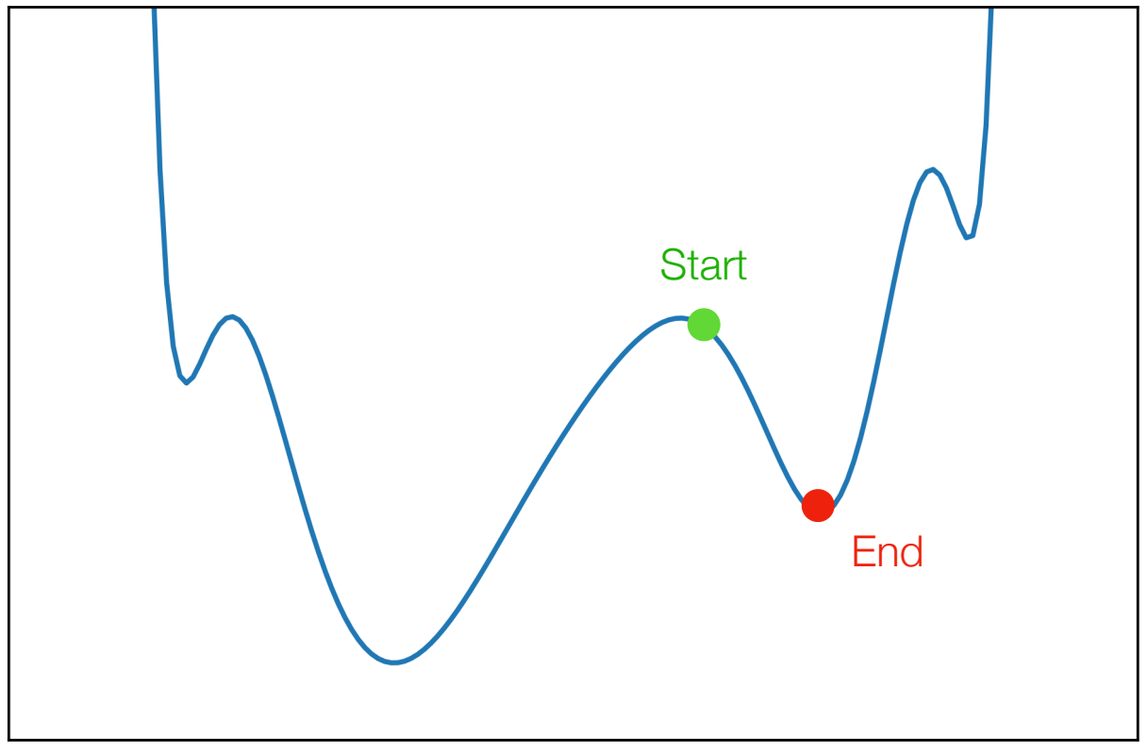


Optimization

Optimization

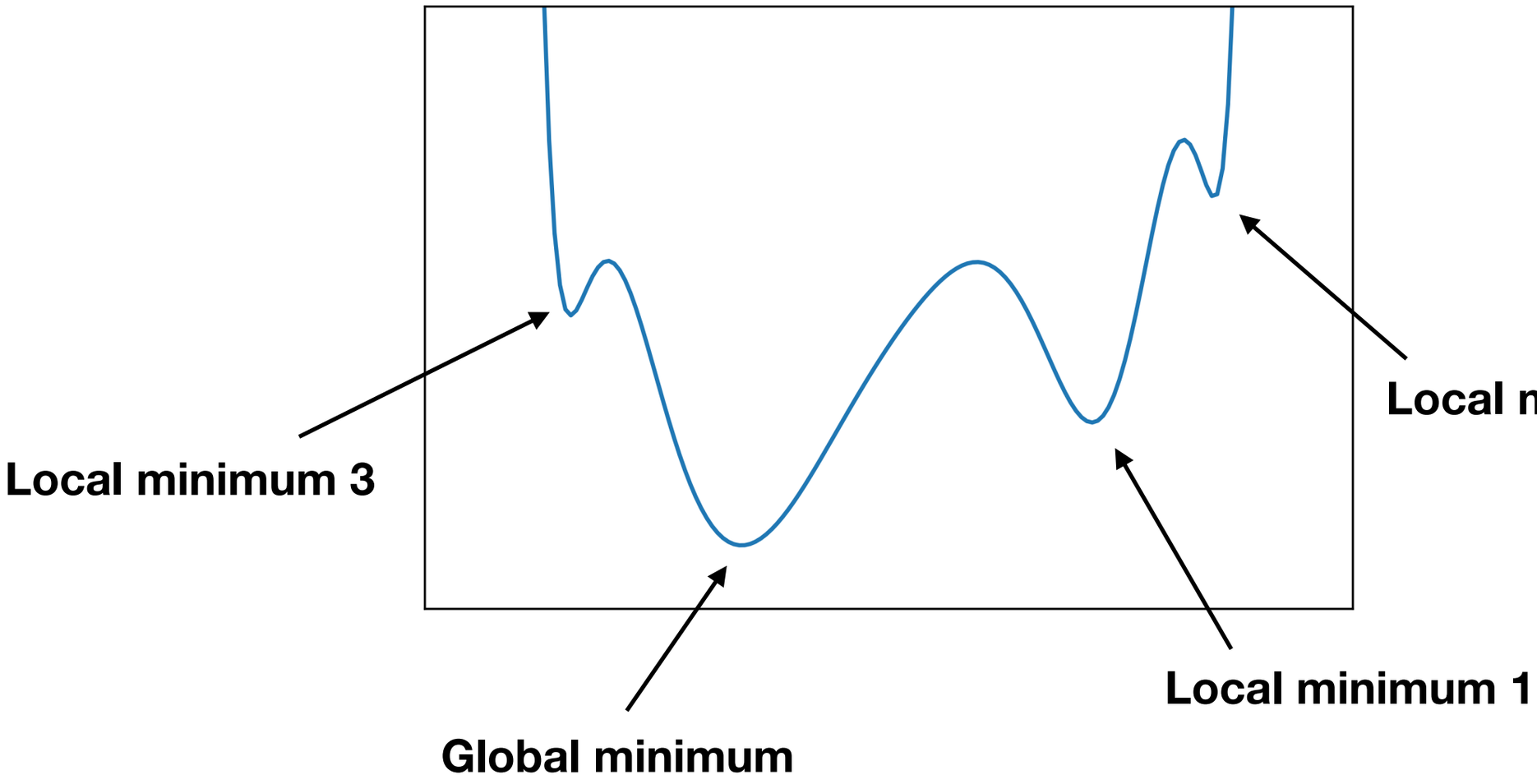
- To understand how underfitting and overfitting occurs and how it can be tackled we have to look at our optimization algorithm: gradient descent.

- Gradient descent starts at a random point and runs into the nearest minimum.



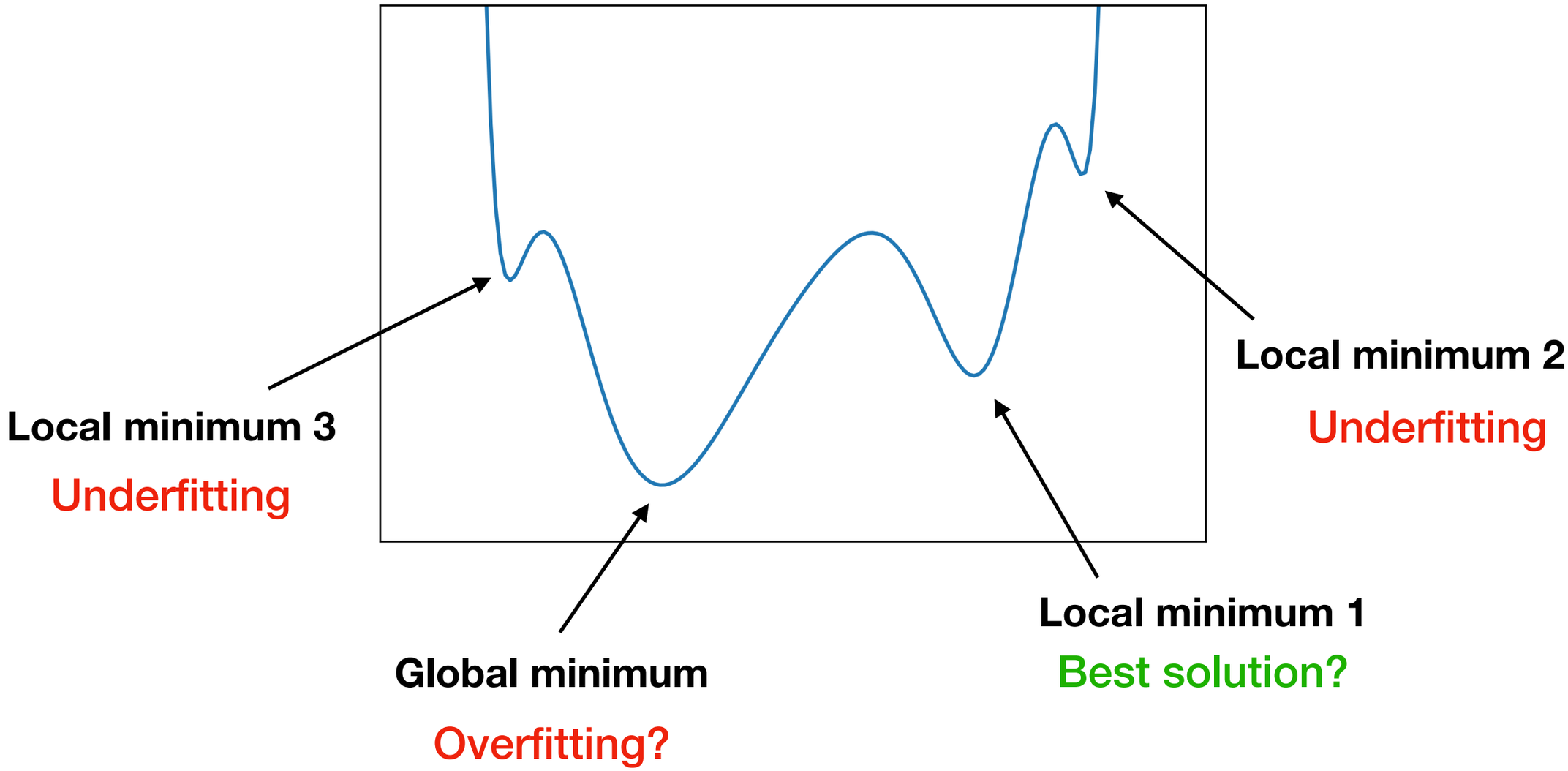
Optimization

- The question is: which minimum would we like to find?



Optimization

Remember that this is only the loss surface for the training data!



Optimization

- That means there are two things we want to avoid:

1. Getting stuck in a local minimum that is simply not a good solution (=underfitting).

   Can be solved by making gradient descent better. (today)

2. Finding a minimum, which might be a solution that overfits the training data.

   Can be solved by other means, called regularization techniques. (in two weeks)

Stochastic Gradient Descent

- The simplest solution to avoid getting stuck is the so called stochastic gradient descent.

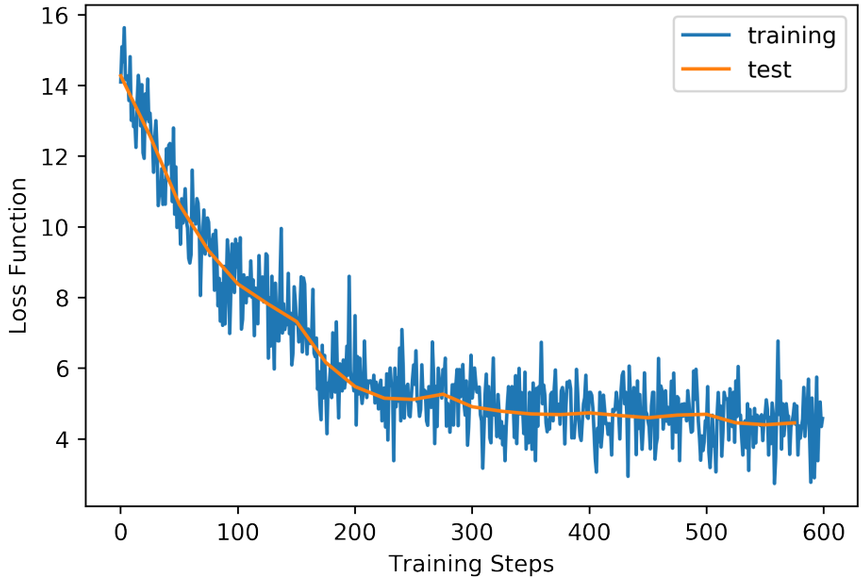
- Instead of computing the loss for the whole dataset in each iteration, you compute it for only one randomly drawn sample at a time.

- In this way there is the chance that you escape the local minimum of the original loss function.

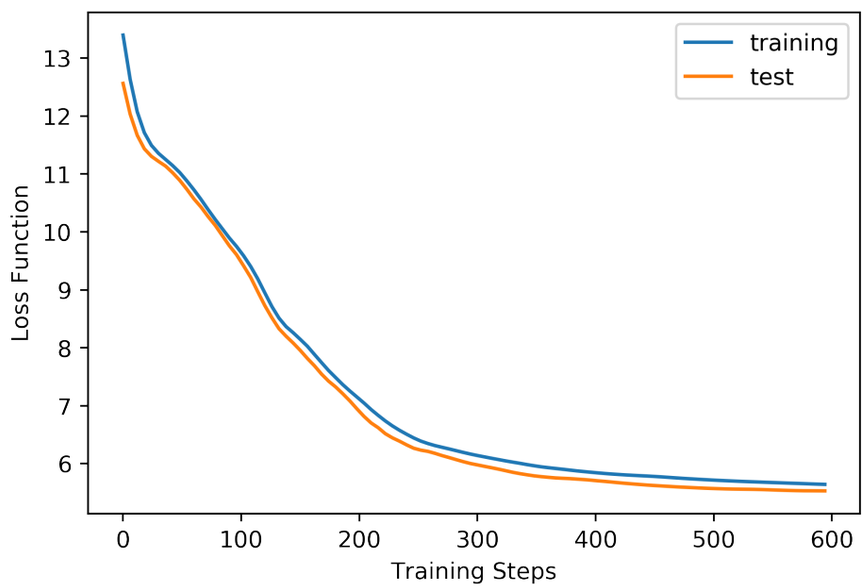
- The original gradient descent that minimizes for the whole dataset is in opposite called full batch gradient descent.

Fluctuations in Loss Function

- For stochastic gradient descent the loss function changes in every update step.
- These changes are visualized by fluctuations of the loss during the training progress.



Stochastic GD



Full Batch GD

Full Batch vs Stochastic

Full Batch Gradient Descent	Stochastic Gradient Descent
Minimizes the same loss surface in each iteration.	Minimizes a very different loss surface in each iteration,
Gradients show a clear direction.	Gradients might heavily differ between different training steps.
Guaranteed to converge to a minimum.	Might never converge.
Gets stuck in local minima.	Has a chance of walking around local minima.
Computationally slow as you have to process the whole dataset in each step.	Computationally fast. Only one sample per iteration.

Mini-Batch Gradient Descent

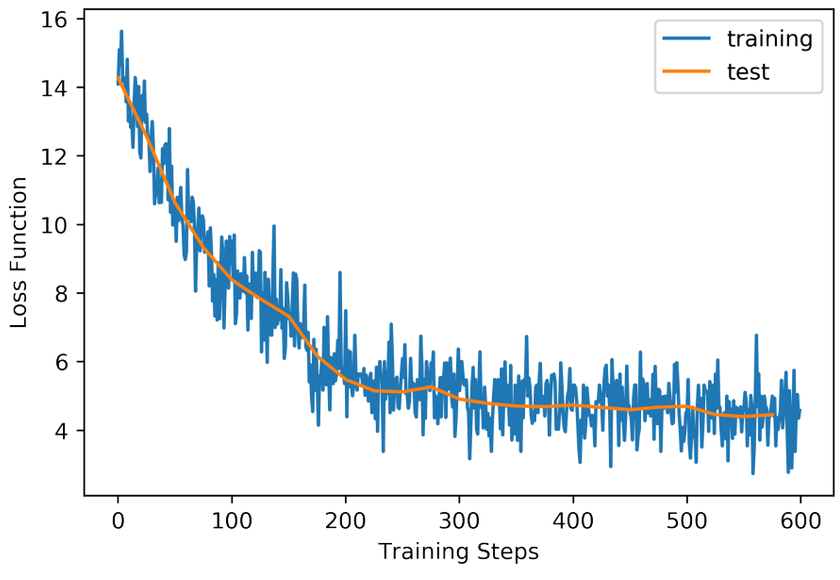
- Now stochastic gradient descent in its pure form can be very stochastic.

- Because you always show completely different samples the network might not be able to find a solution that fits all the samples and instead jump from solution to solution.

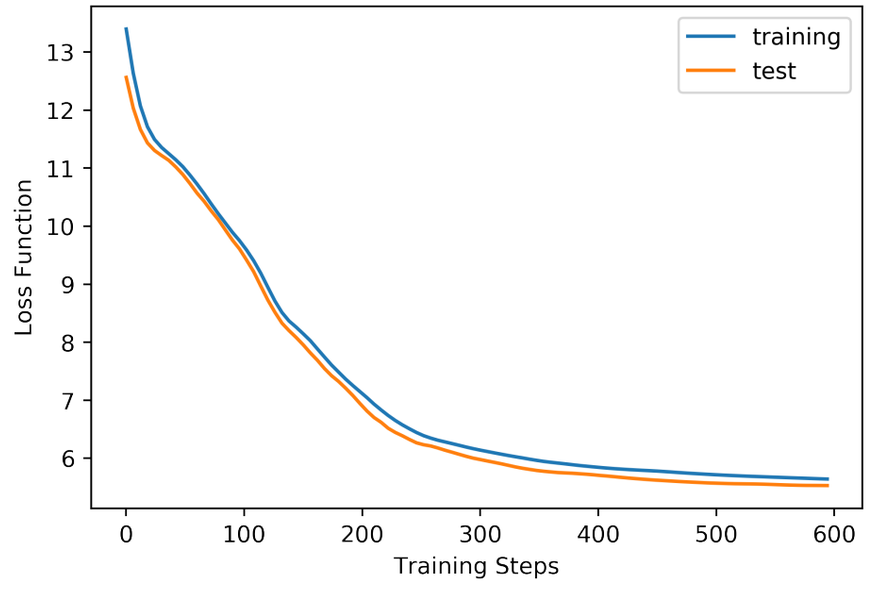
- The solution to this problem is a trade-off called mini-batch gradient descent.

- Instead of showing the whole dataset or only one sample, in each iteration, we show a small batch of e.g. 16, 32, 64, 128, … samples.

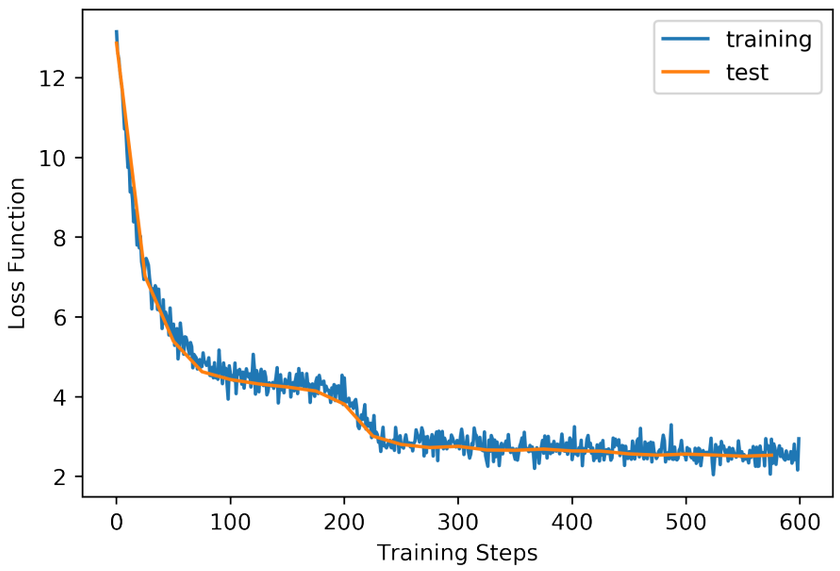
Fluctuations in Loss Function



Stochastic GD



Full Batch GD



Mini-Batch GD

Optimization Algorithm

Let's write down the whole algorithm once:

1. Randomly initialize parameters θ.
2. for epoch in 1…N:
    1. Chunk dataset in batches of desired size (full, mini, stochastic).
    2. for each batch in dataset:
        1. Compute forward step.
        2. Compute gradient.
        3. Update parameters θ.

Training Step

Epoch

Optimizers

Optimizers

There are many other variants of optimizers that make gradient descent even more efficient:

I will introduce three popular ones:

- Momentum
- Rmsprop
- Adam

Standard Gradient Descent

- Compute the gradient for given subset of the data.
- Walk small step in the opposite direction.

Small learning rate

$$\theta_t = \theta_{t-1} - \gamma \nabla L_\theta$$

Gradient

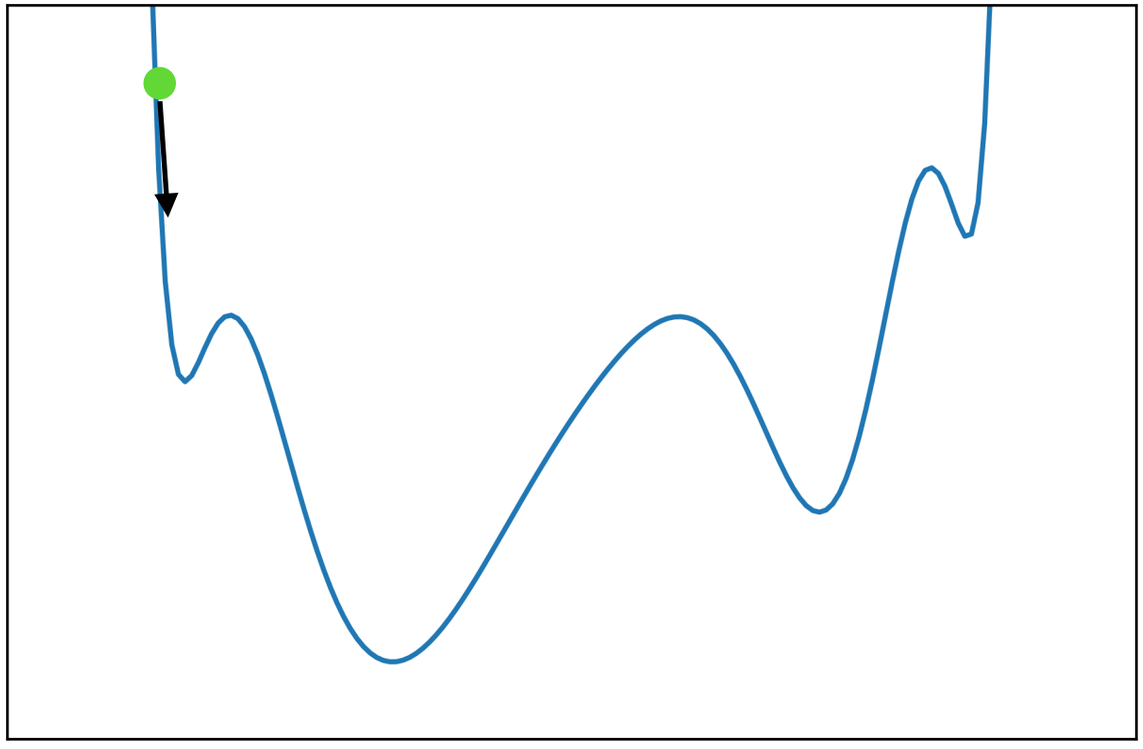
Descent

Momentum - Intuition

We don’t want the parameters to get stuck in the first local minimum.

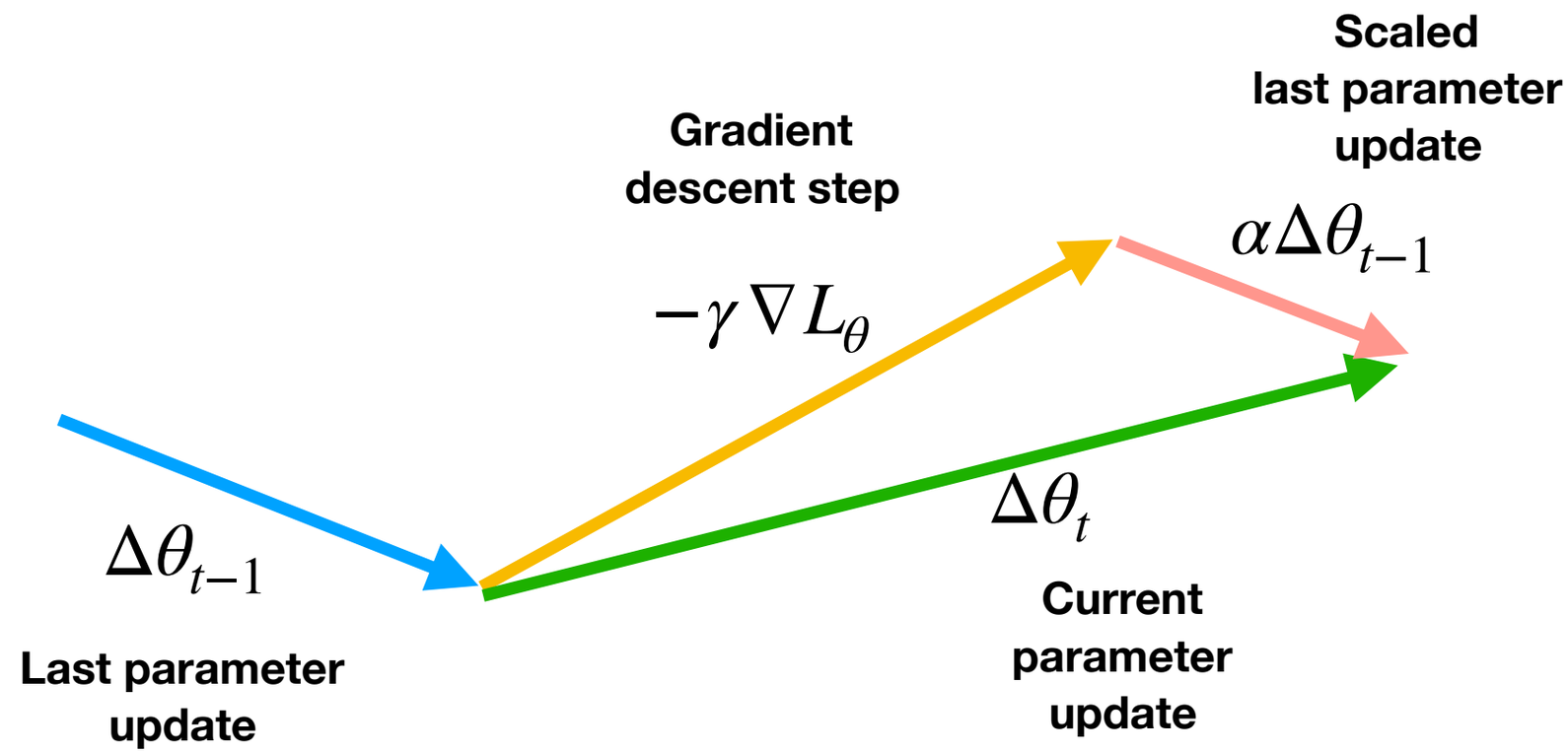
What if we could model that the parameters behave like a ball that rolls down a hill?

A ball would jump over that first hill!



Momentum

- This property can be modeled by adding a small fraction of the previous update



$$\theta_t = \theta_{t-1} + \Delta\theta_t \qquad \Delta\theta_t = -\gamma \nabla L_\theta + \alpha \Delta\theta_{t-1}$$

RMSProp

- It is known that high values for parameters can hurt the performance.

- An intuitive solution would be to weaken the update of weights that received large updates in the recent past.

Running average of recent past:

One parameter

Magnitude of Weight Update

$$v(\theta_i)_t = \beta v(\theta_i)_{t-1} + (1-\beta)(\nabla L_{\theta_i})^2$$

Forgetting term

Update for specific parameter θ_i :

$$\theta_{i,t} = \theta_{i,t-1} - \frac{1}{\sqrt{v(\theta_i)_t}} \gamma \nabla L_{\theta_i}$$

Adam

- Why not merge Momentum and RMSprop?

- Use a running average of the recent direction (momentum) and of the recent magnitude of updates (RMSprop)

$$m(\theta_i)_t = \beta_1 m(\theta_i)_{t-1} + (1 - \beta_1) \nabla L_{\theta_i}$$
$$v(\theta_i)_t = \beta_2 v(\theta_i)_{t-1} + (1 - \beta_2)(\nabla L_{\theta_i})^2$$

- For practical reasons these terms are normalized

$$\hat{m}(\theta_i)_t = \frac{m(\theta_i)_t}{1 - (\beta_1)^t} \qquad \hat{v}(\theta_i)_t = \frac{v(\theta_i)_t}{1 - (\beta_2)^t}$$

Adam

- Resulting in the weight update

$$\theta_{i,t} = \theta_{i,t-1} - \gamma \frac{\hat{m}(\theta_i)_t}{\sqrt{\hat{v}(\theta_i)_t} + \epsilon}$$

Learning rate

Gradient direction

Scale by recent magnitude of updates

Term to avoid division by 0

- By far the most popular optimizer, which we will use from now on!

Visualization

Not available due to copyright reasons.

Blog post: https://ruder.io/optimizing-gradient-descent/index.html

Conclusion & Outlook

Conclusion

- We looked at which tasks a deep neural network can solve: every task that you can express as a mathematical mapping.

- We looked at important building blocks for building a classifier: **one-hot, softmax, cross-entropy**.

- We saw that gradient descent has a problem with getting stuck in local minima and looked at solutions as **stochastic gradient descent** or the **Adam optimizer**.

Outlook

- Next week we will dive into **convolutional neural networks**.

- These are a very popular variant of DNNs, which are good for dealing with spatially structured data as images.

See you next week!

References

[ylc] Y. LeCun et al., "Gradient-based learning applied to document recognition." *Proceedings of the IEEE*, 86(11):2278-2324, November 1998.