Implementing ANNs with TensorFlow

Session 07 - Training Deep Neural Networks

Midterm Exam

- Wednesday 18.12.2019 in 35/E01: 12.15-14.15
- Simple questions!
- No questions about code!
- No cheatsheet, calculator or similar!
- Please be there at 12.00!
- Bring your student identity card.

Midterm Exam QnA

- There will be two QnAs for the exam.
- Leon QnA on Friday!
- The lecture slot on Tuesdays will also be a QnA, by me.
- Sent Leon or me your questions in advance, so we can prepare.

Last Weeks

- In session 5 we had a closer look at the optimization algorithm gradient descent.

- We discussed how could improve it, by making sure that it does not get stuck in local minima (e.g. SGD, Momentum, Adam)

- But there are dozens more problems that can occur while training a neural network.

Agenda

1. No Learning At All
    - Basic Checks
    - Architecture
    - Activation Functions
    - Exploding/Vanishing Gradients
    - Zero-Centered Drives
    - Input Normalization
    - Weight Initialization
    - Batch Normalization

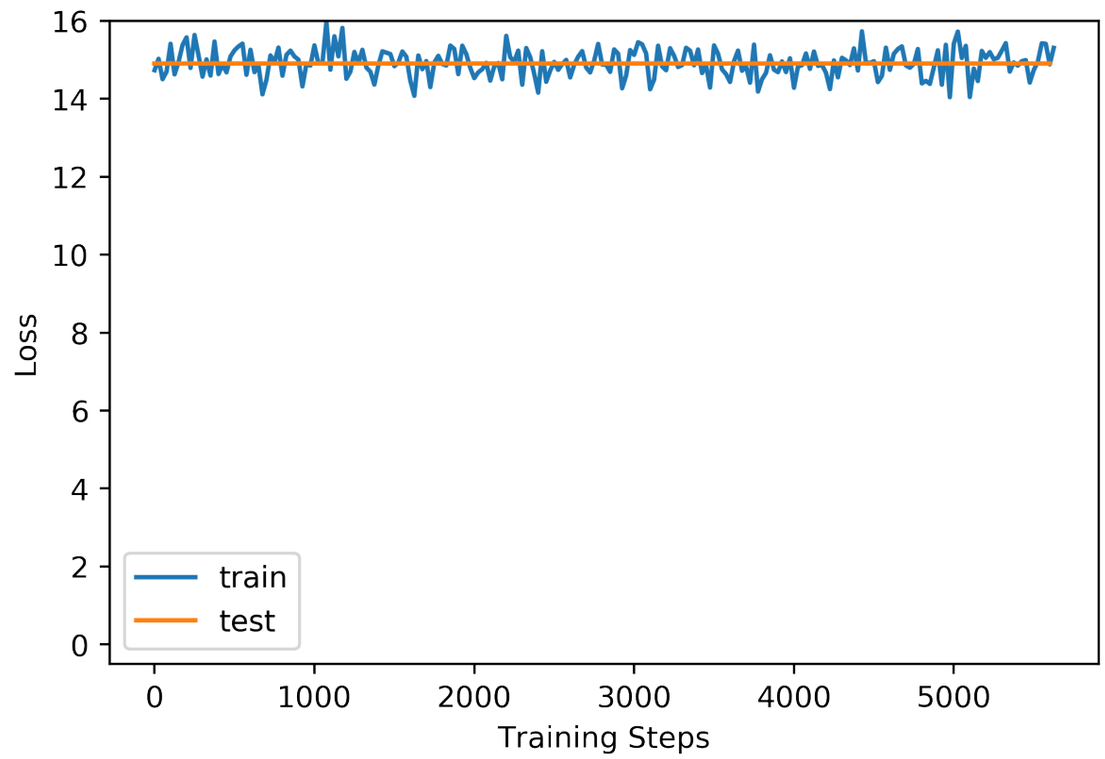
Agenda

2. Learning Rate & Batch Size
3. Overfitting
    - Early Stopping
    - Data Augmentation
    - Transfer Learning
    - L2 Regularization
    - DropOut
4. Cheating Neural Networks
    - Imbalanced Datasets
    - Metrics
    - Exploiting your Dataset

Problem 1: No Learning

No Learning

- You chose a dataset that you would like to train a deep neural network on.
- But it does not learn!
- What is going wrong?



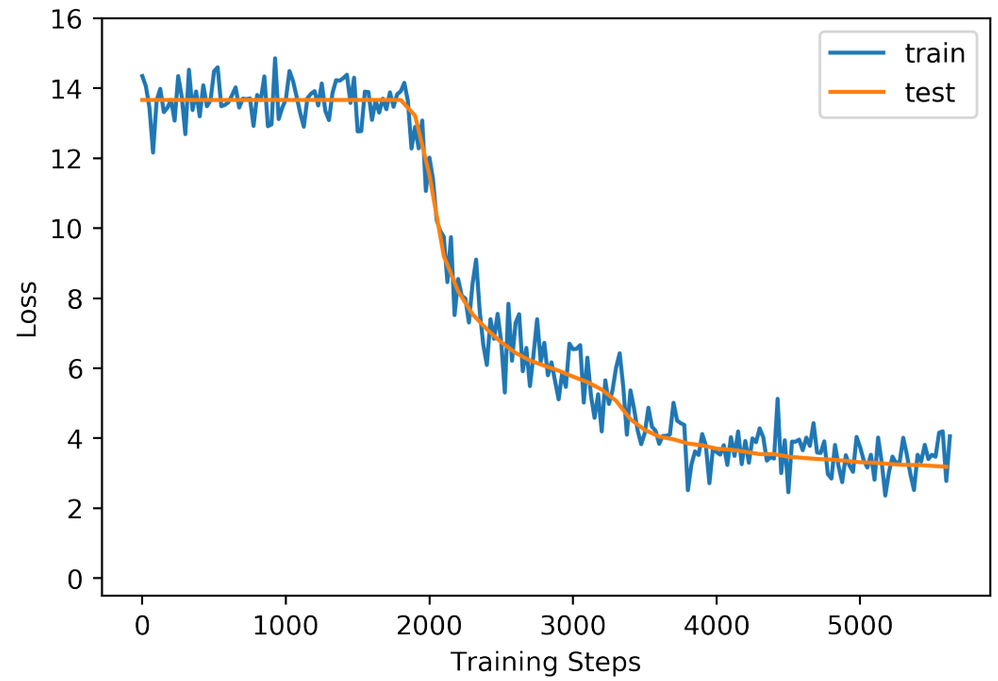
Re-run

- Neural networks are randomly initialized.
- Always run a second (or even third) time.
- Try random seed and find a seed that works.
- A random seed fixes the random initialization, i.e. you always start from the same point.

```
tf.keras.backend.clear_session()
tf.random.set_seed(1234)
```

More Epochs

- It could also be that you simply didn't train long enough.



- Increase the number of epochs:

```
for epoch in range(12):
    for (x,t) in train_dataset:
```

Sensible Mapping

Rethink whether your mapping actually can be learned from the given data.

Emotion? ✓

Age? ?

Height? ✗

Architectural Considerations

Architecture

- There are a few aspects about the architecture of the network that we can change:
    - Type of layers (e.g. fully connected, convolutional)
    - Number of layers (depth)
    - Number of neurons per layer (width)

Depth & Width

- In general increasing the depth and width of a network gives it more representational power, i.e. the network can approximate more complex functions.

- Note: Increasing depth is in terms of approximating complex functions more efficient than increasing width.

- But both come with problems:

  - Increasing depth: Deep networks are difficult to train (see next slides).

  - Increasing width: Wide networks easily overfit the data (see end of this lecture).

Finding a Good Architecture

- There is no recipe! Most of deep learning is about intuition and experience.

- If your network doesn't learn at all it is probably rather because of a too complex architecture than because of a too simple one.

- Start with something simple. You can always increase depth and width later to improve performance.

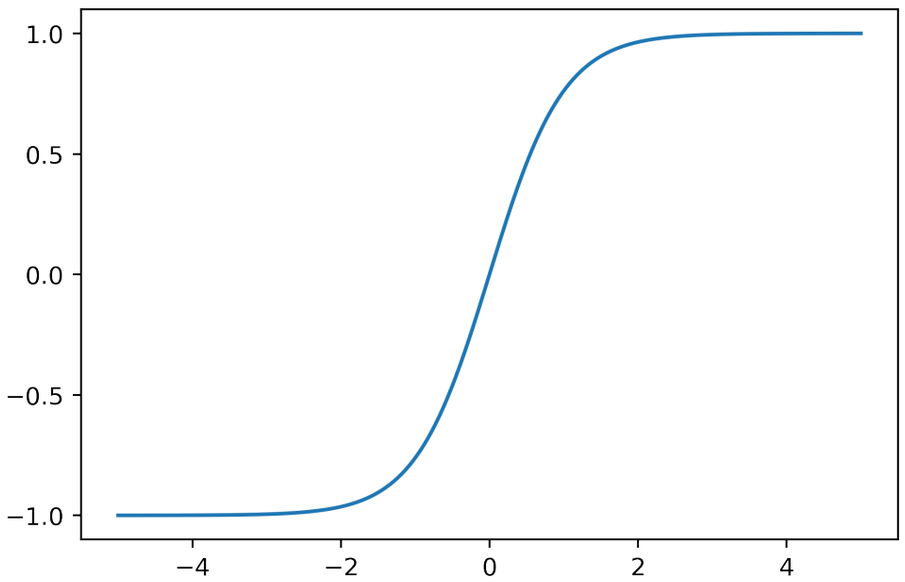
- Try to find paper/projects that tackle tasks similar to yours and get inspired by their architecture.

Activation Functions

Activation Functions

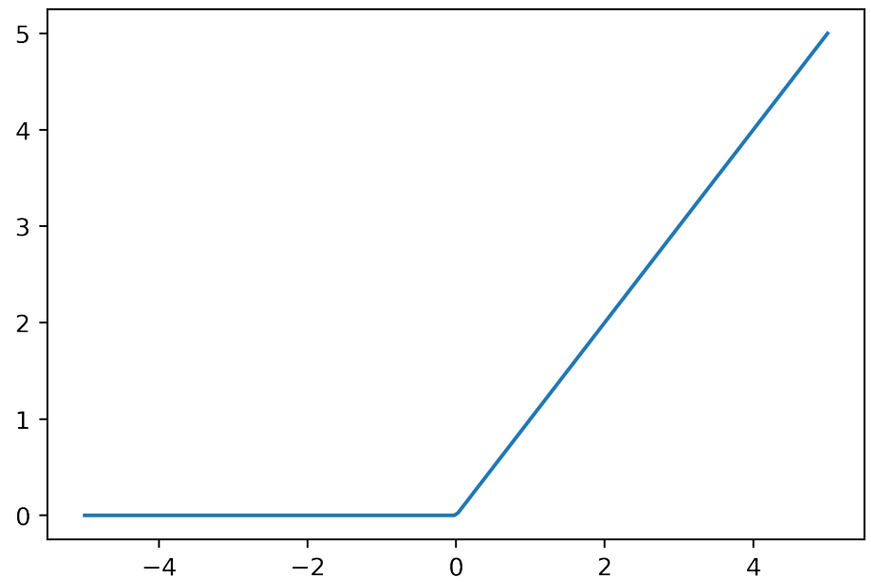
- The activation function is also an essential part of the network that we can change.

- Most common:



TanH (tangens hyperbolicus)

$$\sigma(x) = tanh(x)$$



ReLU (rectified linear unit)

$$\sigma(x) = \begin{cases} x \ if \ x \geq 0, \\ 0 \ else \ . \end{cases}$$

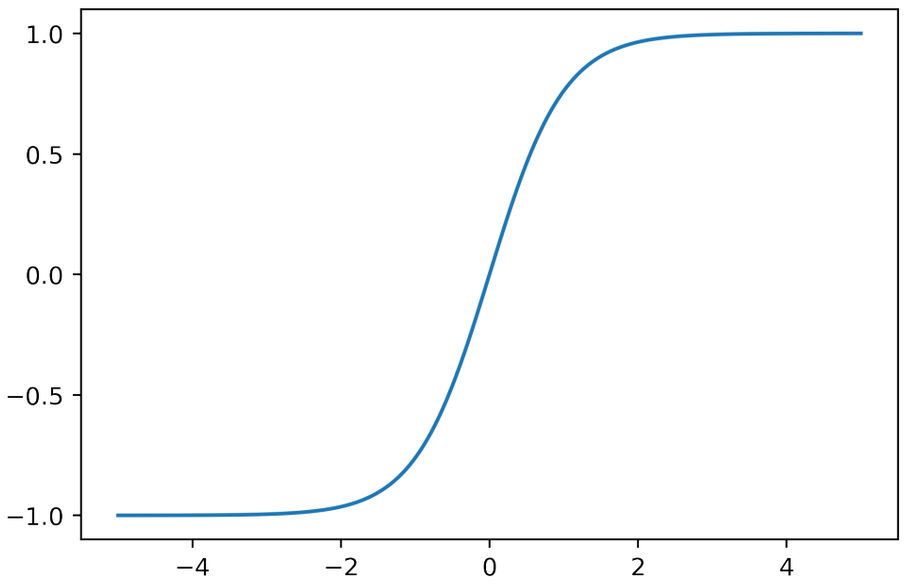
Choose an Activation Functions

- Again the choice is mostly intuition.
- Get inspired by other projects.
- ReLU is the most common used function and usually a good first choice!

Problems

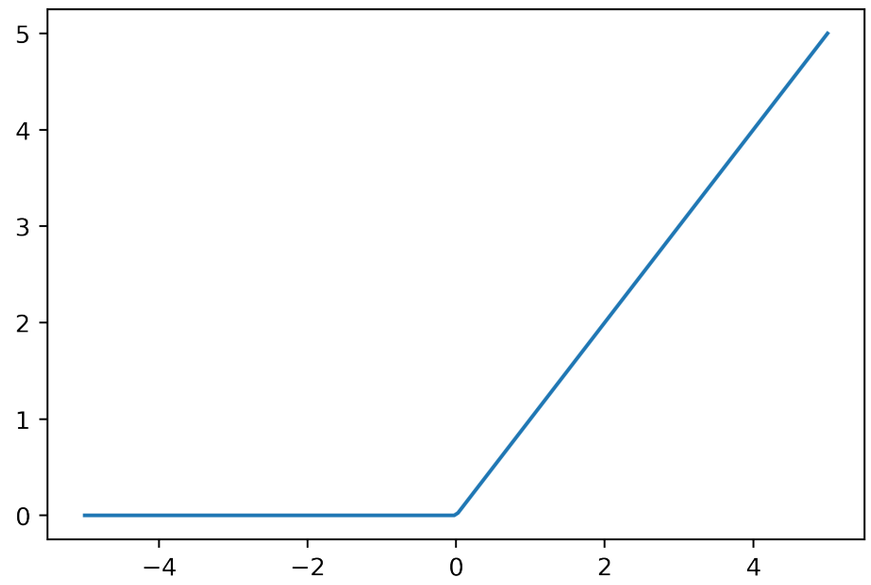
- Activation functions can cause problems.

- To understand these we look at the two representatives: TanH and ReLU



TanH

Double-saturating function



ReLU

Single-saturating function

Backpropagation

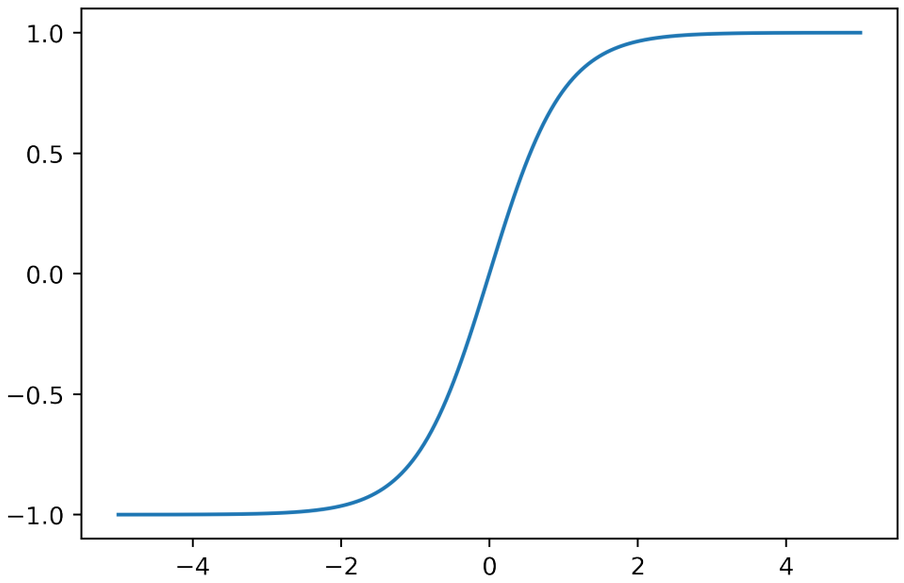
- The activation function can cause problems during backpropagation.
- To calculate delta we multiply with the derivative of the activation function

$$\frac{\partial L}{\partial w_{ij}^{(l)}} = \delta_i^{(l)} \; a_j^{(l-1)}$$

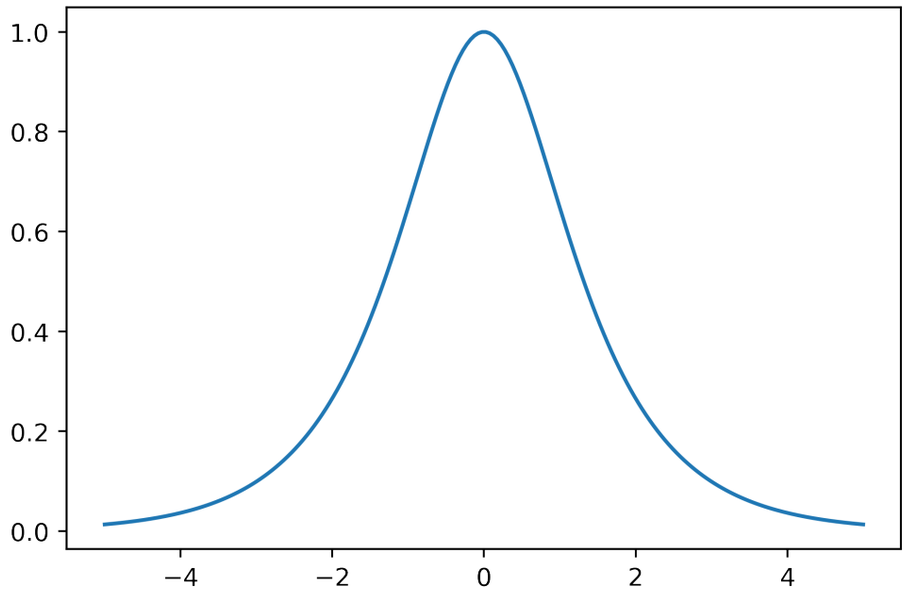
$$\delta_i^{(l)} = \begin{cases} -(t_i - y_i) \; \sigma'(d_i^{(l)}) & \text{if } l = N, \\ \left(\sum_{k=1}^{m} \delta_k^{(l+1)} w_{ki}^{(l+1)} \right) \sigma'(d_i^{(l)}) & \text{else}. \end{cases}$$

Vanishing Gradients

- Vanishing gradients is a problem of double-saturating activation functions.



TanH



Derivative of TanH

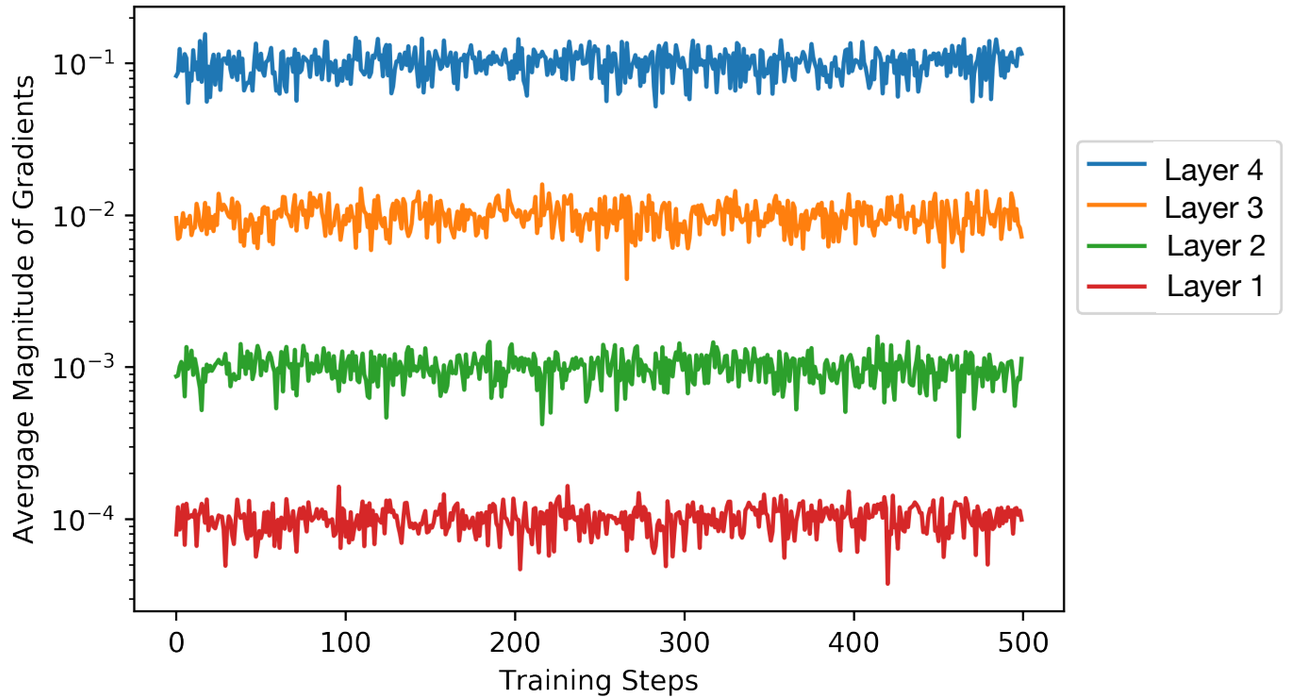
- We can see that the derivative saturates to zero on both sides.

Vanishing Gradients

$$\frac{\partial L}{\partial w_{ij}^{(l)}} = \delta_i^{(l)} \; a_j^{(l-1)}$$

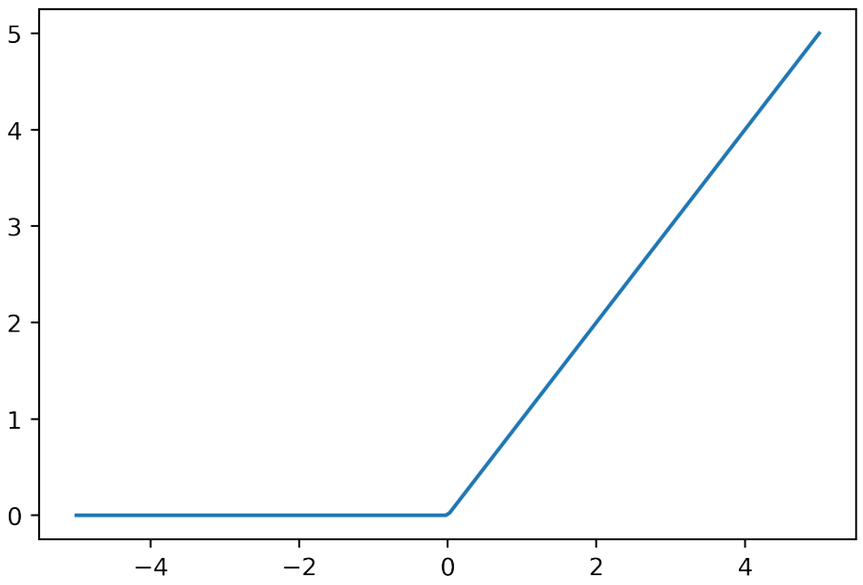
$$\delta_i^{(l)} = \begin{cases} -(t_i - y_i) \; \sigma'(d_i^{(l)}) & \text{if } l = N, \\ \left(\sum_{k=1}^{m} \delta_k^{(l+1)} w_{ki}^{(l+1)} \right) \sigma'(d_i^{(l)}) & \text{else}. \end{cases}$$

- In each backprop step (for each delta) we multiply with the derivative.
- If a lot of drives are either very positive or negative we often multiply with a value near 0.
- The delta values for the first layers (and with them the gradients) might get 0 for early layers.
- Thus the weights in early layers might not get any updates and thus do not learn.

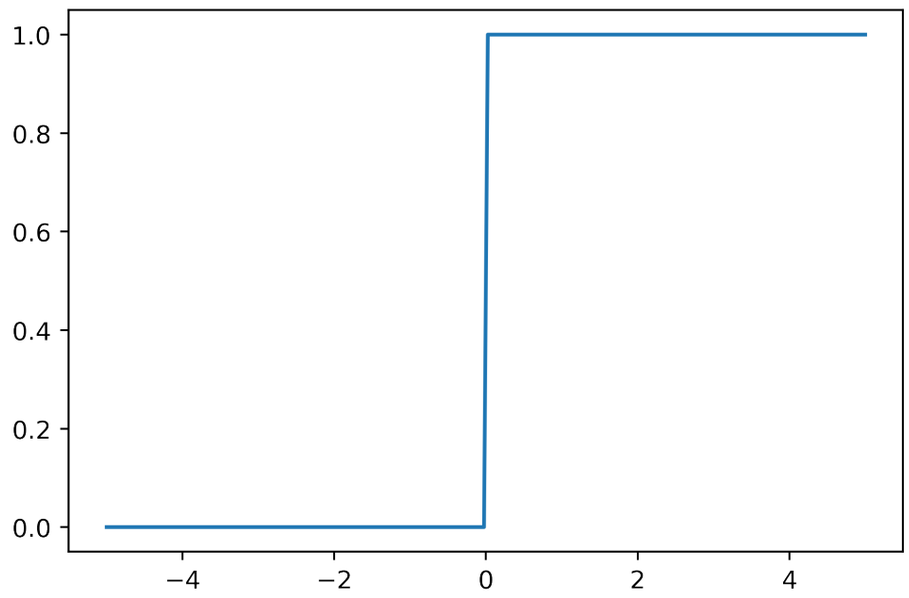


Exploding Gradients

- Exploding gradients is a problem of single-/non-saturating activation functions.



ReLU



Derivative of ReLU

- We can see that the derivative is either 0 or 1.

Exploding Gradients

$$\frac{\partial L}{\partial w_{ij}^{(l)}} = \delta_i^{(l)} \; a_j^{(l-1)}$$

$$\delta_i^{(l)} = \begin{cases} -(t_i - y_i) \; \sigma'(d_i^{(l)}) & \text{if } l = N, \\ \left(\sum_{k=1}^{m} \delta_k^{(l+1)} w_{ki}^{(l+1)} \right) \sigma'(d_i^{(l)}) & \text{else}. \end{cases}$$

- If a lot of drives are positive we often multiply by 1.
- Thus the deltas are never scaled.
- The delta values for the first layers (and with them the gradients) might get very large for early layers.
- Thus the weights in early layers might get too large updates and not learn anything useful.

Extracting the Gradients in TF

```
gradients = tape.gradient(loss, mlp.trainable_variables)
# Extract gradients for certain weights:
gradients[0] # Weight matrix of first layer.
gradients[1] # Bias weights of first layer.
gradients[2] # Weight matrix of first layer.
# ...
# Compute average magnitude:
gradient_avg_W1 = np.mean(np.abs(gradients[0]))
# Or compute the norm of the matrix/vector:
gradient_norm_W1 = np.linalg.norm(gradients[0])
```

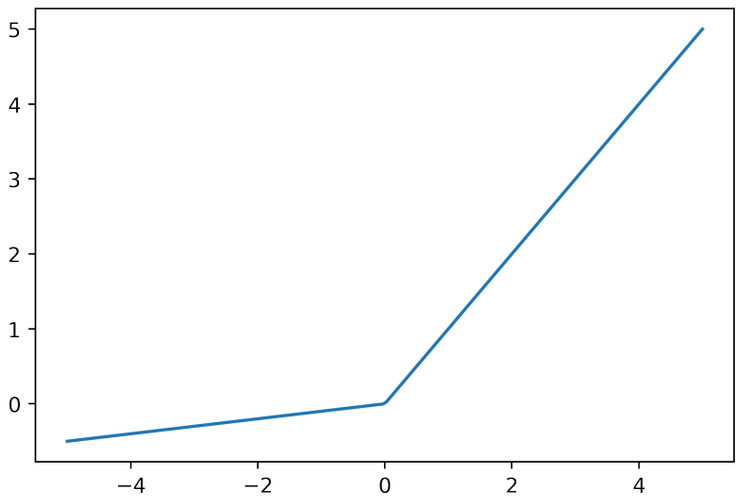
Dying ReLUs

- A last problem with the ReLU activation function is that neurons can “die”.
- Imagine a neuron is not activated by any of the inputs, which means the drive of the neuron is always negative.
- The delta value is always 0, as we multiply with the derivative of the drive.
- Thus the weights never receive updates. The neuron will never be updated.

$$\frac{\partial L}{\partial w_{ij}^{(l)}} = \delta_i^{(l)} \; a_j^{(l-1)} \qquad \delta_i^{(l)} = \begin{cases} -(t_i - y_i) \; \sigma'(d_i^{(l)}) & if \; l = N, \\ \left(\sum_{k=1}^{m} \delta_k^{(l+1)} w_{ki}^{(l+1)} \right) \sigma'(d_i^{(l)}) & else \, . \end{cases}$$

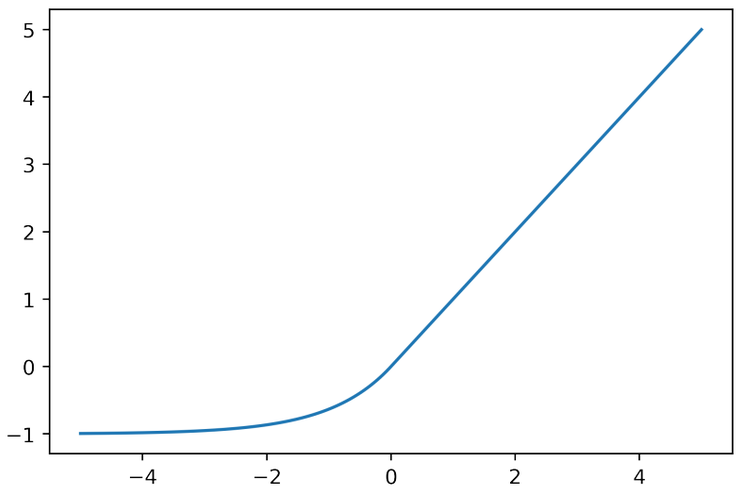
Activation Functions

Improvements over ReLU



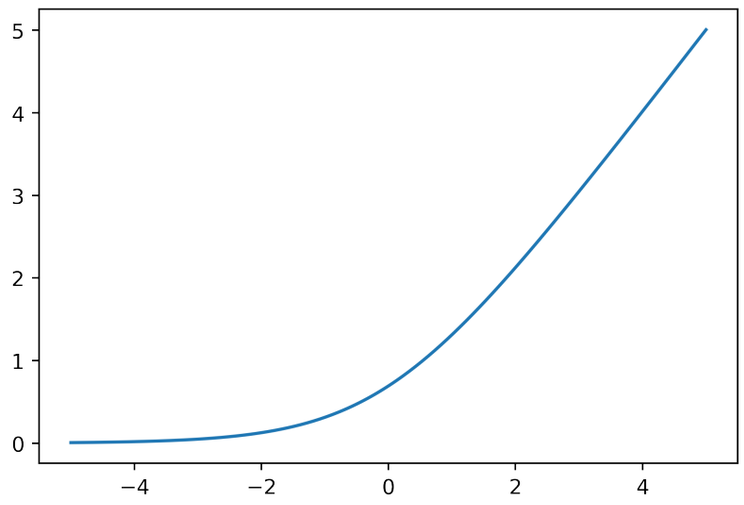
Leaky ReLU

$$\sigma_\alpha(x) = \begin{cases} x & if\ x \geq 0, \\ \alpha x & else\,. \end{cases}$$



ELU

$$\sigma_\alpha(x) = \begin{cases} x & if\ x \geq 0, \\ \alpha(1 - e^x) & else\,. \end{cases}$$

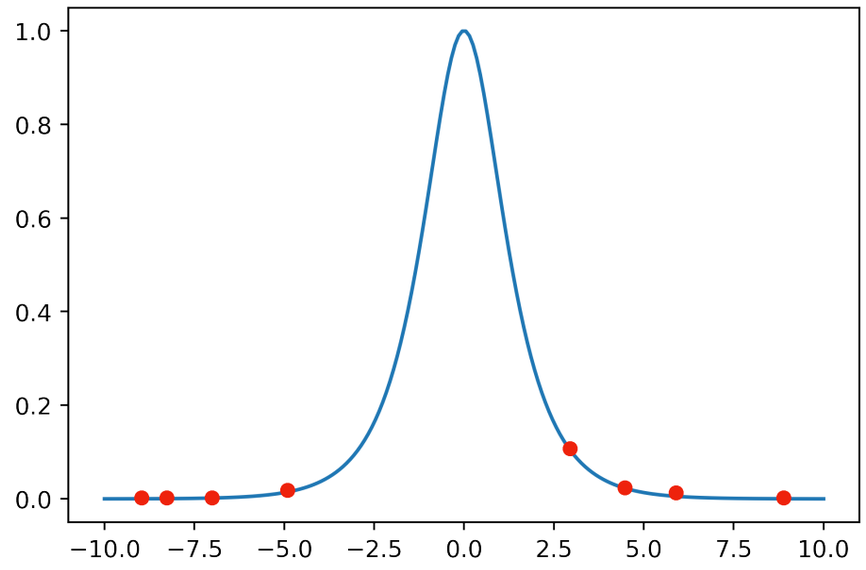


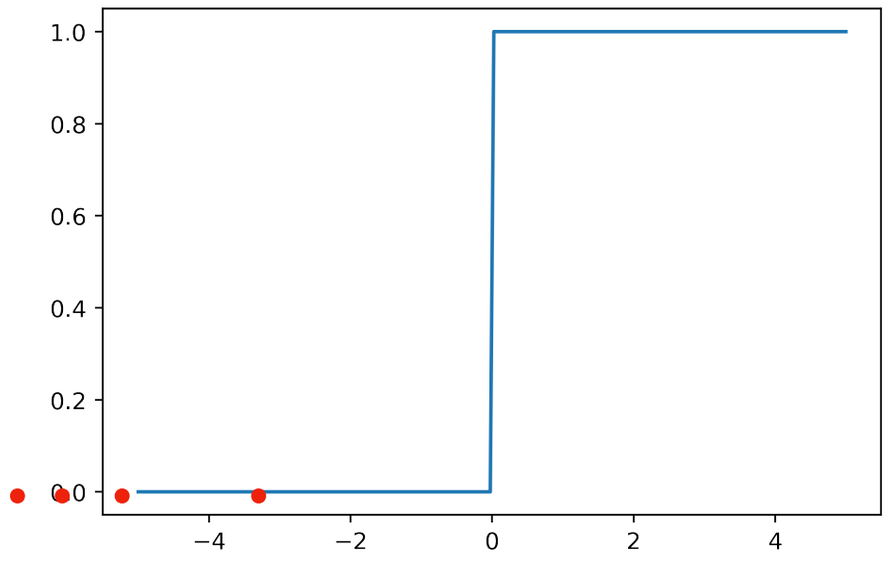
SoftPlus

$$\sigma(x) = ln(e^x + 1)$$

Solution

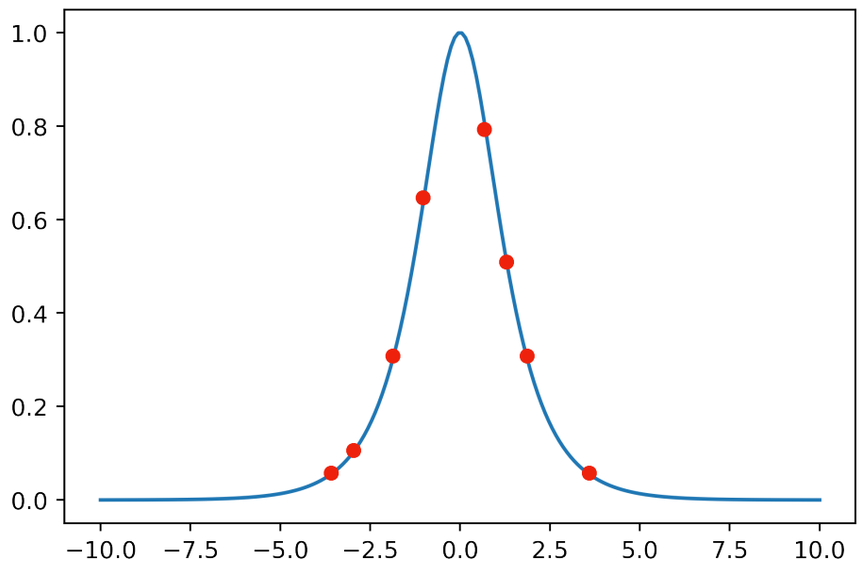
- Is there something we can do against these problems?
- The reason for these problems is that:
  - Vanishing gradients: the drives are too positive/ negative
  - Exploding gradients/Dying: too many drives are positive/ negative

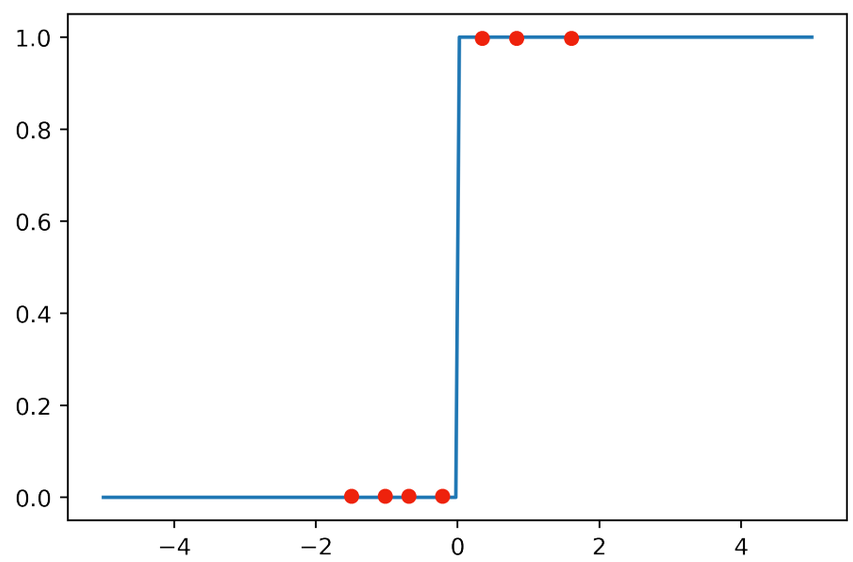




Solution

We could solve both problems if the drives are closely centered around zero.





Zero-Centered Drives

Zero-Centered Drives

- How can we make sure the drives are centered around zero?

- By centering all involved parts around zero.

$$\vec{d}^{(l)} = W^{(l)}\vec{a}^{(l-1)} + \vec{b}^{(l)}$$

Input Normalization

- In the first layer the drive is dependent on the input, that we feed into the network: $\overrightarrow{d}^{(1)} = W^{(1)}\overrightarrow{x} + \overrightarrow{b}^{(1)}$
- To center the input around zero we can simply normalize it.
- Compute mean and average of each component for whole dataset.

Mean $\mu_i = \frac{1}{N}\sum_k x_i^k$

$$x_{i,normalized} = \frac{x_i - \mu_i}{\sigma_i}$$

Standard Deviation $\sigma_i = \sqrt{\frac{1}{N-1}\sum_k (x_i^{(k)} - \mu_i)^2}$

Input Normalization

- Code for TensorFlow:

```
# train_images.shape: (60000,28,28)
mean = np.mean(train_images, axis=0)
stddev = np.std(train_images, axis=0)
train_images_normalized = (train_images-mean)/stddev
```

- Remember to perform the same normalization for the validation data during inference!

```
for (x,t) in test_dataset:
    x = (x-mean)/stddev
```

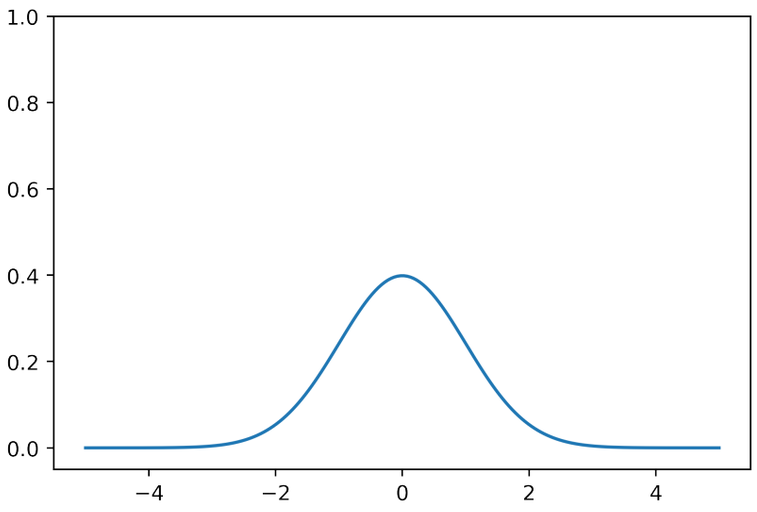
Weight Initialization

- We also have to make sure that the weights are centered closely around zero

$$\vec{d}^{(l)} = W^{(l)} \vec{a}^{(l-1)} + \vec{b}^{(l)}$$

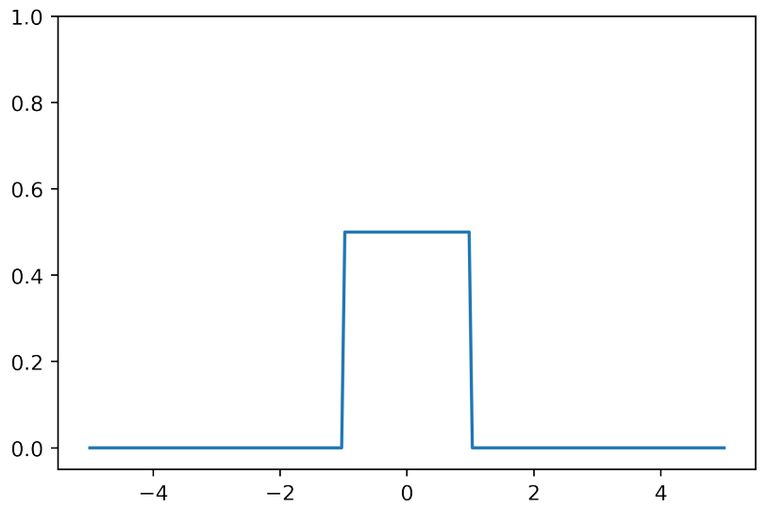
- Two suitable random initializations:

Normal Random Distribution



$$\mathcal{N}(0, \sigma^2)$$

Uniform Distribution



$$\mathcal{U}(-b, b)$$

Weight Initialization

- Code for TensorFlow:

```
self.output_layer = tf.keras.layers.Dense(
                        units=10,
                        activation=tf.keras.activations.softmax,
                        kernel_initializer=tf.keras.initializers.RandomNormal(mean=0.0, stddev=0.05)
                    )
```

- Different initializers can be found here:

https://www.tensorflow.org/api_docs/python/tf/keras/initializers

Weight Initialization

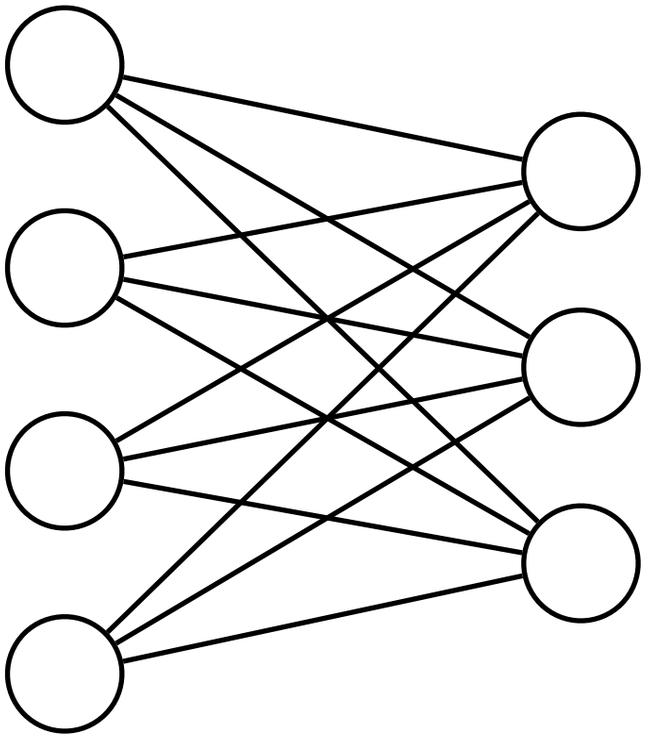
- But how to choose the standard deviation (normal distribution) or the minimum/maximum value (uniform distribution)

- Two different research papers found that it is beneficial to make these values dependent on the number of neurons in the two layers that are connected by the weights.

- The number of neurons are called **fan_in** and **fan_out**.

E.g.

$$fan\_in = 4$$



$$fan\_out = 3$$

Xavier-/Glorot-Initialization

Paper

- Glorot normal initialization:

$$\mathcal{N}(0,\sigma^2) \quad \text{with} \quad \sigma = \sqrt{\frac{2}{fan\_in + fan\_out}}$$

- Glorot uniform initialization:

$$\mathcal{U}(-b, b) \quad \text{with} \quad b = \sqrt{\frac{6}{fan\_in + fan\_out}}$$

He-Initialization

Paper

- He normal initialization:

$$\mathcal{N}(0,\sigma^2) \quad \text{with} \quad \sigma = \sqrt{\frac{2}{fan\_in}}$$

- He uniform initialization:

$$\mathcal{U}(-b,b) \quad \text{with} \quad b = \sqrt{\frac{6}{fan\_in}}$$

Bias Initialization

- We also need to initialize the bias weights.

$$\vec{d}^{(l)} = W^{(l)}\vec{a}^{(l-1)} + \vec{b}^{(l)}$$

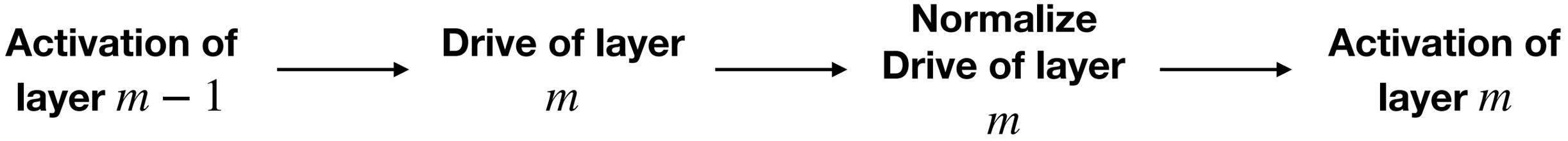
- They are usually simply initialized as zeros.

```
self.output_layer = tf.keras.layers.Dense(
                        units=10,
                        activation=tf.keras.activations.softmax,
                        kernel_initializer=tf.keras.initializers.RandomNormal(mean=0.0, stddev=0.05),
                        bias_initializer=tf.keras.initializers.Zeros
                    )
```

Batch Normalization

- There is also a very explicit solution to make sure the drives are zero-centered: normalize the drives!

- This technique is called BatchNorm.



Batch Normalization

- It is called batch normalization, because we normalize the drives with the mean and standard deviation from the given mini-batch.

- Given a mini-batch (n samples) we obtain n drive vectors: $\left(\overrightarrow{d}_i \right)_{i=1}^{n}$

- Calculate the mean and standard deviation vectors (component-wise): $\overrightarrow{\mu}, \overrightarrow{\sigma}$

- Normalize component-wise: $$normalized(\overrightarrow{d}_i) = \frac{\overrightarrow{d}_i - \overrightarrow{\mu}}{\overrightarrow{\sigma}}$$

Batch Normalization

- Lastly batch normalization includes new learnable parameters for each neuron: scale and shift.
- Because sometimes the previously done normalization is not useful.
- Scale and shift component wise:

$$batchnorm(\vec{d}_i) = \vec{\gamma}\frac{\vec{d}_i - \vec{\mu}}{\vec{\sigma}} + \vec{\beta}$$

Scale **Shift**

Batch Normalization

- During inference (test time), seeing only one sample at a time, we can't compute the mean/standard deviation of the drives.

- Thus during training we update an estimated average, which is then used during inference.

Old estimated average

$$\overrightarrow{\mu}_{inf,t} = \alpha \overrightarrow{\mu}_{inf,t-1} + (1-\alpha)\overrightarrow{\mu}_{t}$$

Current value

$$\overrightarrow{\sigma}_{inf,t} = \alpha \overrightarrow{\sigma}_{inf,t-1} + (1-\alpha)\overrightarrow{\sigma}_{t}$$

New estimated average

Momentum for Average (e.g. 0.99)

Batch Normalization

- Batch Normalization seems to bring many advantages:
  - Stable gradients.
  - Works as a regularizer.
  - Allows for larger learning rate.
- Yet it is not really clear why!

Batch Normalization in TF

- Batch normalization is applied before the activation function, thus the layer should not apply any activation function itself.

```
self.conv_layer_1 = tf.keras.layers.Conv2D(
                        filters=32,
                        kernel_size=3,
                        activation=None,
                        input_shape=(32,32,3)
                    )
self.batch_norm = tf.keras.layers.BatchNormalization()
```

- The call method of your model, needs an argument that indicates whether you are currently training or testing.

- This argument is also fed to the batch normalization call.

```
def call(self, x, is_training):
    x = self.conv_layer_1(x)
    x = self.batch_norm(x, training=is_training)
    x = tf.nn.relu(x)
```

- After applying the batch norm layer you can apply the activation function.

Learning Rate
&
Batch Size

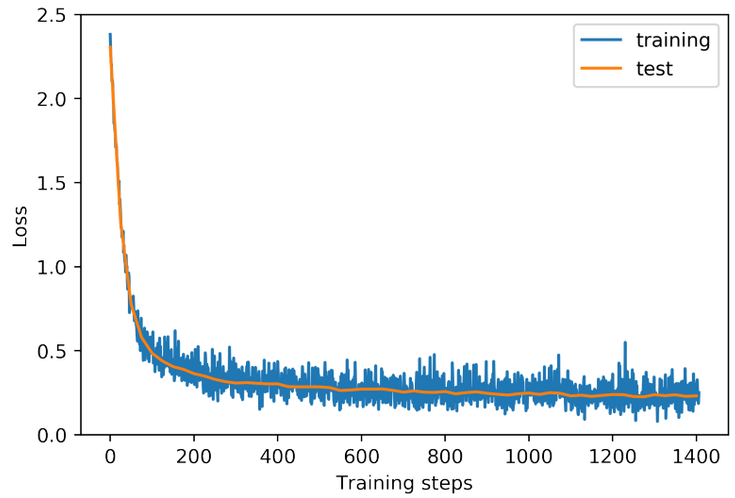
Learning Rate

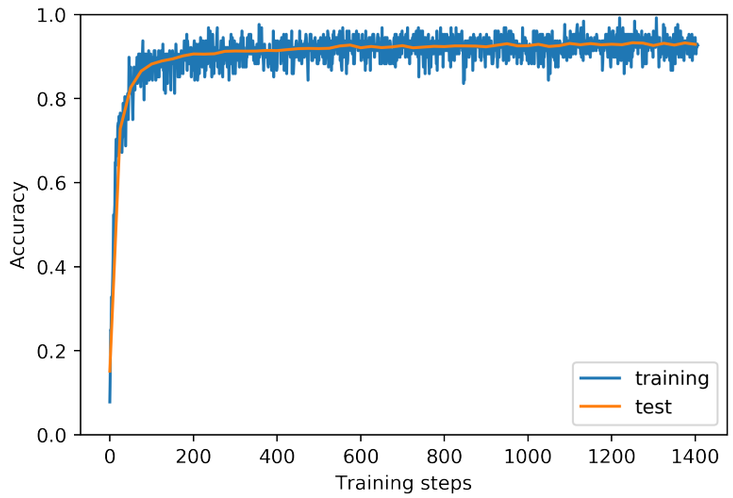
The learning rate is an important hyperparameter, which influences the training dynamics.

Learning Rate: Optimal

e.g. 0.001

Slow and steady convergence.

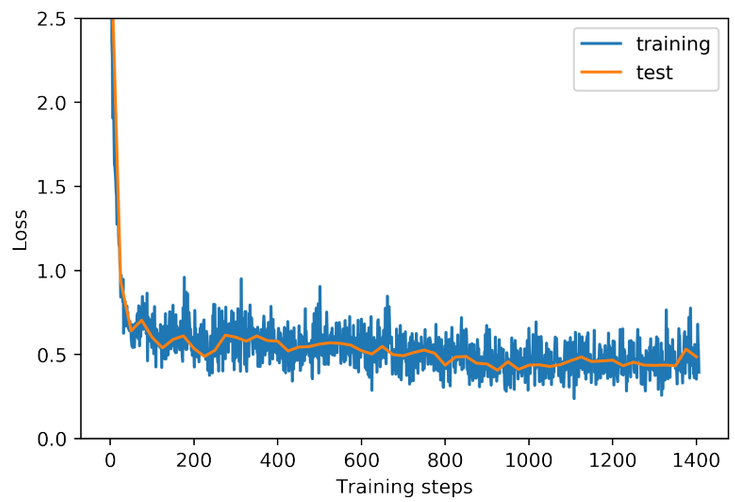


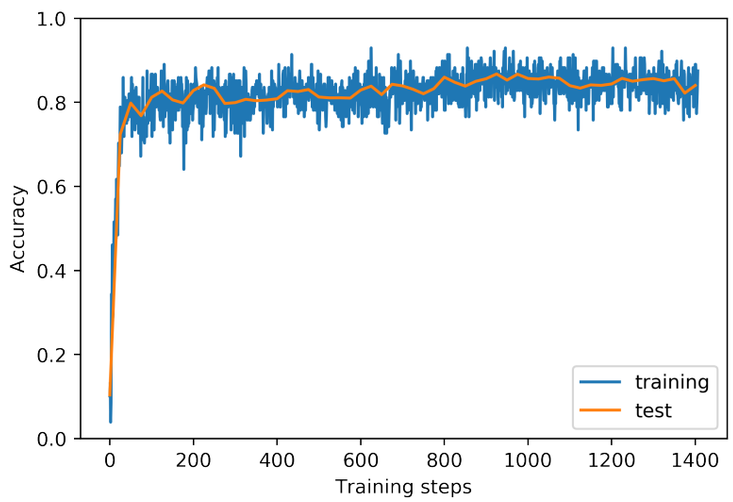


Learning Rate: Slightly too high

e.g 0.01

Fast but non-optimal convergence.





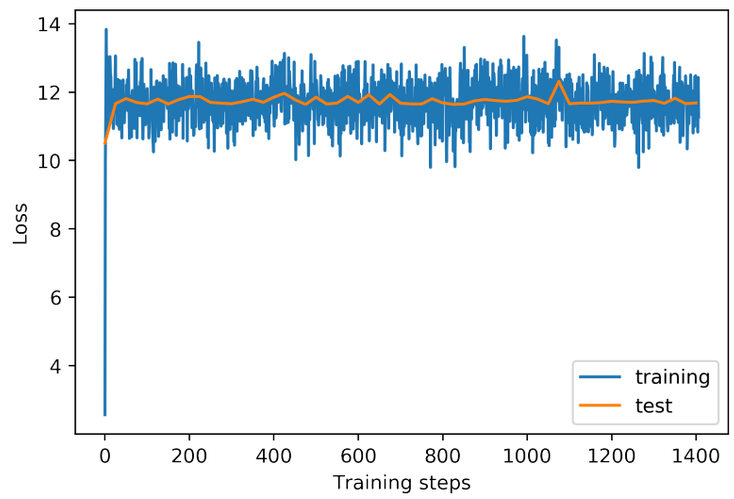
Learning Rate

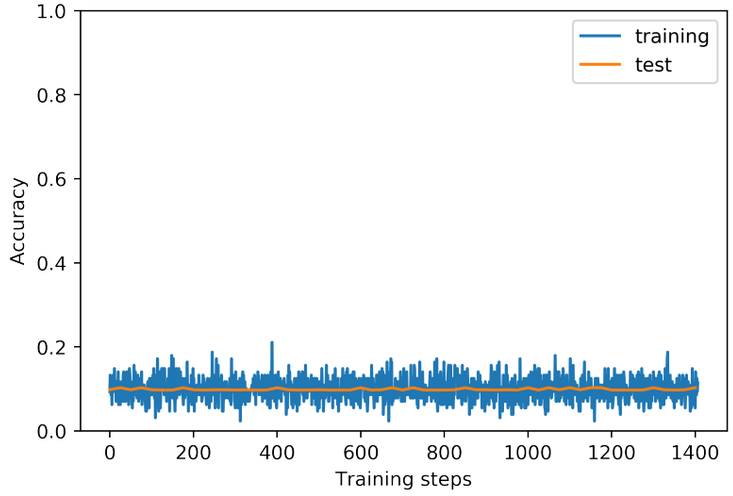
The learning rate is an important hyperparameter, which influences the training dynamics.

Learning Rate:
Way too high

e.g. 0.1

No learning.

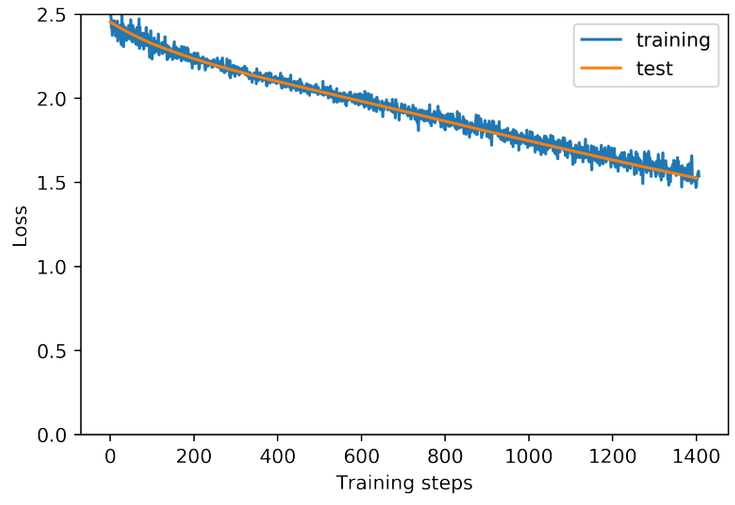


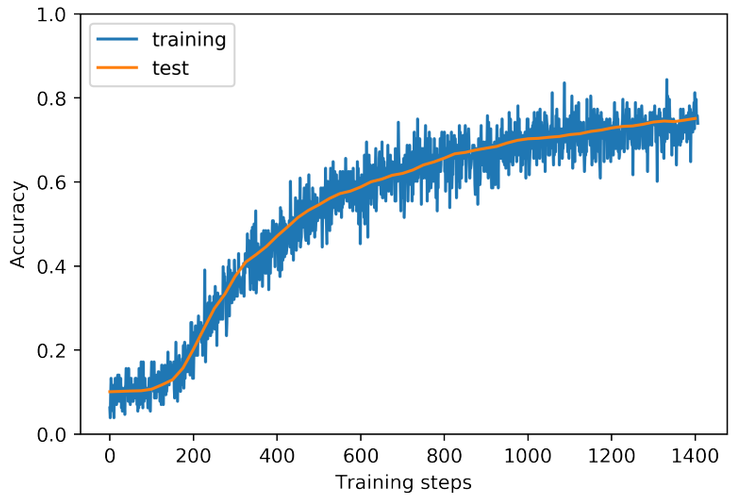


Learning Rate:
Too low

e.g 0.00001

Slow non-effective convergence.





Finding a Good Learning Rate

- As with most hyperparameters you can only find a good solution by trial.
- Start with some suitable choice (e.g. 0.001).
- Observe training dynamics and try higher or lower values.

Decaying Learning Rate

- It can be helpful to reduce the learning rate during training.
- Large learning rate in beginning helps faster convergence.
- Smaller learning late later helps to find better solutions.
- Can be done either manually (stop training and train on with reduced learning rate or automatically using a learning rate schedule (Docs).

```
initial_learning_rate = 0.1
lr_schedule = tf.keras.optimizers.schedules.ExponentialDecay(
    initial_learning_rate,
    decay_steps=100000,
    decay_rate=0.96,
    staircase=True)
optimizer = tf.keras.optimizers.Adam(learning_rate = lr_schedule)
```

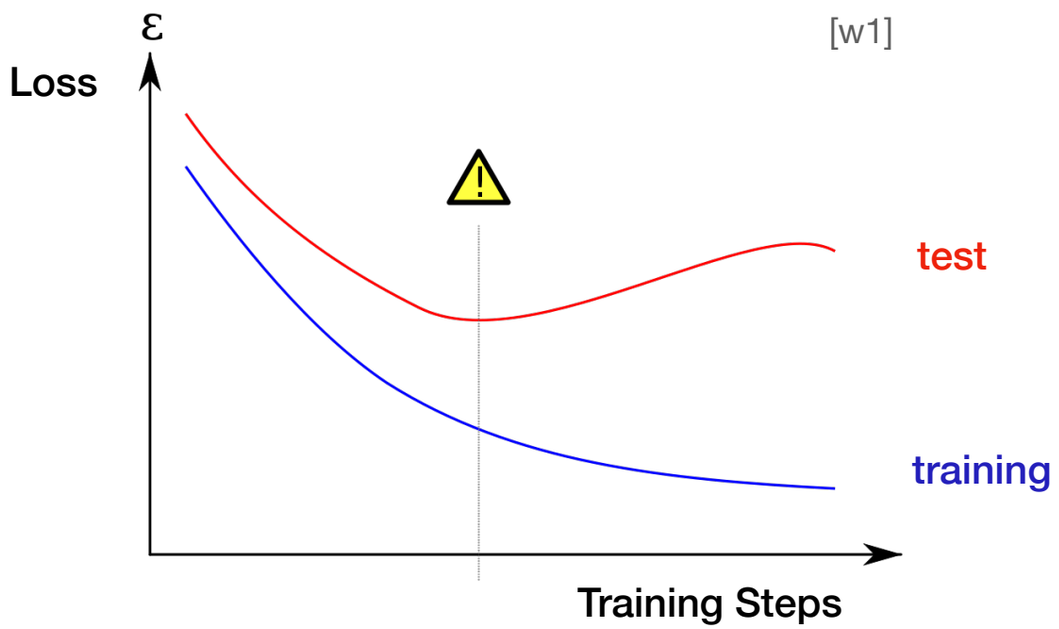
Batch Size

- The batch size can influence the training dynamics in a similar way.
- Again there is no recipe on how to find a good one, except for trial and error.
- Usually a good point to start is batch size 128.
- In general the batch size is correlated to the learning rate: e.g. a large batch size yields stable gradients, which would allow you to use a larger learning rate.

Problem 2: Overfitting

Overfitting

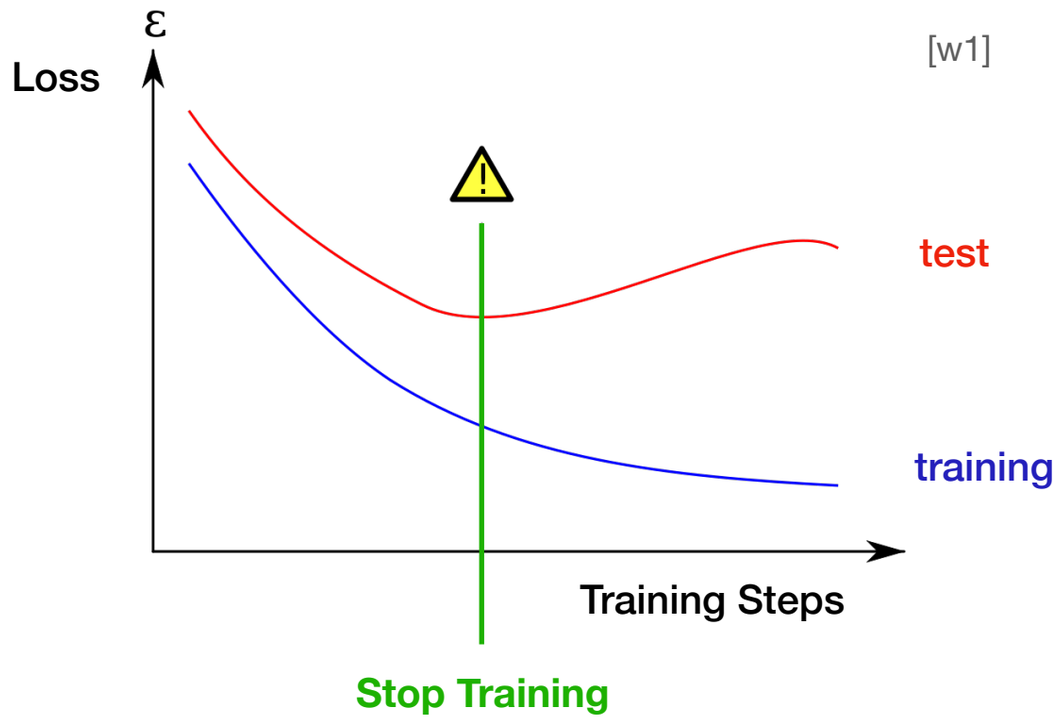
- Using all previously showed tips you should be able to make your neural network learn as effective as possible.
- But there is a second problem that you can encounter: the neural network overfits the training data.
- While it is improving performance on the training set it start to get worse on the test set.



Early Stopping

- A very simple solution is early stopping.

- It simply means to stop training as soon as you observe overfitting.



- But this does not prevent overfitting.

- Methods which prevent overfitting are regularization techniques.

Data Augmentation

- A common reason for overfitting is that you don’t have enough datapoint or too few variance in your training data.

- There is a simple solution to make your dataset larger and increase variance in the data: data augmentation.

- It means to apply small transformations to your training data, which do not change the target value.

Data Augmentation Images

- For images these data augmentations are simple image transformations.

Original

Random Crop

Horizontal Flip

Rotate & Crop

Color Transformation

Data Augmentation in TensorFlow

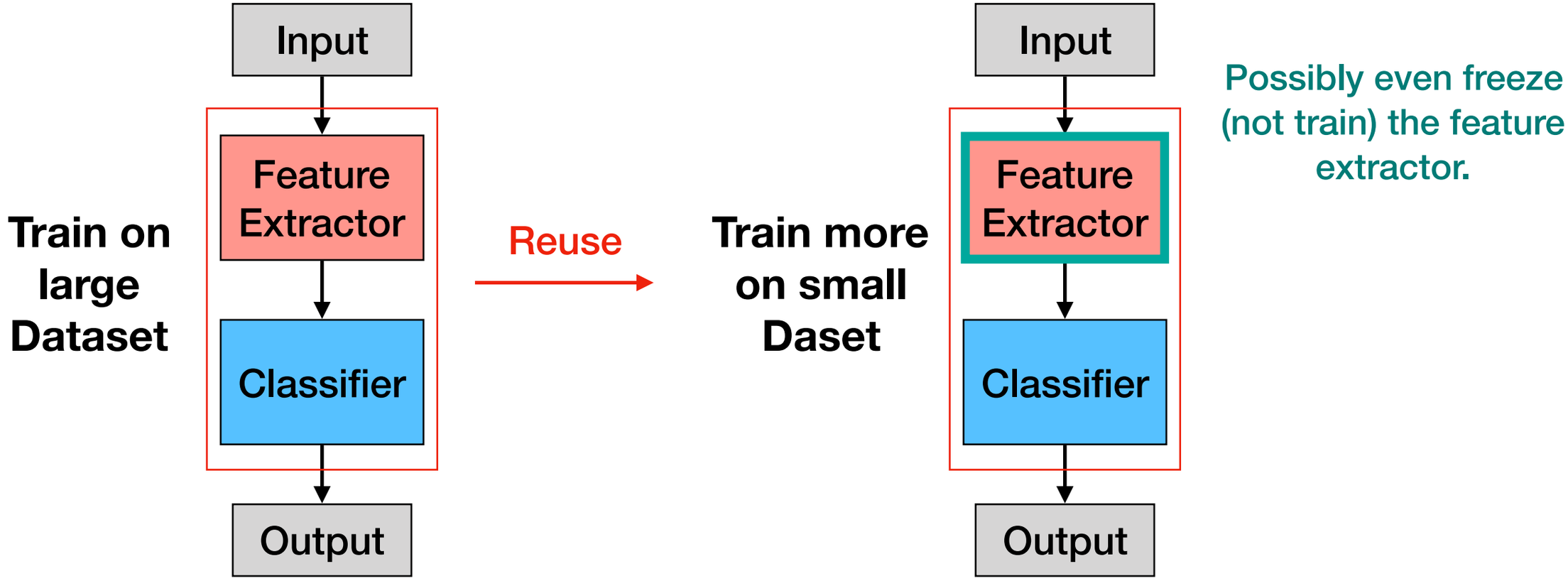
```
data_augmentation = tf.keras.preprocessing.image.ImageDataGenerator(
                                                    rotation_range=45,
                                                    width_shift_range=0.1,
                                                    height_shift_range=0.1,
                                                    brightness_range=0.1,
                                                    shear_range=0.2,
                                                    zoom_range=0.2,
                                                    horizontal_flip=True
                                                    )

train_generator = train_datagen.flow(
                            x=train_images,
                            y=train_labels,
                            batch_size=32,
                            shuffle=True,
                            )
```

```
for (x,t) in train_generator:
    with tf.GradientTape() as tape:
        output = model(x)
        ...
```

Transfer Learning

- But if you have too little data, data augmentation won't be able to solve your problem.

- Networks trained on larger datasets can be re-used: called transfer learning.



L2 Regularization

- In the case of overfitting it is often observed that parameters get very large.
- Thus a common regularizer is L2 regularization, which punishes large weights.
- This punishment term is simply included in the loss function.

Regularization parameter (e.g. 0.01)

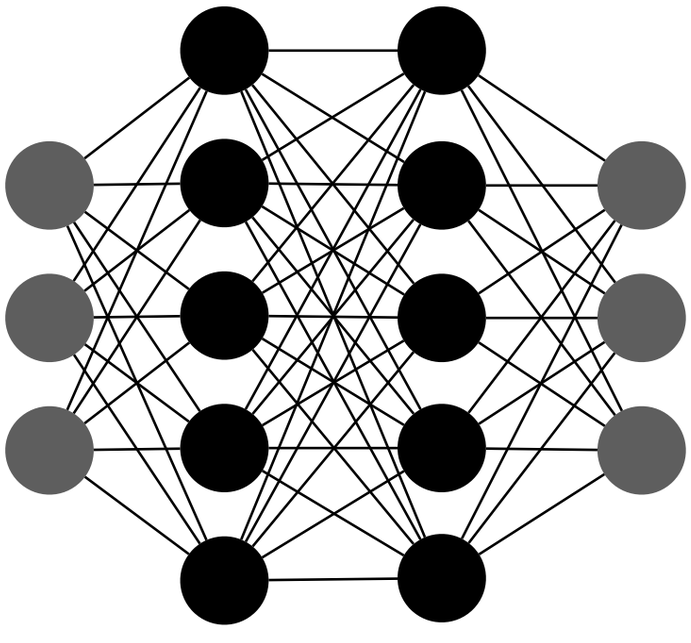
$$l2\_loss = loss + \frac{1}{2}\lambda \sum_{w} w^2$$

```
loss = cross_entropy_loss(t, output)
l2_loss = 0
for w in model.trainable_variables:
    l2_loss += tf.nn.l2_loss(w)
total_loss = loss + 0.01 * l2_loss
```

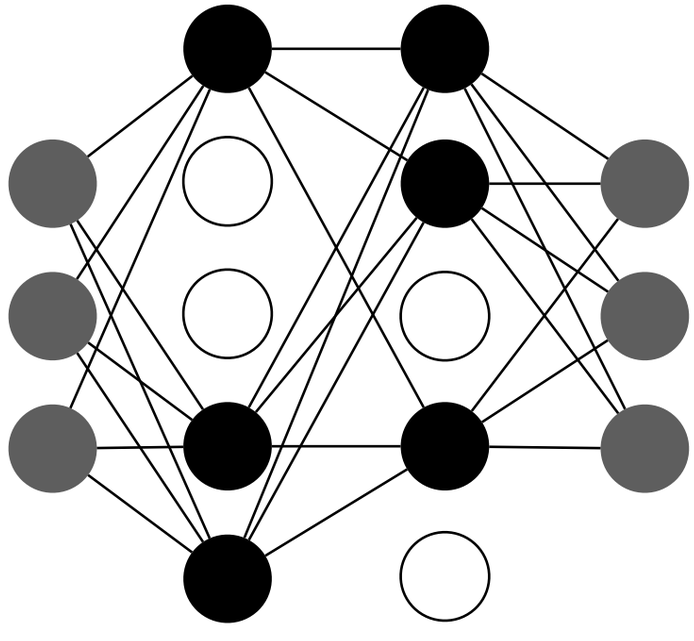
Dropout

- One very effective regularizer is called DropOut.
- It simply means that during a training step you set for the layer some random neurons to zero.
- This means that the network can’t rely to much on particular feature detectors and thus does not overfit.

Original Network

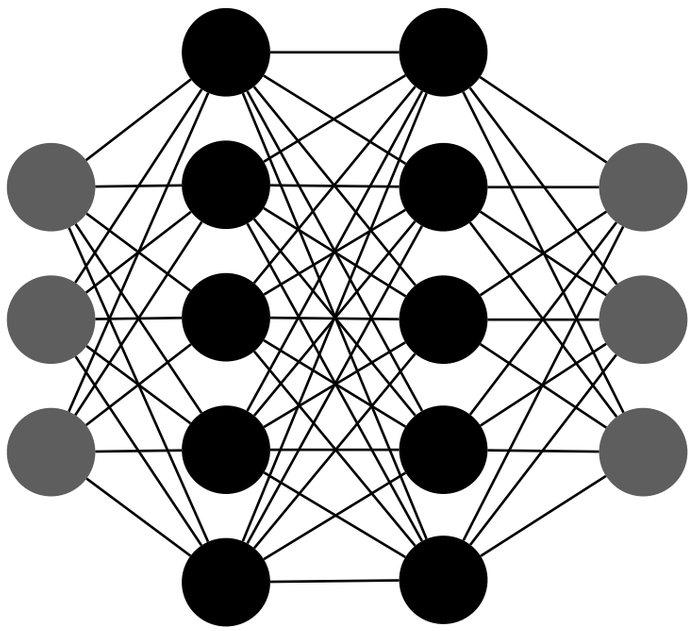


Dropout 20 % both hidden layers



Dropout

- In each iteration different randomly drawn neurons are dropped.
- This means we train a different subnetwork of the original network every time.
- During inference, we do not drop neurons. We want to use the information, which is stored in “all subnetworks”
- How do we do that? We simply use all neurons and scale their activations, with the probability that they would have persisted (otherwise the drive of the next layer is too large)



Dropout in TF

- Define a dropout layer, with the probability to drop each neuron.

```
self.conv_layer_1 = tf.keras.layers.Conv2D(
                        filters=32,
                        kernel_size=3,
                        activation=tf.keras.activations.relu,
                        input_shape=(32,32,3)
                    )
self.max_pool_1 = tf.keras.layers.MaxPool2D()
self.drop_out_1 = tf.keras.layers.Dropout(0.2)
```

- As in batch normalization the call method of your model requires an argument that indicates whether you are training or testing.

```
def call(self, x, is_training):
    x = self.conv_layer_1(x)
    x = self.max_pool_1(x)
    x = self.drop_out_1(x, training=is_training)
```

- For convolutional layers dropout is applied after the max pooling layer.

Neural Networks will Cheat

Cheating

- Given all these tips and tricks you should be able to train a network, which performs well on your chosen task.
- But there is a last thing to be aware of.
- If a network can cheat, it will usually do so.

Imbalanced Data

- The simplest case of cheating is based on an imbalanced dataset.

- Example: Classify diagnostic values about a patient as *Cancer* vs *NoCancer*.

- Probably you have way more samples of patients without cancer than of patients with cancer (e.g. 90% vs 10%).

- The network can thus achieve a good performance by simply classifying all samples as *NoCancer*.

Metrics for Imbalanced Data

- To test whether our network does this we need new metrics to supervise our model (because accuracy fails).
- Most general: Confusion matrix.
- Can be easily extended to multiple classes.

		Actual Class	
		Cancer	NoCancer
Predicted Class	Cancer	True positives	False positives
	NoCancer	False negatives	True negatives

Metrics for Imbalanced Data

- Given the values of the confusion matrix we can define our metrics:

Accuracy

$$acc = \frac{tp + tn}{tp + fp + fn + tn}$$

How many predictions were correct in total?

Recall

$$rec = \frac{tp}{tp + fn}$$

How many of the actual positive samples were classified correctly?

Precision

$$prec = \frac{tp}{tp + fp}$$

How many of the predicted positive samples were actually positive?

And there are many more: https://www.tensorflow.org/api_docs/python/tf/metrics

Training on Imbalanced Data

- These metrics help you to detect whether the network is actually performing good.
- If we find that the data is in fact exploiting the imbalanced data structure, how can we prevent that?

Weighted Loss

- First possibility is the weighted loss.

- Compute the ratio of how many samples are in the two classes:

$$w_+(=0.1) \qquad w\_(=0.9)$$

- Use these weights to punish wrong classifications for rare class harder:

Positive Sample $\longrightarrow$ $loss$ $\longrightarrow$ $loss_{weighted} = (1 - w_+) \cdot loss$

Negative Sample $\longrightarrow$ $loss$ $\longrightarrow$ $loss_{weighted} = (1 - w\_) \cdot loss$

TF: https://www.tensorflow.org/api_docs/python/tf/nn/weighted_cross_entropy_with_logits

Balanced Mini-Batches

- Second solution is: training on balanced mini-batches.
- The dataset as a whole is imbalanced, but we only train on one mini-batch at a time.
- We can make sure that these mini-batches are always balanced.
- But be aware that the model might overfit on the rare data.

Cheating Possibilities in your Data

- Lastly there is the possibility that your model uses hints, that accidentally occur in your training data.

- Example:

    - Trained on images of dumbbells

    - Accidentally all images might contain arms.

    - So the network might classify arms instead of dumbbells.

Any questions?

Conclusion

- We investigated which reasons can cause your network to fail at learning and what you can do about it.
- You can change aspects of the network (e.g. depth, width, activation function.)
- You can play around with the training parameters (e.g. batch size, learning rate).
- To make sure that your drives are well distributed it makes sense to normalize your data, initialize the weights according to e.g. Xavier and use batch normalization.

Conclusion

- Further your network can suffer from overfitting which you can prevent with regularization techniques (e.g. early stopping, data augmentation, dropout).

- Lastly you have to make sure that the network actually learned something (check other metrics than accuracy, check whether there are superficial clues that the network might use).

Outlook

- In the homework you will try to solve a dataset as good as possible using everything that you learned so far.
- Homework is due to after the Christmas break.
- After Christmas we will start with more fun applications of neural networks.

See you next week!

Resources

[w1] Gringer at Wikipedia (https://de.wikipedia.org/wiki/Datei:Overfitting_svg.svg)