Application of CNNs

Overfitting, Regularization
Practical example: Flavia

Table of contents

- Application of CNNs: Flavia as an example
  - CNN, number of parameters, shapes of parameters and layers
- Regularization methods
  - Dropout
  - Early Stopping
  - Data augmentation
  - …
- Application and Evaluation

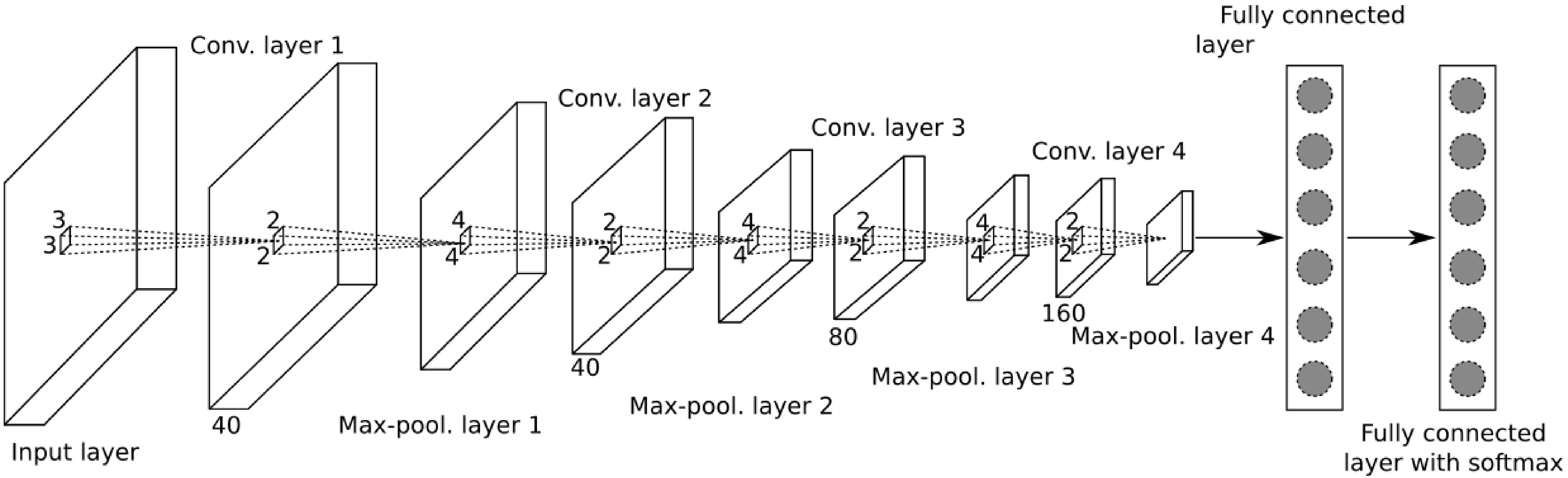
Application: Leaf classification

- Classification of leaves
- Using a deep CNN for classification
- Introducing several techniques like Dropout, data augmentation, transfer learning
- Based on *Wick & Puppe: Leaf Identification Using a Deep Convolutional Neural Network (2017)*

Application: The Flavia data set

- Photographs of leaves from 32 tree types
- Task: Classify the type of tree from the leaf image
- 1,907 photos, so about 60 per class
- 10 photos per class for testing
- Simplification: Leaf stalk removed
- Input size is $100 \times 100 \times 3$ (300 in the paper)

Application: The CNN architecture



Layer	Input	Conv1	Pool1	Conv2	Pool2	Conv3	Pool3	Conv4	Pool4	Dense	Dense
Shape	100x100x3	100x100x40	50x50x40	50x50x40	25x25x40	25x25x80	12x12x80	12x12x160	6x6x160	500	32
Params	-	3x3x3x40	-	4x4x40x40	-	4x4x40x80	-	4x4x80x160	-	5760x500	500x32

Application: CNN in Keras

```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Input, Dense, Conv2D, MaxPooling2D, Dropout, Flatten


model = Sequential([
    Input(shape=(target_size, target_size, 3)),
    Conv2D(40, (3, 3), padding='same', activation='relu'),
    MaxPooling2D(pool_size=(2, 2), strides=2),
    Conv2D(40, (4, 4), padding='same', activation='relu'),
    MaxPooling2D(pool_size=(2, 2), strides=2),
    Conv2D(80, (4, 4), padding='same', activation='relu'),
    MaxPooling2D(pool_size=(2, 2), strides=2),
    Conv2D(160, (4, 4), padding='same', activation='relu'),
    MaxPooling2D(pool_size=(2, 2), strides=2),
    Flatten(),
    Dense(500, activation='relu'),
    Dense(num_classes, activation='softmax', name='softmax'),
])
```

Application : The CNN architecture

Layer	Input	Conv1	Pool1	Conv2	Pool2	Conv3	Pool3	Conv4	Pool4	Dense	Dense
Shape	100x100x3	100x100x40	50x50x40	50x50x40	25x25x40	25x25x80	12x12x80	12x12x160	6x6x160	500	32
Params	-	3x3x3x40	-	4x4x40x40	-	4x4x40x80	-	4x4x80x160	-	5760x500	500x32

Number of parameters:

Input

- Without Bias

$$1080 + 25{,}600 + 51{,}200 + 204{,}800 + 2{,}880{,}000 + 16{,}000 = 3{,}179{,}180$$

- Bias:

$$40 + 40 + 80 + 160 + 500 + 32 = 852$$

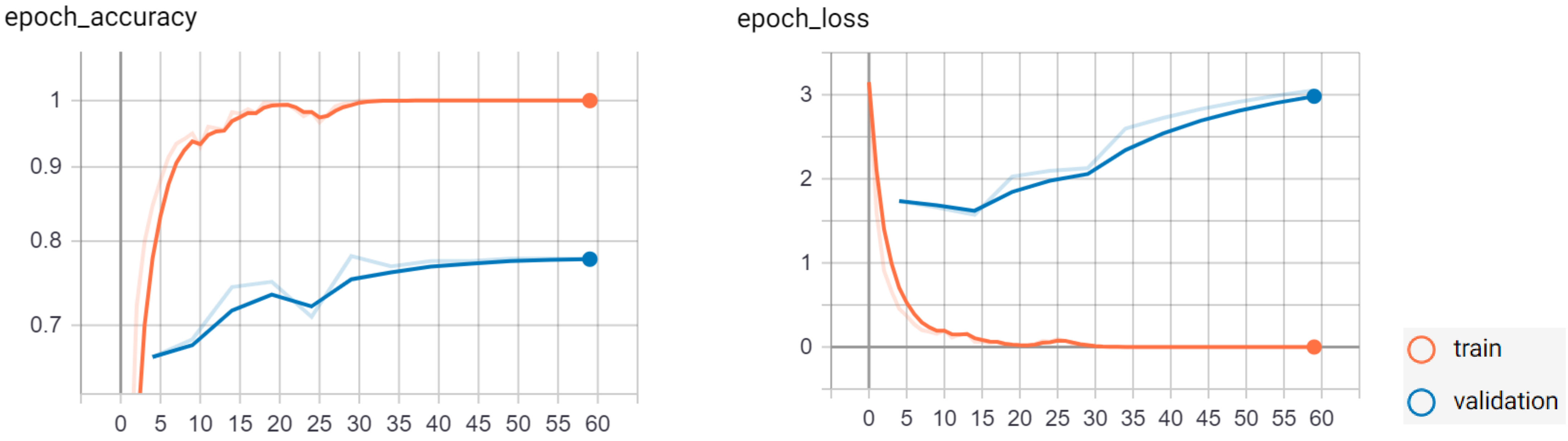
- Total:

$$3{,}180{,}032$$

Application: Hyperparameters

- Input size $100 \times 100 \times 3$
- 30 (to 60) epochs
- Batch-Size 32
- Learning-Rate 0.001
- Solver: Adam

Application: Results



Accuracy and Loss for train and validation after 60 training epochs

Application: Results

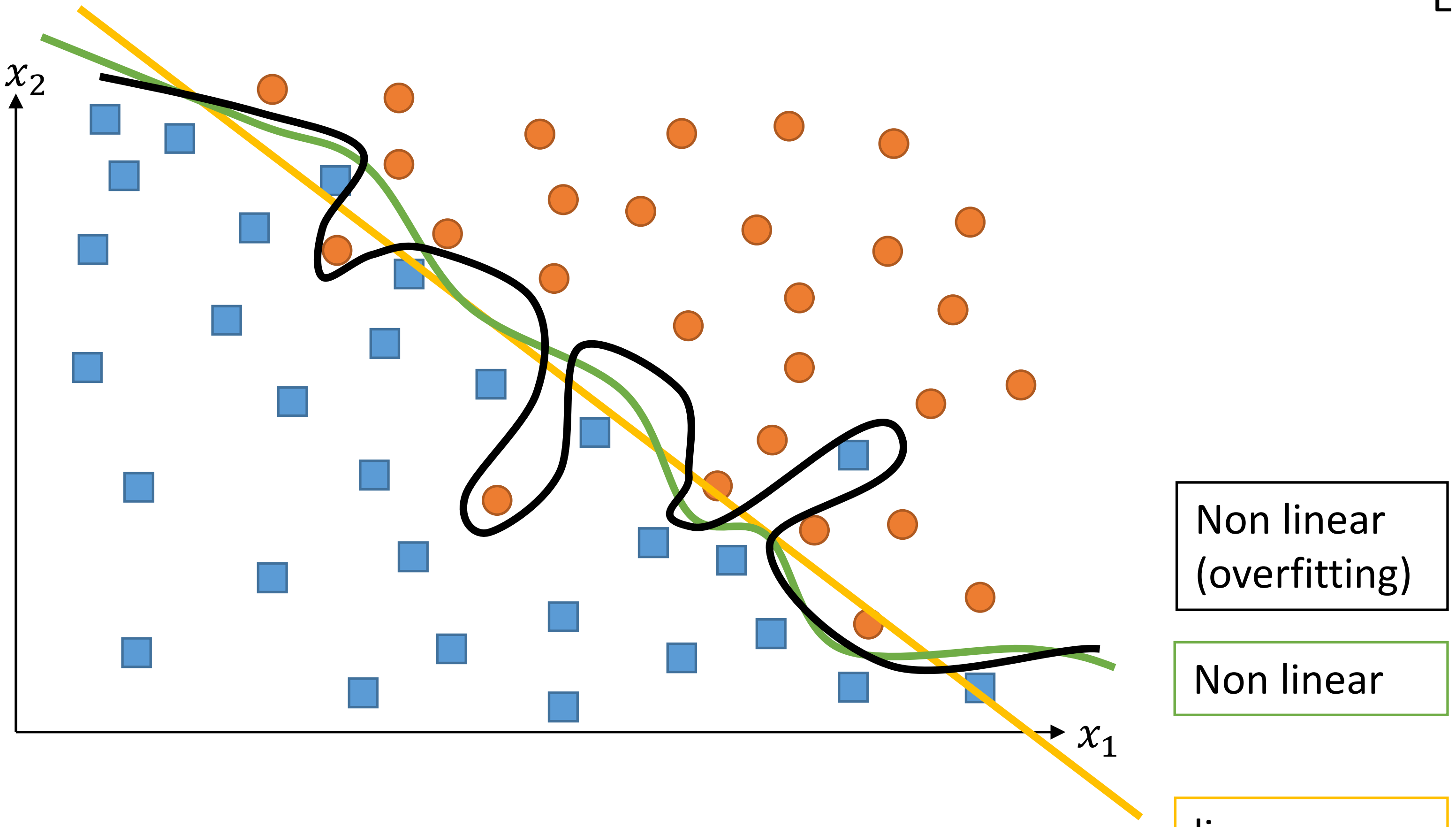
Epoche	Training-Accuracy	Validierung-Accuracy
0	2.0%	2.8%
5	84.6%	66.6%
10	95.0%	68.4%
20	99.6%	75.0%
30	99.6%	78.1%

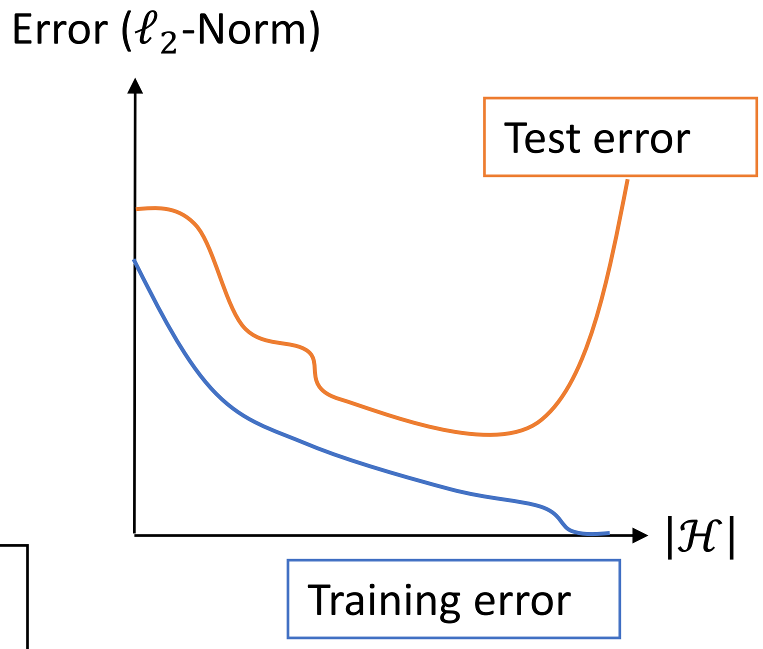
Observations:

- Training data after 30 epochs at almost 100%
- Validation data after 30 epochs only at 78%
- In generel model will perform much better on training data!

⇒ Overfitting!

Overfitting





Overfitting

- Problem: Empirical Loss and expected Loss are different
  - Or: Training error und test/generalization error are different
- The bigger the dataset, the smaller the difference
- The bigger the hypothesis/number of features are, the easier it is to find a hypothesis which simply memorizes the training dataset (small training error, big generelization error, overfitting)

- Reject unnecessary hypotheses/features (e.g. with prior knowledge)
- Bigger dataset
- **Regularization**

Deep Learning - Regularization

- In general: Method to reduce overfitting and support the optimization

- Specifically in Deep Learning: Additional terms in the optimization function (Loss function $\hat{L}$)

Regularization

- Regularization term (ℓ_1,ℓ_2 Norm)
- Noise:
  - Input data
  - Weights
  - Output
- Data augmentation
- Early-Stopping
- Dropout
- Batch Normalization

Regularization term as a hard constraint

- Training goal:

$$\min_f \hat{L}(f) = \min_f \frac{1}{N}\sum_{i=1}^{N} \ell(f, \vec{x}_i, y_i), \qquad f \in \mathcal{H}$$

- Parametrized:

$$\min_\theta \hat{L}(\theta) = \min_\theta \frac{1}{N}\sum_{i=1}^{N} \ell(\theta, \vec{x}_i, y_i), \qquad \theta \in \Omega$$

Regularization term as a hard constraint

- Regularisierung measure $R(\Omega)$

$$\min_\theta \hat{L}(\theta) = \min_\theta \frac{1}{N} \sum_{i=1}^{N} \ell(\theta, \vec{x}_i, y_i)\,, \qquad R(\theta) \leq r$$

- Example ℓ_2 Regularization:

$$\min_\theta \hat{L}(\theta) = \min_\theta \frac{1}{N} \sum_{i=1}^{N} \ell(\theta, \vec{x}_i, y_i)\,, \qquad \|\theta\|_2^2 \leq r^2$$

Regularization term as a soft constraint

- Hard constraint is equivalent to soft constraint:

$$\min_{\theta} \hat{L}(\theta) = \min_{\theta} \frac{1}{N} \sum_{i=1}^{N} \ell(\theta, \vec{x}_i, y_i) + \lambda R(\theta)\,, \qquad \lambda > 0$$

- Example ℓ_2 Regularization:

$$\min_{\theta} \hat{L}(\theta) = \min_{\theta} \frac{1}{N} \sum_{i=1}^{N} \ell(\theta, \vec{x}_i, y_i) + \lambda \|\theta\|_2^2\,, \qquad \lambda > 0$$

Regularization term in practise (weight-decay)

- ℓ_2 regularization:

$$\min_\theta \hat{L}_R(\theta) = \min_\theta \hat{L}(\theta) + \frac{\lambda}{2}\|\theta\|_2^2$$

Gradient:

$$\nabla \hat{L}_R(\theta) = \nabla \hat{L}(\theta) + \lambda\theta$$

Update rule:

$$\theta \leftarrow \theta - \eta\nabla\hat{L}_R(\theta) = \theta - \eta\nabla\hat{L}(\theta) - \eta\lambda\theta = (1-\eta\lambda)\theta - \eta\nabla\hat{L}(\theta)$$

- ℓ_1 regularization:

$$\min_\theta \hat{L}_R(\theta) = \min_\theta \hat{L}(\theta) + \frac{\lambda}{2}|\theta|$$

Gradient:

$$\nabla \hat{L}_R(\theta) = \nabla \hat{L}(\theta) + \lambda \operatorname{sign}\theta$$

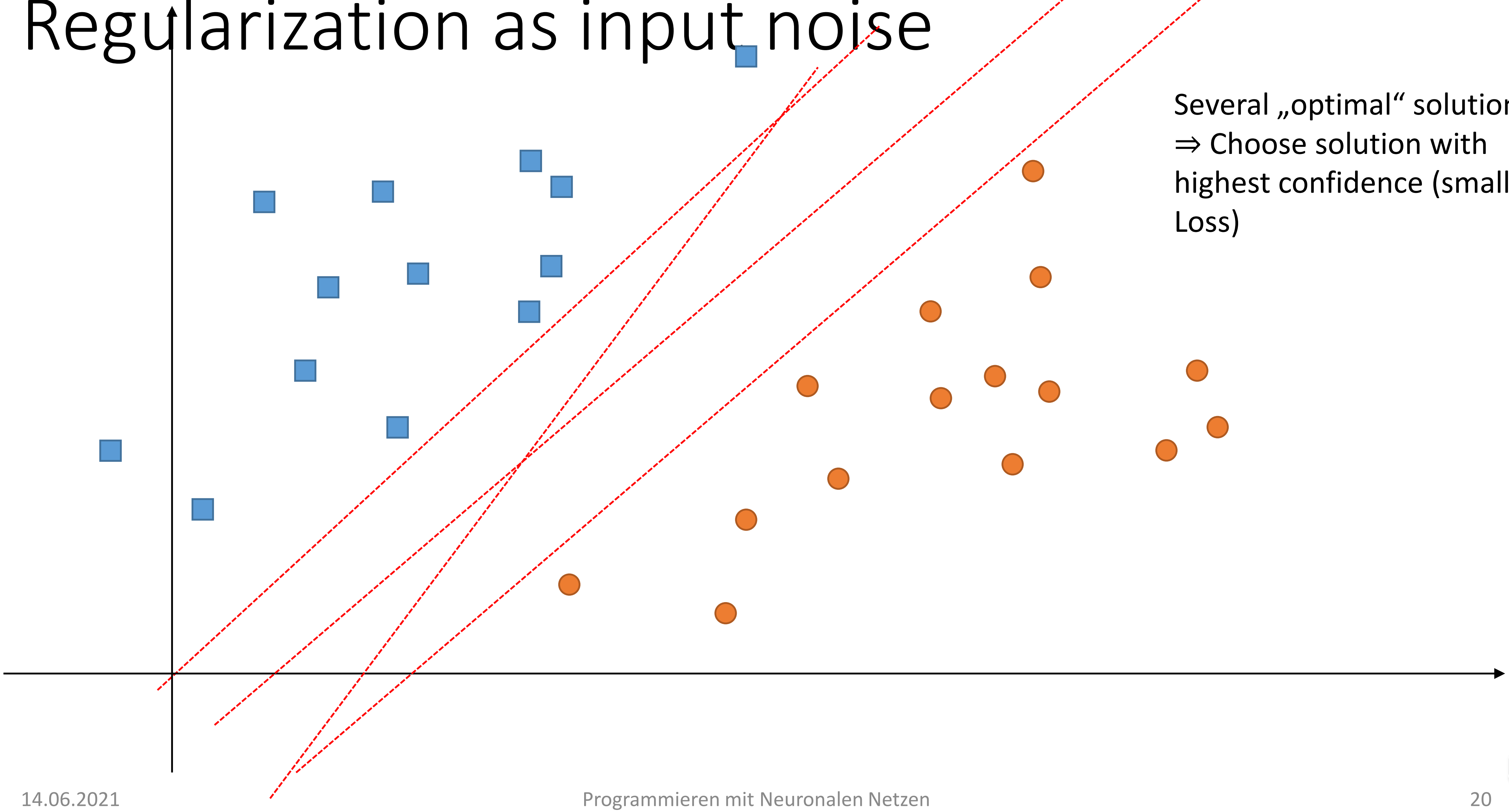
Update rule:

$$\theta \leftarrow \theta - \eta\nabla\hat{L}_R(\theta) = \theta - \eta\nabla\hat{L}(\theta) - \eta\lambda \operatorname{sign}\theta$$

Regularization term in Keras

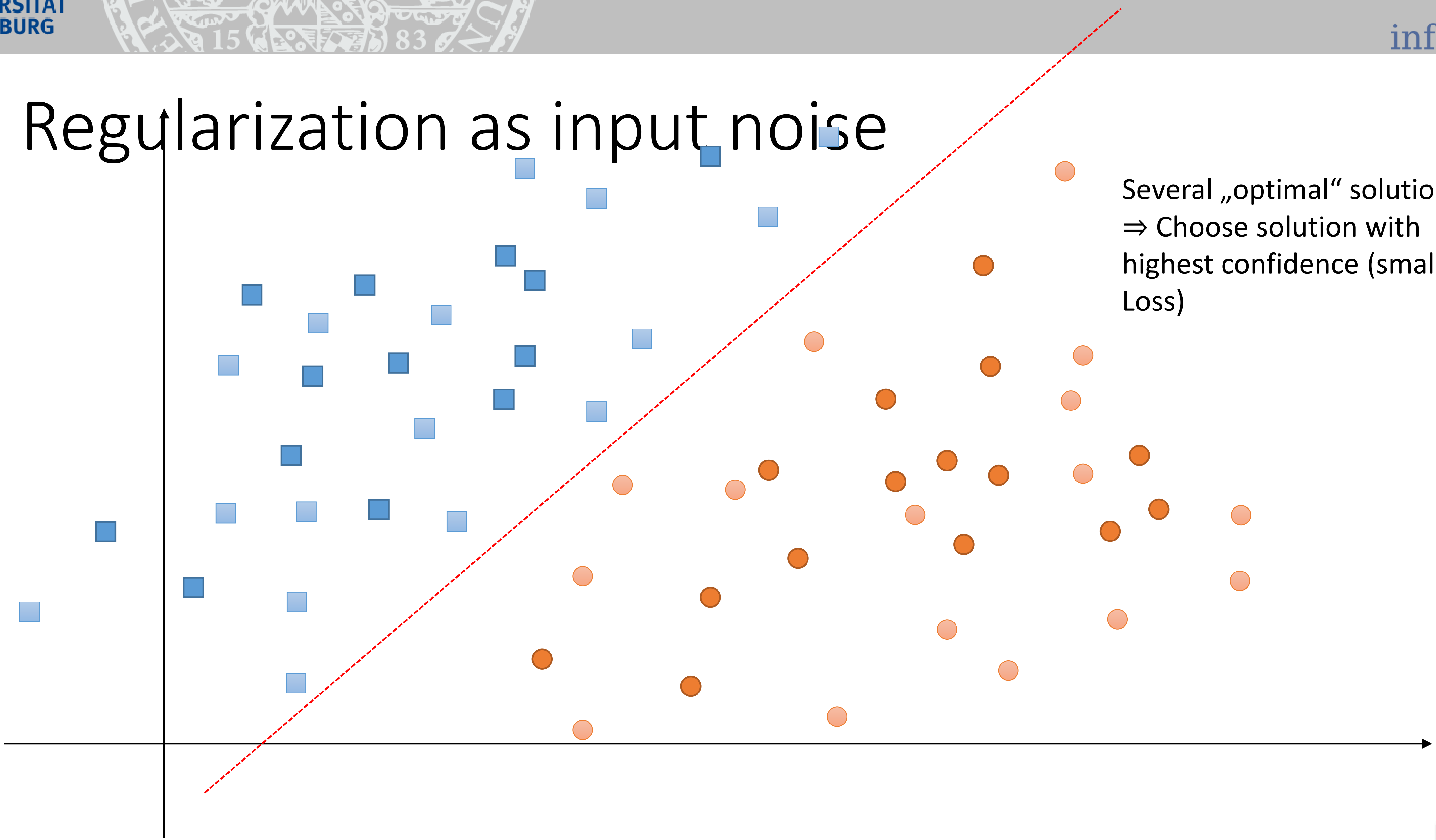
```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.regularizers import l2

lambda = 0.01
model = Sequential([
    Input(shape=784),
    Dense(units=500, activation='relu', kernel_regularizer=l2(lambda), bias_regularizer=l2(lambda)),
    Dense(units=500, activation='relu', kernel_regularizer=l2(lambda), bias_regularizer=l2(lambda)),
    Dense(units=10, activation='softmax', kernel_regularizer=l2(lambda), bias_regularizer=l2(lambda))
    ])
```

Regularization as input noise



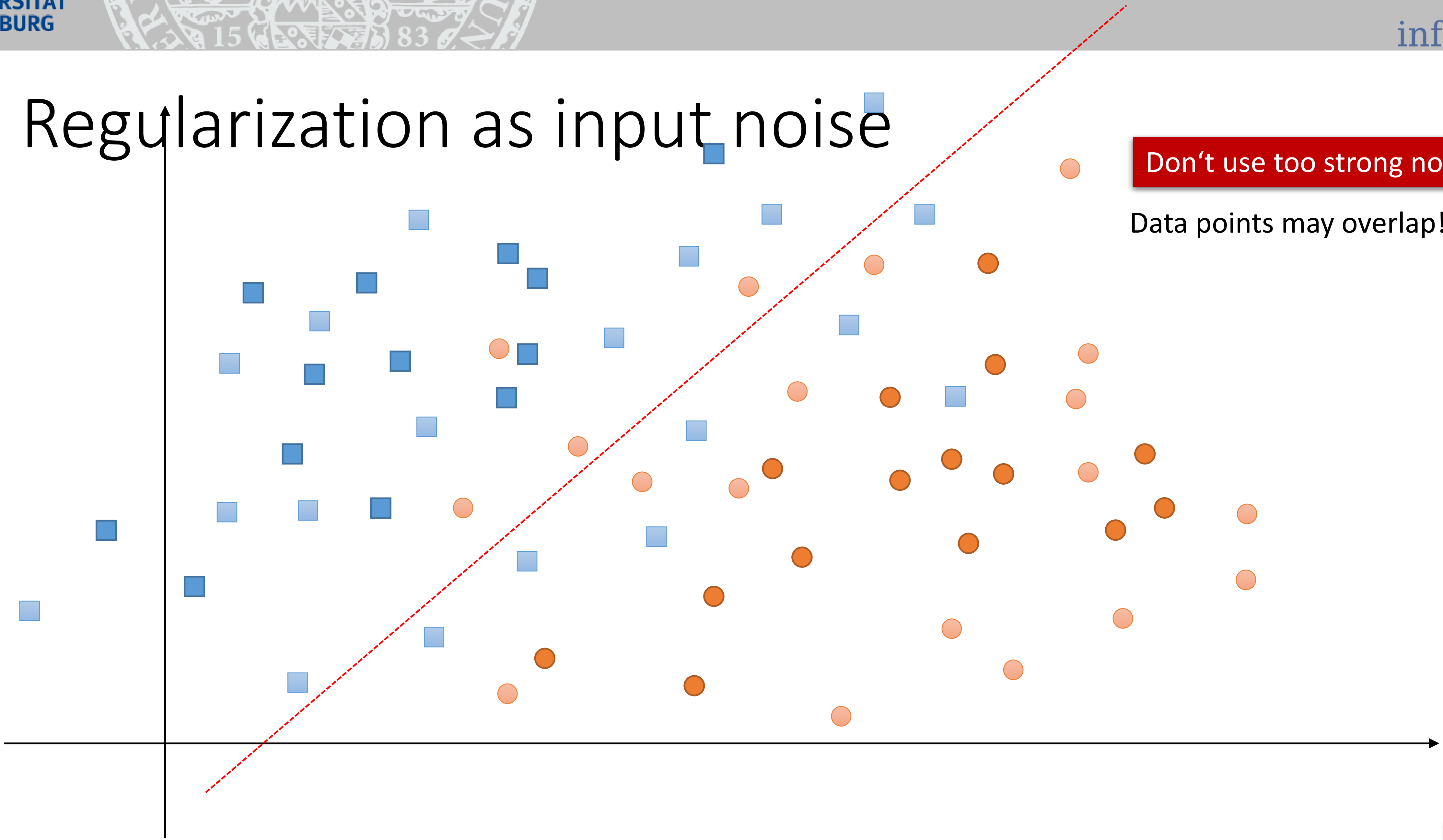
Several „optimal“ solutions?
⇒ Choose solution with highest confidence (smallest Loss)

Regularization as input noise



Several „optimal" solutions?
⇒ Choose solution with highest confidence (smallest Loss)

Regularization as input noise

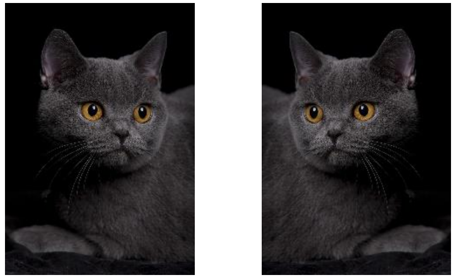


Don‘t use too strong noise!

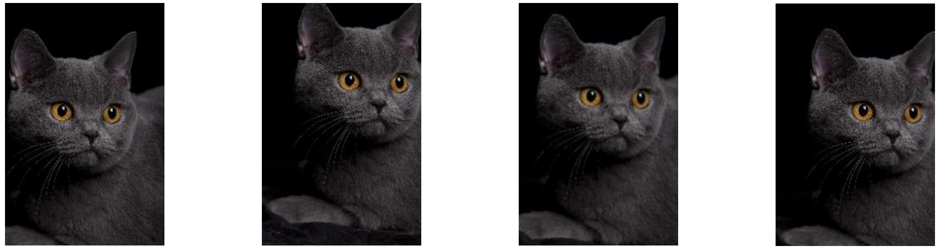
Data points may overlap!

Data augmentation

- Flip:



- Crop:



- Rotate:

Data augmentation

- Brightness/contrast

- Color

- And much more

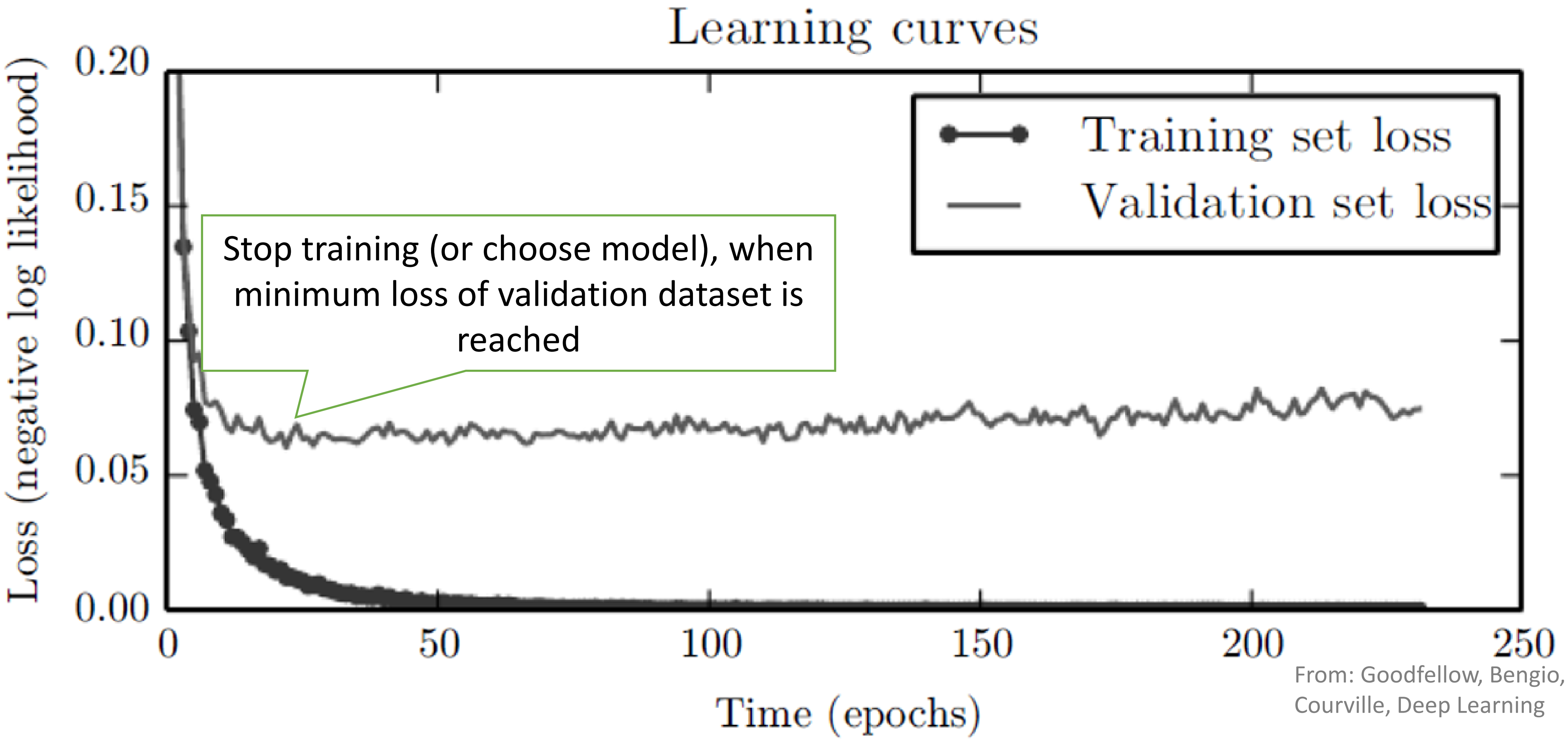
Data augmentation

- Important:

Label must stay the same!

- Example flip or rotation:

Early Stopping



From: Goodfellow, Bengio, Courville, Deep Learning

Early Stopping

- Training duration is hyperparameter
- Validation set for optimization

- Advantage:
  - Efficient: The currently best weights on the validation set must be saved until the error on the validation set does not improve for a while (i.e. a few epochs)
  - Easy: Model and algorithm do not need to be changed!

- Disadvantage:
  - Validation dataset necessary (less data for training)

Early Stopping

Strategy for using validation data for training:

1. Train using the training set and use validation to find and save the optimal training duration (Early Stopping)
2. Repeat the first step for an arbitrary amount of times. The „real“ optimal training duration is then e.g. the mean of all early stopping points
3. Train a new model on training and validation data together until the early stopping point

Recommended addition: [Early Stopping – but when?]

Dropout

- For every iteration:
  - Randomly choose a set of nodes (or weights)
  - Set them to zero
  - Ignore these weights during update

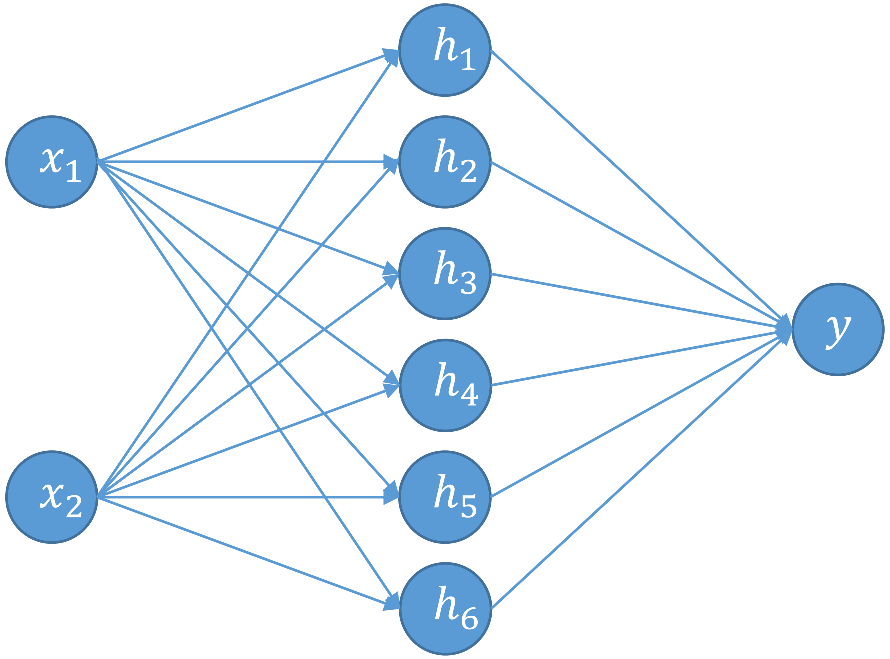
- Ratio of „Dropout“ weights to all weights is the **Dropout Rate**

  Typically 0,2 for input and 0,5 for hidden layers.

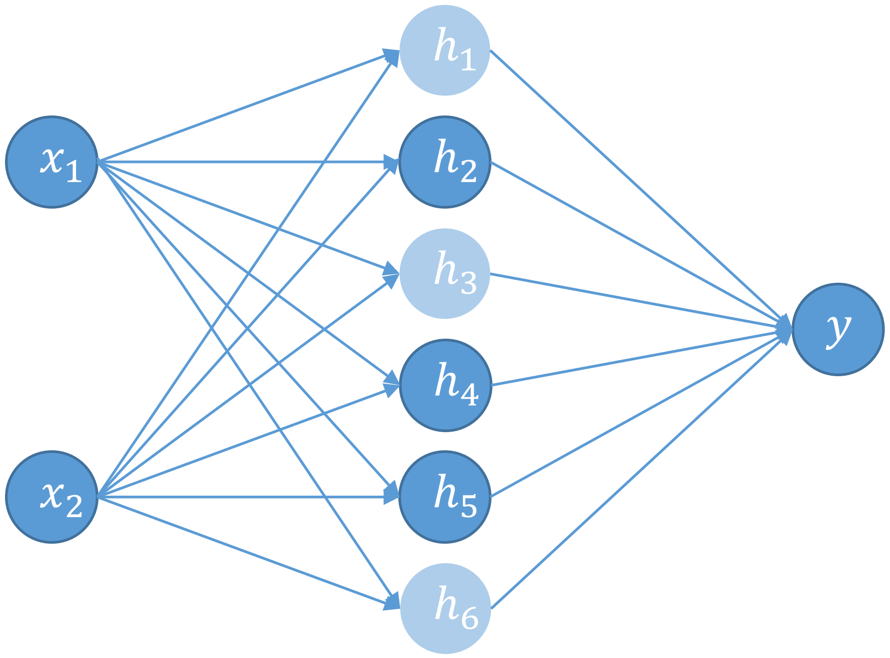
- Network must learn more robust model

Dropout visualized

No Dropout



Dropout (0,5)



Batch Normalization

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_\mathcal{B} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_\mathcal{B}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_\mathcal{B})^2 \qquad \text{// mini-batch variance}$$

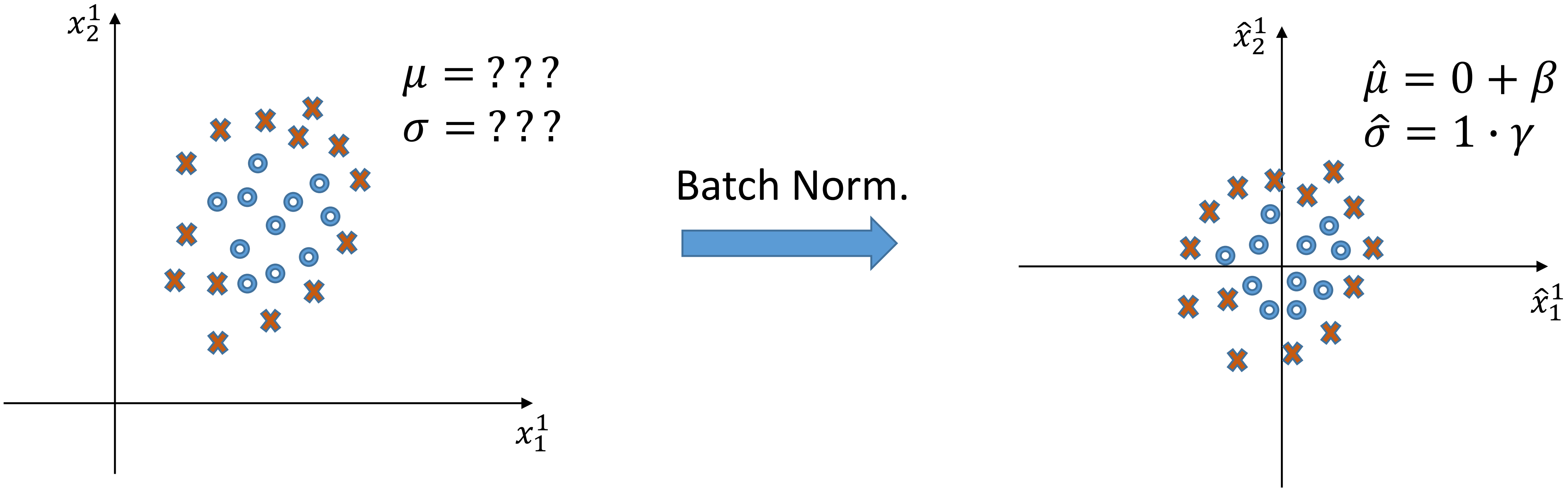
$$\widehat{x}_i \leftarrow \frac{x_i - \mu_\mathcal{B}}{\sqrt{\sigma_\mathcal{B}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

- Reparametrization of (hidden) units
- Normalisierung along feature axis (for every feature map)
- Training: μ and σ from minibatch
  → running statistic for test
  Test: Use the „trained“ μ and σ
- New training parameters

What does Batch Normalization do?

- Accelerates training
- Transforms distribution of hidden units



Covariate Shift

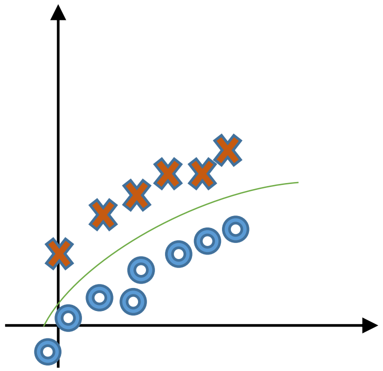
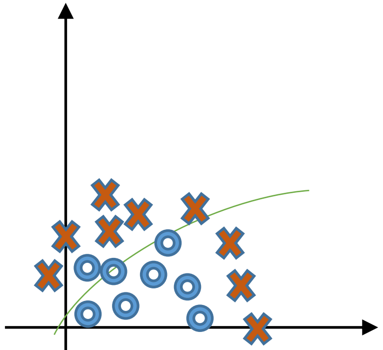
- Input: Distribution changes between datasets → Problem

Training:

$y = 1$ $y = 0$




Covariate-Shift

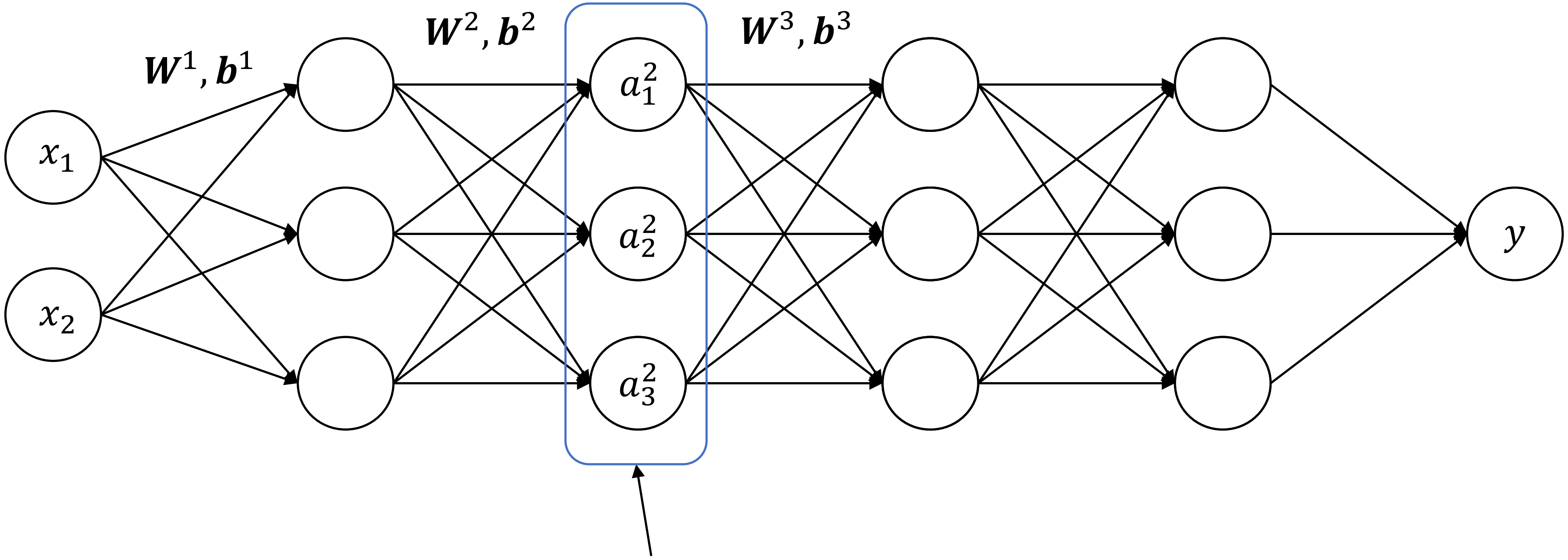



Test:

$y = 1$ $y = 0$

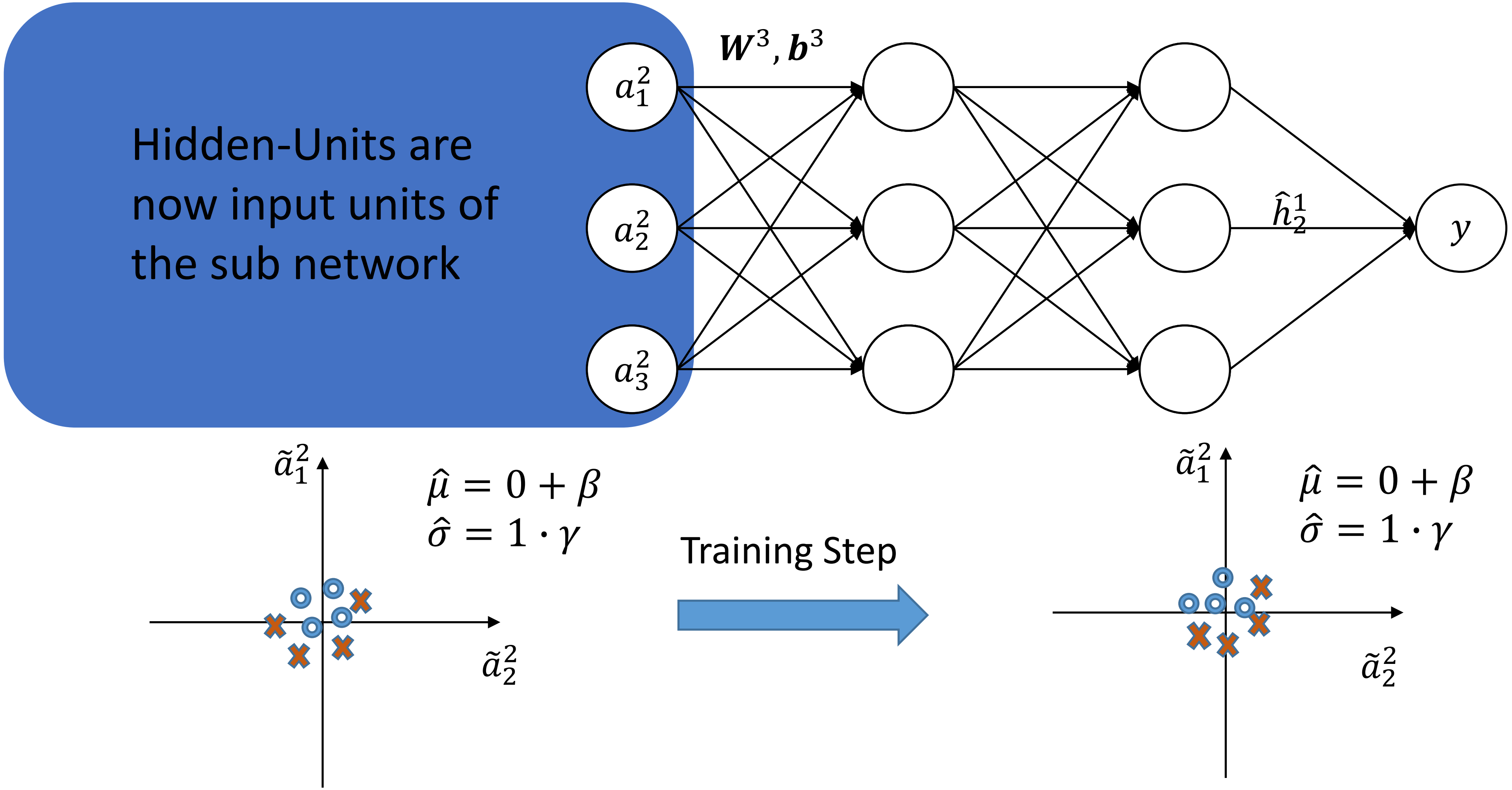



Internal Covariate Shift for Neural Networks



Distribution changes
constantly during training

Interner Covariate Shift bei Neuronalen Netzen



Batch Normalization: Regularization

- Reminder: $\widehat{x_i} = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$
- μ_B and σ_B **only** computed from **minibatch**
  → Introduces noise into the computed $\hat{x}_i$
- Effect similar to Dropout
- Can lead to errors for smaller batch sizes

Regularization in practise

- ℓ_2 Regularization
- Early stopping
- Dropout
- Data augmentation where possible
- (Batch/Group Normalization)

Application: Results (Recap)

Epoch	Training Accuracy	Validation Accuracy
0	2.0%	2.8%
5	84.6%	66.6%
10	95.0%	68.4%
20	99.6%	75.0%
30	99.6%	78.1%

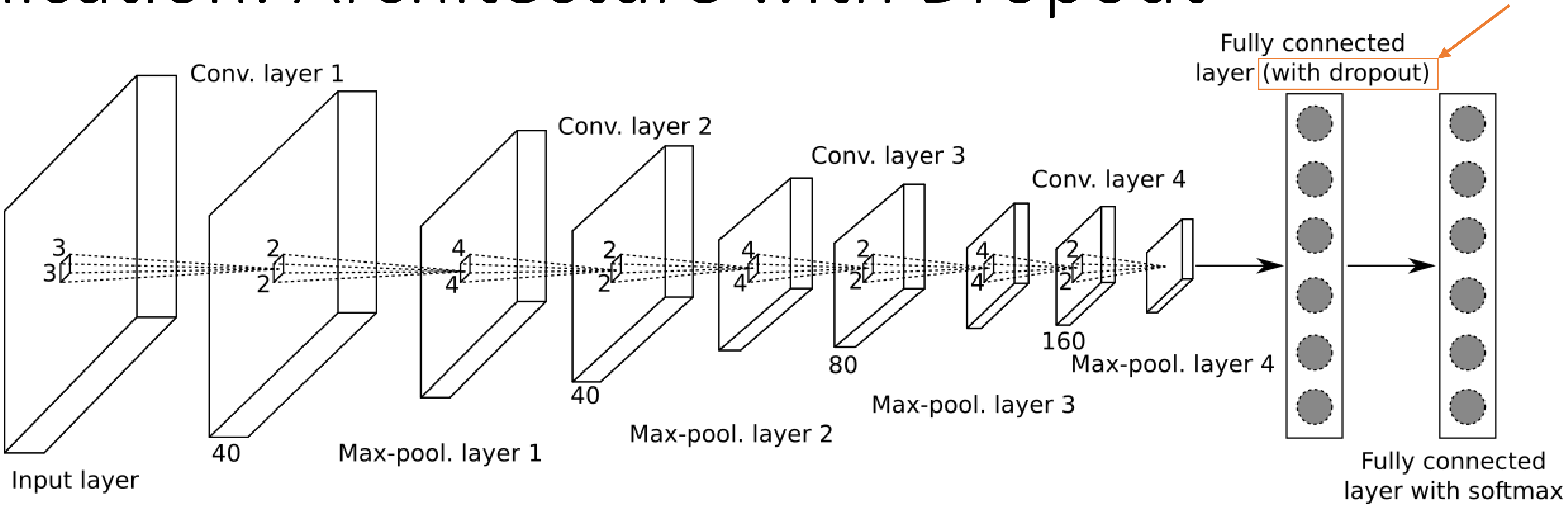
Observations:

⇒ Overfitting!

Applying regularization (usually) leads to:

- Reduces difference of error between training and validation
- Reduces training accuracy

Application: Architecture with Dropout



Layer	Input	Conv1	Pool1	Conv2	Pool2	Conv3	Pool3	Conv4	Pool4	Dense	Dense
Shape	100x100x3	98x98x40	49x49x40	46x46x40	23x23x40	20x20x80	10x10x80	7x7x160	4x4x160	500	32
Params	-	3x3x3x40	-	4x4x40x40	-	4x4x40x80	-	4x4x80x160	-	2560x500	500x32

Application: Dropout in Keras

```
dropout_dense = 0.5
dropout_conv = 0.2
model = Sequential([
    Input(shape=(target_size, target_size, 3), format=),
    Conv2D(40, (3, 3), padding='same', activation='relu'),
    MaxPooling2D(pool_size=(2, 2), strides=2),
    Dropout(dropout_conv),
    Conv2D(40, (4, 4), padding='same', activation='relu'),
    MaxPooling2D(pool_size=(2, 2), strides=2),
    Dropout(dropout_conv),
    Conv2D(80, (4, 4), padding='same', activation='relu'),
    MaxPooling2D(pool_size=(2, 2), strides=2),
    Dropout(dropout_conv),
    Conv2D(160, (4, 4), padding='same', activation='relu'),
    MaxPooling2D(pool_size=(2, 2), strides=2),
    Dropout(dropout_conv),
    Flatten(),
    Dense(500, activation='relu'),
    Dropout(dropout_dense),
    Dense(num_classes, activation='softmax', name='softmax'),
])
```

- Example only uses dropout_dense
- Dropout after conv/pool can help (Standard in many big architectures)

→ Hyperparameter tuning and dataset size important

→ Stochastic effect

Application: Results with Dropout

	Default		Dropout	
Epoch	Training	Validation	Training	Validation
0	2.0%	2.8%	6.1%	4.1%
5	84.6%	66.6%	80.8%	66.3%
10	95.0%	68.4%	93.1%	73.8%
20	99.6%	75.0%	97.4%	71.9%
30	99.6%	78.1%	99.0%	79.1%

Application: Results with Dropout

- Dropout is easy and efficient
- Should be used always in practise
- Here: relative improvement by 1%
- Training takes longer with dropout, but generalization results are better
- But: Training data at 99.0%, validation only at 79.1%

→ Difference between training and validation is still too big!

Application: Data augmentation

Original

Rotation

Flip

Scaling

Application: Augmentation in Keras

```
train_datagen = ImageDataGenerator(
    rotation_range=360,
    zoom_range=0.1,
    # brightness_range=[0.9, 1.1],
    horizontal_flip=True,
    vertical_flip=True,
    rescale=1./255,
    fill_mode='nearest‘)

train_generator = train_datagen.flow_from_directory(
    train_data_dir,
    seed=seed,
    color_mode="rgb",
    shuffle=True,
    target_size=(target_size, target_size),  # all images will be resized to 100x100
    batch_size=batch_size,
    class_mode='categorical')
```

Augmentation: Which values?

Processing method	Range (examples)
Rotation	360°
Zoom	0.1
Width/Height Shift	0.2
Shear	20°
Horizontal/Vertical Flip	True/False
Brightness	[0.9, 1.1]

„Correct“ values are hyperparameter tuning

But: Process in such a way, that a human would choose the same label!

Application: Results with augmentation

	Dropout		Augmentation	
Epoch	Training	Validation	Training	Validation
0	6.1%	4.1%	3.1%	3.1%
5	80.8%	66.3%	26.0%	29.7%
10	93.1%	73.8%	54.8%	46.9%
20	97.4%	71.9%	72.3%	63.1%
30	99.0%	79.1%	80.7%	74.7%
60	-	-	85.6%	82.5%

Application: Results with augmentation

- Slower learning, since data is more diverse

→ Must be trained longer

- Small difference between training and validation:

⇒ Bigger model with more parameters possible

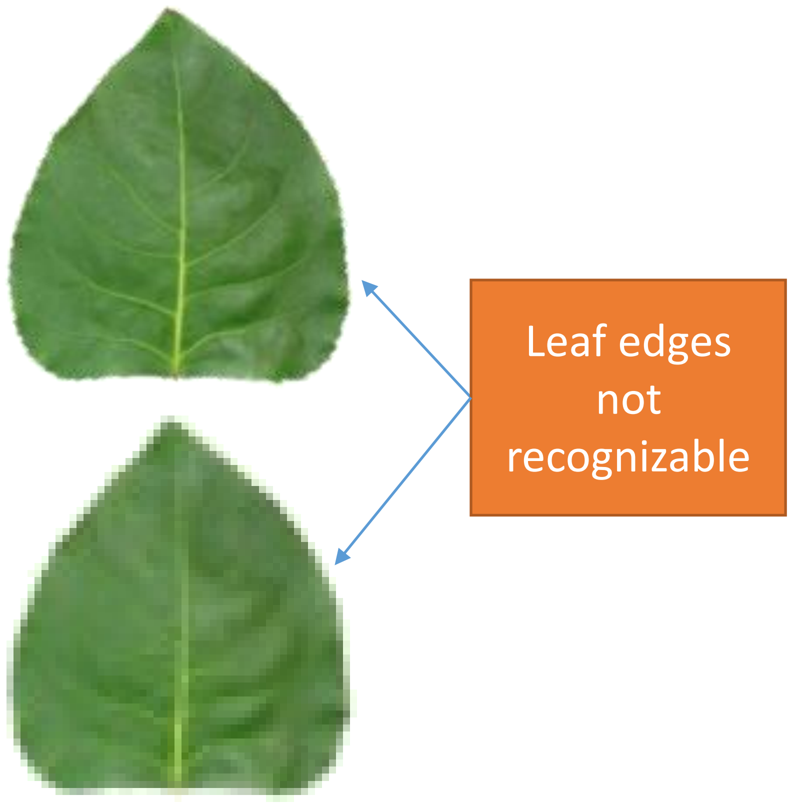
- Better results, but longer training duration
- Paper uses 300 pixels as input dimension (and more augmentations)

Application: Results with bigger input size

	Augmentation		Bigger input size	
Epoch	Training	Validation	Training	Validation
0	3.1%	3.1%	2.3%	2.8%
5	26.0%	29.7%	35.4%	47.5%
10	54.8%	46.9%	59.4%	54.4%
20	72.3%	63.1%	72.9%	59.1%
30	80.7%	74.7%	80.9%	75.9%
60	85.6%	82.5%	90.2%	86.7%

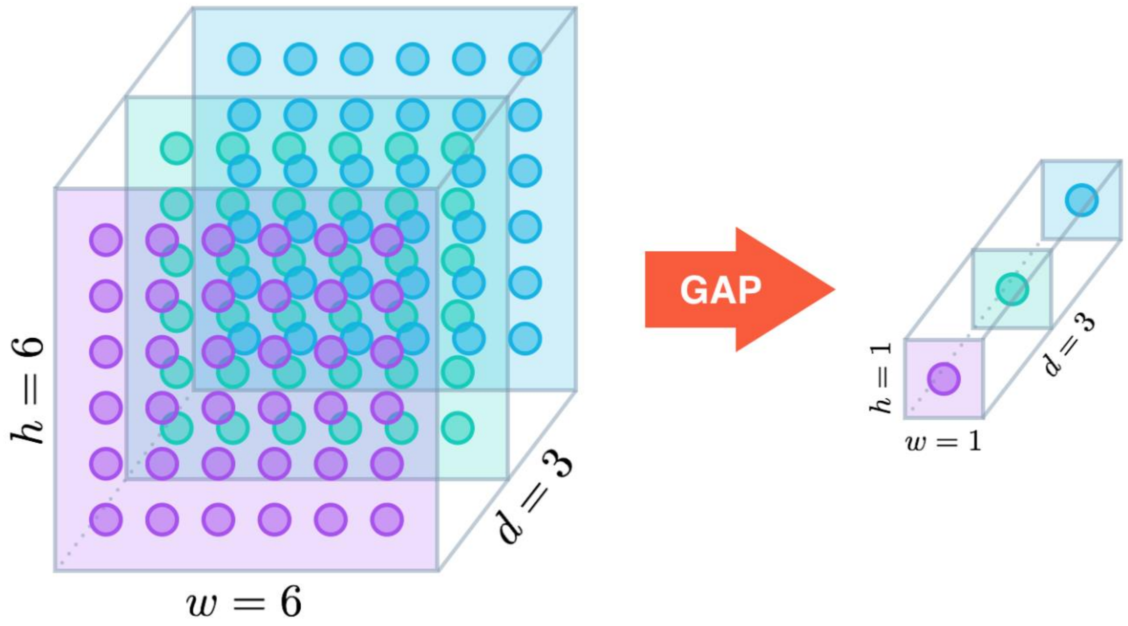
Application: Results with bigger input size

- Bigger input size leads to better results (after small number of iterations already)
- Small images los detailed information of structure at leaf edges and texture
- This information is relevant to classify the leaf
- Important: Scale as much as possible (abstraction), but only to the point that all relevant information are still detectable (details)



Input: Arbitrary input sizes

- Global Average Pooling instead of flattening
- Mapping of every Feature Map to the mean
- Independent of input size→ Arbitrary images
- Reularization effect
- Reduces parameter load



Transferlearning/Pretraining

Choosing the right starting conditions for training

Transferlearning/Pretraining

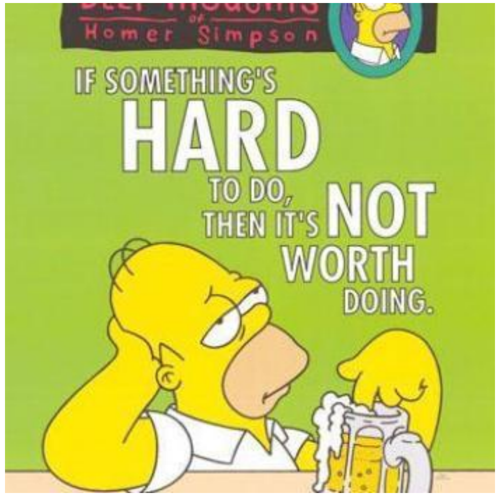
Problem:

- Initializing the weights (starting point) has great influence on the results
- Extreme cases: None or all images correctly classifed

Idea:

- Pretrain the (same) network on other (similar) data
- Use these weights as starting values
- Assumption: Weights already learned useful features/kernels which are applicable to other data

Pretraining on Caltech 256



- 256 Classes
- 30,608 Images
- Around 120 per class
- Scaled to 100x100 (analogous to Flavia)
- Training for 20 Epochs

Pretraining on Caltech 256 - results

	Default		Augmentation	
Epoch	Training	Validation	Training	Validation
0	0.4%	0.4%	0.4%	0.4%
5	29.6%	20.8%	23.0%	18.4%
10	53.3%	21.6%	32.4%	26.5%
20	78.7%	21.3%	43.0%	30.3%

Pretraining on Caltech 256 - results

- Data augmentation on Caltech data very helpful
- Accuracy on training data lower
- Accuracy on validation data at around 32% (very small, but baseline of 0.4%)
- Longer training possible since training data plateau has not been reached yet (but this is enough for pretraining)

Application 1: Use pretrained model

Problem:

- Caltech dataset has 256 classes, Flavia only 32

	Layer	Input	Conv1	Pool1	Conv2	Pool2	Conv3	Pool3	Conv4	Pool4	Dense	Dense
Flavia	Shape	100x100x3	100x100x40	50x50x40	50x50x40	25x25x40	25x25x80	12x12x80	6x6x160	6x6x160	500	32
	Params	-	3x3x3x40	-	4x4x40x40	-	4x4x40x80	-	4x4x80x160	-	5760x500	500x32
Caltech	Shape	100x100x3	100x100x40	50x50x40	50x50x40	25x25x40	25x25x80	12x12x80	6x6x160	6x6x160	500	256
	Params	-	3x3x3x40	-	4x4x40x40	-	4x4x40x80	-	4x4x80x160	-	5760x500	500x256

Solution

- Copy all but the last layer of parameters

Keras: Load and change model

```
# Load model, copy all layers but the last and add a new dense layer
loaded_model = load_model(caltech_model_path)
model = Sequential()
for layer in loaded_model.layers[:-1]:
    model.add(layer)

model.add(Dense(num_classes, activation='softmax', name='softmax'))
```

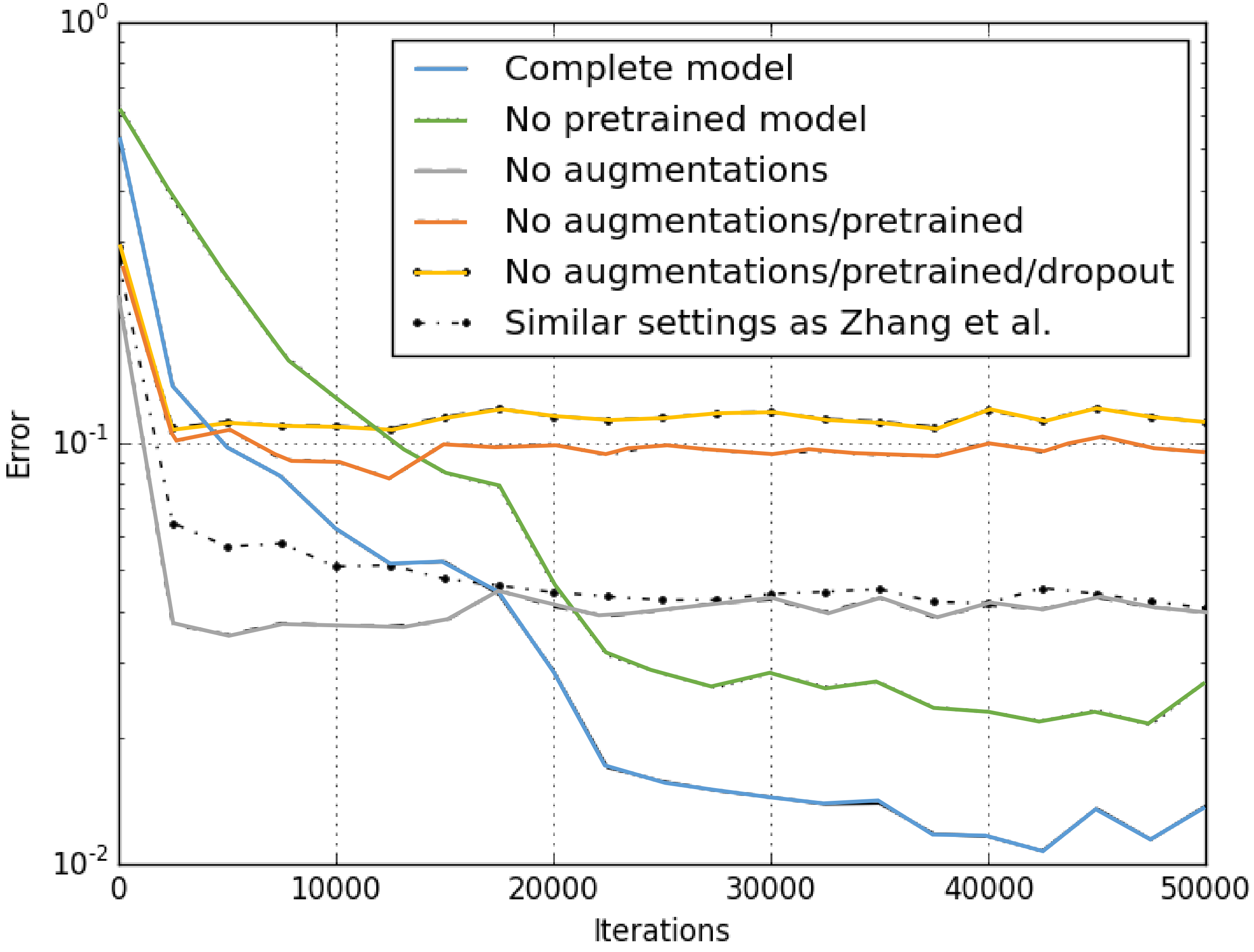
Application 1: Transfer Learning

	Augmentation		Pretraining	
Epoch	Training	Validation	Training	Validation
0	3.1%	3.1%	3.1%	3.1%
5	26.0%	29.7%	50.1%	58.8%
10	54.8%	46.9%	70.8%	68.1%
20	72.3%	63.1%	83.0%	75.0%
30	80.7%	74.7%	87.0%	80.0%
60	85.6%	82.5%	91.4%	86.3%

Application 1: Results with bigger input size

	Augmentation		Transfer Learning	
Epoch	Training	Validation	Training	Validation
0	2.3%	2.8%	3.1%	3.1%
5	35.4%	47.5%	48.5%	63.8%
10	59.4%	54.4%	69.5%	77.8%
20	72.9%	59.1%	81.6%	86.3%
30	80.9%	75.9%	86.0%	89.7%
60	90.2%	86.7%	91.9%	92.5%

Application 1: Results Overview



Observations:

- Small improvement with dropout (orange vs yellow)
- Augmentations (green vs orange) lead to flat progression, but intersect the curves without augmentation later
- Pretraining (grey vs orange and blue vs green) leads to a steep progression at first, since useful features have already been learned and the output layer can be adapted to the actual data quickly

Improving the prediction

- Data augmentation learns to predict leaves in different representations
- Idea for prediction:
  - Augment the prediction itself a few times (e. g. 10)
  - Expected result: 0, 0, 0, 0, 13, 0, 0, 17, 13, 0
  - Choose the most common label (mode) → 90% vs. 94.1% Accuracy (100 vs 300 input size)
- Similar approach: Voting
  - Train several models (as diverse as possible)
  - Mode of results

Summary

- Standard model 78.1 %
- With Dropout 79.1 %
- With Augmentation 82.5 %
- With Transfer Learning 86.3%
- With Mode of prediction 90 %

Outlook

Exercise and next lecture

Outlook: Exercise

1. CNN
   - MNIST Data
   - How to use a CNN
2. Regularization
   - Reducing the dataset to small amount of data per class (e.g. 100)
   - Implement regularization: dropout and data augmentation
   - Pretraining (Training on Fashion-MNIST, and CIFAR-100)
3. Evaluation
   - Compute the different evaluation metrics
   - Present your results (e.g. plot the confusion matrix)

Outlook: Next lecture

Image segmentation

- So far only pure classification of whole image

→Localisation? Several objects on one image? Identifying regions?

- Image segmentation
  - Networks for pixelwise classification