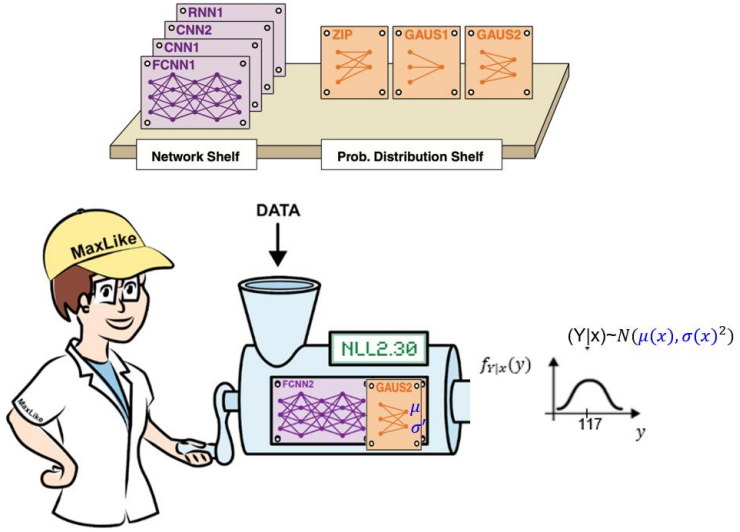
WBL Probabilistic Deep Learning:: Day 2

Preliminary version

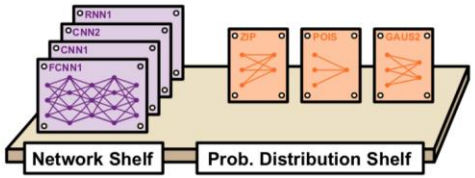
Beate Sick, Oliver Dürr

Day 2: NN training and different NN architectures

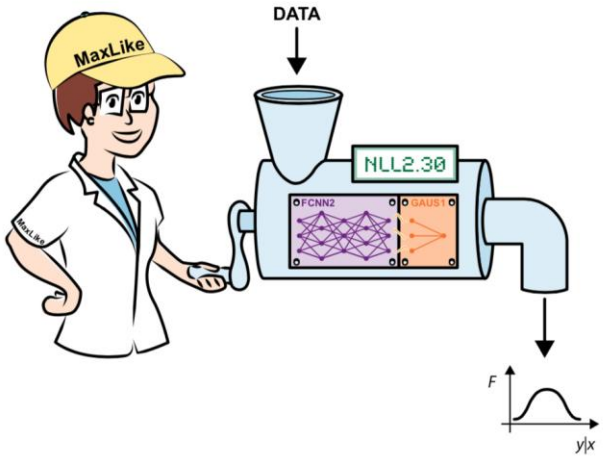


Topics of today

- Training of NN
  - Negative Log Likelihood (NLL) as loss function
  - Optimization via stochastic gradient descent (SGD) or a variant (e.g. adam)
- NN architectures
  - Fully connected NN for tabular data
  - Convolutional Neural Networks (CNN) for images
  - Next time: Transformer



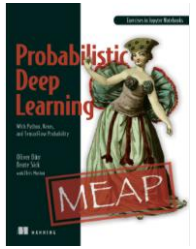
- Tricks of the trade
  - Early stopping
  - Input data normalization
  - Batchnorm
  - Dropout
  - Data augmentation



- Working with pretrained (foundation) models
  - Feature extraction followed by a RF or fcNN or ….
  - Finetuning, transfer learning

Literature

- Course website
  - https://tensorchiefs.github.io/dlwbl_eth25/



- Probabilistic Deep Learning: Our probabilistic take
  - https://www.manning.com/books/probabilistic-deep-learning

- Deep Learning with Python, Second Edition
  - https://www.manning.com/books/deep-learning-with-python-second-edition

- Keras Documentation:
  - https://keras.io/

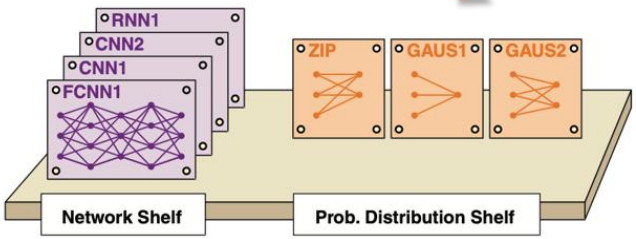
- Other Courses
  - Convolutional Neural Networks for Visual Recognition http://cs231n.stanford.edu

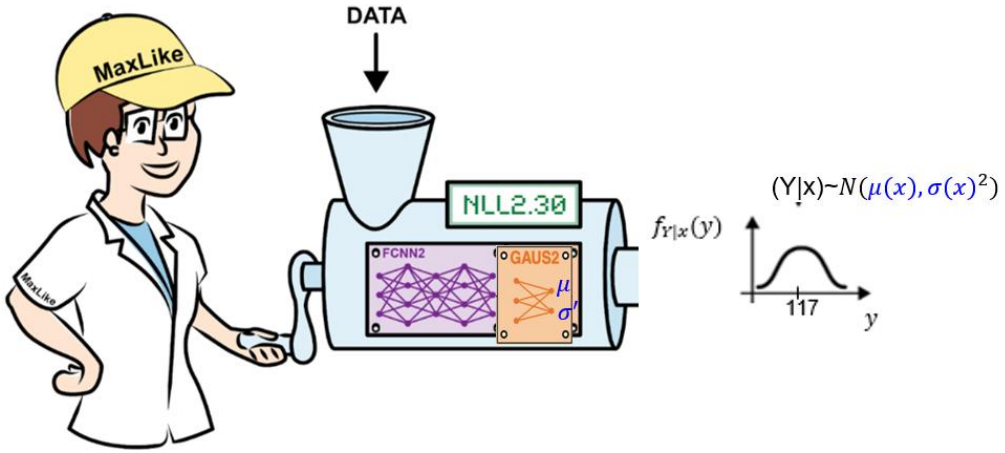
Topics (may change)

- Half day 1
  - Introduction to probabilistic DL
  - Introduction to Keras with pytorch backend
- Day 2
  - Loss function and optimization
  - Convolutional Neural Networks (CNN) for image data
  - Working with pretrained models
  - Transfer learning
  - Projects
- Day 3
  - Transformer Architectures
  - Model evaluation
  - NN-based logistic regression with tabular and image data
  - Poisson regression with NNs
  - Projects
- Half day 4
  - Project presentations
  - tba

Date
03.02.2025
10.02.2025
17.02.2025
24.02.2025

NLL is used as loss function which depends on outcome distribution



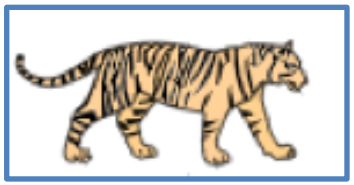


Training: Tune the weights to minimize the loss

Input $x^{(i)}$

True class $y^{(i)}$

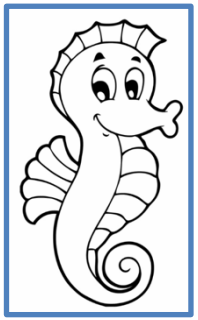
predicted class
(class with highest predicted probability)



Tiger → Lion 👎

Neural network with many weights w

Tiger → Tiger 👍



Seehorse → Seehorse 👍

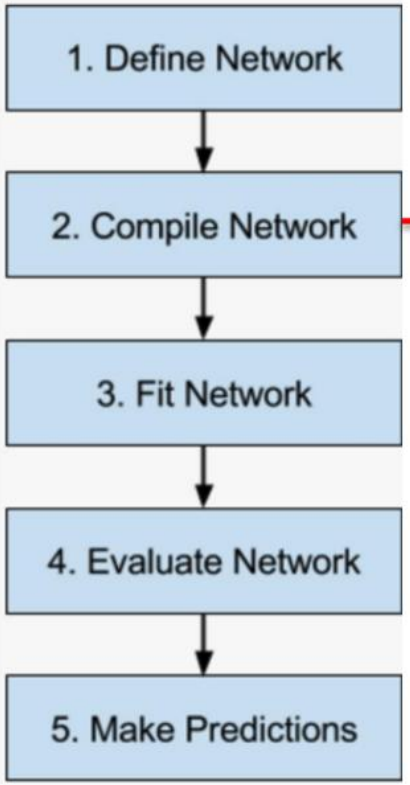
...

Trainingsprinciple:
Weights are tuned so a loss functions gets minimized.

$$\hat{w} = \underset{w}{\operatorname{argmin}}\ \operatorname{loss}\left(\left\{y^{(i)}, x^{(i)}\right\}, w\right)$$

Recall: in Keras we choose a loss function

Keras




```
model.compile(optimizer=SGD(learning_rate=0.01)
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

Loss Function to optimize
Here: Binary CE

Losses in Keras

keras.io/api/losses/

Developer guides

Code examples

Keras 3 API documentation

- Models API
- Layers API
- Callbacks API
- Ops API
- Optimizers
- Metrics
- **Losses**
  - Probabilistic losses
  - Regression losses
  - Hinge losses for "maximum-margin" classification
- Data loading
- Built-in small datasets
- Keras Applications
- Mixed precision
- Multi-device distribution
- RNG API
- Utilities

The purpose of loss functions is to compute the quantity that a model should seek to minimize during training.

Available losses

Note that all losses are available both via a class handle and via a function handle. The class handles enable you to pass configuration arguments to the constructor (e.g. `loss_fn = CategoricalCrossentropy(from_logits=True)`), and they perform reduction by default when used in a standalone way (see details below).

Probabilistic losses

- BinaryCrossentropy class
- BinaryFocalCrossentropy class
- CategoricalCrossentropy class
- CategoricalFocalCrossentropy class
- SparseCategoricalCrossentropy class
- Poisson class
- CTC class
- KLDivergence class
- binary_crossentropy function
- categorical_crossentropy function
- sparse_categorical_crossentropy function
- poisson function
- ctc function
- kl_divergence function

Regression losses

- MeanSquaredError class
- MeanAbsoluteError class
- MeanAbsolutePercentageError class
- MeanSquaredLogarithmicError class
- CosineSimilarity class

► Keras 3 API documentation / Losses / Probabilistic losses

Probabilistic losses

BinaryCrossentropy class [so

```
keras.losses.BinaryCrossentropy(
    from_logits=False,
    label_smoothing=0.0,
    axis=-1,
    reduction="sum_over_batch_size",
    name="binary_crossentropy",
    dtype=None,
)
```

► Keras 3 API documentation / Losses / Regression losses

Regression losses

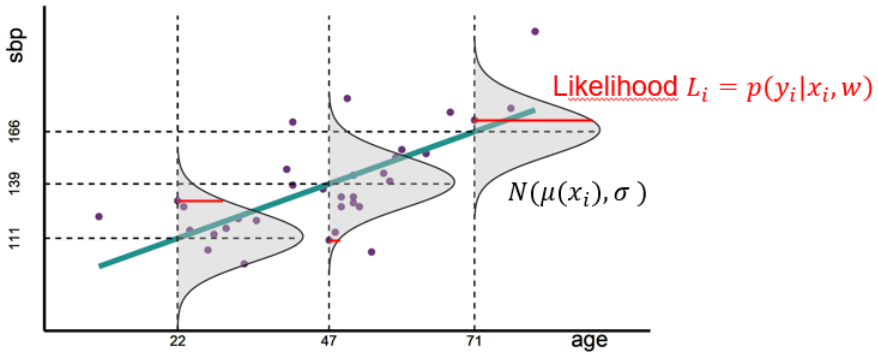
MeanSquaredError class [source]

```
keras.losses.MeanSquaredError(
    reduction="sum_over_batch_size", name="mean_squared_error", dtype=None
)
```

Which loss to use in probabilistic DL?

Negative Log Likelihood!

Recap: Maximum Likelihood (one of the most beautiful ideas in statistics)



Ronald Fisher in 1913
Also used before by Gauss, Laplace

Tune the parameters weights of the network such, that observed data (training data) is most likely under the predicted outcome distribution.

We assume iid training data → multiplication of likelihood over all data points:

$$\hat{w} = \underset{w}{\operatorname{argmax}} \prod_{i=1}^{N} L_i = \underset{w}{\operatorname{argmax}} \prod_{i=1}^{N} p(y_i|x_i, w)$$

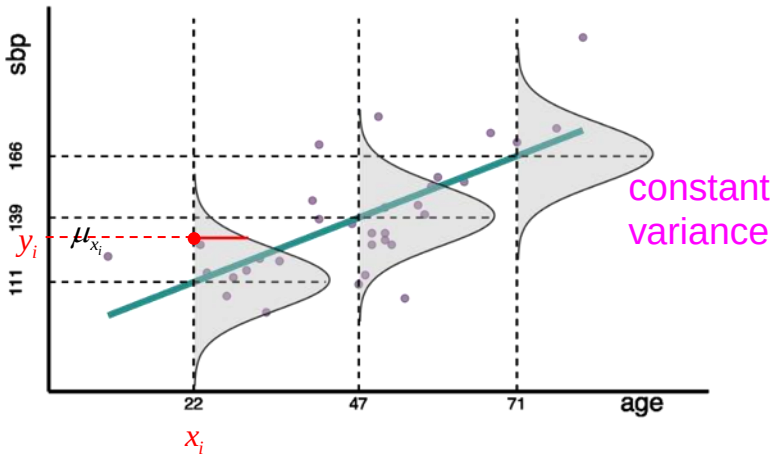
Practically: Use Negative Log-Likelihood NLL as loss and minimize NLL

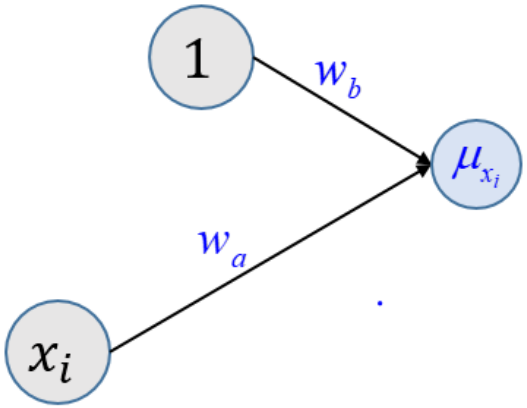
take log; minimize after multiplication with -1, note that NLL in DL is usually the mean-negative log-likelihodd

$$\hat{w} = \underset{w}{\operatorname{argmin}} \operatorname{NLL}\left(y^{(i)}, x^{(i)}, w\right) = \underset{w}{\operatorname{argmin}} \frac{1}{N} \sum_{i=i}^{N} -\log(p(y_i|x_i, w))$$

Fit "linear regression NN" via Maximum likelihood principle

$$Y_{X_i} \sim \mathrm{N}(\mu_{x_i} = w_a \cdot x_i + w_b \ , \sigma^2)$$





ML-principle:

$$\mathbf{w}_{\mathrm{ML}} = \underset{w}{\operatorname{argmax}} \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi\sigma^2}} \, \mathrm{e}^{-\frac{\left(y_i - \mu_{x_i}\right)^2}{2\sigma^2}}$$

$$= \underset{w}{\operatorname{argmin}} \sum_{i=1}^{n} -\log\left(\frac{1}{\sqrt{2\pi\sigma^2}}\right) + \frac{\left(y_i - \mu_{x_i}\right)^2}{2\sigma^2}$$

Negative Log-Likelihood (NLL)

$$(\hat{w}_a, \hat{w}_b)_{\mathrm{ML}} = \underset{w_a, w_b}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} \left(y_i - (w_a \cdot x_i + w_b)\right)^2$$

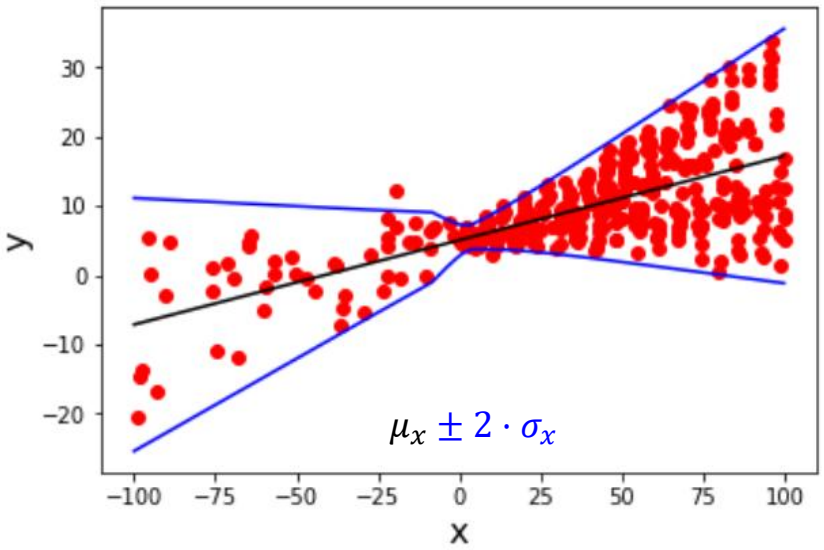
Minimize MSE loss

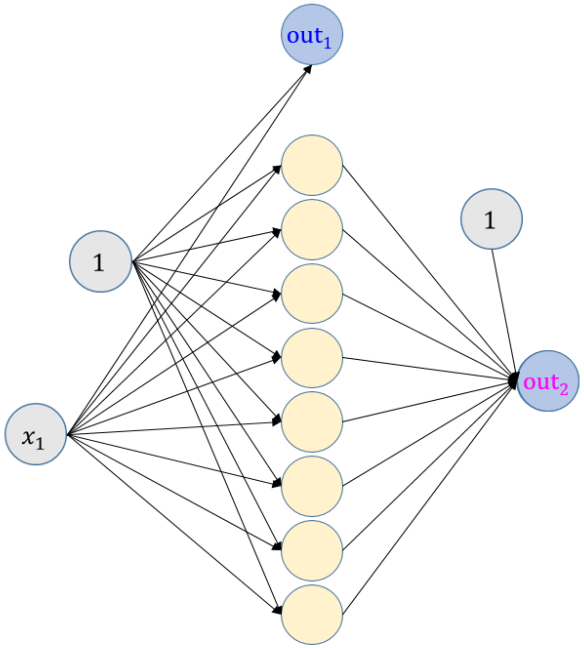
$\hat{w}_a$ $\hat{w}_b$

Minimizing NLL loss $\overset{\sigma \ const}{\Longrightarrow}$ Minimizing MSE loss

Linear regression with flexible non-constant variance

$$Y_{X_i} = (Y|X_i) \sim N(\mu_{x_i}, \sigma_x^2)$$





$$\mu_{x_i} = \text{out}_{1_i}$$

$$\sigma_{x_i} = e^{\text{out}_{2_i}}$$

Minimize the mean negative log-likelihood (NLL) on train data:

$$NLL(w) = \frac{1}{n} \sum_{train-data} -log(p(y_i|x_i, w)$$

gradient descent with NLL loss

$$\hat{\mathbf{w}}_{\text{ML}} = \underset{w}{\text{argmin}} \sum_{i=1}^{n} -\log\left(\frac{1}{\sqrt{2\pi\sigma_{x_i}^2}}\right) + \frac{\left(y_i - \mu_{x_i}\right)^2}{2\sigma_{x_i}^2}$$

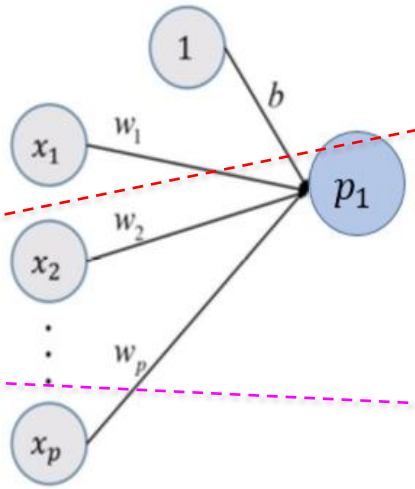
Note: we do not need to know the "ground truth for σ" – the likelihood does the job!

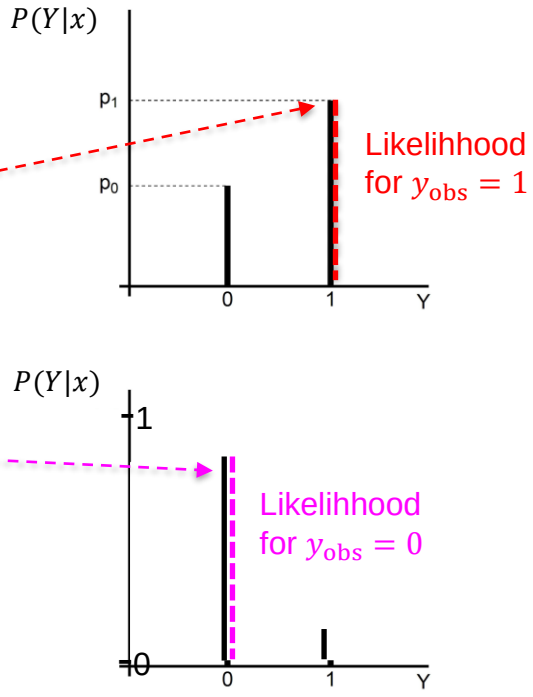
Maximum likelihood principle in logistic regression

probabilistic prediction

Training data set

x1	…	xp	y
0.4	…	-1.3	0
1.1	…	0.2	1
:	:	:	:
0.6	…	-0.9	0





Given (=conditioned on) the input featurs x of an observation i, a well trained NN

- should predict large $p_1 = P(Y = 1|x)$ if the observed class is $y_i = 1$
- should predict small p_1 hence large $p_0 = P(Y = 0|x)$ if observed is $y_i = 0$

→ The likelihood (for the observed outcome) or LogLikelihood should be large

$$\text{LogLikelihood} = \sum_{i=1}^{n} [y_i \log(p_{1i}) + (1 - y_i) \log(1 - p_{1i})]$$

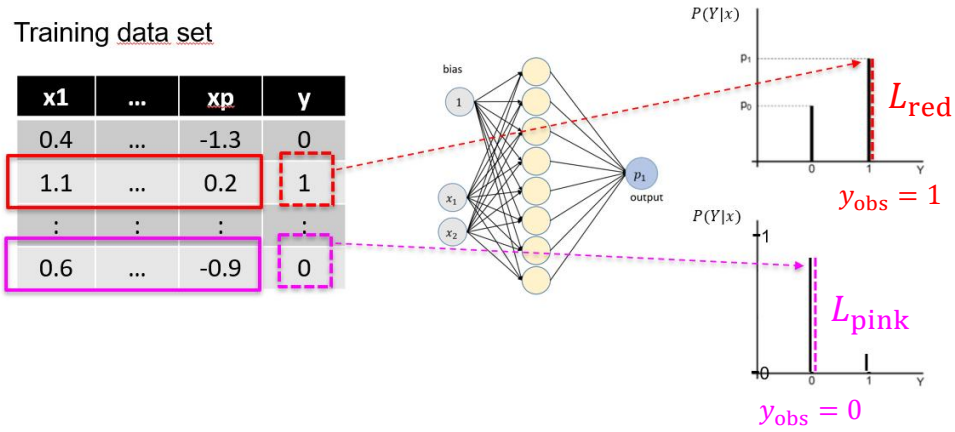
Notation trick in statistics – it selects correct log-probability since $y_i \in \{0,1\}$

Fitting a "logistic regression NN"

The Likelihood of an observation i is the likelihood (probability), that the predicted probability distribution $P(Y|x_i)$ assigns to the observed outcome y_i.

Note: the predicted $P(Y \mid x_i)$ and the corresponding likelihood for the observed y_i depends on the data-point (x_i, y_i) and the model parameter values

→ The higher the likelihood, the better is the model prediction $P(Y|x)$



L_{red} is likelihood of red observation with $y_{obs} = 1$

L_{pink} is likelihood of pink observation with $y_{obs} = 0$

Maximum likelihood principle:

Statistical models are fit to maximize the average LogLikelihood

$$L = \frac{1}{N}\sum L_i = \frac{1}{N}\sum [y_i \log(p_{1i}) + (1 - y_i)\log(1 - p_{1i})]$$

we often use the simplified notation average logLik : $\mathbf{L = \frac{1}{N}\sum log(p_i)}$

Predicted probability for the observed outcome y_{obs}

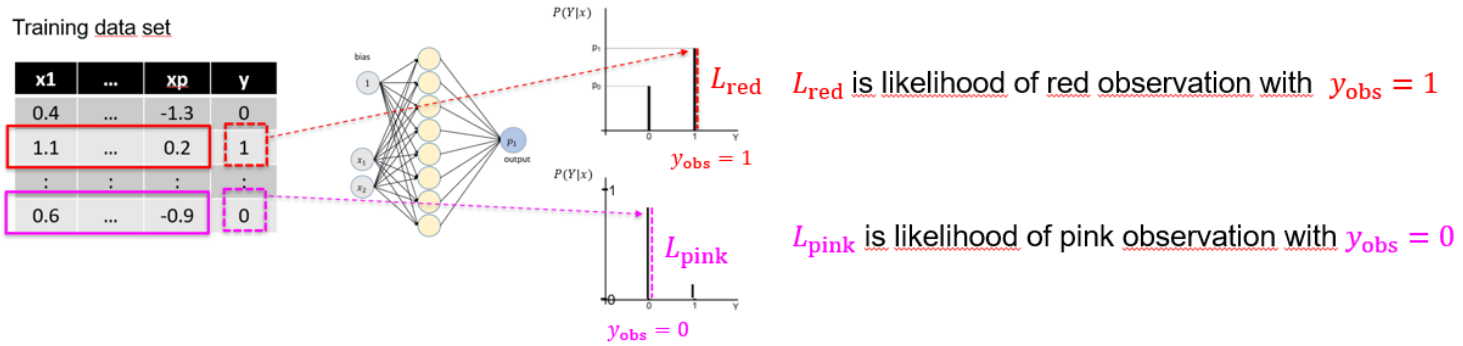
NLL Loss for probabilistic binary classification

In DL we aim to minimize a loss function $L(\text{data}, w)$ which depends on the weights

→ Instead of maximizing the average LogLikelihood

we minimize the averaged Negative LogLikelihood (NLL):

$$\textbf{loss} = \textbf{NLL} = -\frac{1}{N}\sum \log(L_i) = -\frac{1}{N}\sum \log(p_i)$$



The best possible value of the NLL contribution of an observation i is $-\log(\mathrm{L_i}) = -\log(1) = 0$

The worst possible value of the NLL contribution of an observation i $-\log(\mathrm{L_i}) = -\log(0) = \infty$

Note: In Keras the NLL for binary outcome is called 'binary_crossentropy',
if we do probabilistc binary classification with 1 output node with sigmoid activation.
If we would use 2 output nodes and softmax than the NLL is called 'categorical_crossentropy'

Define the NLL loss for a probabilistic model

- Some keras losses correspond to NLL, e.g.
  - Linear Regression with constant variance
    - Mean Squared Error (keras.losses.mse)
  - Multiclass classification
    - Categorical_cross_entropy (keras.losses.categorical_crossentropy)

- For a given outcome distribution family, you can build the NLL loss
  1. Define a model with as many outputs as the number of parameters of the outcome distribution family
  2. Link the outputs of the NN to the parameters of the outcome distribution
  3. Use negative log likelihood as loss function

See https://github.com/tensorchiefs/dlwbl_eth25/blob/master/notebooks/01_hello_prob_dl.ipynb

1. Define a model with as many outputs as the number of parameters of the distribution

```
# Define the model
inputs = Input(shape=(1,))
hidden = layers.Dense(10, activation="relu")(inputs)
# ... more hidden layers ...
outputs = layers.Dense(2)(hidden)  # <--- Outputs mean and log(sd)
model = Model(inputs=inputs, outputs=outputs)
```

[8] ✓ 0.0s Python

2. Link the outputs of the model to the parameters of the distribution

```
# Wrapper function to convert model output to a PyTorch Normal distribution
@staticmethod
def output_to_gaussian_distribution(output):
    mean = output[:, 0:1]   # Gets the first column while keeping dimensions (like output[, 1, drop=FALSE] in R)
    log_sd = output[:, 1:2] # Gets the second column while keeping dimensions
    scale = torch.exp(log_sd) # Inverse link function to ensure positive scale
    return Normal(loc=mean, scale=scale)
```

[7] ✓ 0.0s Python

3. Define NLL

```
# Custom Negative Log-Likelihood Loss
def negative_log_likelihood(y_true, output):
    dist = output_to_gaussian_distribution(output)
    return -dist.log_prob(torch.tensor(y_true)).mean()

# Compile the model
model.compile(optimizer="adam", loss=negative_log_likelihood)
```

[10] ✓ 0.0s Python

Exercise Define a custom loss function

Notebook: 02

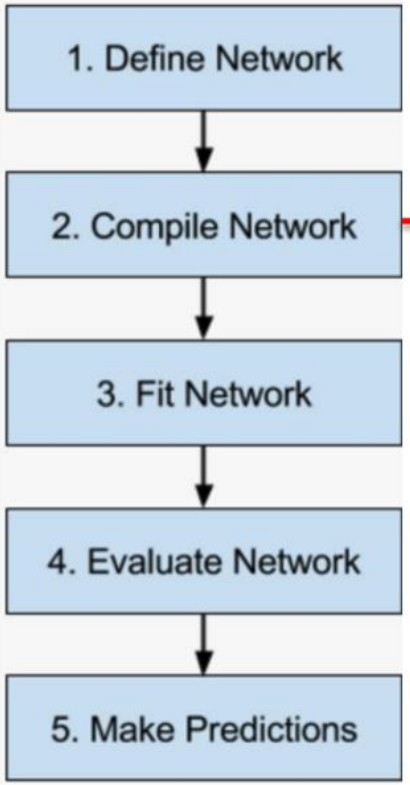
https://github.com/tensorchiefs/dlwbl_eth25/blob/master/notebooks/02_custom_loss.ipynb

Fit a NN via Stochastic Gradient Descent

Recall: in Keras we choose a loss function

Keras

Which optimizer should be used?
Here Stochastic Gradient Descent

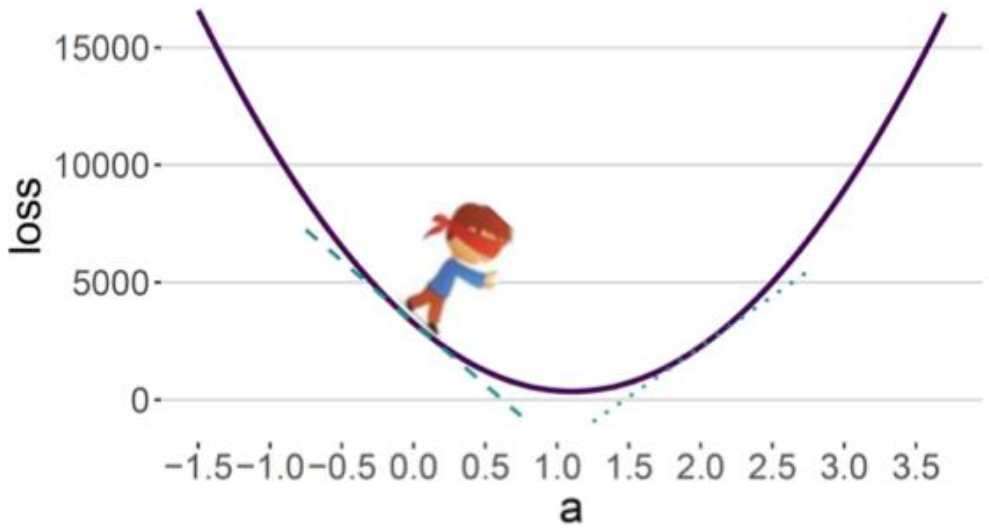



```
model.compile(optimizer=SGD(learning_rate=0.01)
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

Loss Function to optimize
Here: Binary CE

Idea of gradient descent

- Shown loss function for a single parameter a

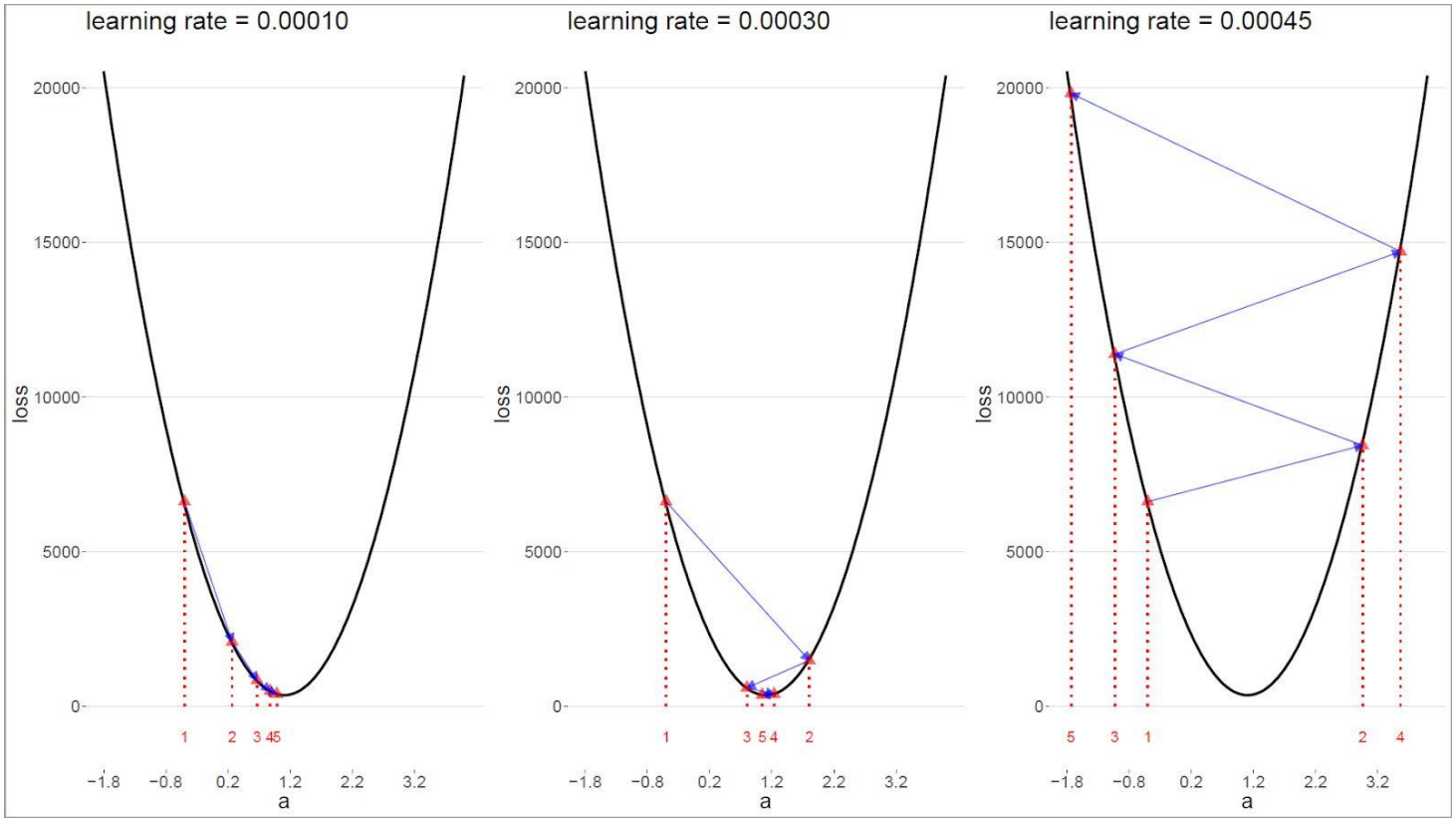


- Take a large step if slope is steep (you are away from minimum)
- Slope of loss function is given by gradient of the loss w.r.t. a
- Iterative update of the parameter a

$$a^{(t)} = a^{(t-1)} - \epsilon \left. \frac{\partial L(a)}{\partial a} \right|_{a=a^{(t-1)}}$$

learning rate

The learning rate is a very important parameter for DL



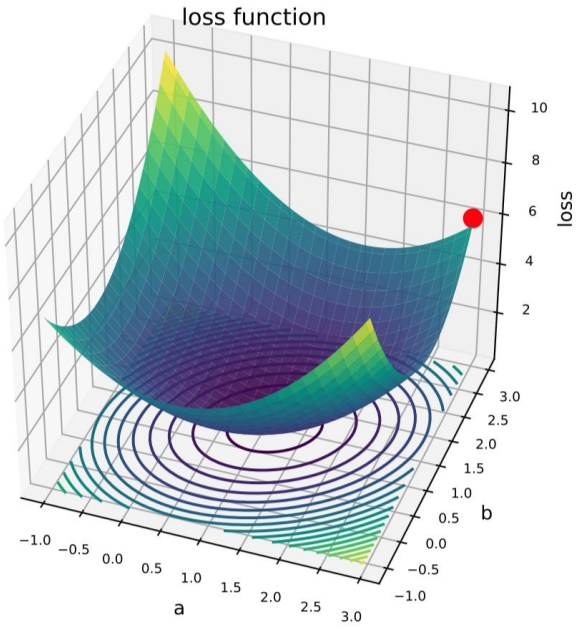
good – little slow

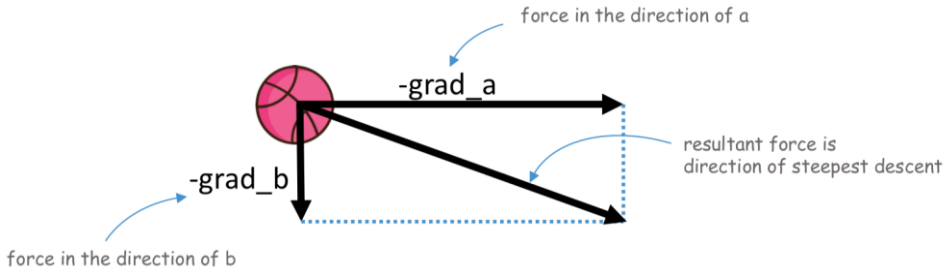
good

learning rate too high

If the loss diverges to infinity: Don’t panic, lower the learning rate!

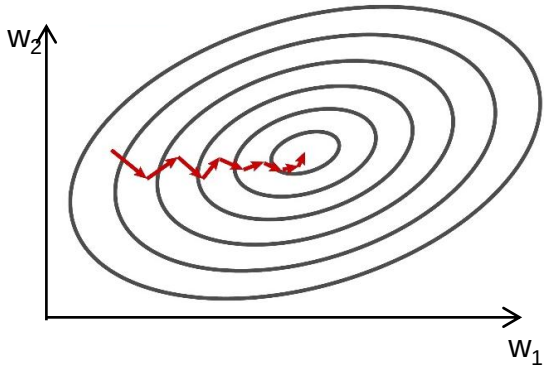
In two dimensions





Gradient is perpendicular to contour lines

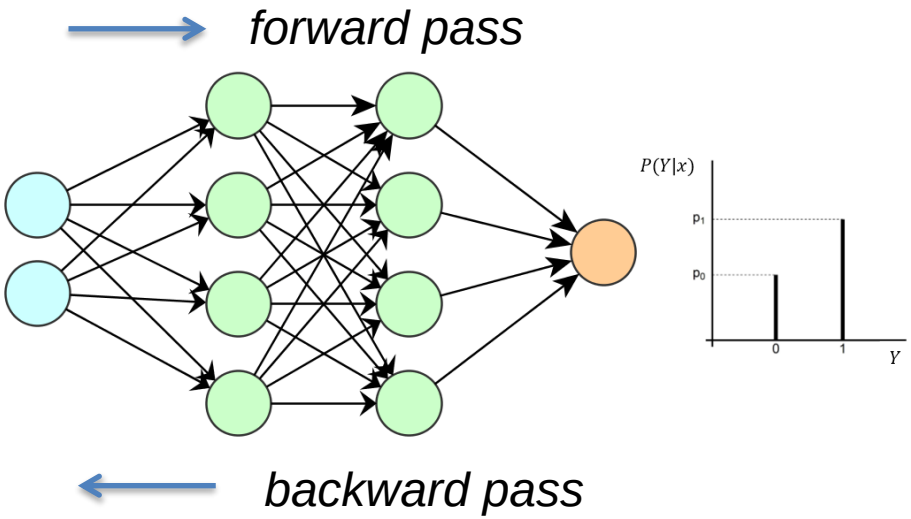
$$w_i^{(t)} = w_i^{(t-1)} - \epsilon \left. \frac{\partial L(\mathbf{w})}{\partial w_i} \right|_{w_i = w_i^{(t-1)}}$$



Remark:
DL frameworks can do the differntiation see notebook.
In DL the Loss is computed as mean NLL over all examples in a mini-batch, then weight update is performed
→ Since minibatches hold random part of all data, we talk about Stochastic Gradient Descent (SGD).
→ 'SG'D is the basic of most optimizers, but usually 'adam', an improved updating procedure is used in DL. See https://keras.io/api/optimizers/ for an overview.

Backpropagation

- We efficiently train the weights in a NN via forward and backward pass
  - Forward Pass propagate training example through network
    - Predicts as output $P(Y|x_i)$ for each input x_i in the batch given the NN weights w
      → With $P(Y|x_i)$ and the observed y_i we compute the loss $L = \left(\text{NLL} = -\frac{1}{N}\sum log(L_i)\right)$
  - Backward pass propagate gradients through network
    - Via chain rule all gradients $\frac{\partial L(\mathbf{w})}{\partial w_k}$ are determined
      → update the weights $w_i{}^{(t)} = w_i{}^{(t-1)} - \epsilon \frac{\partial L(\mathbf{w})}{\partial w_i}\Big|_{w_i = w_i{}^{(t-1)}}$



Typical Training Curve: ReLU is used for fast training

Loss curves for NN with ReLu or sigmoid

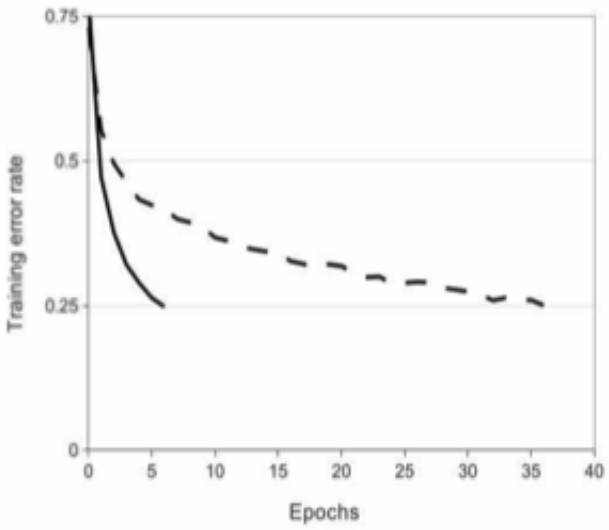
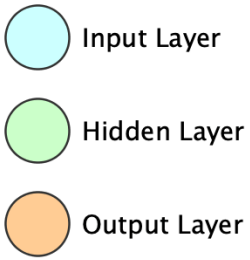
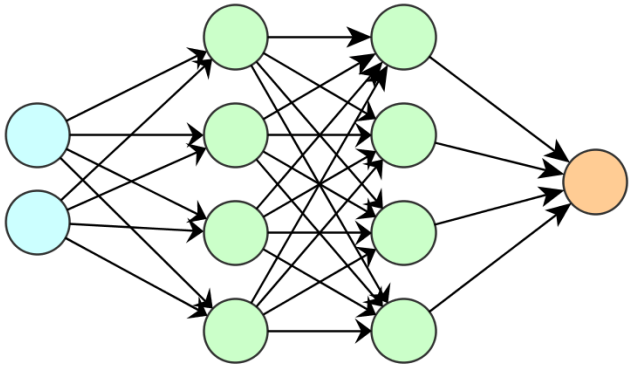
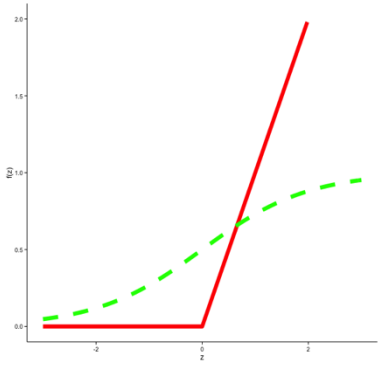


Figure 1: A four-layer convolutional neural network with ReLUs **(solid line)** reaches a 25% training error rate on CIFAR-10 six times faster than an equivalent network with tanh neurons

Source:
Alexnet
Krizhevsky et al 2012







Activation functions
Green: sigmoid.
Red: ReLU

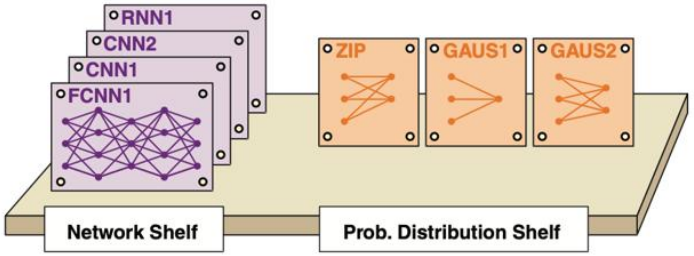
Remark:
In deep NNs ReLu is preferred since training is faster, in very shallow NNs sigmoids can be better to achieve smoothness.

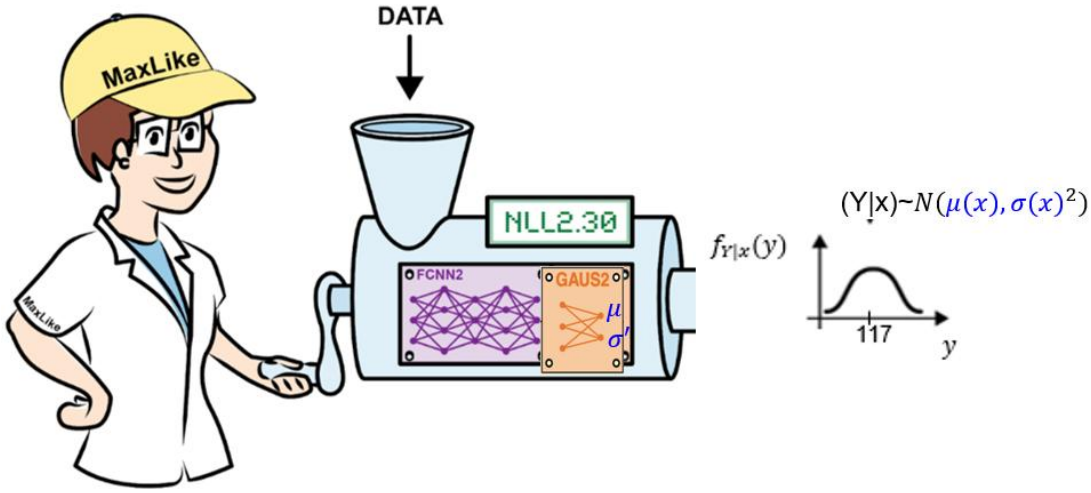
Epochs: ”each training examples is used once”

The miracle of gradient descent in DL

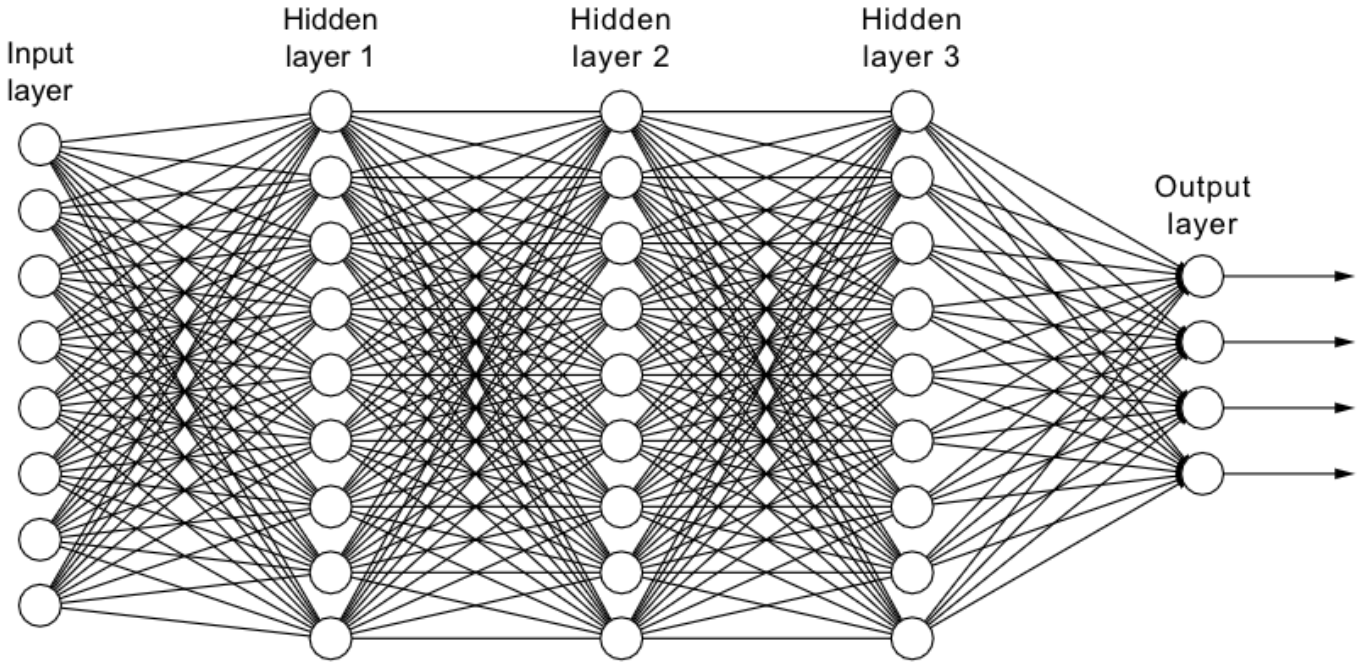
Loss surface in DL (is not convex) but SGD magically also works for non-convex problems.

NN architectures



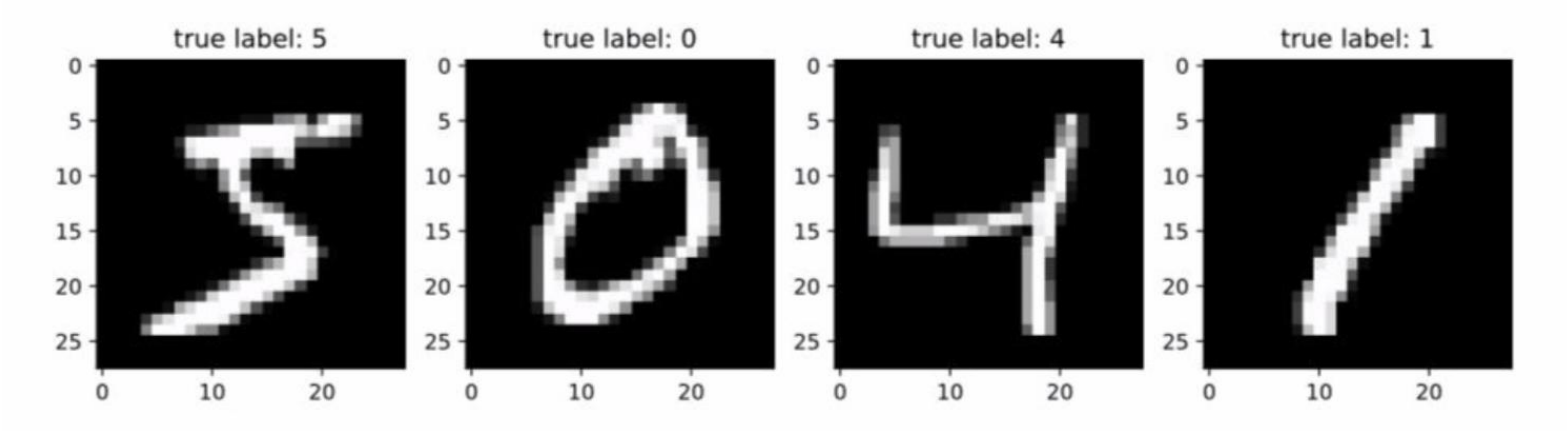


A fully connected NN

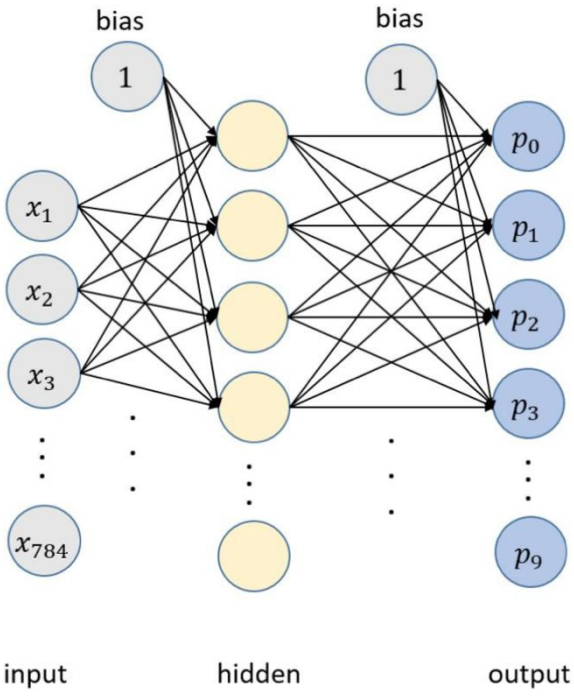


Probabilistic classification

- Usually in DL the model predicts a probability for each possible class
- Example:
  - Banknote from exercise – classes: “real” or “fake”
  - Typical example Number from hand-written digit – classes: 0, 1, …, 9



Classification: Softmax Activation



$p_o, p_1 \ldots p_9$ are probabilities for the classes 0 to 9.

Activation of last layer z_i incomming

$$p_i = \frac{e^{z_i}}{\sum_{j=0}^{9} e^{z_j}}$$

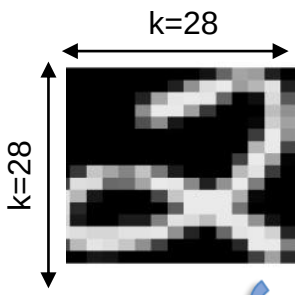
Makes outcome positive

Ensures that pi's sum up to one

This activation is called softmax

Figure 2.12: A fully connected NN with 2 hidden layers. For the MNIST example, the input layer has 784 values for the 28 x 28 pixels and the output layer out of 10 nodes for the 10 classes.

NLL for multi-class = cross-entropy

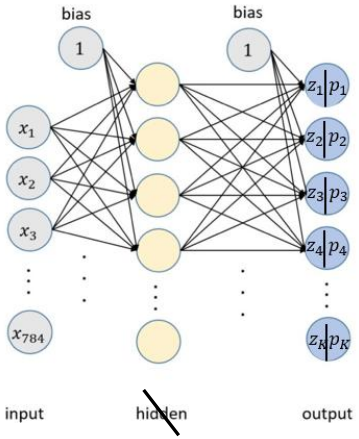


Flatten image$_i$

Input x_i

24.0
−5.1
12.2
89.9
:
3.2

$\mathbf{x}_{(1,k^2)} \cdot \mathbf{W}_{(k^2,10)} + \mathbf{b}_{(1,10)} = \mathbf{z}_{(1,10)}$



Score or logit z_i

-3.89
-3.18
-0.80
-2.20
-2.44
-1.05
-4.60
-3.48
-2.09
-2.44

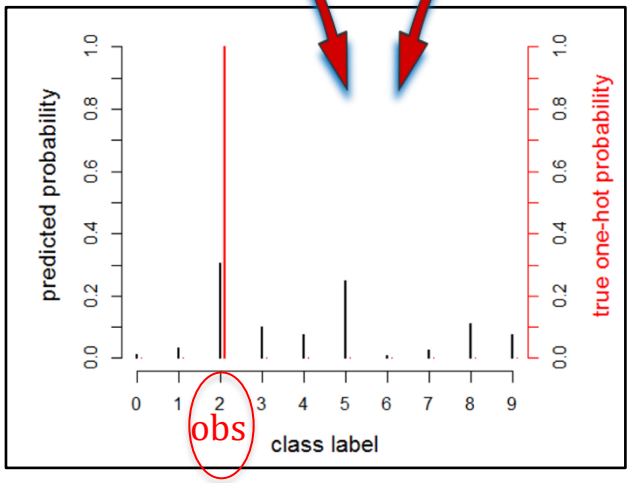
$\hat{p}_k = \frac{e^{z_k}}{\sum_j e^{zj}}$

probability p_i

0.02
0.04
0.31
0.10
0.08
0.26
0.01
0.03
0.11
0.08

1-hot labels y_i

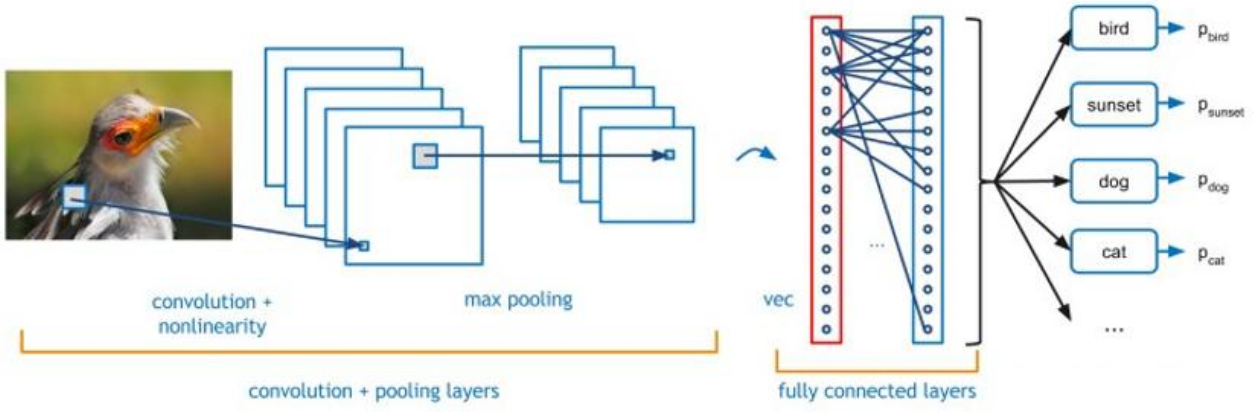
	index
0	0
0	1
1	2
0	3
0	4
0	5
0	6
0	7
0	8
0	9



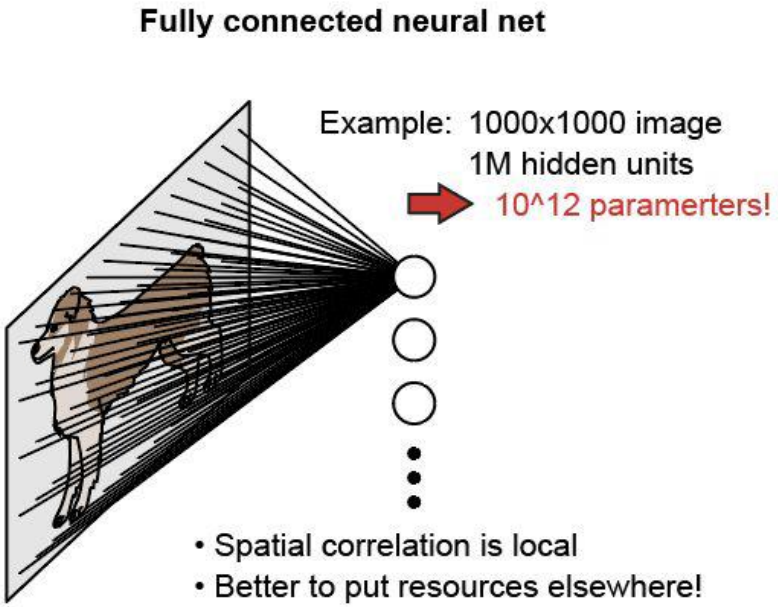
$$\text{loss} = \text{NLL} = \frac{1}{N_{\text{batch}}} \sum_{i=1}^{N_{\text{batch}}} l_i \quad \text{, with} \quad l_i = -\sum_{k=1}^{10} {}^{obs}y_{ik} \cdot \log(p_{ik})$$

Loss = NLL= cross-entropy ($-\sum p_i^{\text{obs}} \log p_i^{\text{pred}}$) averaged over all images in mini-batch

Convolutional Neural Networks for image data



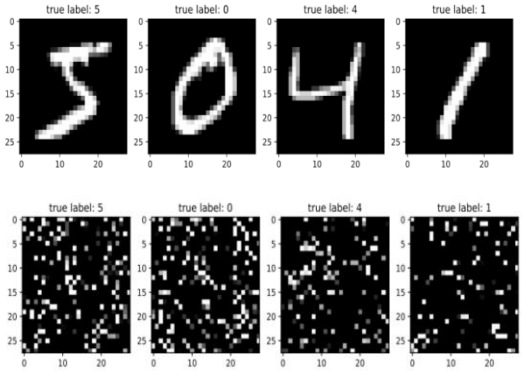
Is a fully connected NN a good architecture for images?



$$z = b + \sum_i x_i w_i$$

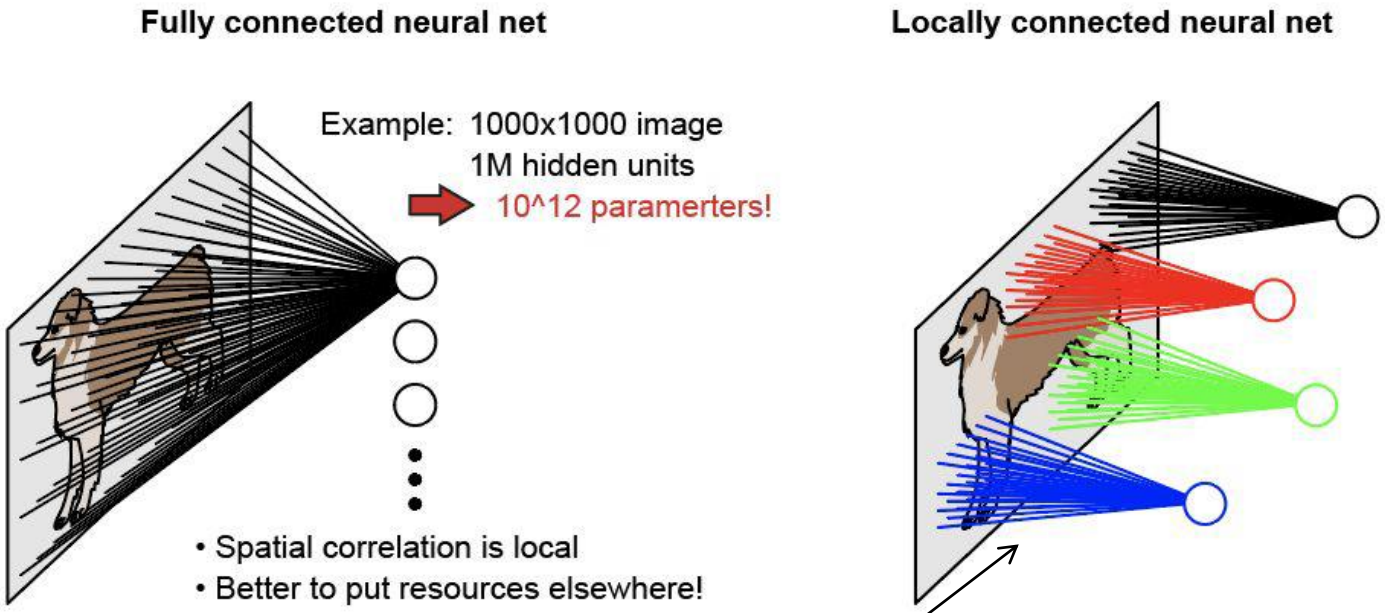
A fcNN does not take into account the spacial relationships of the pixels.

If you reshuffle the pixels of in all images in the same manner, the images look weired but the fcNN trains as with original images



04_fcnn_mnist_shuffled.ipynb

Convolution extracts local information using few weights



Shared weights:

by using the same weights for each patch of the image we need much less parameters than in the fully connected NN and get from each patch the same kind of local feature information such as the presence of a edge.

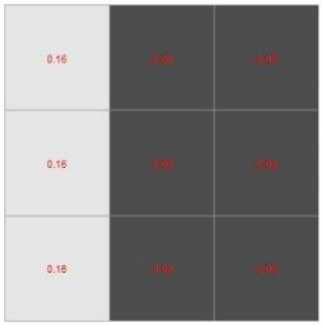
Convolutional networks use neighborhood information and replicated local feature extraction

In a locally connected network the calculation rule

$$z = b + \sum_i x_i w_i$$

Pixel values in a small image patch are element-wise multiplied with weights of a small filter/kernel:

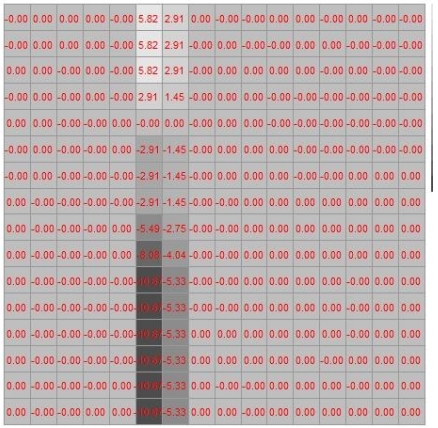
w_1	w_2	w_3
w_4	w_5	w_6
w_7	w_8	w_9



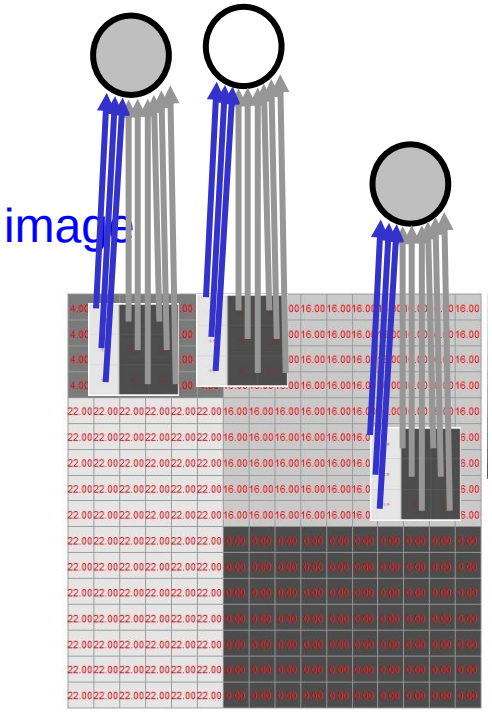
The filter is applied at each position of the image and it can be shown that the result is maximal if the image pattern corresponds to the weight pattern.

The results form again an image called feature map (=activation map) which shows at which position the feature is present.

feature/activation map



image



Exercise: Do one convolution step by hand

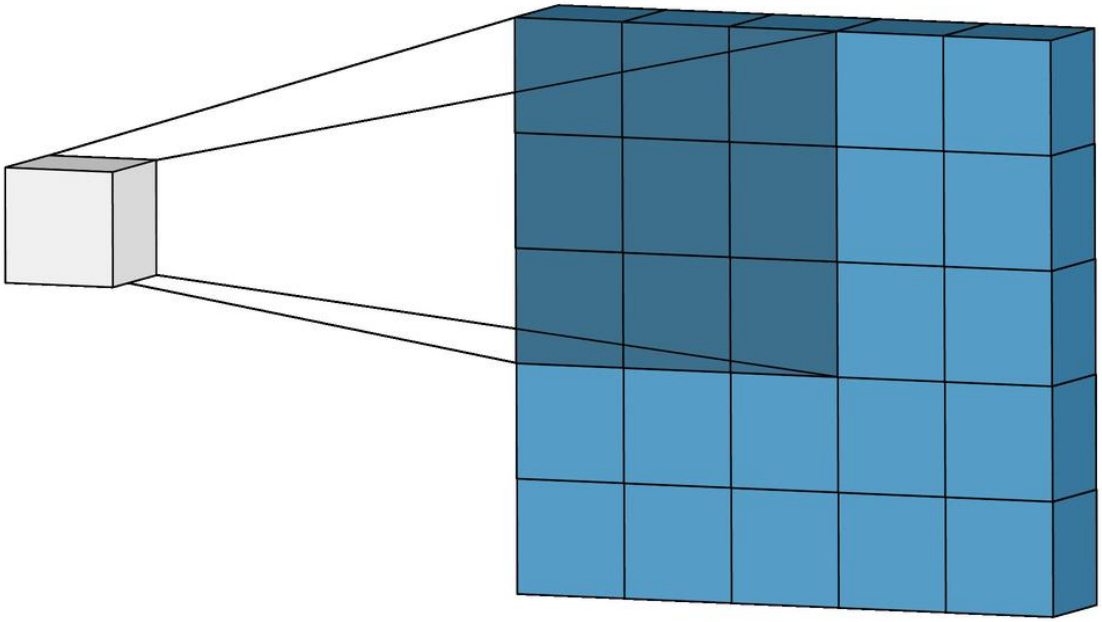
The kernel is 3x3 and is applied at each valid positon
– how large is the resulting activation map?

The small numbers in the shaded region are the kernel weights.
Determine the position and the value within the resulting activation map.

3	3	2	1	0
0_0	0_1	1_2	3	1
3_2	1_2	2_0	2	3
2_0	0_1	0_2	2	2
2	0	0	0	1

36

Applying the same 3x3 kernel at each image position



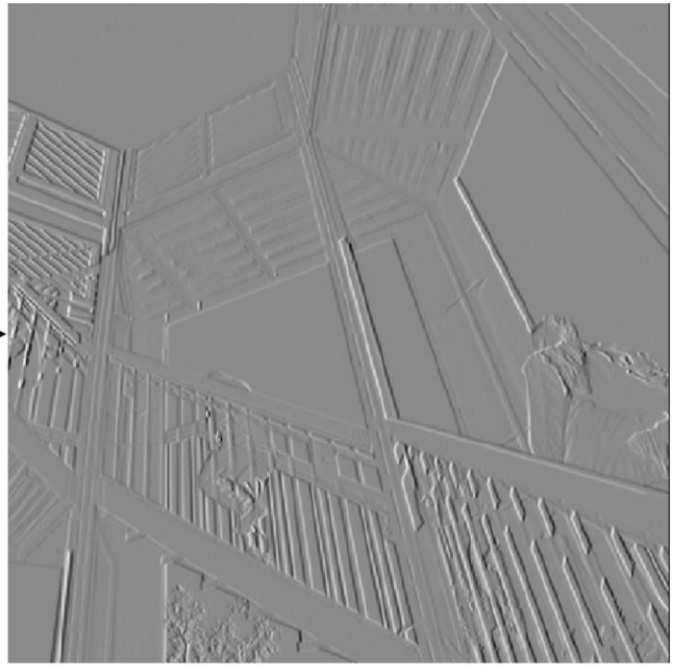
Applying the 3x3 kernel on a certain position of the image yields one pixel within the activation map where the position corresponds to the center of the image patch on which the kernel is applied.

Taken from the great website on convolution: https://towardsdatascience.com/intuitively-understanding-convolutions-for-deep-learning-1f6f42faee1

Example of designed Kernel / Filter

$$\begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix}$$

Horizontal Sobel kernel



Applying a vertical edge detector kernel

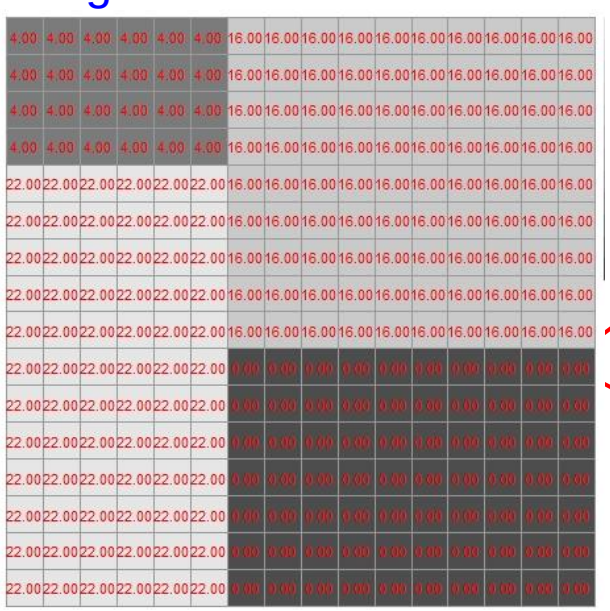
Taken from the great website on convolution: https://towardsdatascience.com/intuitively-understanding-convolutions-for-deep-learning-1f6f42faee1

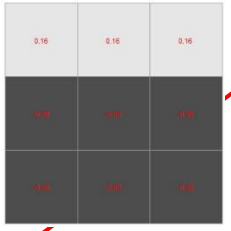
Convolutional networks use neighborhood information and replicated local feature extraction

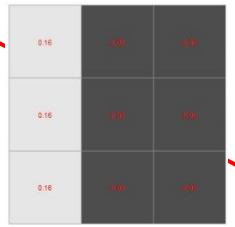
filtering = convolution

kernel 1

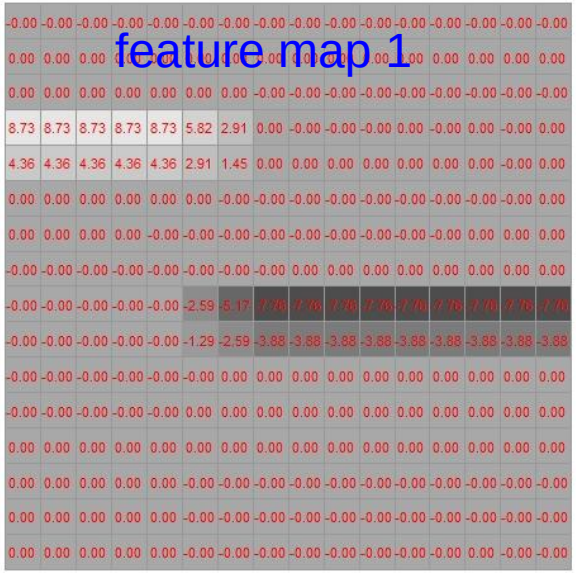
image







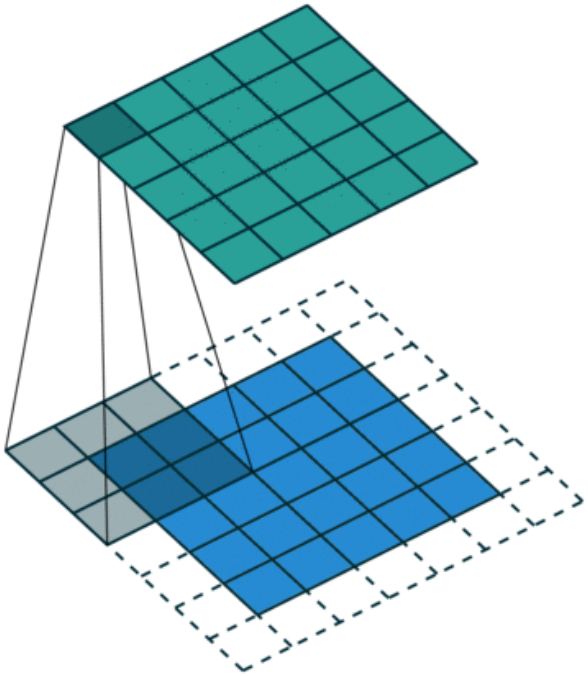
feature map 1



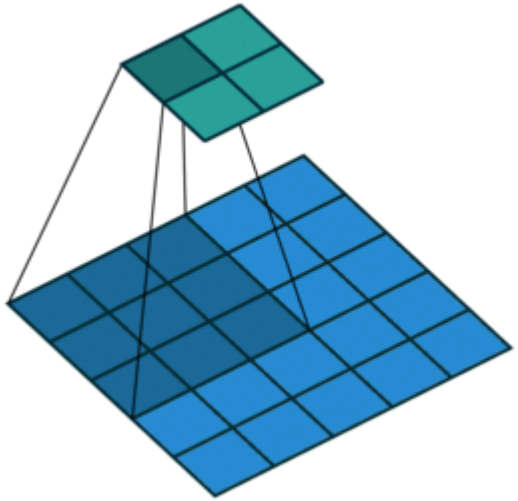
feature map 2

The weights of each filter are randomly initiated and then adapted during the training.

CNN Ingredient I: Convolution



Zero-padding to achieve
same size of feature and input

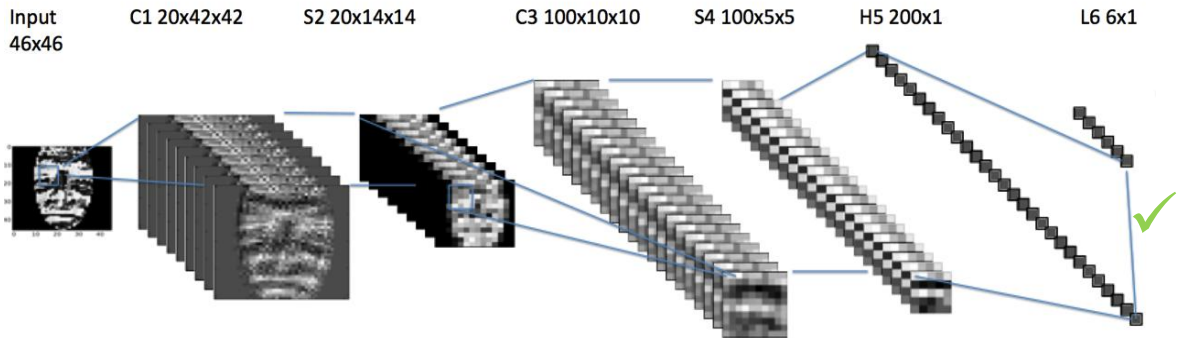


no padding to only use
valid input information

The *same* weights are used at each position of the input image.

Taken from the great website on convolution: https://towardsdatascience.com/intuitively-understanding-convolutions-for-deep-learning-1f6f42faee1

CNN ingredient II: Maxpooling Building Blocks reduce size



conv1 maxpool conv2 maxpool flatten dense

4x4

12	20	30	0
8	12	2	0
34	70	37	4
112	100	25	12

→

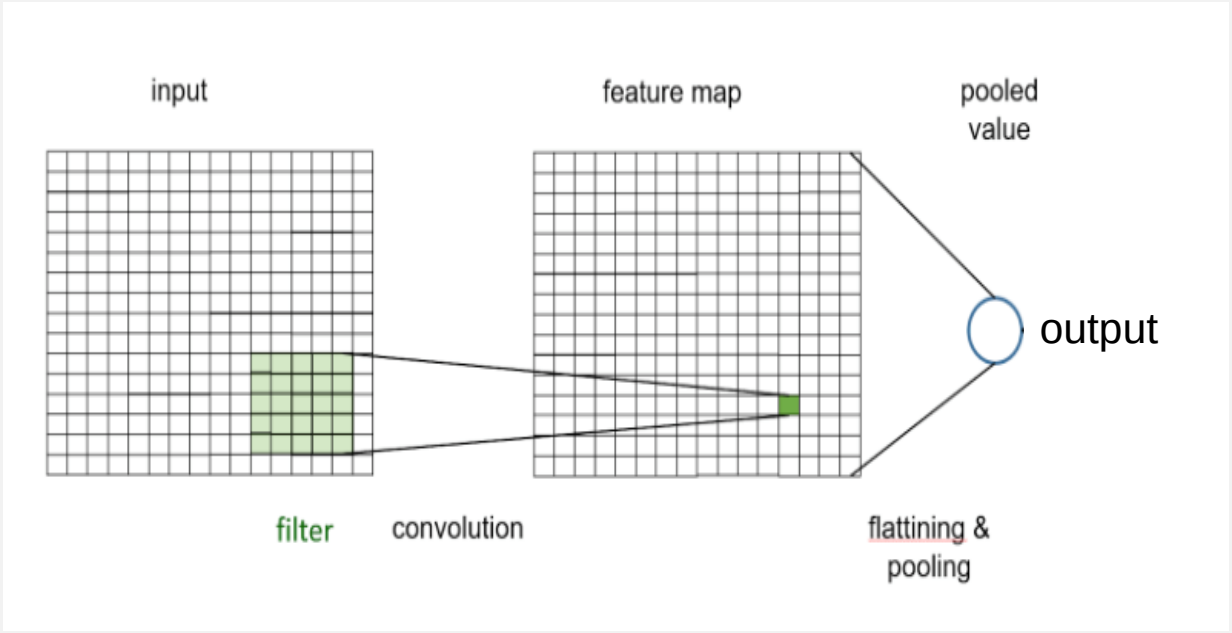
2x2

20	30
112	37

Simply join e.g. 2x2 adjacent pixels in one by taking the max.
→ less weights in model
→ Less train data needed
→ increased performance

Hinton: „The pooling operation used in convolutional neural networks is a big mistake and the fact that it works so well is a disaster“

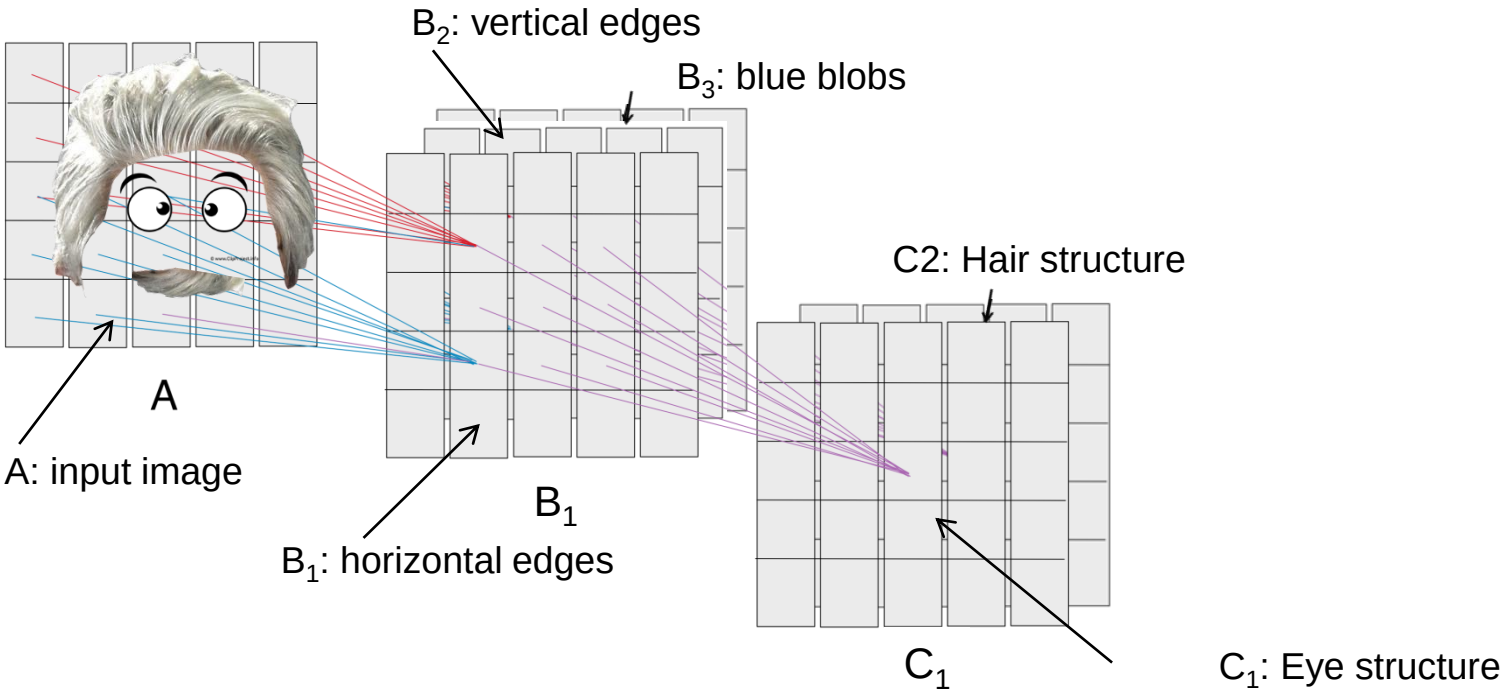
Exercise: Artstyle Lover



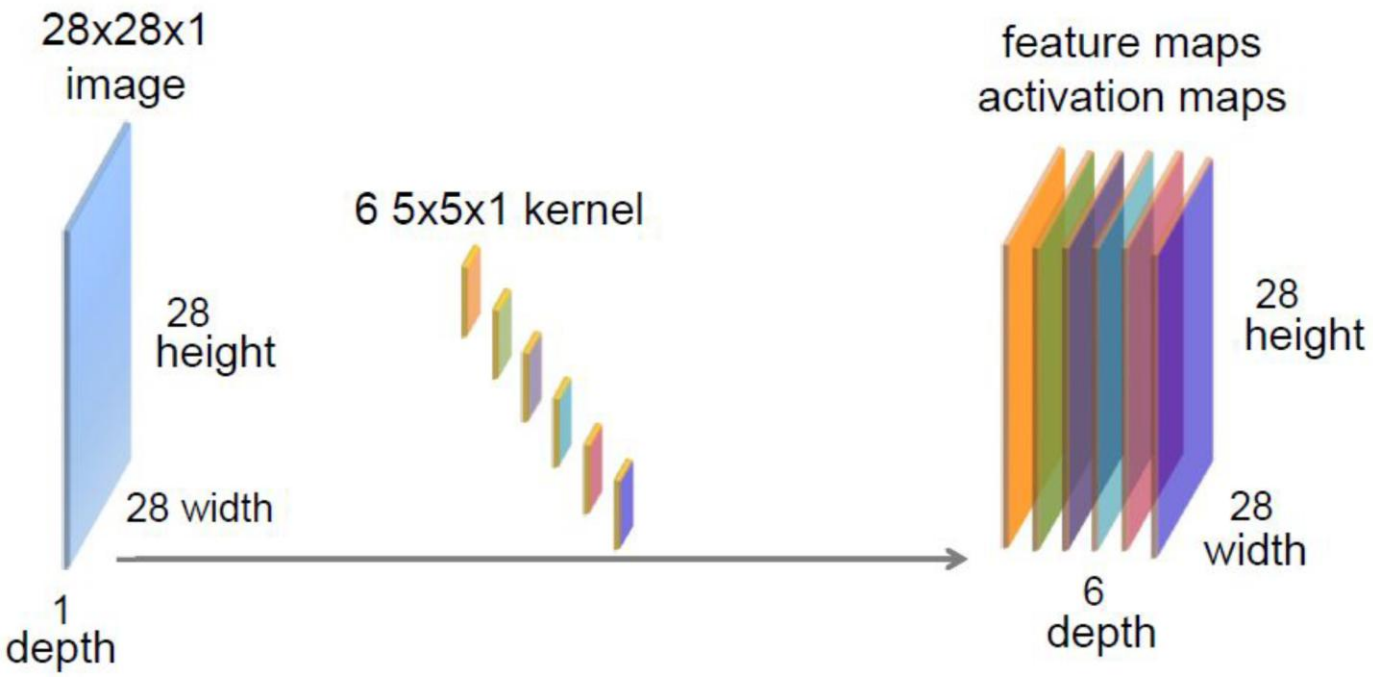
Open NB in: https://github.com/tensorchiefs/dlwbl_eth25/blob/master/notebooks/02_cnn_edge_lover.ipynb

43

More then one kernel (Motivatoion)



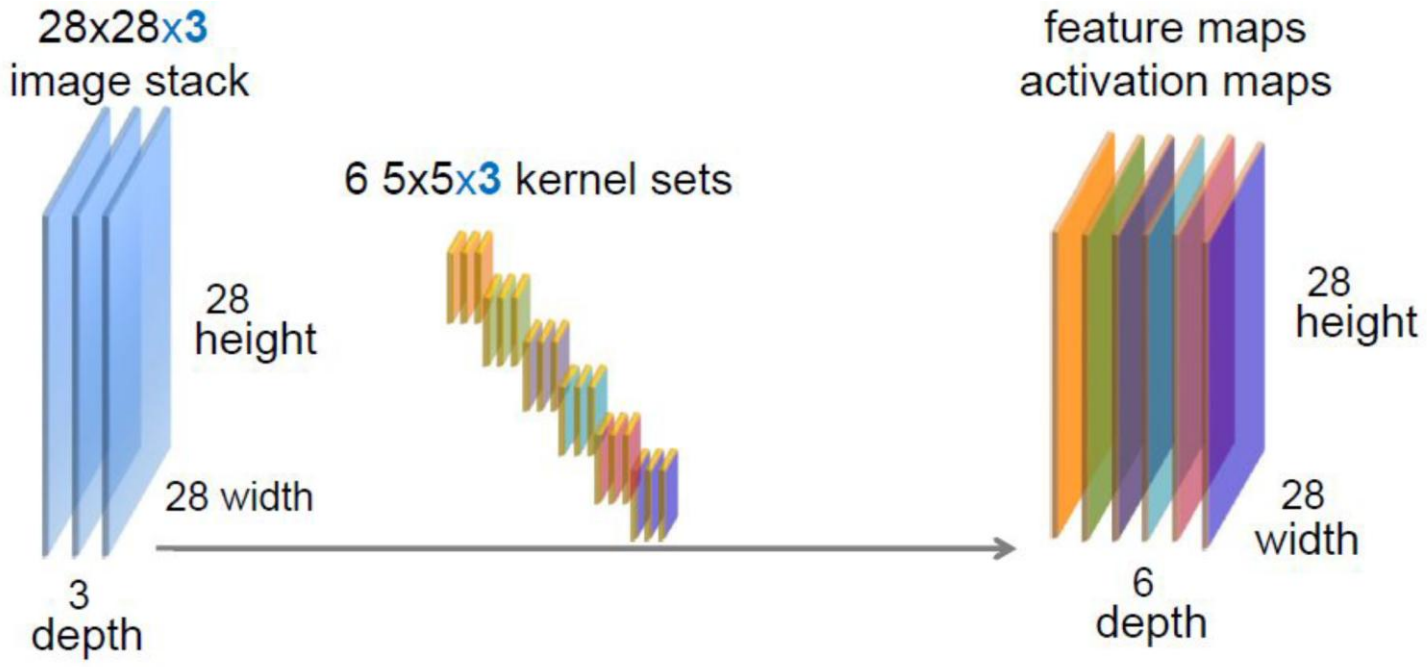
Convolution layer with a 1-chanel input and 6 kernels



Convolution of the input image with 6 different kernels results in 6 activation maps.

If the input image has only one channel, then each kernel has also only one channel.

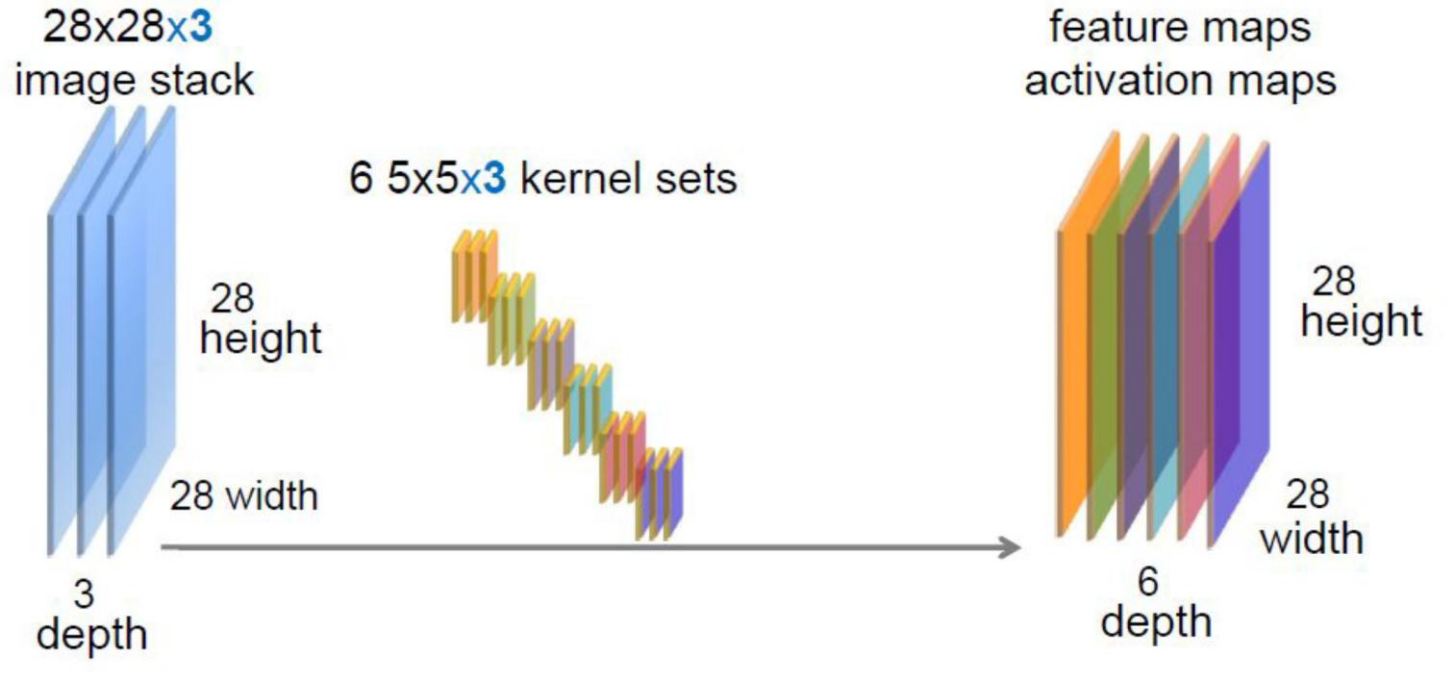
Convolution layer with a 3-chanel input and 6 kernels



Convolution of the input image with 6 different kernels results in 6 activation maps.

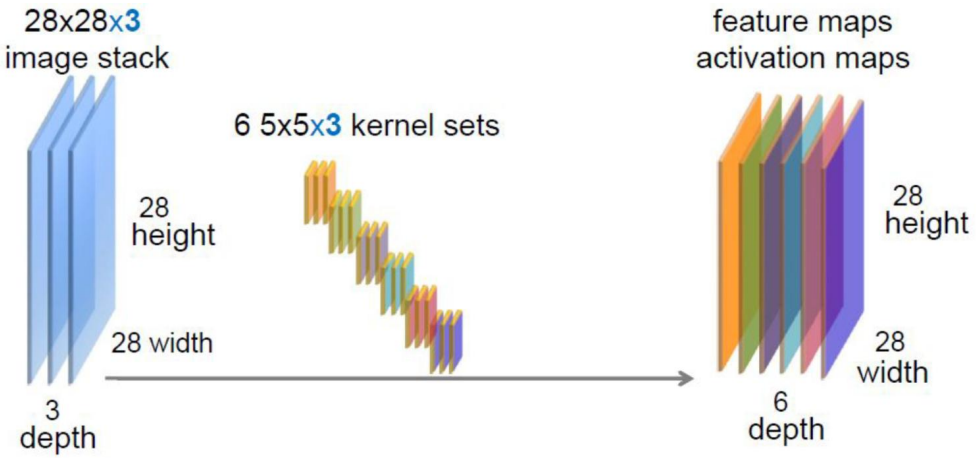
If the input image has 3 channels, then each filter has also 3 channels.

Convolution layer with a 3-chanel input and 6 kernels



How many weights?

Solution



6*5*5*3 + 6 = 456

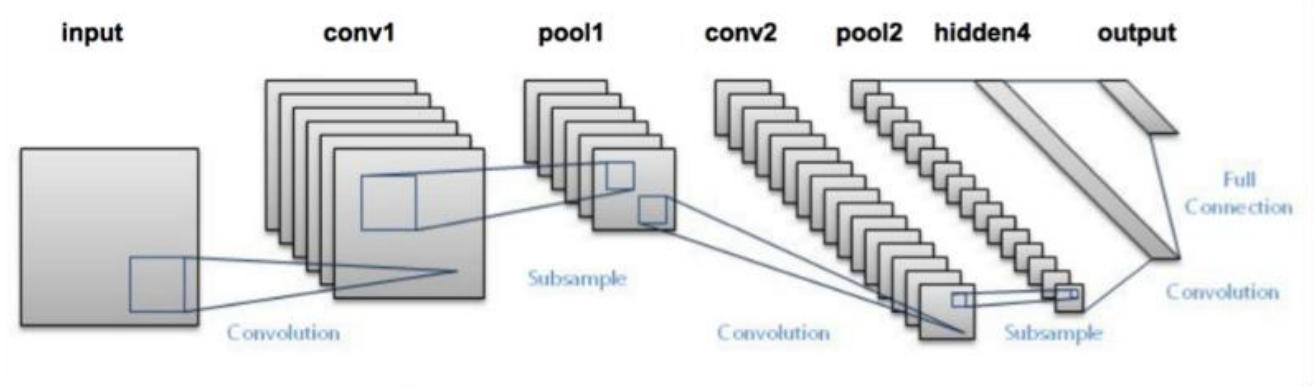
```
# Number of weights in Convolution
model = Sequential()
model.add(Convolution2D(6,kernel_size=(5,5),padding='same',input_shape=(28,28,3)))
model.summary()
```

Model: "sequential_11"

Layer (type)	Output Shape	Param #
conv2d_44 (Conv2D)	(None, 28, 28, 6)	456

Total params: 456 (1.78 KB)
Trainable params: 456 (1.78 KB)
Non-trainable params: 0 (0.00 B)

CNN for MNIST



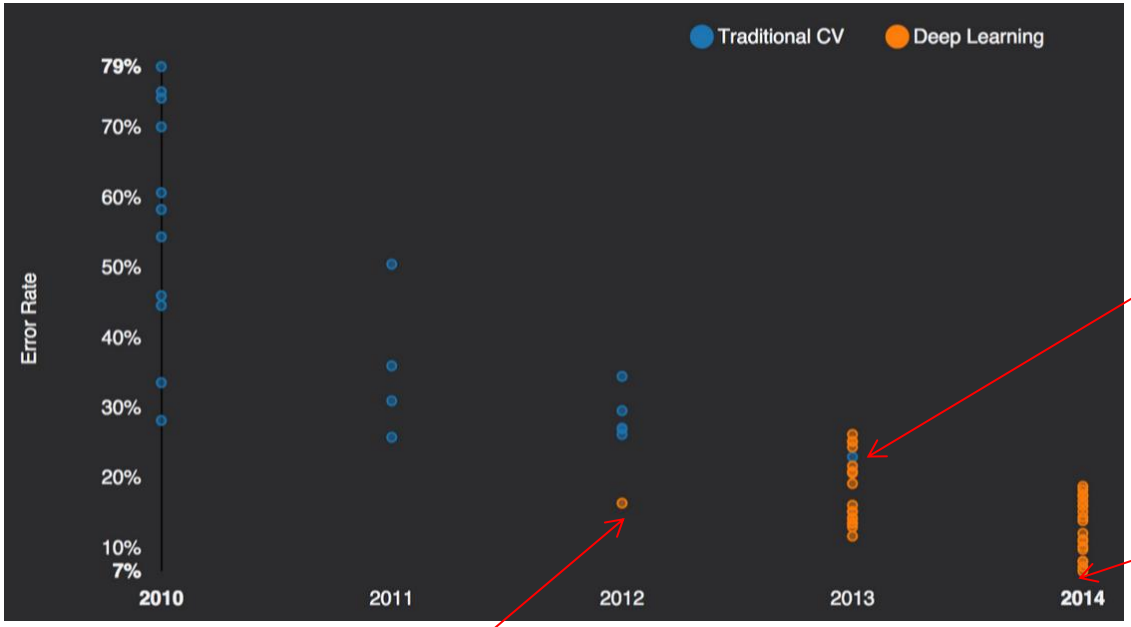

```
model = Sequential()

model.add(Convolution2D(filters=8,kernel_size=(3,3),
                        padding='same',input_shape=(28,28,1)))
model.add(Activation('relu'))
model.add(Convolution2D(16, (3, 3),padding='same'))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Flatten())
model.add(Dense(40))
model.add(Activation('relu'))
model.add(Dense(10))
model.add(Activation('softmax'))

# compile model and intitialize weights
model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])
```

What is a good CNN architecture?

CNN breakthrough in 2012: Imagenet challenge

1000 classes
1 Mio samples

...



Human: 5% misclassification

Only one non-CNN approach in 2013

GoogLeNet 6.7%

A. Krizhevsky
first CNN in 2012
Und es hat zoom gemacht

Figure: https://medium.com/global-silicon-valley/machine-learning-yesterday-today-tomorrow-3d3023c7b519

52

Review of ImageNet winning CNN architectures

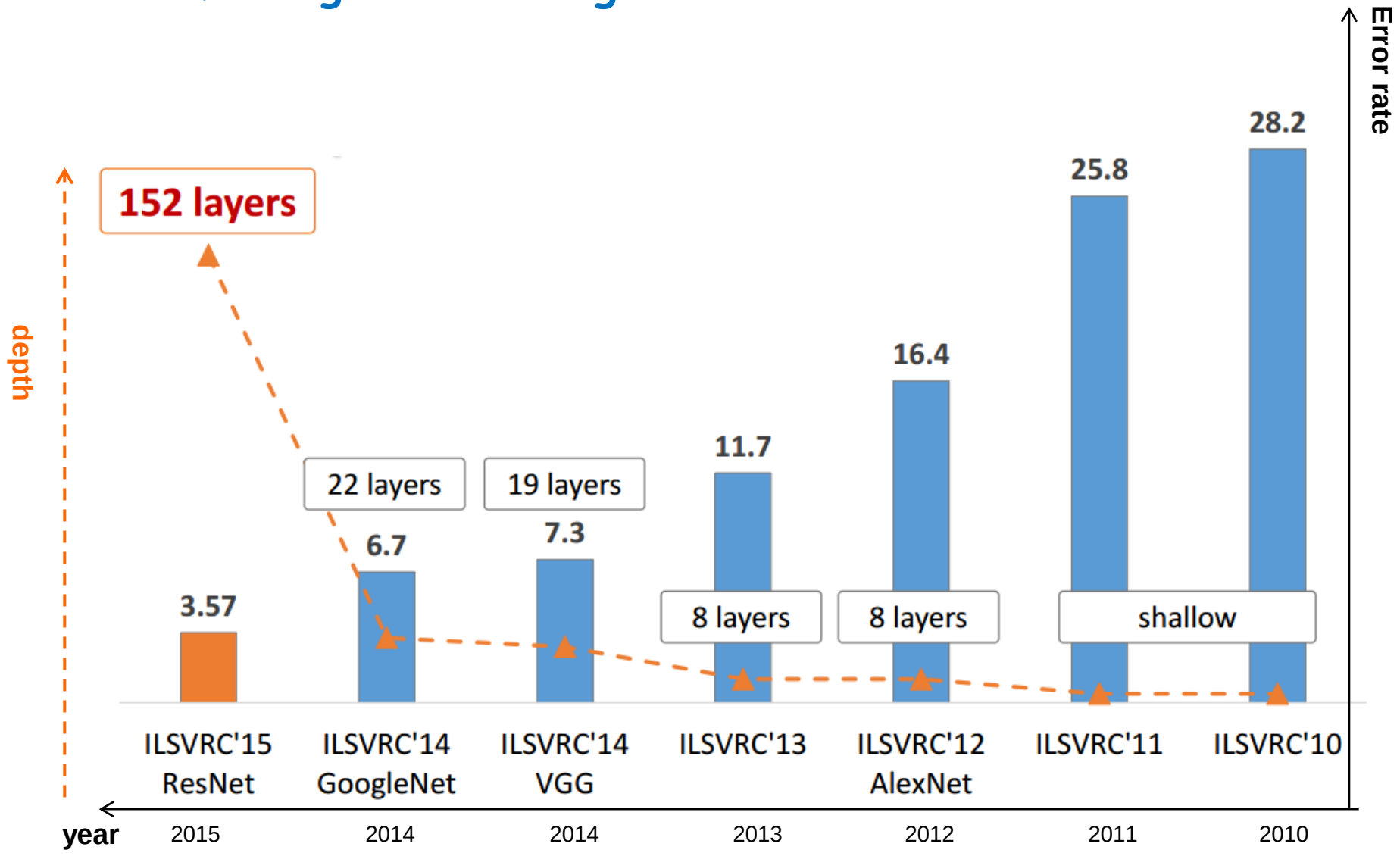


Image credits (modified): K.He, X.Zhang, S.Ren, J.Sun. “Deep Residual Learning for Image Recognition”. arXiv 2015

LeNet-5 1998: first CNN for ZIP code recognition

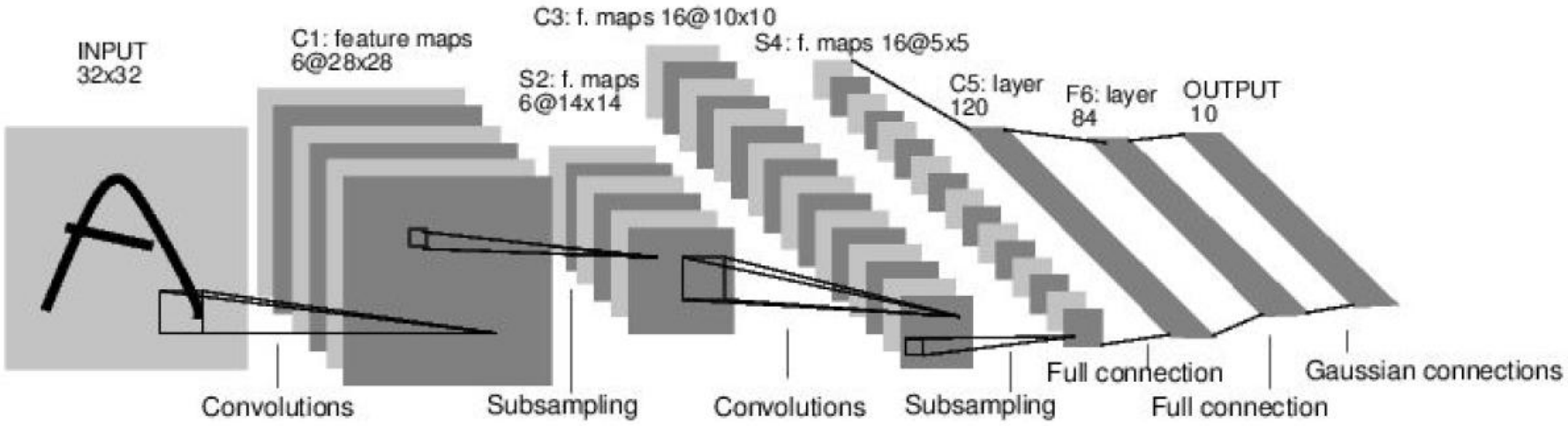


Image credits: http://yann.lecun.com/exdb/publis/pdf/lecun-98.pdf

Conv filters were 5x5, applied at stride 1
Subsampling (Pooling) layers were 2x2 applied at stride 2
i.e. architecture is [CONV-POOL-CONV-POOL-CONV-FC]

Demo von 1993 Yann LeCun

http://yann.lecun.com/exdb/lenet/index.html

The trend in modern CNN architectures goes to small filters

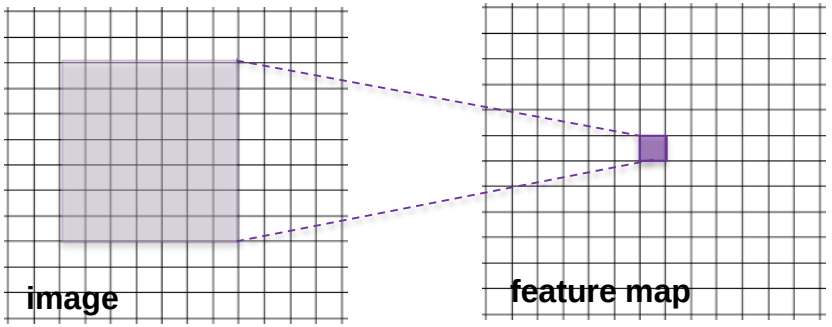
Why do modern architectures use very small filters?

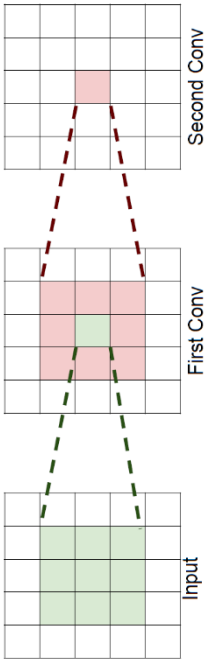
Determine the receptive field in the following situation:

1) Suppose we have **one** 7x7 conv layers (stride 1)

49 weights

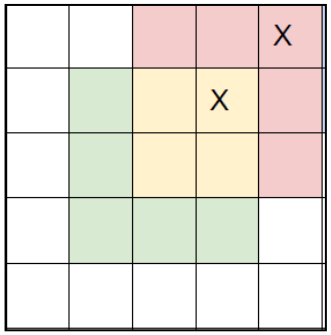
Answer1): 7x7





2) Suppose we stack **two** 3x3 conv layers (stride 1)

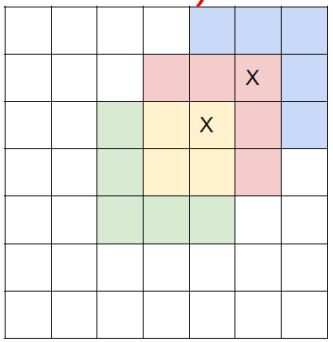
Answer 2): 5x5



3) Suppose we stack **three** 3x3 conv layers (stride 1)

3*9=27 weights

Answer 3): 7x7



We need less weights for the same receptive field when stacking small filters!

"Oxford Net" or "VGG Net" 2014 2nd place

- 2nd place in the imageNet challenge
- More traditional, easier to train
- Small pooling
- Stacked 3x3 convolutions before maxpooling

  -> large receptive field
- no strides (stride 1)
- ReLU after conv. and FC (batchnorm was not introduced)
- Pre-trainined model is available (see excercise)

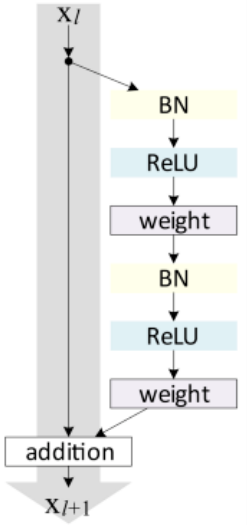
http://arxiv.org/abs/1409.1556

"ResNet" from Microsoft 2015 winner of imageNet

152 layers

ResNet basic design (VGG-style)

- add shortcut connections every two
- all 3x3 conv (almost)



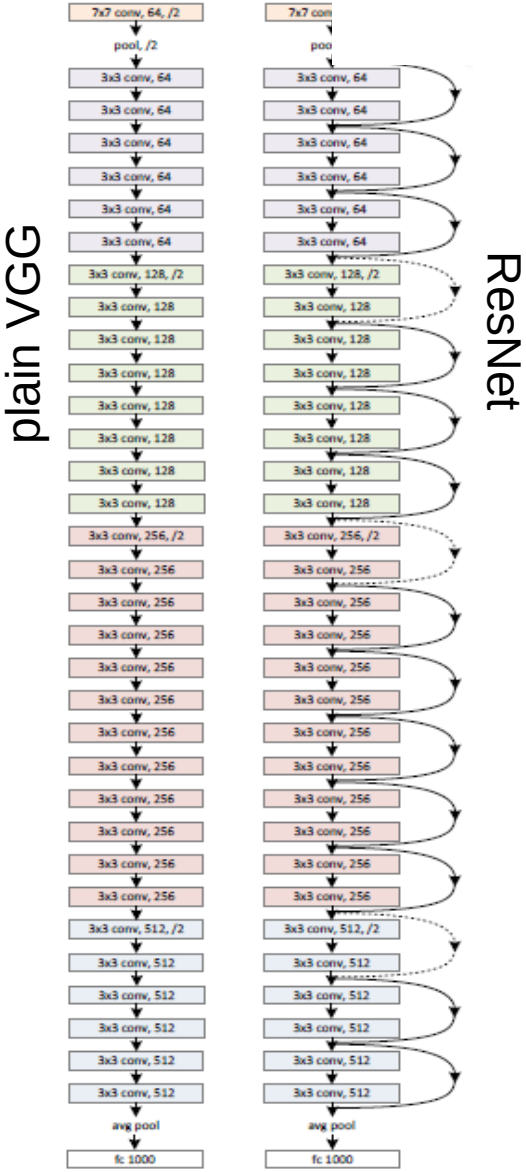
$F(x_l)$

152 layers:
Why does this train at all?

This deep architecture could still be trained, since the gradients can skip layers which diminish the gradient!

$H(x_l)=x_{l+1} = x_l + F(x_l)$

F(x) is called "residual" since it only learns the "delta" which is needed to add to x to get H(x)

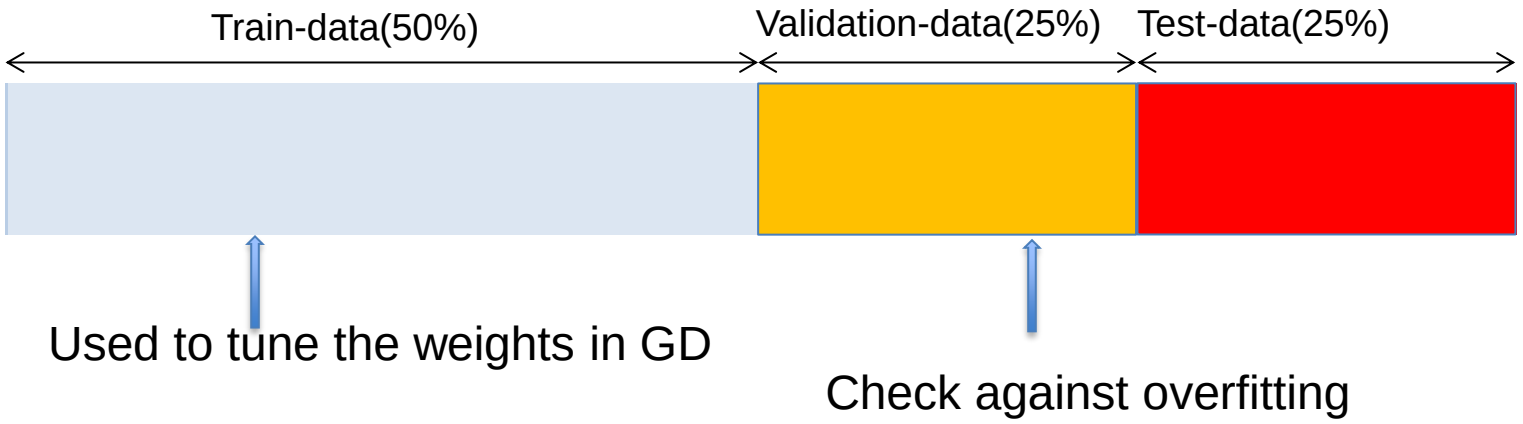


credits: https://chatbotslife.com/resnets-highwaynets-and-densenets-oh-my-9bb15918ee32

Tricks of the trade

- Early stopping
- Input Standardization
- Batch Norm Layer
- Dropout
- Data augmentation

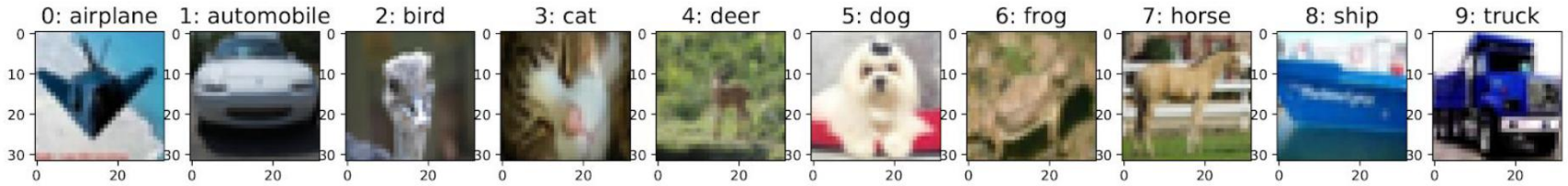
Best practice: Split in Train, Validation, and Test Set



Best practice: Lock an extra test data set away, and use it only at the very end, to evaluate the chosen model, that performed best on your validation set.
Reason: **When trying many models, you probably overfit on the validation set**.

Determine performance metrics, such as MSE, to evaluate the predictions **on new validation or test data**

CIFAR10 study with tensorflow notebook



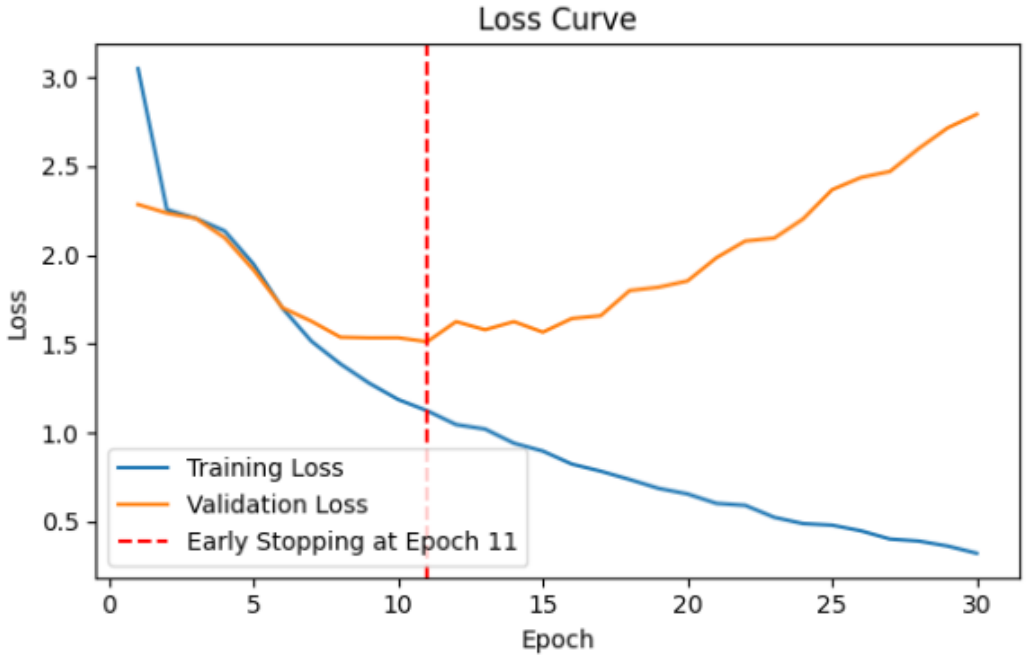
We define a CNN to classify cifar10 images (we have 10 classes)

Notebook:
https://github.com/tensorchiefs/dl_course_2020/blob/master/notebooks/07_cifar10_norm_sol.ipynb

60

Loss curve and early stopping

Very common check: Plot loss in train and validation data vs epoch of training.



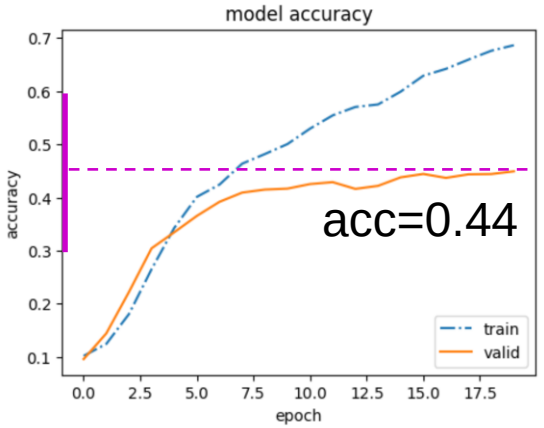

```
Training completed all epochs.
Best model weights were restored from epoch 11
```

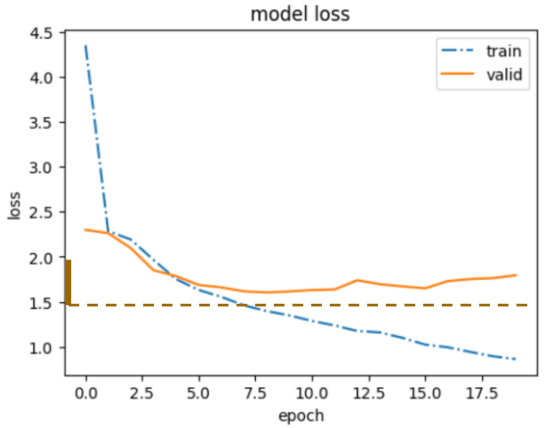
- If training loss does not go down to zero: model is not flexible enough
- Use weights @minimum of validation before overfitting
- Early stopping: stop to training if validation loss does not improve anymore

Take-home messages from CIFAR10 CNN study

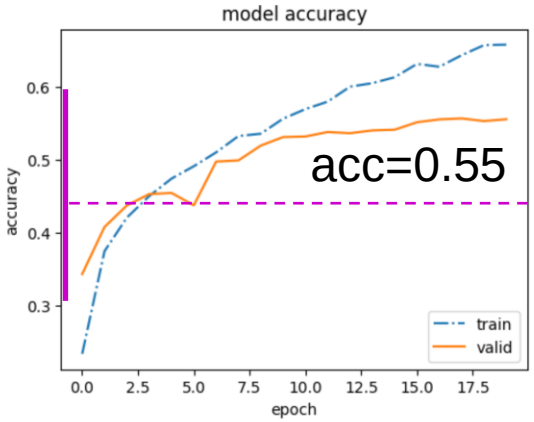
- DL does not need a lot of preprocessing, but working with standardized (small-valued) input data often helps.

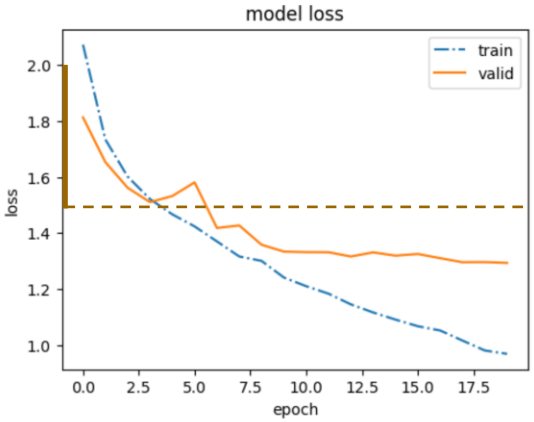
Without normalizing
the input to the CNN





With normalizing
(pixel-value/255)
the input to the CNN

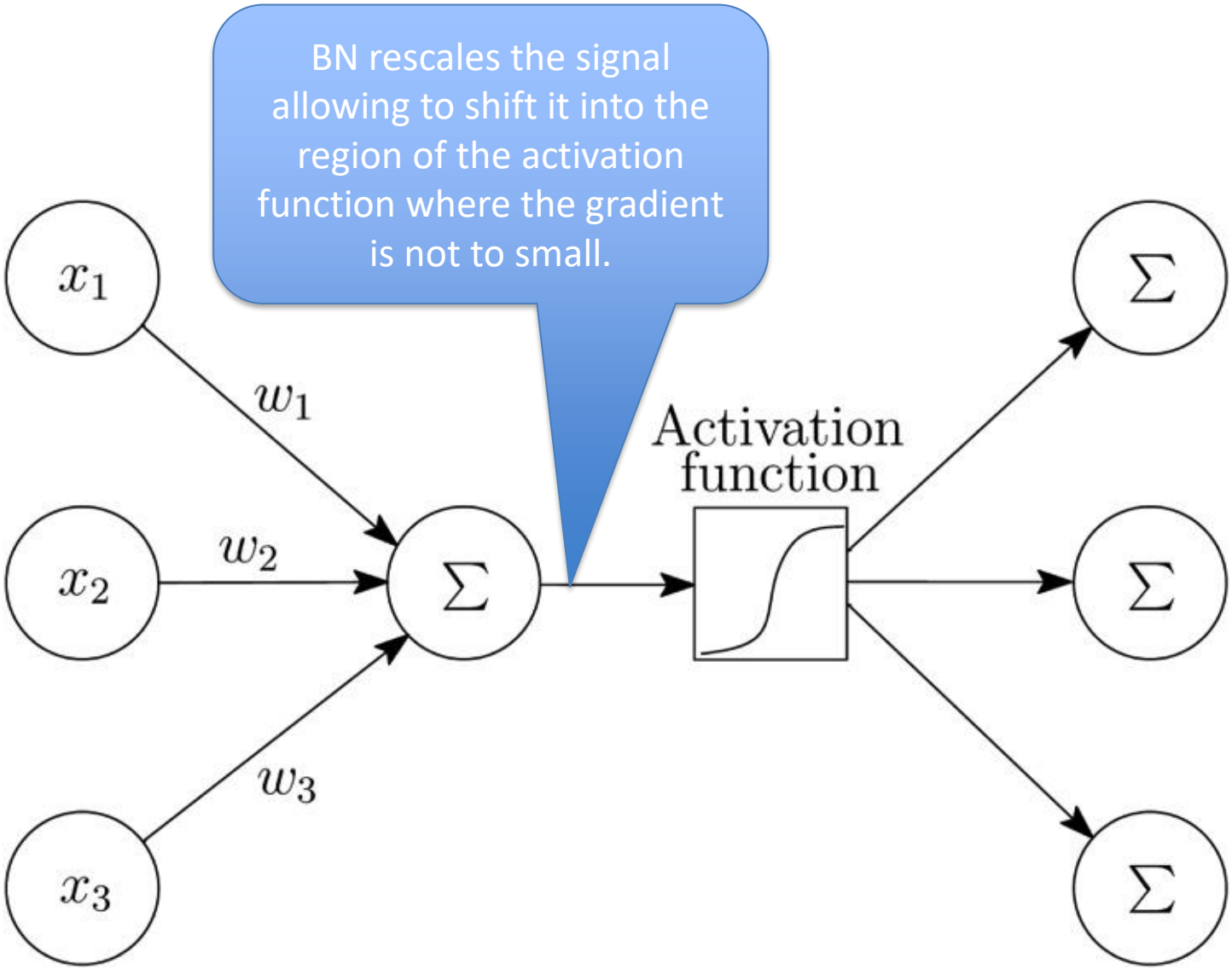




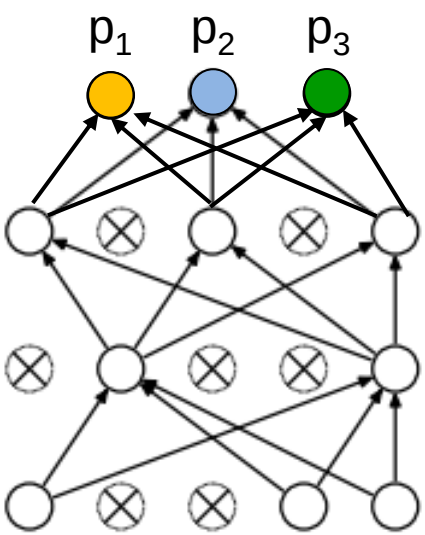
Regularization to avoid overfitting

- Batchnorm layer
- Dropout layer

What is the idea of Batch-Normalization (BN)



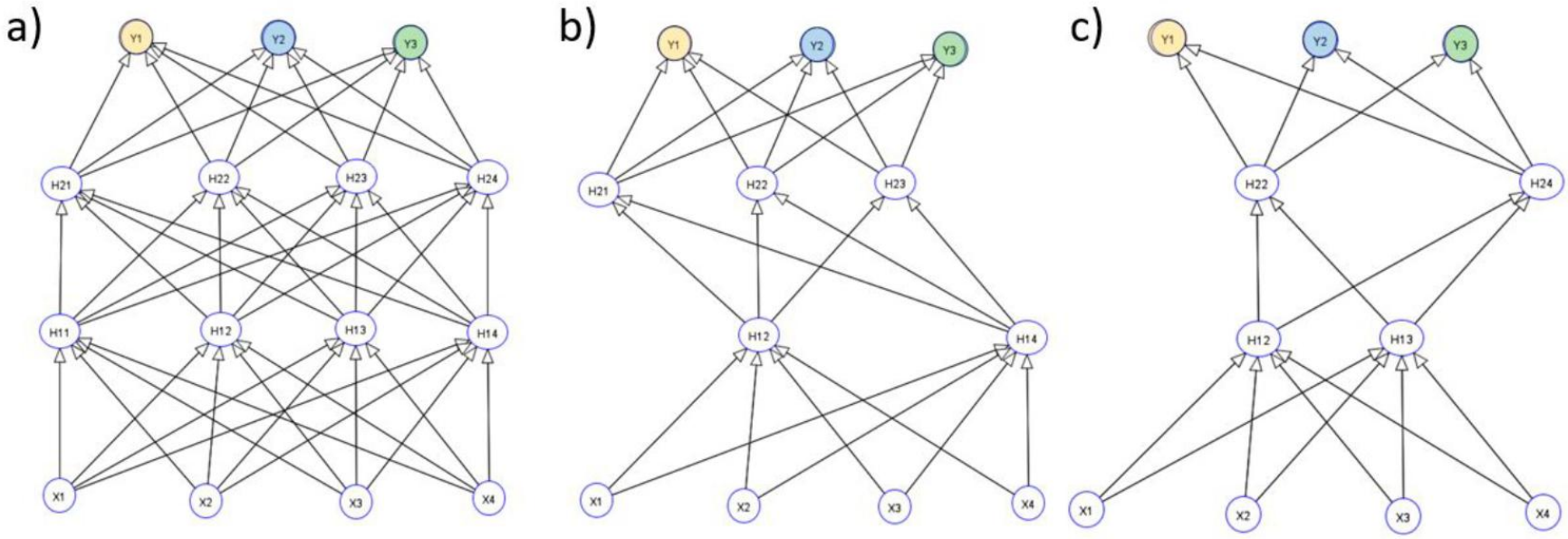
Dropout helps to fight overfitting



Using dropout during training implies:

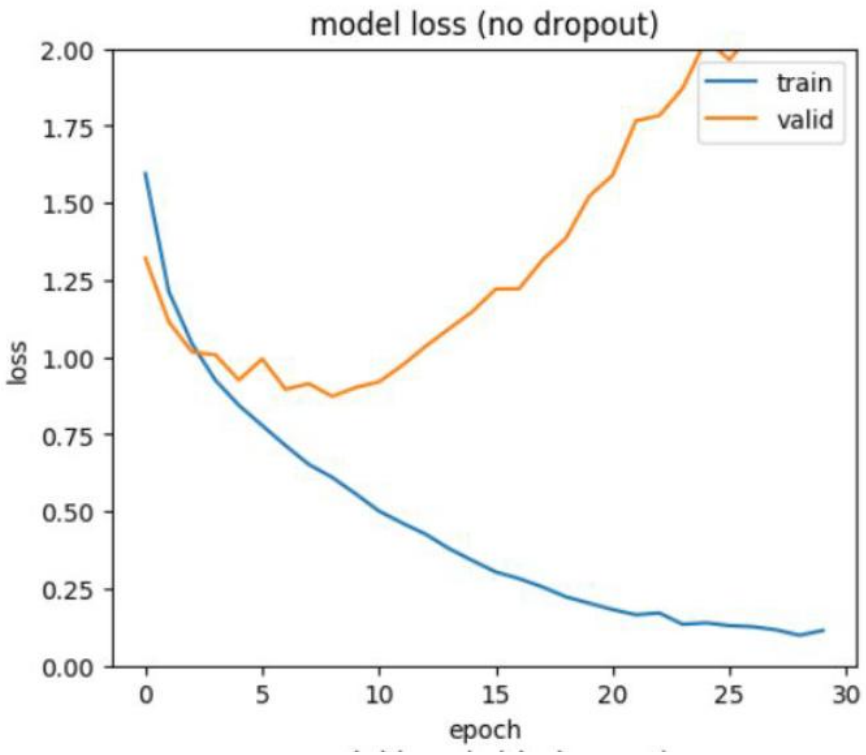
- In each training step only weights to not-dropped units are updated → we train a sparse sub-model NN
- For predictions with the trained NN we freeze the weights corresponding to averaging over the ensemble of trained models we should be able to “reduce noise”, “overfitting”
- JFI: To get same expected output in training (with dropout) and after training (test time - without dropout), the weights are multiplied after training by the dropout probability p=0.5.

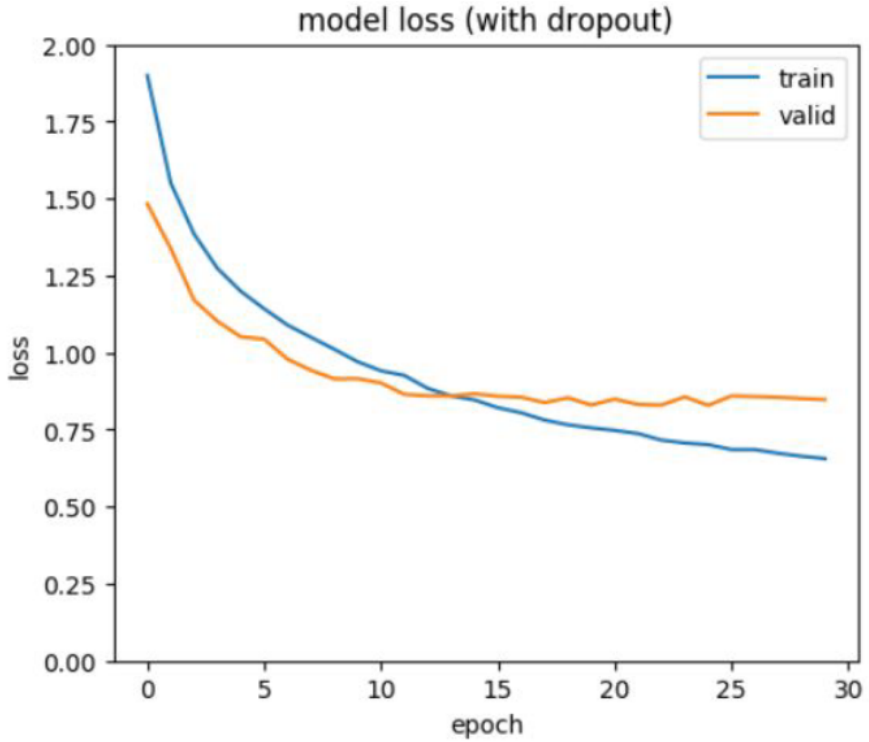
Dropout



Three NNs:
a) shows the full NN with all neurons (as used when NN is trained),
b) and c) show two versions of a thinned NN where some neurons are dropped (as done during training with dropout). Dropping neurons is the same as setting all connections that start from these neurons to zero.

Dropout fights overfitting in a CIFAR10 CNN





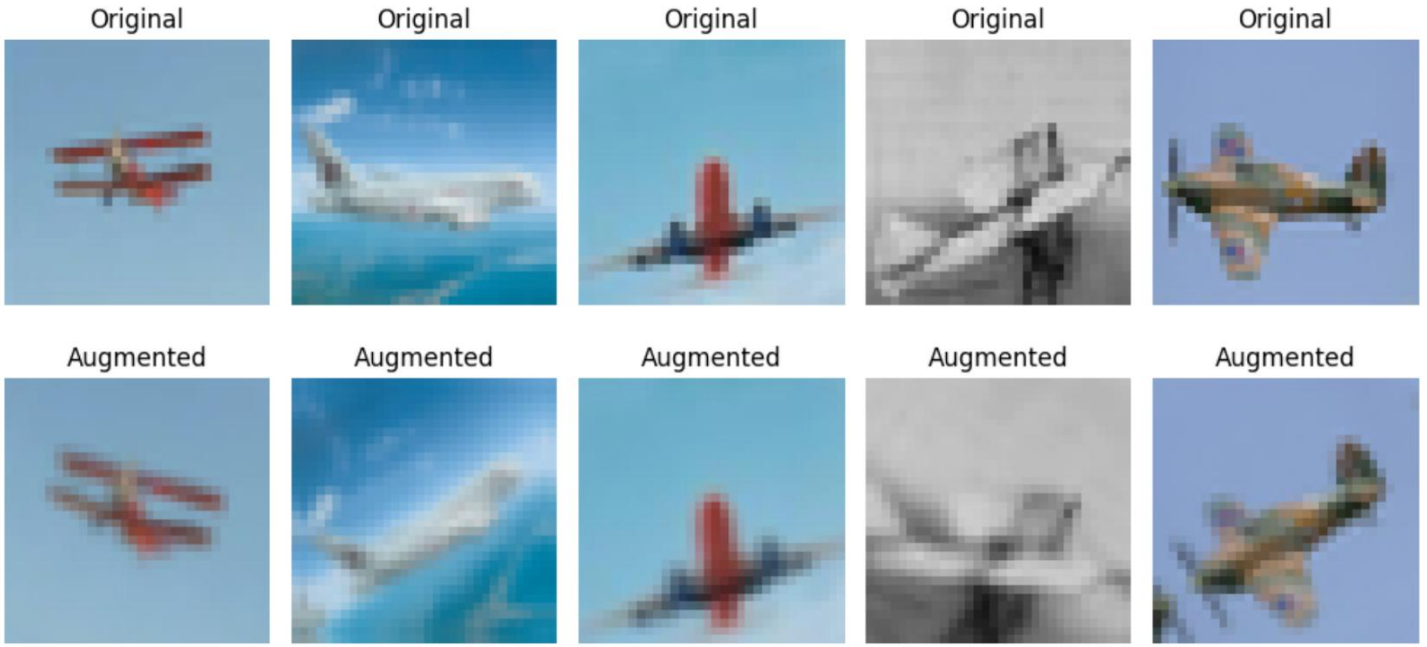
What to do in case of limited data?

- Data augmentation
- Pretrained (foundation) models

Fighting overfitting by Data augmentation:

During training random operations are done on the fly

- Rotate image within an angle range
- Flip image: left/right, up, down
- resize
- Take patches from images
- ….



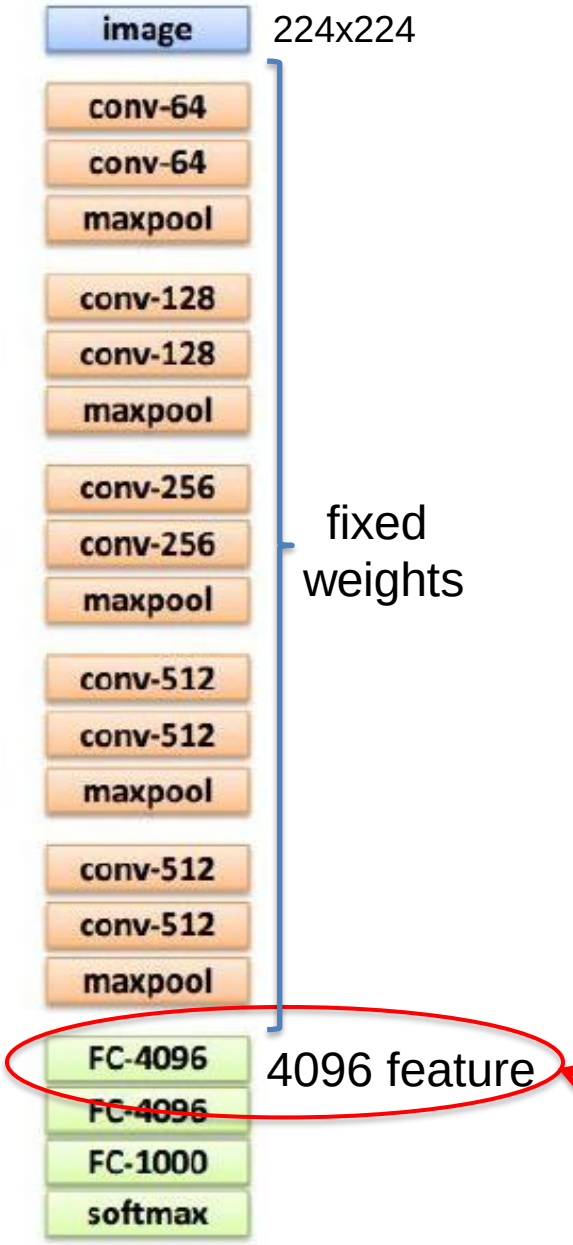
In Code

```python
# Define data augmentation pipeline
from keras.layers import RandomFlip, RandomRotation, RandomContrast, RandomZoom
data_augmentation = keras.Sequential([
    #RandomFlip("horizontal"),       # Randomly flip images horizontally
    RandomRotation(0.1),            # Randomly rotate images by 10%
    RandomZoom(0.1),                # Randomly zoom in on images
    #RandomContrast(0.1),           # Adjust contrast randomly
], name='Augmentation')
```

```python
# Functional API Model Definition
x = Input(shape=input_shape)
x = data_augmentation(x)
# First Convolutional Block
x = Convolution2D(8, kernel_size, padding='same')(x)
...
```

See: https://github.com/tensorchiefs/dlwbl_eth25/blob/master/notebooks/02_transfer_learning.ipynb

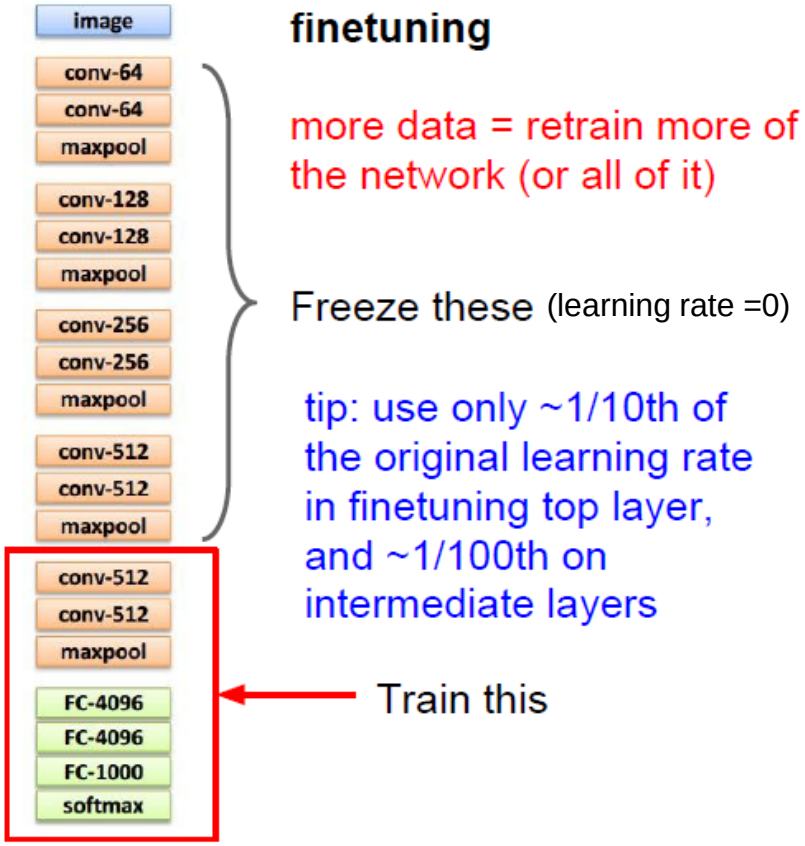
Use pre-trained CNNs for feature generation



- Load a pre-trained CNN – e.g.g VGG16
  developed by Karen Simonyan and Andrew Zisserman in 2014
- Images are resized to required size
- Rescaling of the pixel values to “VGG range”
- Do a forward pass and fetched features that are used as CNN representations, dump these features into a file on disk
- Use these CNN features as input to a simple classifier – e.g. fc NN, RF, SVM …
  (here it is easily possible to adapt to the new number of class labels)

Transfer learning beyond using off-shelf CNN feature

e.g. medium data set (<1M images)



The strategy for fine-tuning depends on the size of the data set and the type of images:

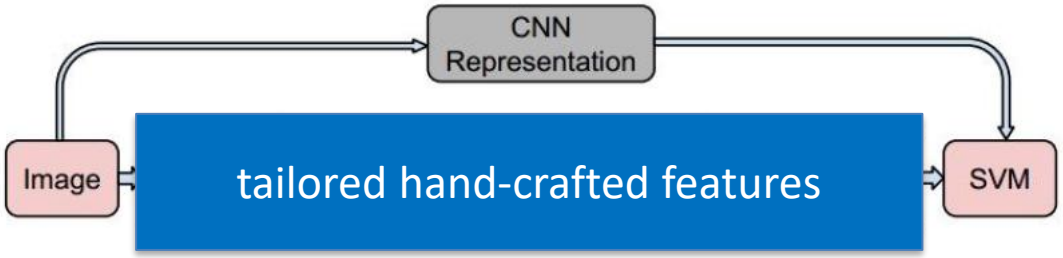
	Similar task (to imageNet challenge)	**Very different task** (to imageNet challenge)
little data	Extract CNN representation of one top fc layer and use these features to train an external classifier	You are in trouble - try to extract CNN representations from different stages and use them as input to new classifier
lots of data	Fine-tune a few layers including few convolutional layers	Fine-tune a large number of layers

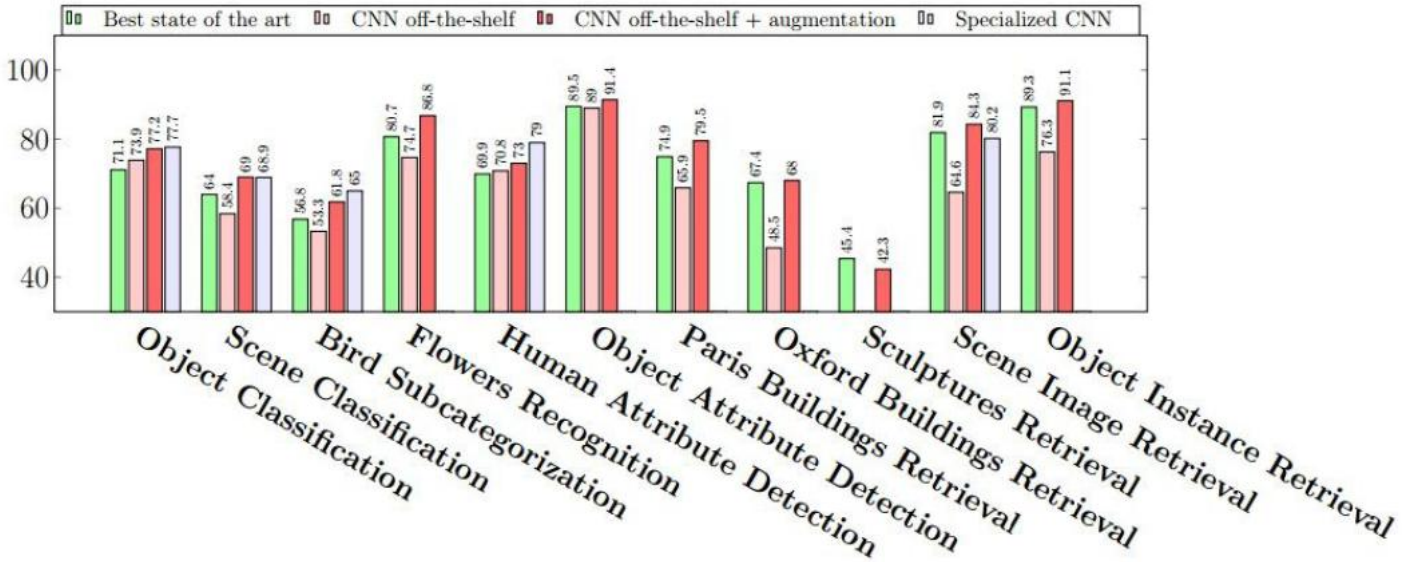
Where to get pretrained CNNs like VGG16 or ResNet: https://keras.io/api/applications/
Hint: first retrain only fully connected layer, only then add convolutional layers for fine-tuning.

Slide credits (modified): cs231n

Performance of off-the-shelf CNN features when compared to tailored hand-crafted features

CNN's distributed and compositional features generalize well to new tasks

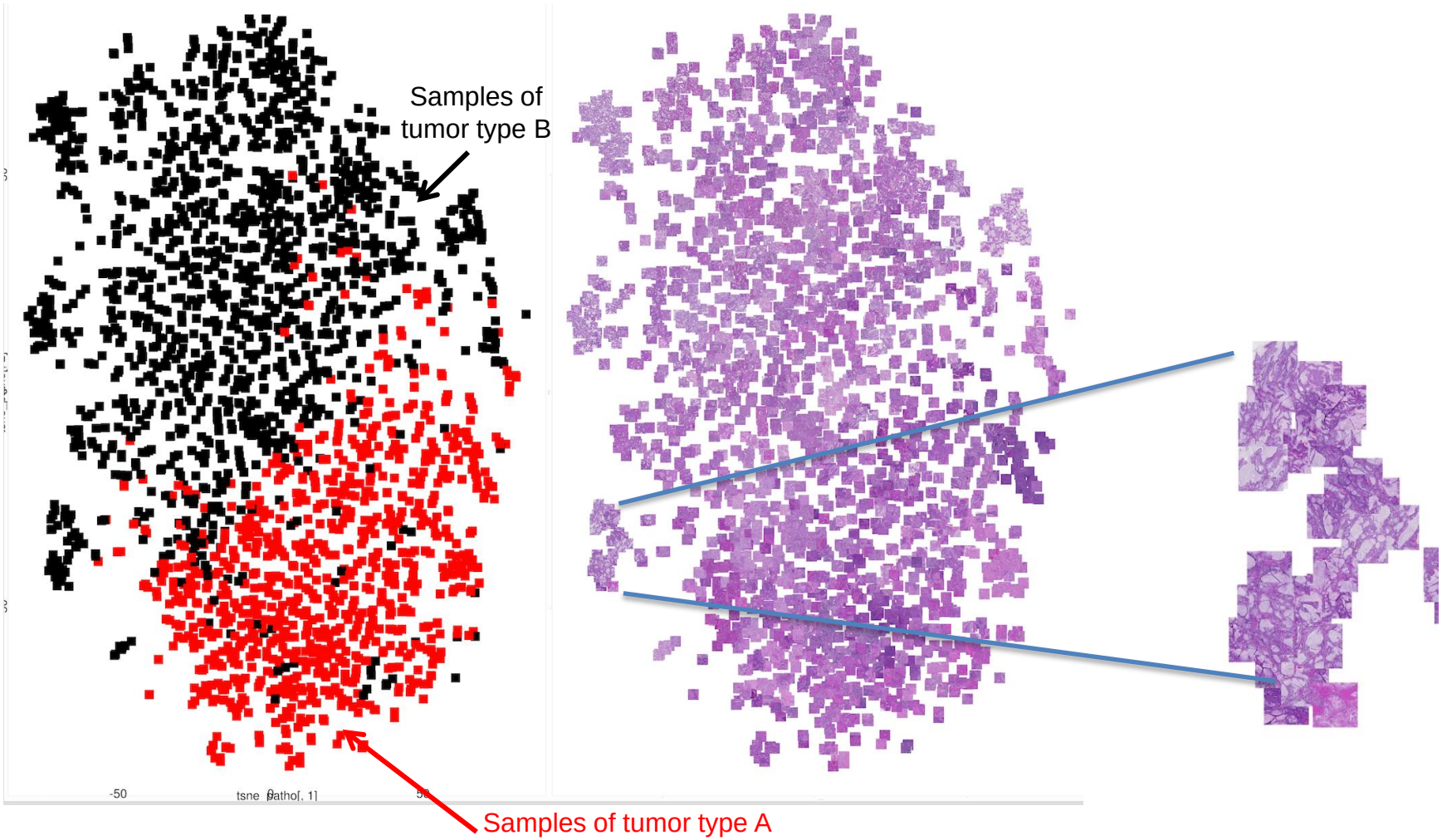




"Astonishingly, we report consistent superior results compared to the highly tuned state-of-the-art systems in all the visual classification tasks on various datasets."

CNN Features off-the-shelf: an Astounding Baseline for Recognition [Razavian et al, 2014]

Use features from a pretrained VGG as input to t-SNE

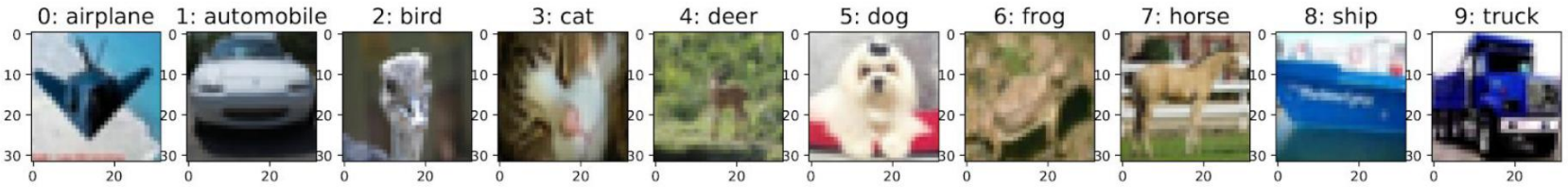


→ Different tissue types cluster together in t-SNE: we could use knn as classifier

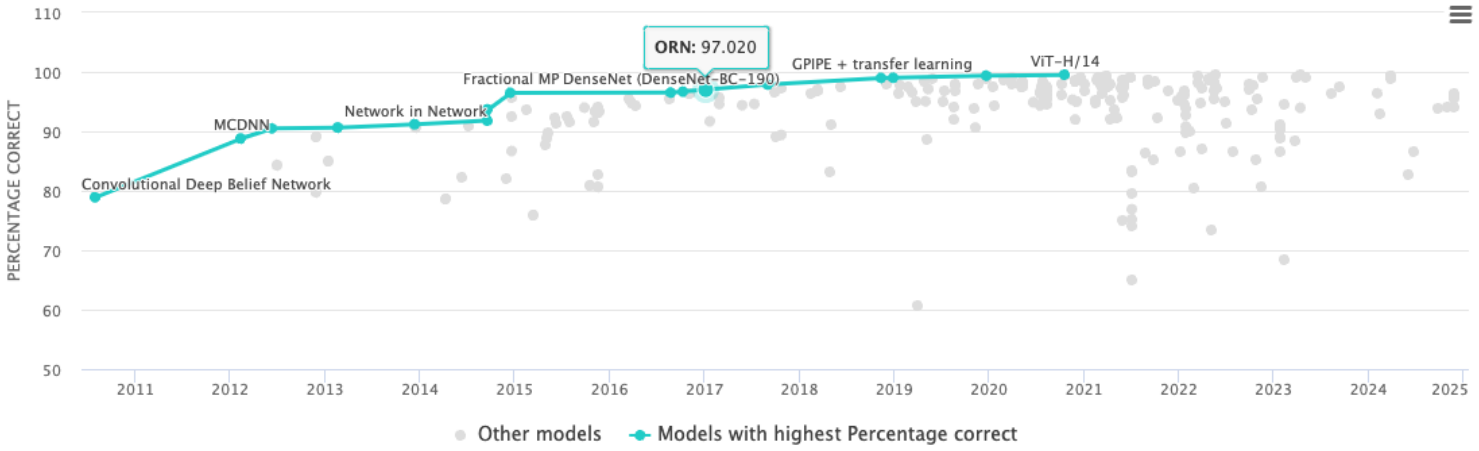
→ VGG features even work on images that are far away from the 1000 imageNet classes

Case Study: CIFAR10 Dataset

Cifar10 Data



10 Classes, with 5'000 images for training



Some examples:

VGG-19 with GradInit	2021	94.71
ResNet-18	2022	95.55

https://paperswithcode.com/sota/image-classification-on-cifar-10

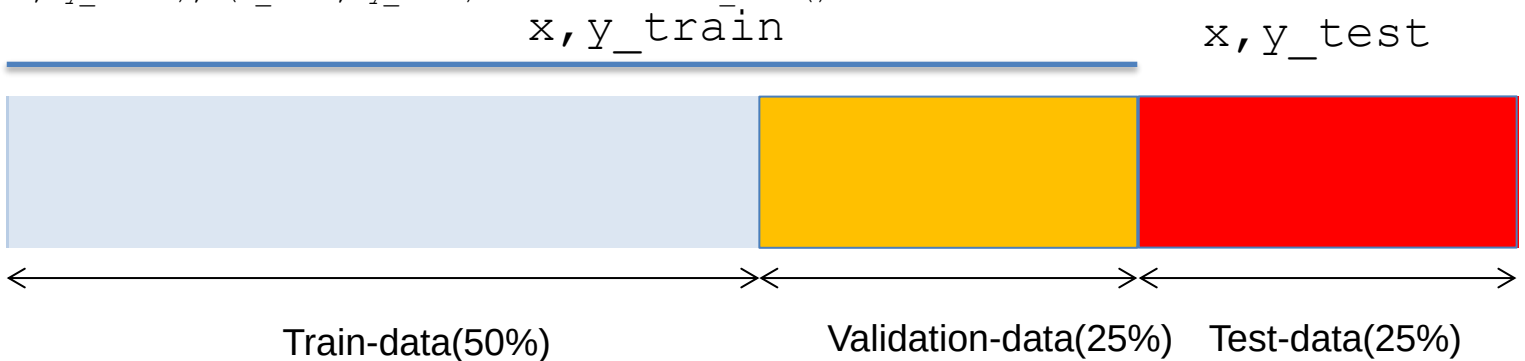
Dataset Splitting in Machine Learning Competitions

- Training Set
  - The set you get for training, you can do what you want with it.
- Test Set:
  - Final evaluation set, never seen by the model during training (e.g., 10,000 images in CIFAR-10).

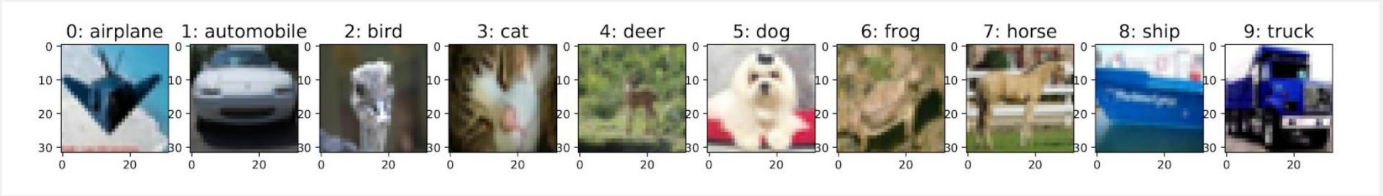
Note: CIFAR-10 and many other datasets do NOT provide a separate validation set.

- Typical Workflow
  - Hyperparameter Tuning
    - **Create validation set** from training set
      - Helps in tuning hyperparameters and preventing overfitting.
      - Common practice: split 80% training / 20% validation (e.g., 40,000 train / 10,000 validation).
  - Final Training: After hyperparameter tuning, models are often retrained on the entire training set
  - Evaluation on test set
    - Sometimes test set is secret and you to upload you predictions (private test set)

```
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
```



Exercise: Transfer learning with CIFAR10 data



rf on vgg features

transfer learning on vgg features

cnn from scratch with data augmentation

79