

# Machine Intelligence:: Deep Learning

## Week 3

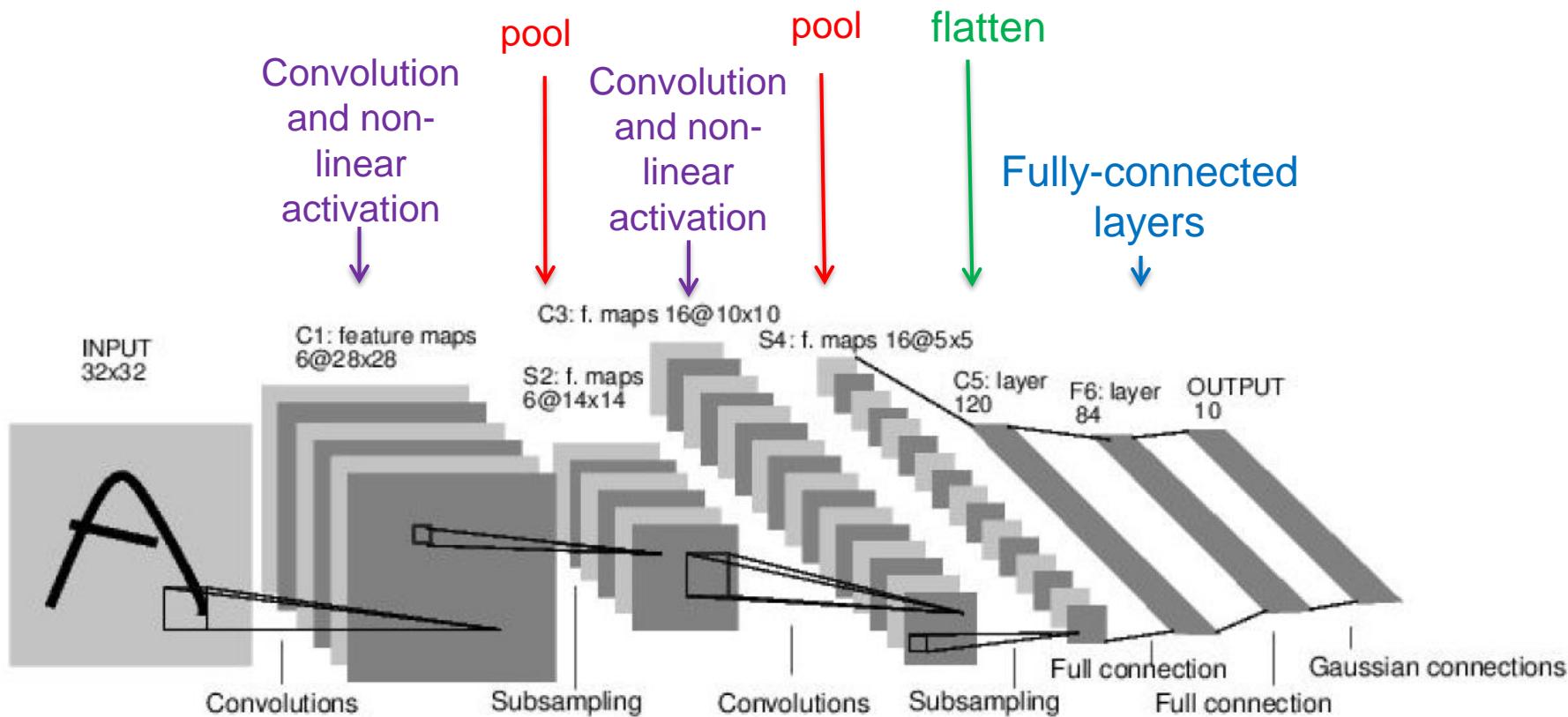
*Beate Sick, Oliver Dürr, Pascal Bühler*

Institut für Datenanalyse und Prozessdesign  
Zürcher Hochschule für Angewandte Wissenschaften

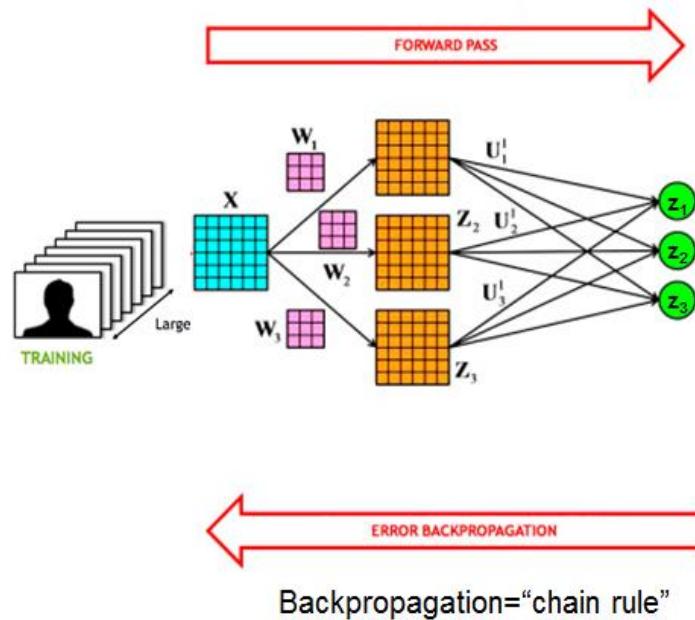
# Topics of today

- Tricks of the trade to achieve good-performing CNNs
  - Standardizing / Batch-Norm
  - Dropout
- Look at challenge-winning CNN architectures
- DL with few data:
  - Augmentation
  - Using pretrained CNN for extracting “good” image features
  - Transfer learning
- Biological motivation of a CNN
- Shining light into the black-box CNN
  - Which parts of an image were important for a certain model prediction

# Typical architecture of a simple CNN



# Training of a CNN is based on gradient backpropagation



Learning is done by weight updating:

For the training we need the observed label for each image which we then compare with the output of the CNN.

We want to adjust the weights in a way so that difference between true label and output is minimal.

Minimize Loss-function:

$L = \text{distance}(\text{observed}, \text{output}(w))$

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

↑  
learning rate

# Data Standardization

## Batch Norm Layer

# Data Standardization

- In contrast to other classifiers like trees (or Random Forest) result strongly depends on range of input data
- CNN works best if pixel values of images are in range [0,1]
- Standardization
  - Divide by range
  - Z-Transformation
- Automatic Z-transformation
  - Batch-Normalization
- Make sure to use correct standardization
  - If you “steal” an architecture
  - If you do transfer learning

# Typical preprocessing of image data

CNN works best if pixel values of images are in range [0,1] (see NB 07).

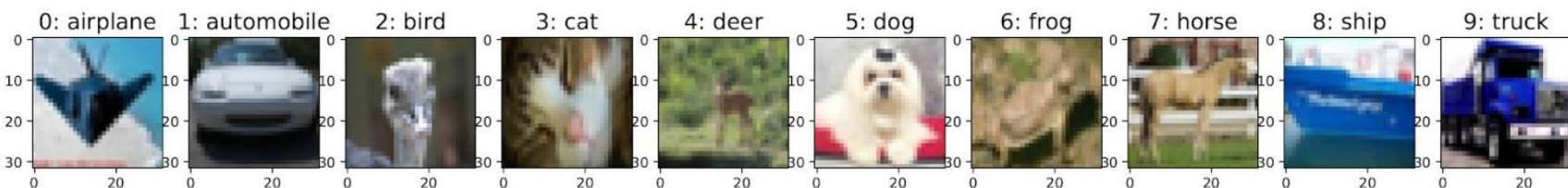
Therefore each pixel value is divided by the maximal possible pixel value:

```
# normalization
X_train_norm = X_train/255
X_val_norm = X_val/255
X_test_norm = X_test/255
```



Since also the initial weights of the NN are small values around zero, this data normalization as preprocessing is reasonable.

# Exercise:



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

Investigate the impact of standardizing/normalizing the data on the performance

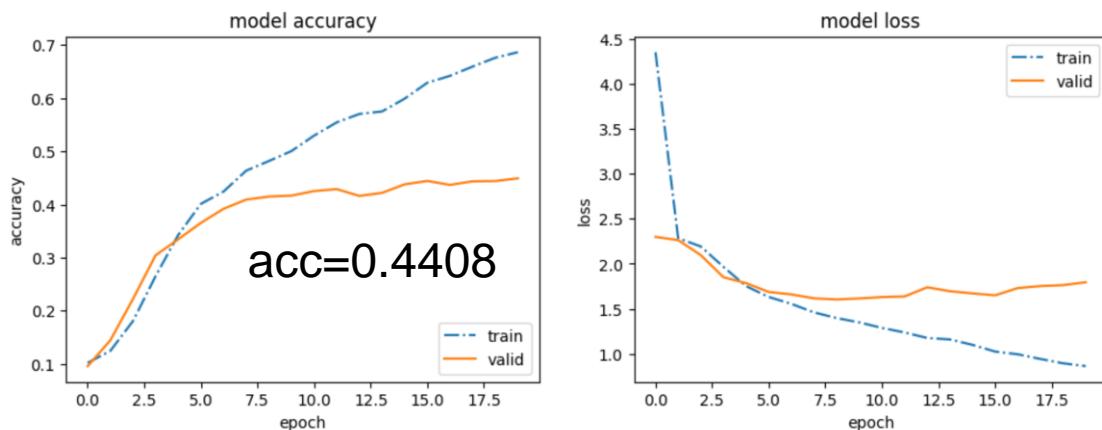
Work through the Notebook 07 and **stop before** the “dropout” section:

[07\\_cifar10\\_tricks\\_sol.ipynb](#)

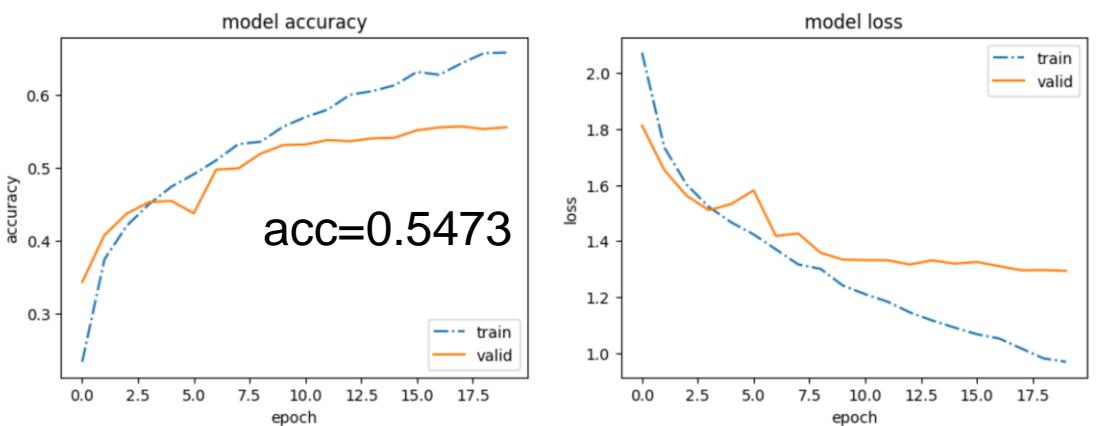
# Take-home messages from NB07

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

Without standardizing  
the input to the CNN

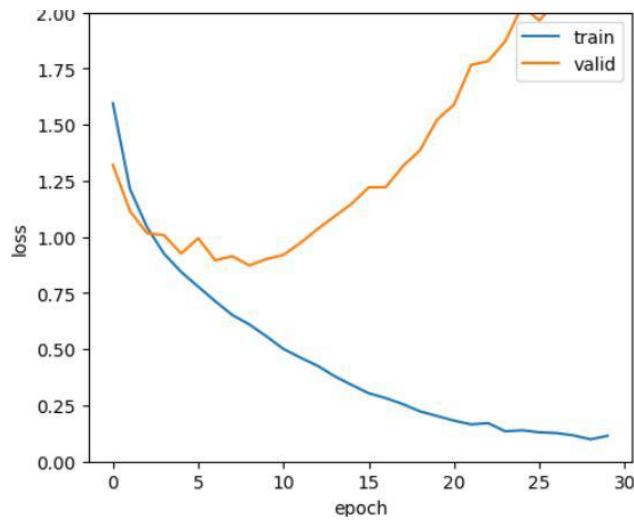


After standardizing  
the input to the CNN



**Overfitting  
Early stopping and  
Dropout during training**

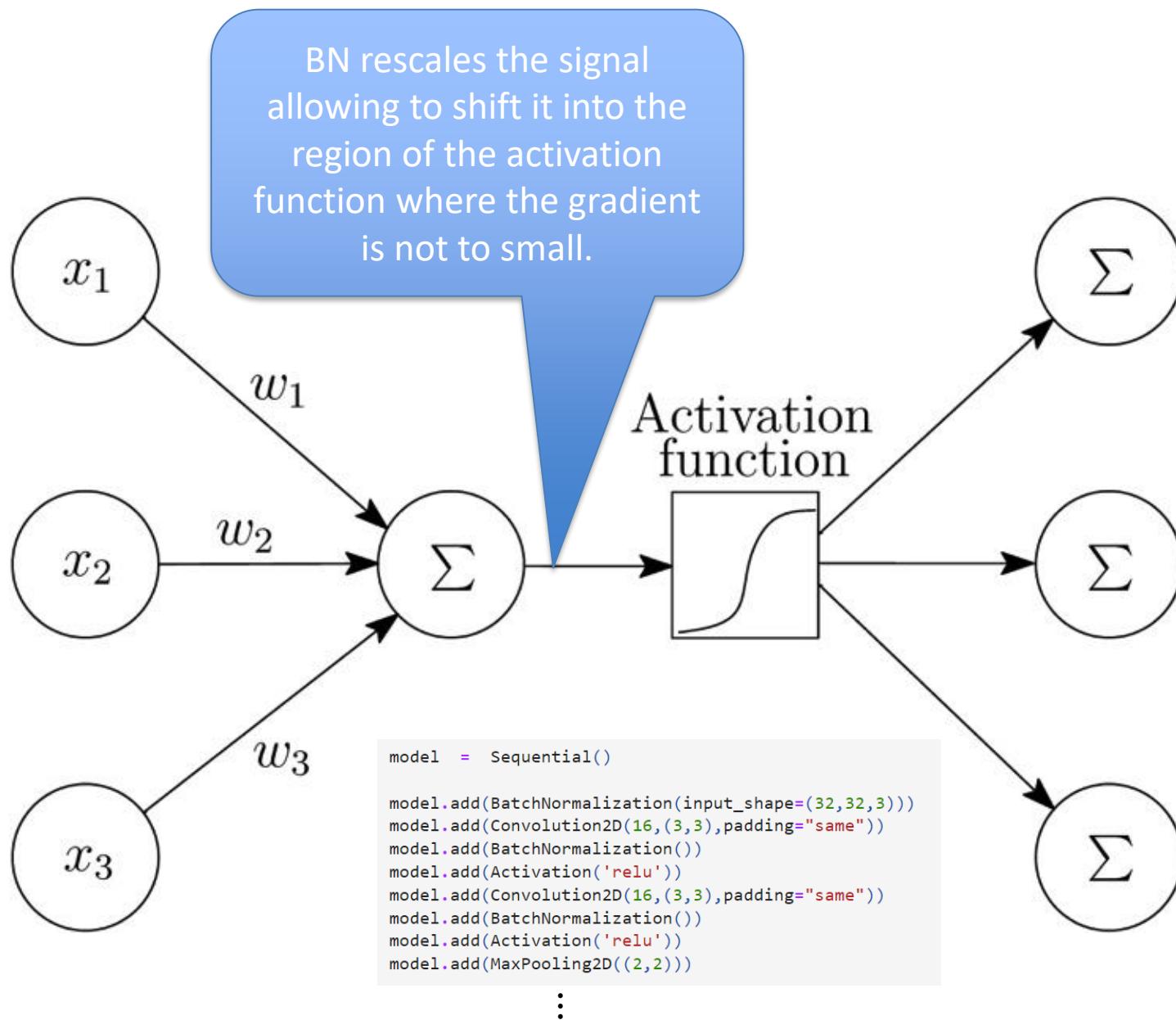
# To avoid overfitting early stopping



```
>>> callback = tf.keras.callbacks.EarlyStopping(monitor='loss', patience=3)
>>> # This callback will stop the training when there is no improvement in
>>> # the loss for three consecutive epochs.
>>> model = tf.keras.models.Sequential([tf.keras.layers.Dense(10)])
>>> model.compile(tf.keras.optimizers.SGD(), loss='mse')
>>> history = model.fit(np.arange(100).reshape(5, 20), np.zeros(5),
...                     epochs=10, batch_size=1, callbacks=[callback],
...                     verbose=0)
>>> len(history.history['loss']) # Only 4 epochs are run.
4
```

See [https://keras.io/api/callbacks/early\\_stopping/](https://keras.io/api/callbacks/early_stopping/)

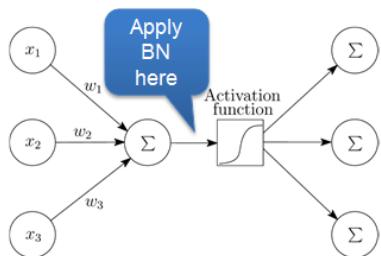
# What is the idea of Batch-Normalization (BN)



# Batch Normalization

- Idea: Introduce batch-norm layers between convolutions.

A BN layer performs a 2-step procedure with  $\alpha$  and  $\beta$  as **learnable** parameter:



$$\text{Step 1: } \hat{x} = \frac{x - \text{avg}_{\text{batch}}(x)}{\text{stdev}_{\text{batch}}(x) + \epsilon}$$

$$\begin{aligned}\text{avg}(\hat{x}) &= 0 \\ \text{stdev}(\hat{x}) &= 1\end{aligned}$$

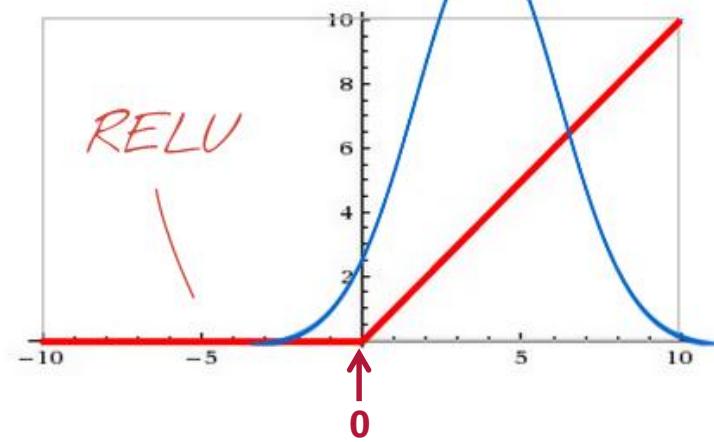
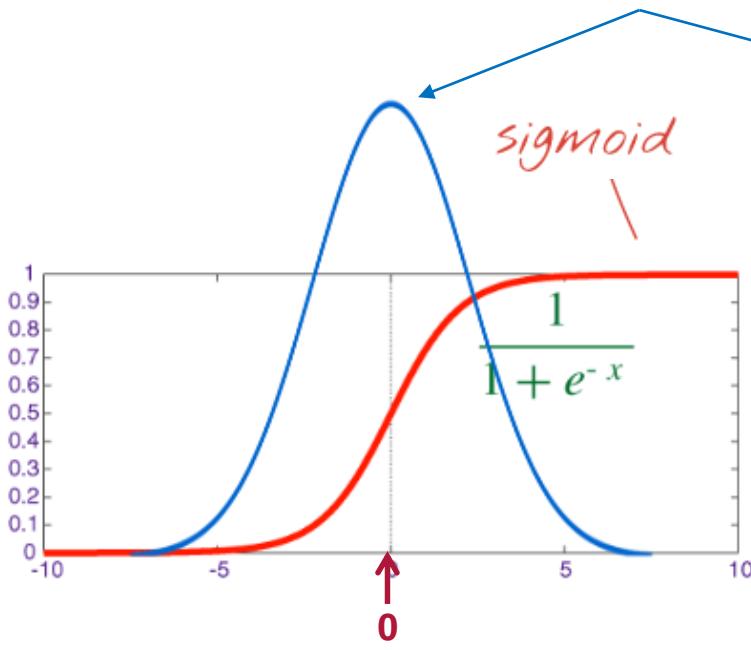
$$\text{Step 2: } BN(x) = \alpha \hat{x} + \beta$$

The learned parameters  $\alpha$  and  $\beta$  determine how strictly the standardization is done. If the learned  $\alpha = \text{stdev}(x)$  and the learned  $\beta = \text{avg}(x)$ , then the standardization performed in step 1 is undone in step 2 and  $BN(x) = x$ .

# Batch Normalization is beneficial in many NN

After BN the input to the activation function is in the sweet spot

Observed distributions of signal after BN before going into the activation layer.



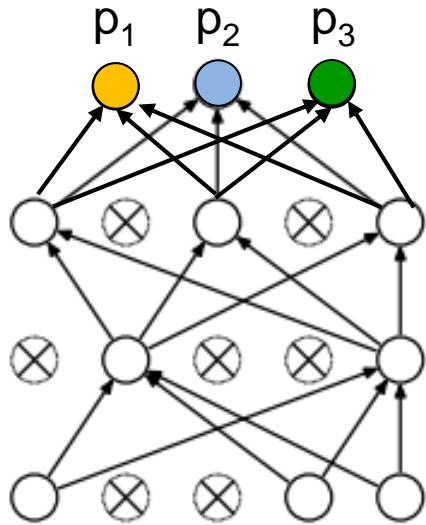
When using BN consider the following:

- Using a higher learning rate might work better
- Use less regularization, e.g. reduce dropout probability

Image credits: Martin Gorner:

[https://docs.google.com/presentation/d/e/2PACX-1vRouwj\\_3cYsmLrNNI3Uq5gv5-hYp\\_QFdeoan2GlxKgIZRSejozruAbVV0IMXBoPsINB7Jw92vJo2EAM/pub?slide=id.g187d73109b\\_1\\_2921](https://docs.google.com/presentation/d/e/2PACX-1vRouwj_3cYsmLrNNI3Uq5gv5-hYp_QFdeoan2GlxKgIZRSejozruAbVV0IMXBoPsINB7Jw92vJo2EAM/pub?slide=id.g187d73109b_1_2921)

# Dropout helps to fight overfitting

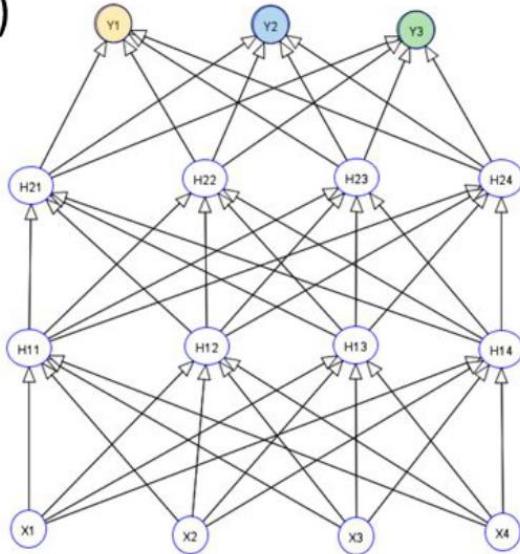


Using **dropout** during training implies:

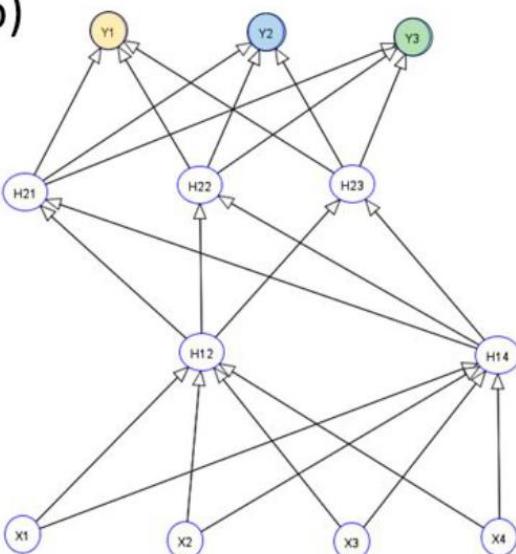
- In each training step only weights to not-dropped units are updated → we train a sparse sub-model NN
- For predictions with the trained NN we freeze the weights corresponding to averaging over the ensemble of trained models we should be able to “reduce noise”, “overfitting”

# Dropout: kind of NN ensemble

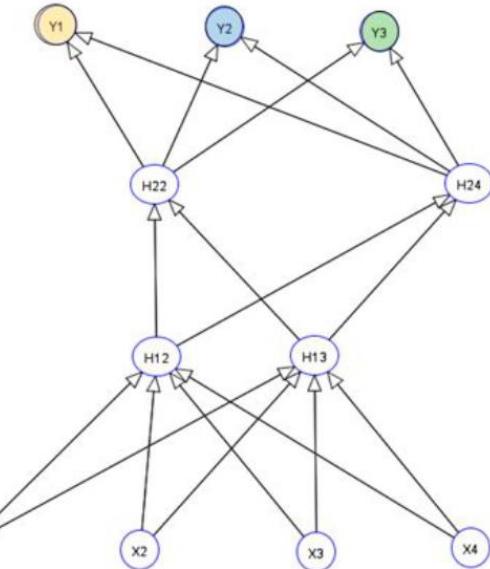
a)



b)



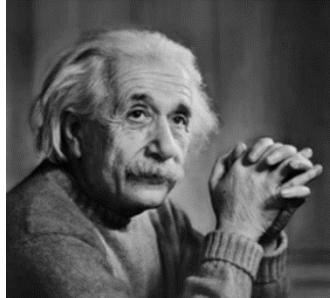
c)



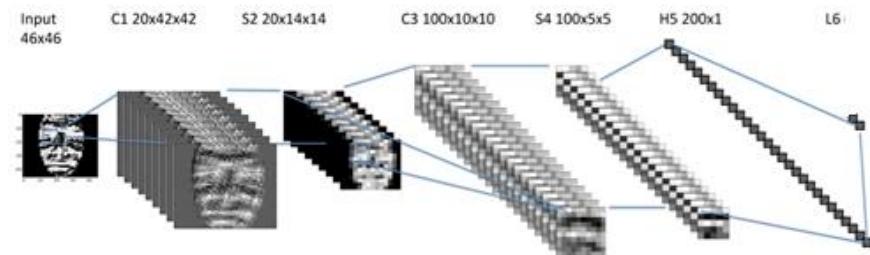
Three NNs: a) shows the full NN with all neurons, b) and c) show two versions of a thinned NN where some neurons are dropped. Dropping neurons is the same as setting all connections that start from these neurons to zero

## Another intuition: Why “dropout” can be a good idea

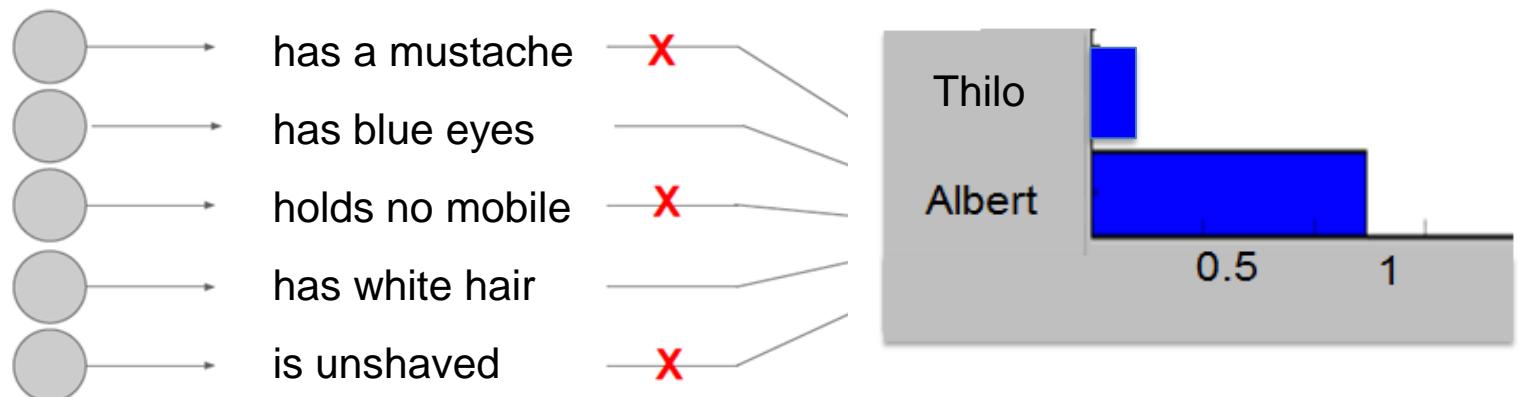
The training data consists of many different pictures of Thilo and Einstein



We need a huge number of neurons to extract good features which help to distinguish Thilo from Einstein

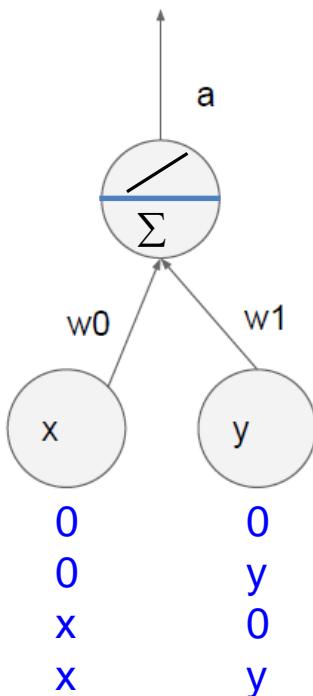


Dropout forces the network to learn redundant and independent features



# Dropout-trained NN require weight adaptation

Use the trained net without dropout during test time



Q: Suppose no dropout during test time ( $x, y$  are never dropped to zero), but a dropout probability  $p=0.5$  during training

What is the expected value for the output **a** of this neuron?

during test  
w/o dropout:

$$a = w_0 * x + w_1 * y$$

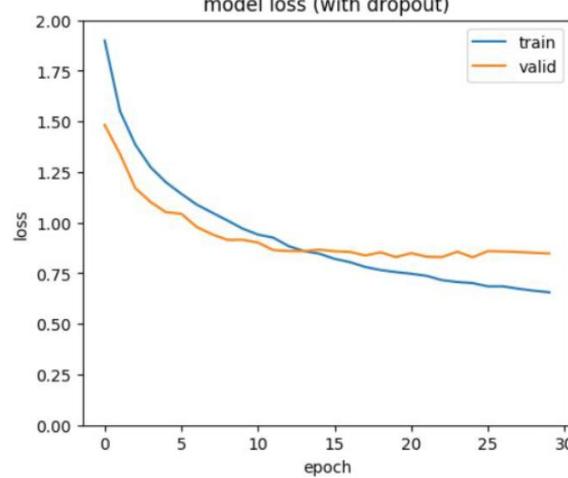
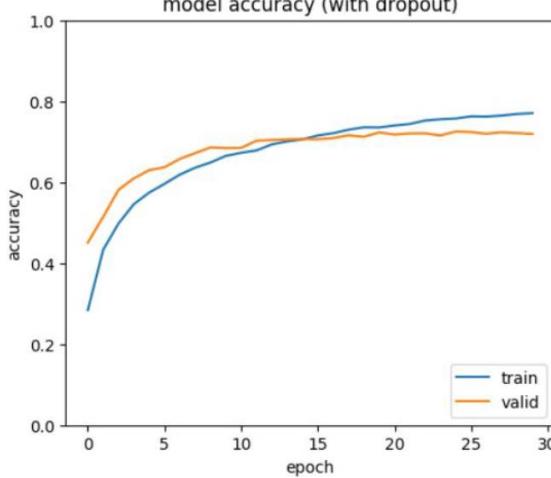
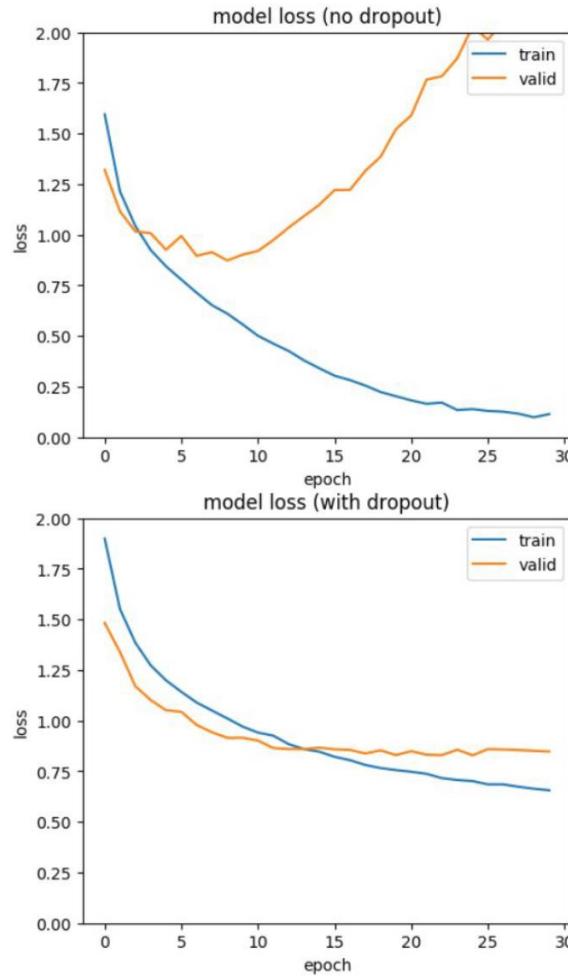
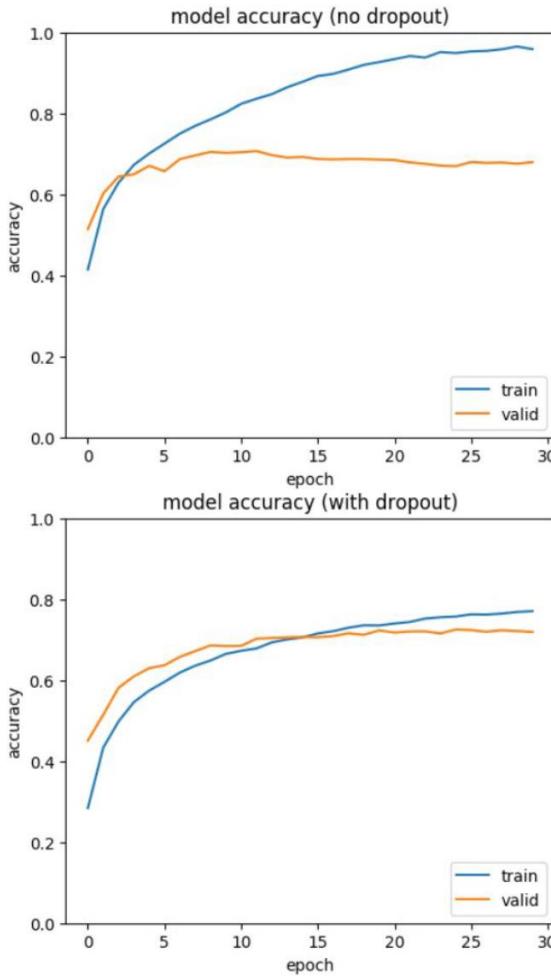
during training  
with dropout  
probability 0.5:

$$\begin{aligned} E[a] &= \frac{1}{4} * (w_0 * 0 + w_1 * 0 + \\ &\quad w_0 * 0 + w_1 * y + \\ &\quad w_0 * x + w_1 * 0 + \\ &\quad w_0 * x + w_1 * y) \\ &= \frac{1}{4} * (2 w_0 * x + 2 w_1 * y) \\ &= \frac{1}{2} * (w_0 * x + w_1 * y) \end{aligned}$$

=> To get same expected output in training and test time, we reduce the weights during test time by multiplying them by the dropout probability  $p=0.5$



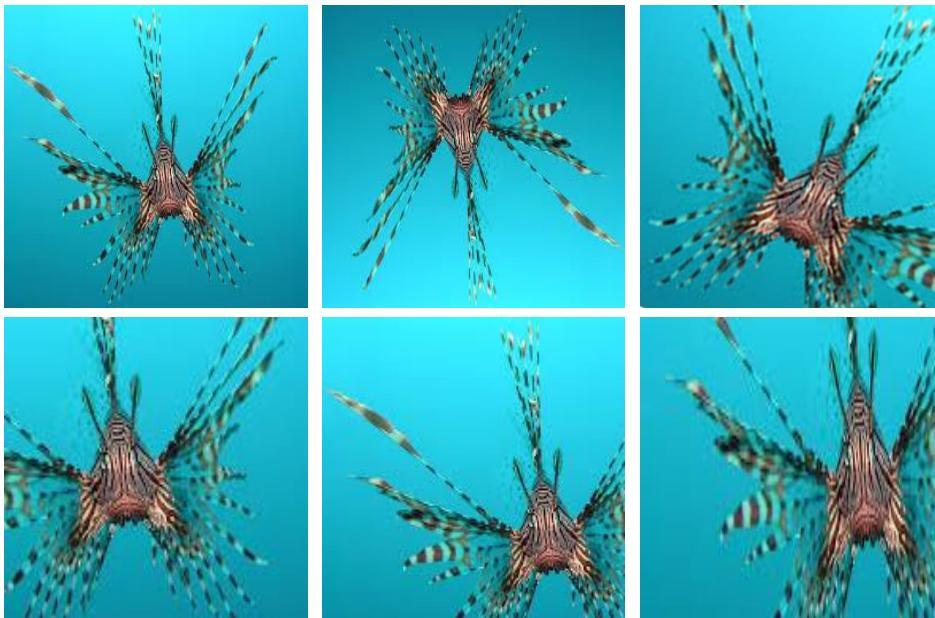
# Dropout fights overfitting in a CIFAR10 CNN



```
model = Sequential()  
  
model.add(Convolution2D(16, (3,3),  
activation="relu",padding="same",  
input_shape=(32,32,3)))  
model.add(Convolution2D(16, (3,3),  
activation="relu",padding="same"))  
model.add(MaxPooling2D((2,2)))  
  
model.add(Convolution2D(32, (3,3),  
activation="relu",padding="same"))  
model.add(Convolution2D(32, (3,3),  
activation="relu",padding="same"))  
model.add(MaxPooling2D((2,2)))  
  
model.add(Flatten())  
model.add(Dropout(0.3))  
model.add(Dense(500))  
model.add(Activation('relu'))  
model.add(Dropout(0.3))  
model.add(Dense(300))  
model.add(Activation('relu'))  
model.add(Dropout(0.3))  
model.add(Dense(100))  
model.add(Activation('relu'))  
  
model.add(Dense(10))  
model.add(Activation('softmax'))
```

# Fighting overfitting by Data augmentation ("always" done): "generate more data" on the fly during fitting the model

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

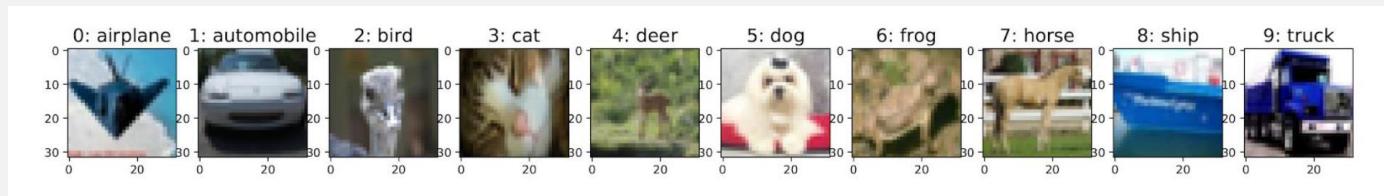


Data augmentation in Keras:

```
from keras.preprocessing.image import ImageDataGenerator  
  
datagen = ImageDataGenerator(  
    rotation_range=10,  
    width_shift_range=0.1,  
    height_shift_range=0.1,  
    horizontal_flip=True,  
    vertical_flip=True,  
    #zoom_range=[0.1,0.1]  
)  
  
train_generator = datagen.flow(  
    x = X_train_new,  
    y = Y_train,  
    batch_size = 128,  
    shuffle = True)  
  
history = model.fit_generator(  
    train_generator,  
    samples_per_epoch = X_train_new.shape[0],  
    epochs = 400,  
    validation_data = (X_valid_new, Y_valid),  
    verbose = 2,callbacks=[checkpointer]  
)
```



# Exercise: Tricks of the Trade: NB 07



Work through the Notebook 07 “dropout” section on:

[07\\_cifar10\\_tricks\\_sol.ipynb](#)

## Content:

- load the original cifar10 data create a train val and test dataset
- visualize samples of cifar10 dataset
- train a random forest on the pixelvalues
- train a cnn from scratch without normalization
- train a cnn from scratch with normalization
- train a cnn from scratch with dropout
- train a cnn from scratch with batchnorm
- train a cnn from scratch with data augmentation
- compare the performances of the models

# Challenge winning CNN architectures

# LeNet-5 1998: first CNN for ZIP code recognition

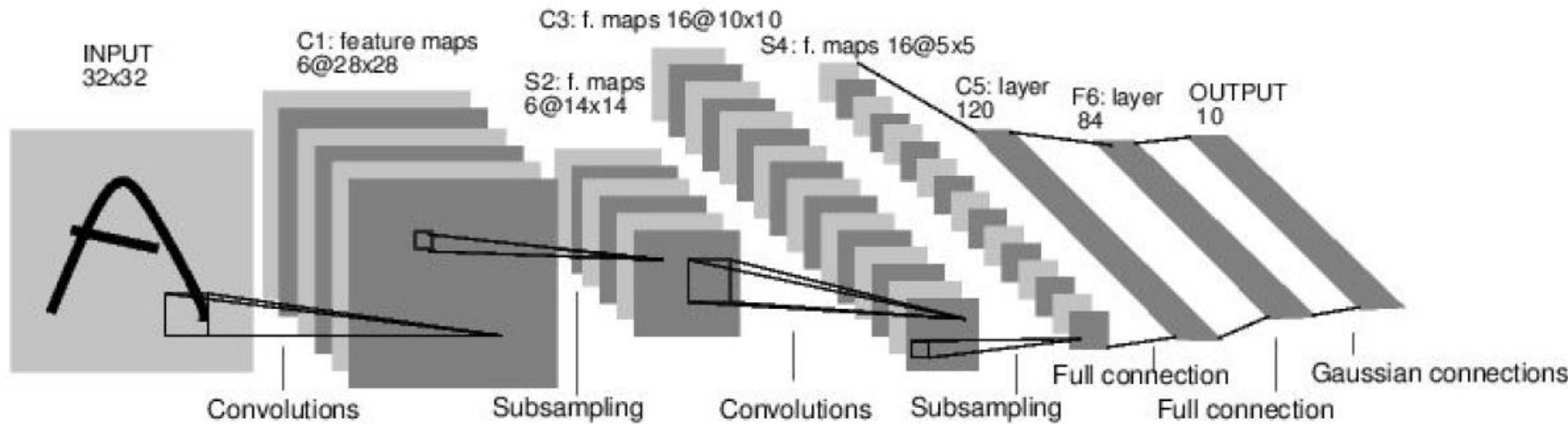
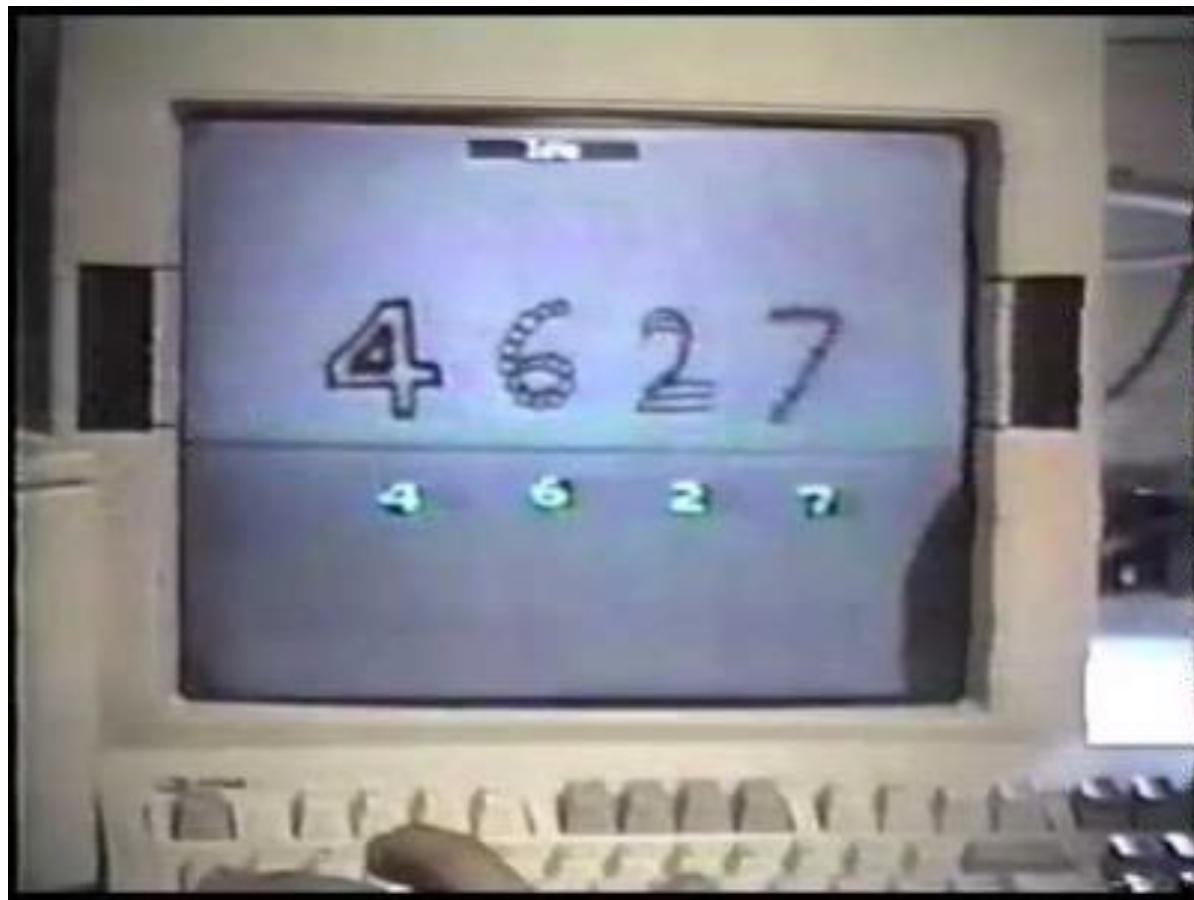


Image credits: <http://yann.lecun.com/exdb/publis/pdf/lecun-98.pdf>

Conv filters were 5x5, applied at stride 1

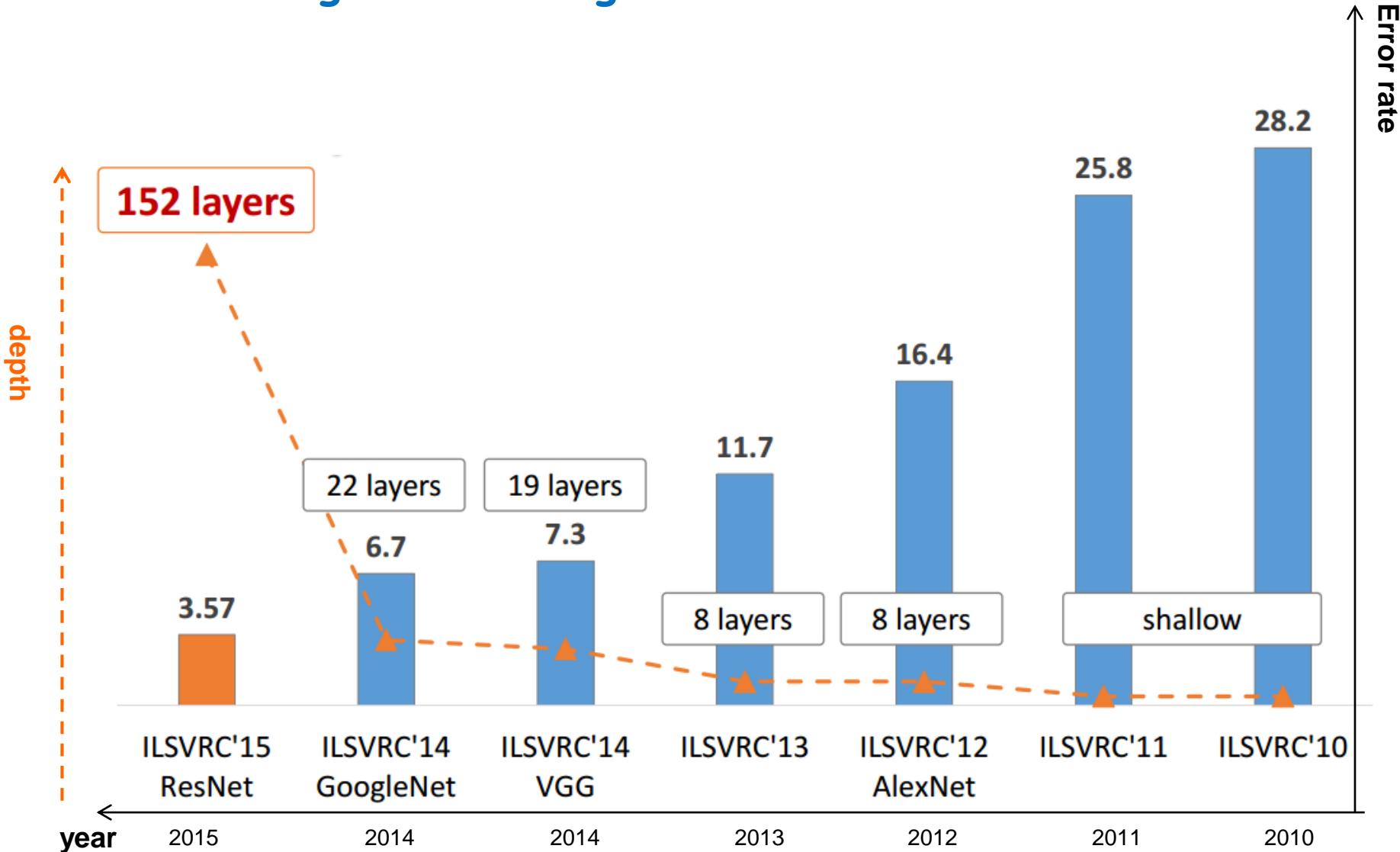
Subsampling (Pooling) layers were 2x2 applied at stride 2  
i.e. architecture is [CONV-POOL-CONV-POOL-CONV-FC]

# The first CNN (Yan LeCun)



[https://www.youtube.com/watch?v=FwFduRA\\_L6Q](https://www.youtube.com/watch?v=FwFduRA_L6Q)

# Review of ImageNet winning CNN architectures

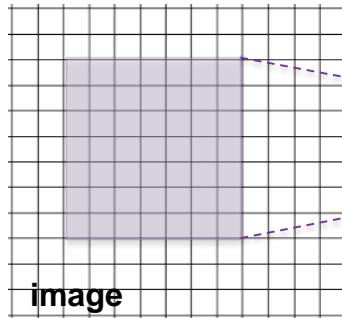


# The trend in modern CNN architectures goes to small filters

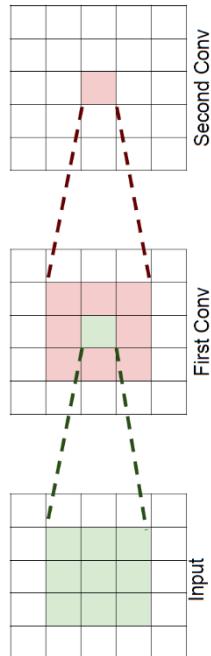
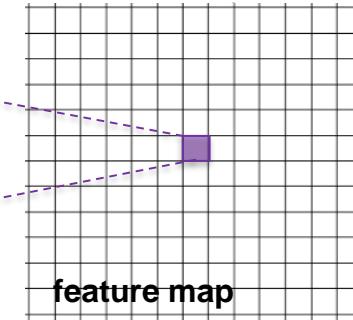
Why do modern architectures use very small filters?

Determine the receptive field in the following situation:

- 1) Suppose we have one  
7x7 conv layers (stride 1)  
49 weights

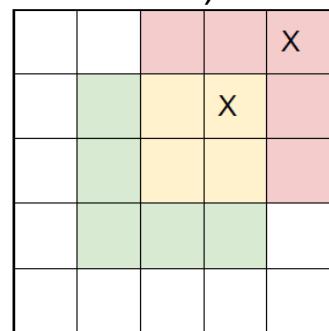


Answer1): 7x7



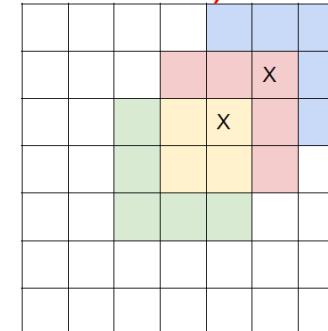
- 2) Suppose we stack two  
3x3 conv layers (stride 1)

Answer 2): 5x5



- 3) Suppose we stack three  
3x3 conv layers (stride 1)  
 $3 \times 9 = 27$  weights

Answer 3): 7x7



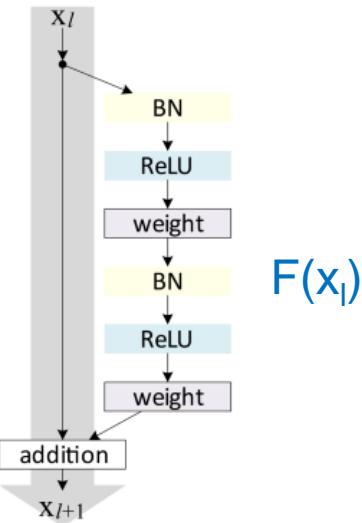
We need less weights for the same receptive field when stacking small filters!

# "ResNet" from Microsoft 2015 winner of imageNet

152  
layers

ResNet basic design (VGG-style)

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



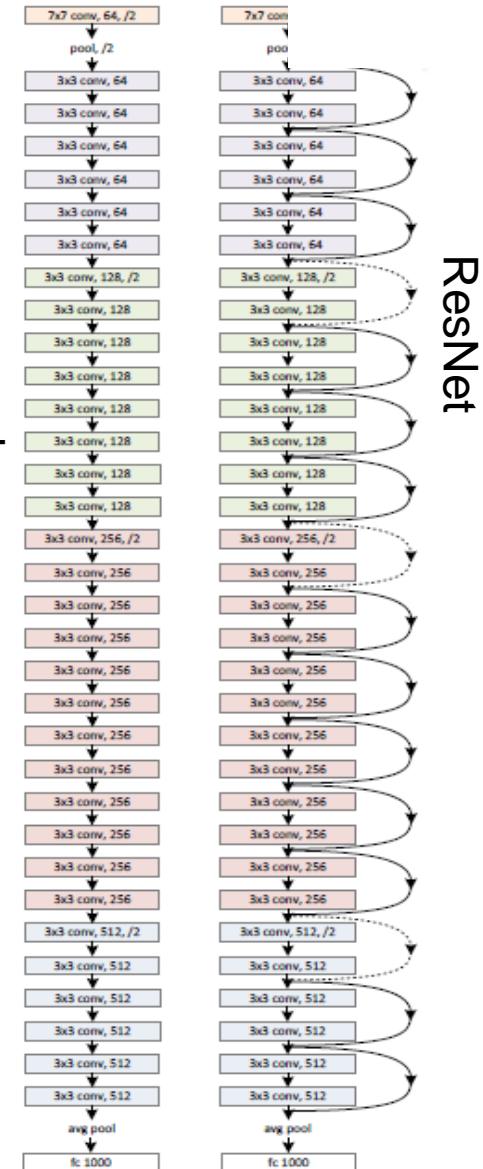
$$H(x_i) = x_{i+1} = x_i + F(x_i)$$

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

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!

plain VGG



# “Oxford Net” or “VGG Net” 2<sup>nd</sup> place

- 2<sup>nd</sup> place in the imageNet challenge
- More traditional, easier to train
- More weights than GoogLeNet
- Small pooling
- Stacked 3x3 convolutions before maxpooling  
-> large receptive field
- no strides (stride 1)
- ReLU after conv. and FC (batchnorm was not used)
- Pre-trained model is available



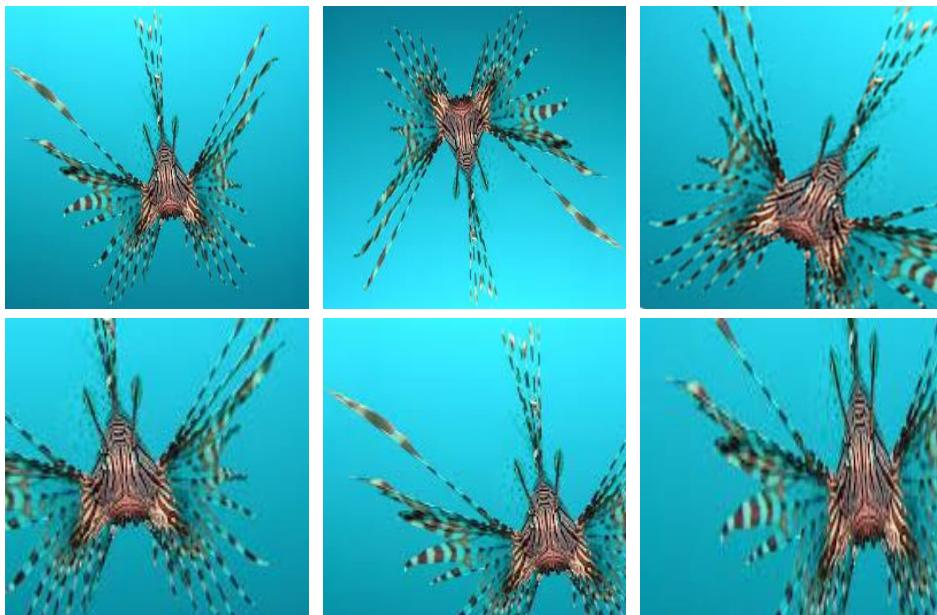
<http://arxiv.org/abs/1409.1556>

# What to do in case of limited data?

- Do augmentation during training of a CNN
- Use shallow learner (e.g. RF) based on image features extracted via pretrained CNNs
- Fine-tune a pretrained CNN on few data (transfer learning)

# Recall: Augmentation

- Generate “more data from the available data” by augmentation



Data augmentation in Keras:

```
from keras.preprocessing.image import ImageDataGenerator

datagen = ImageDataGenerator(
    rotation_range=10,
    width_shift_range=0.1,
    height_shift_range=0.1,
    horizontal_flip=True,
    vertical_flip=True,
    #zoom_range=[0.1,0.1]
)

train_generator = datagen.flow(
    x = X_train_new,
    y = Y_train,
    batch_size = 128,
    shuffle = True)

history = model.fit_generator(
    train_generator,
    samples_per_epoch = X_train_new.shape[0],
    epochs = 400,
    validation_data = (X_valid_new, Y_valid),
    verbose = 2,callbacks=[checkpointer]
)
```

# Use pre-trained CNNs for feature extraction



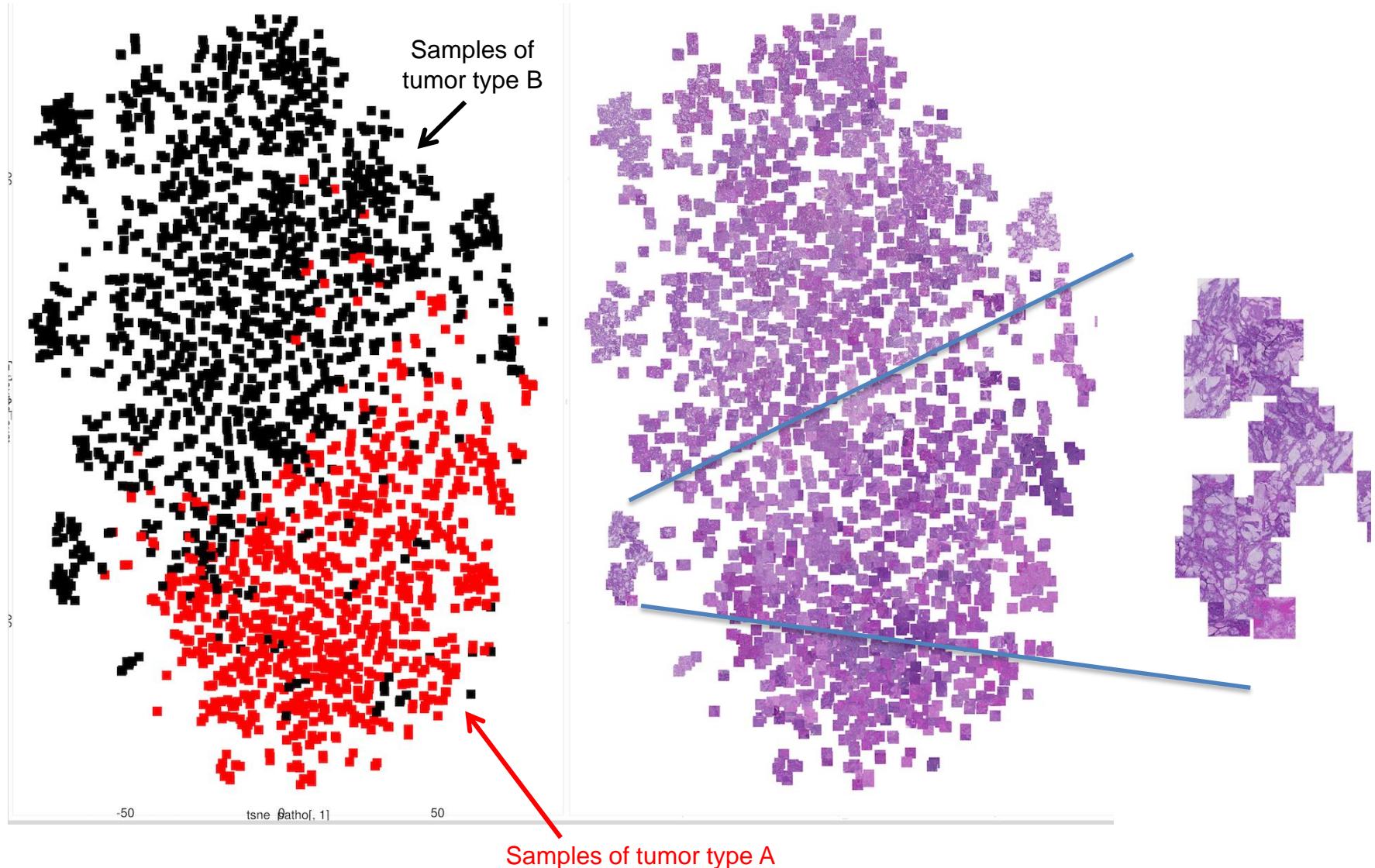
- Load a pre-trained CNN – e.g. VGG16
- Resize image to required size (224x224 for VGG16)
- Rescaling of the pixel values to “VGG range”
- Do a forward pass and **fetched features** that are used as CNN representations, dump these features into a file on disk
- Use these CNN features as input to a simple classifier – e.g. fc NN, RF, SVM ...  
(here it is easily possible to adapt to the new number of class labels)

Number features depends on input shape

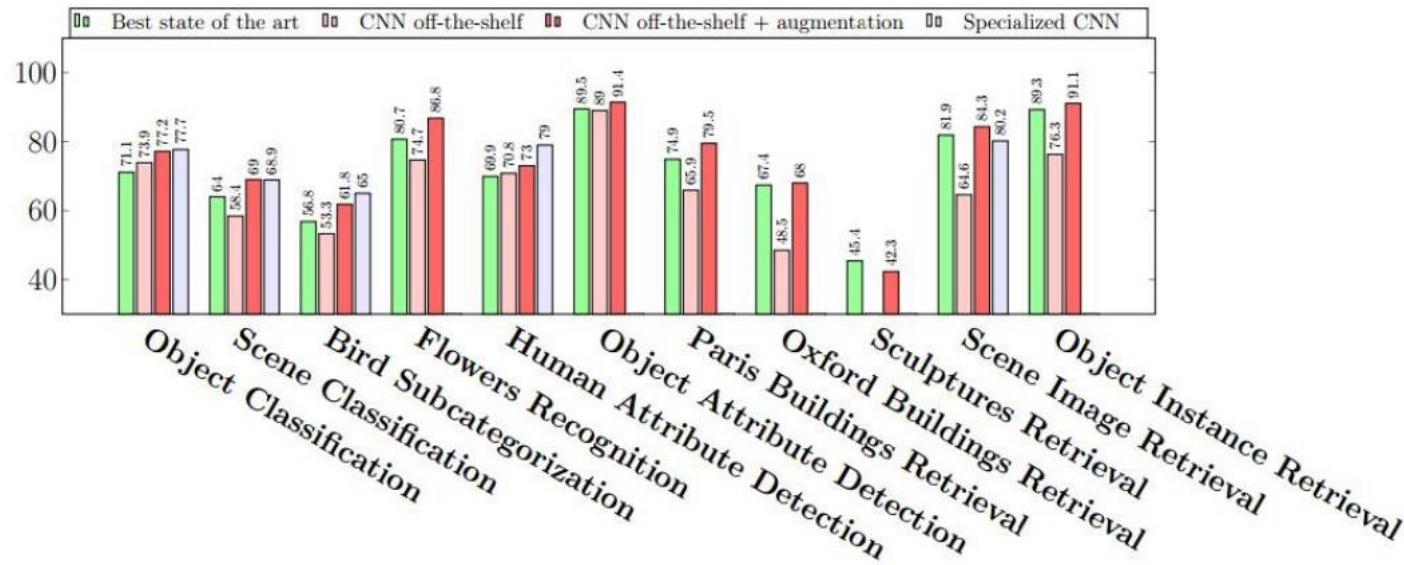
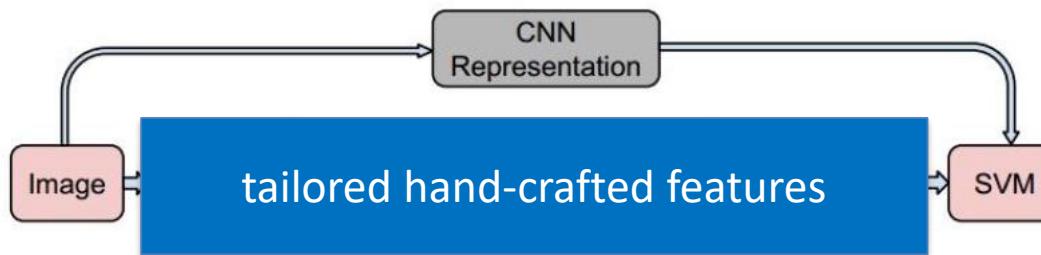
4096 feature

Fetch this CNN feature vector for each image

# Are VGG features useful?



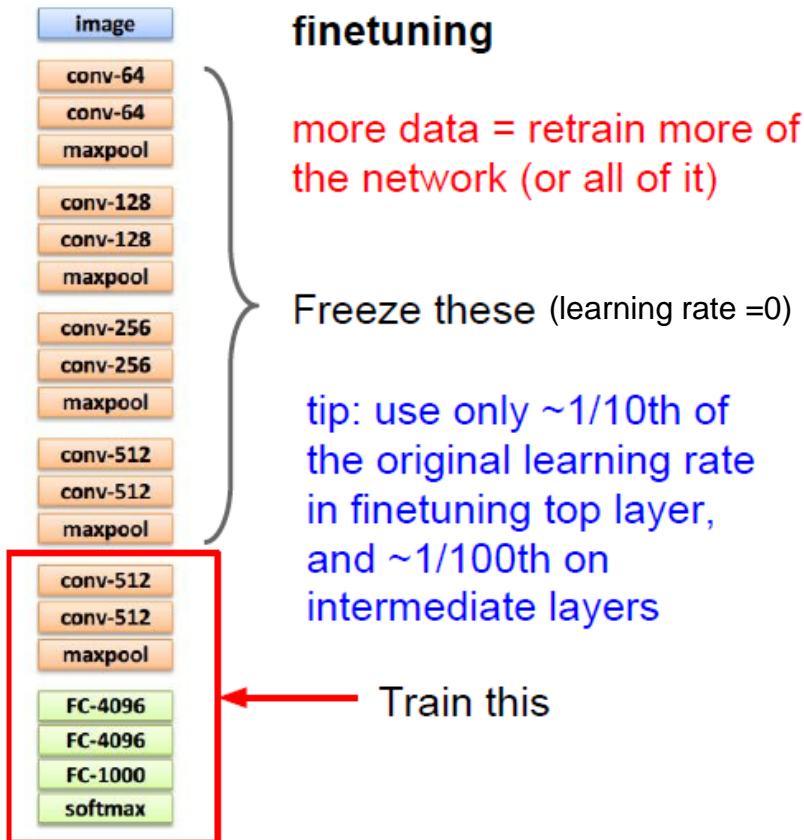
# Performance of off-the-shelf CNN features when compared to tailored hand-crafted features



“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.”

# 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:

|                     | <b>Similar task</b><br>(to imageNet challenge)  | <b>Very different task</b><br>(to imageNet challenge)   |
|---------------------|---|---|
| <b>little data</b>  | 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 |
| <b>lots of data</b> | Fine-tune a few layers including few convolutional layers   | Fine-tune a large number of layers  |

Hint: first retrain only fully connected layer, only then add convolutional layers for fine-tuning.

# Where to find pretrained networks

- <https://tfhub.dev/>

The screenshot shows the TensorFlow Hub website interface. On the left, there's a sidebar with categories: Text (Text, Embedding), Image (Classification, Feature Vector, Generator, Other), Video (Classification), and Publishers (Google, DeepMind). The main content area has a search bar at the top right. Below it, under "Text embedding", there are three results:

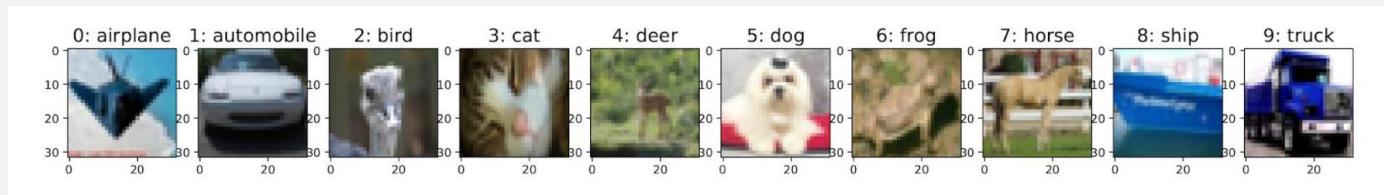
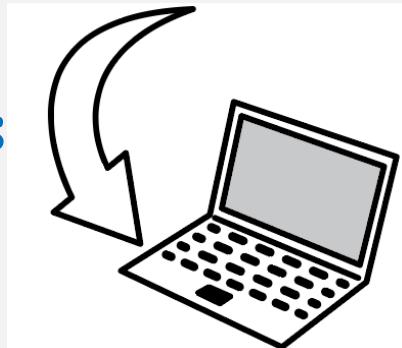
- universal-sentence-encoder-multilingual-large** By Google  
Transformer text-embedding  
16 languages (Arabic, Chinese-simplified, Chinese-traditional, English, French, German, Italian, Japanese, Korean, Dutch, Polish, Portuguese, Spanish, Thai, Turkish, Russian) text encoder.
- universal-sentence-encoder-large** By Google  
English Transformer text-embedding  
Encoder of greater-than-word length text trained on a variety of data.
- universal-sentence-encoder** By Google  
English DAN text-embedding  
Encoder of greater-than-word length text trained on a variety of data.

Below these, there's a link to "View more text embeddings". Under "Image feature vectors", there are two results:

- imagenet/inception\_v3/feature\_vector** By Google  
Inception V3 ImageNet (ILSVRC-2012-CLS) image-feature-vector  
Feature vectors of images with Inception V3 trained on ImageNet (ILSVRC-2012-CLS).
- tf2-preview/mobilenet\_v2/feature\_vector** By Google  
MobileNet V2 ImageNet (ILSVRC-2012-CLS) image-feature-vector  
[TF2] Feature vectors of images with MobileNet V2 trained on ImageNet (ILSVRC-2012-CLS).

# Exercise:

## 08\_classification\_transfer\_learning\_few\_labels



Notebook: [08\\_classification\\_transfer\\_learning\\_few\\_labels.ipynb](#)



### Learning with few data

In case of few data you can work with features that you extract from a pretrained cnn. Data augmentation increases the training data and usually help to improve the performance.

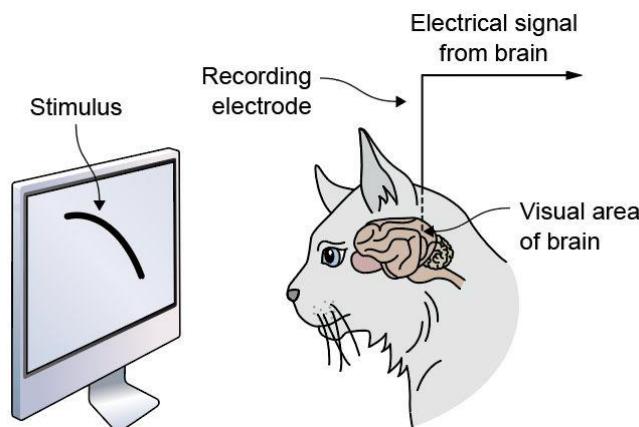
#### Learning with few data

Baseline model: RF on VGG features

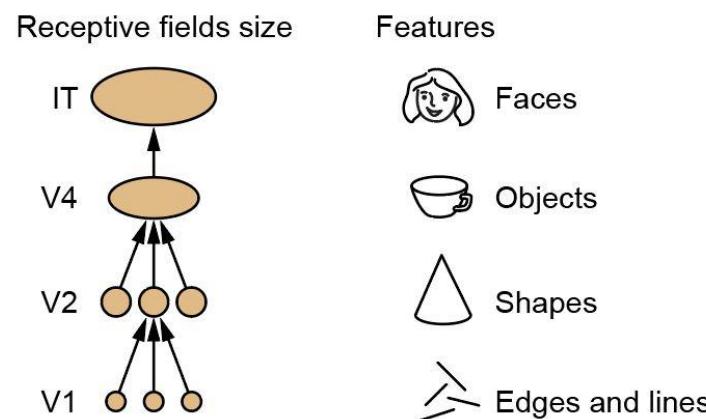
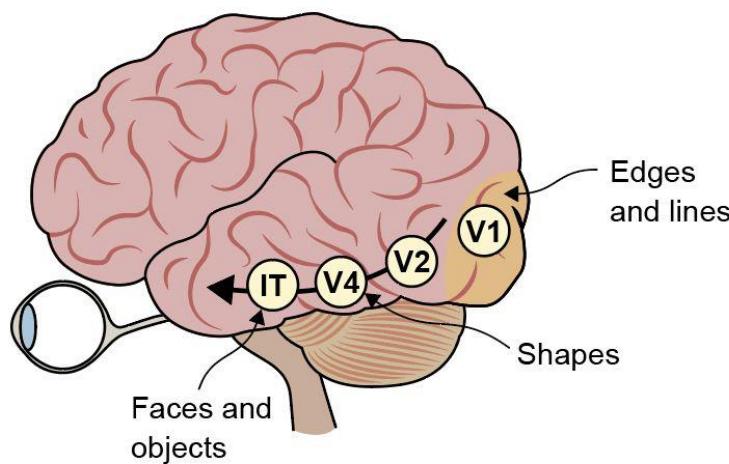
Transfer learning

# Biological Inspiration of CNNs

# How does the brain respond to visually received stimuli?

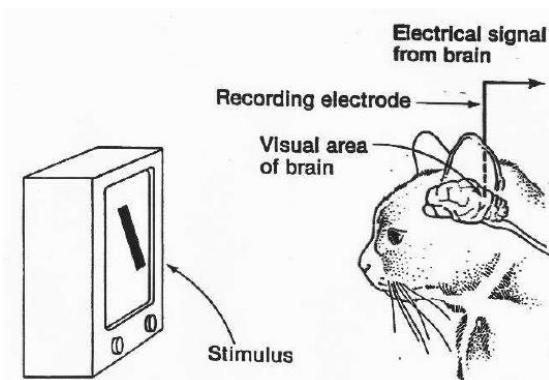


Setup of the experiment of Hubel and Wiesel in late 1950s in which they discovered **neurons** in the visual cortex that **responded** when moving **edges** were shown to the cat.

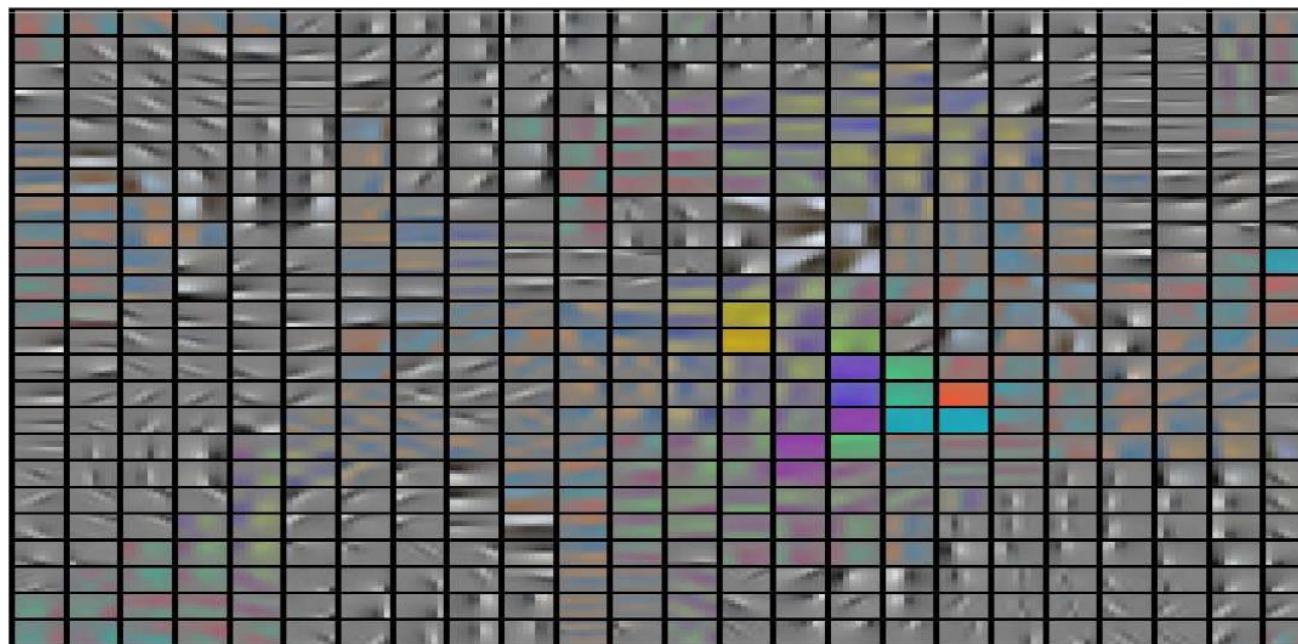
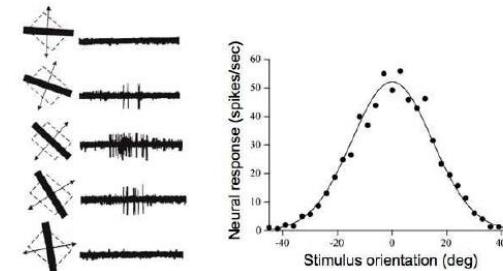


Organization of the visual cortex in a brain, where neurons in different regions respond to more and more complex stimuli

# Compare neurons in brain region V1 in first layer of a CNN



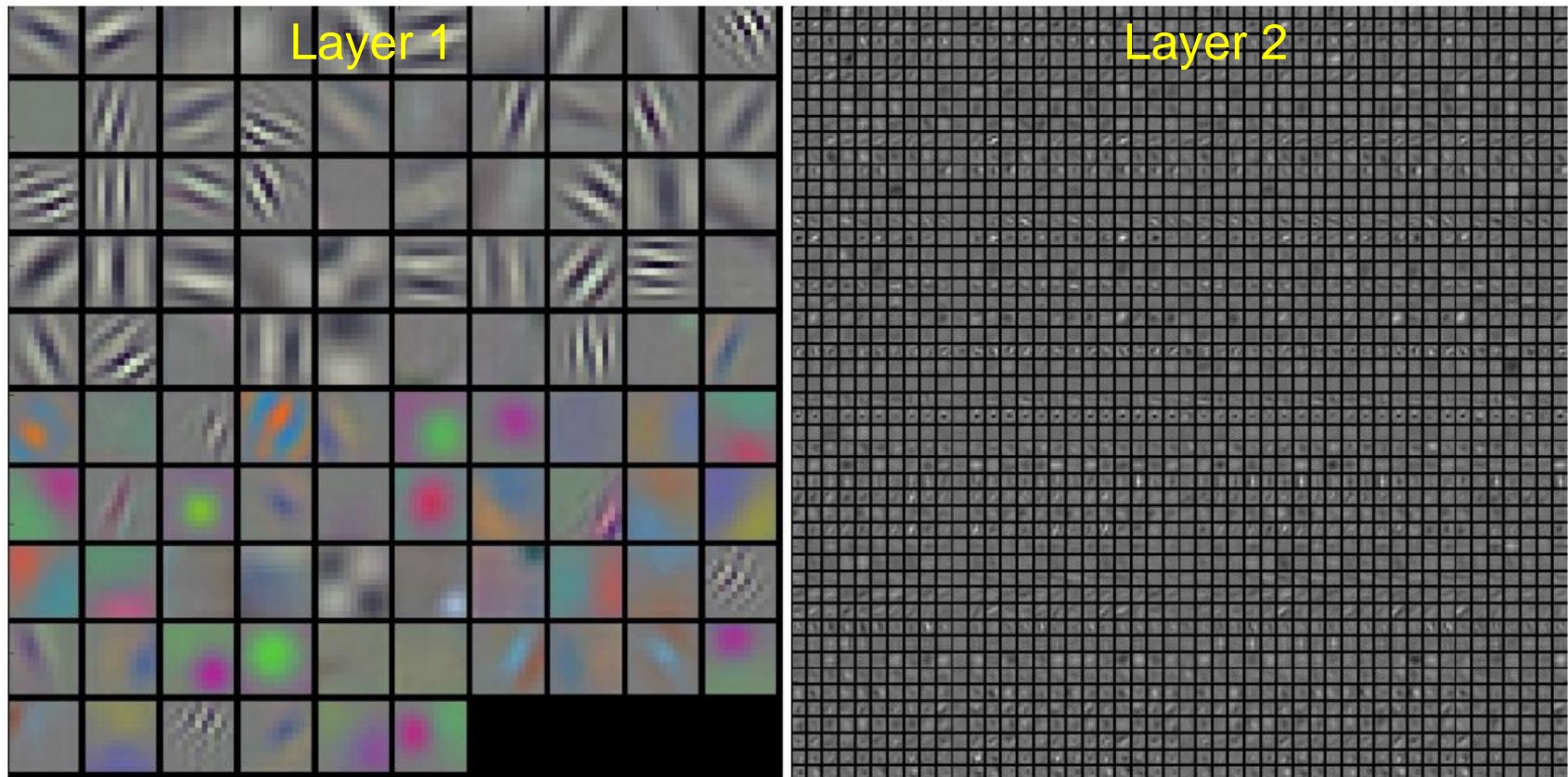
V1 physiology: orientation selectivity



Neurons in brain region V1 and neurons in 1. layer of a CNN respond to similar patterns

# Visualize the weights used in filters

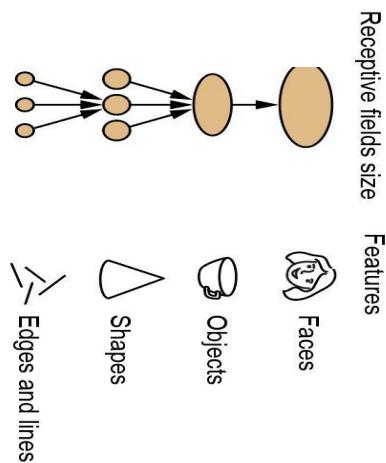
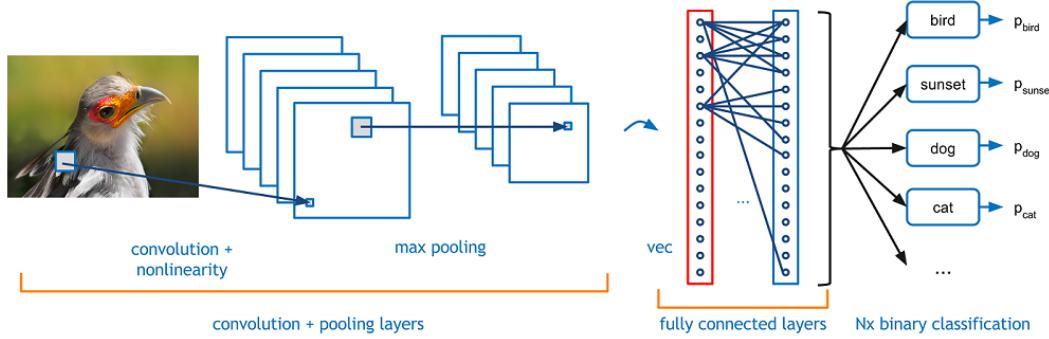
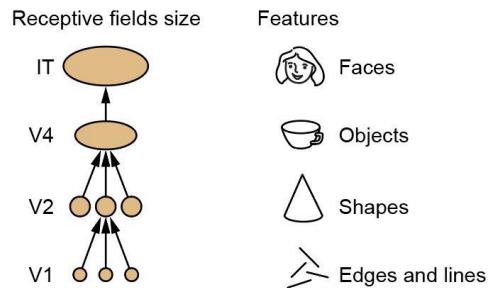
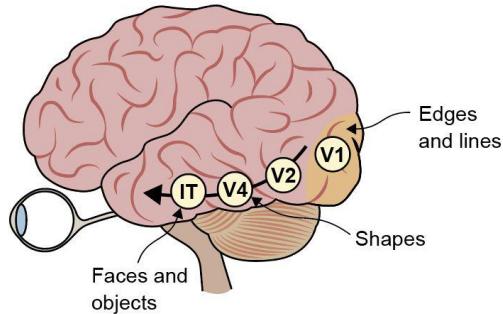
Filter weights from a trained Alex Net



Only in layer 1 the filter pattern correspond to extracted patterns in the image.

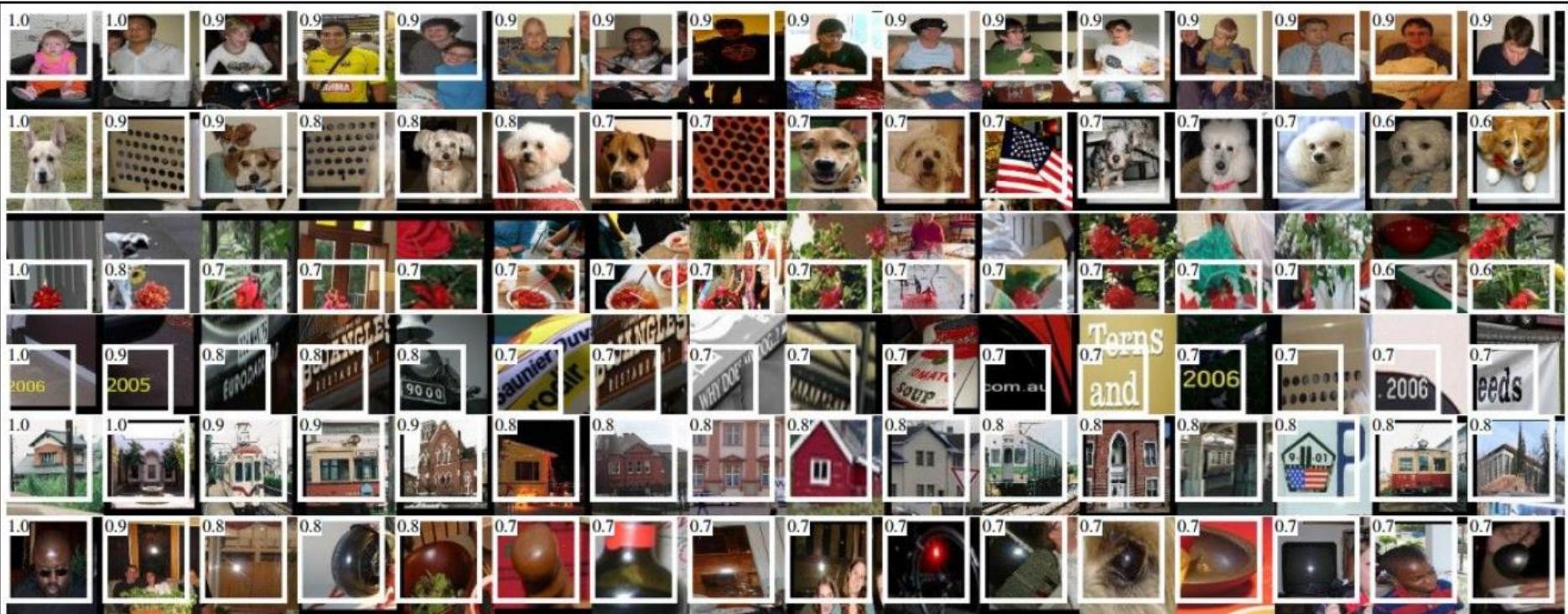
In higher layers we can only check if patterns look noisy, which would indicate that the network that hasn't been trained for long enough, or possibly with a too low regularization strength that may have led to overfitting.

# Weak analogies between brain and CNNs architecture

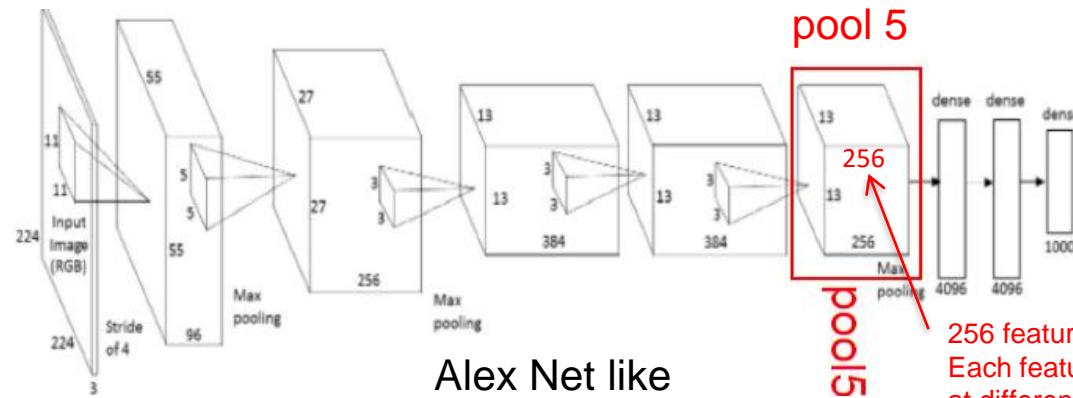


# Where does a CNN look at?

# Visualize patches yielding high values in activation maps



**Figure 4: Top regions for six pool<sub>5</sub> units.** Receptive fields and activation values are drawn in white. Some units are aligned to concepts, such as people (row 1) or text (4). Other units capture texture and material properties, such as dot arrays (2) and specular reflections (6).



Here we show image patches that activate maps in layer 5 most.

256 feature maps generated by 256 different 3x3 filters. Each feature map consists of equivalent neurons looking at different positions of the input volume.

# What kind of image (patches) excites a certain neuron corresponding to a large activation in a feature map?

10 images from data set leading to high signals 6 feature maps of **conv6**



10 images from data set leading to high signals 6 feature maps of **conv9**

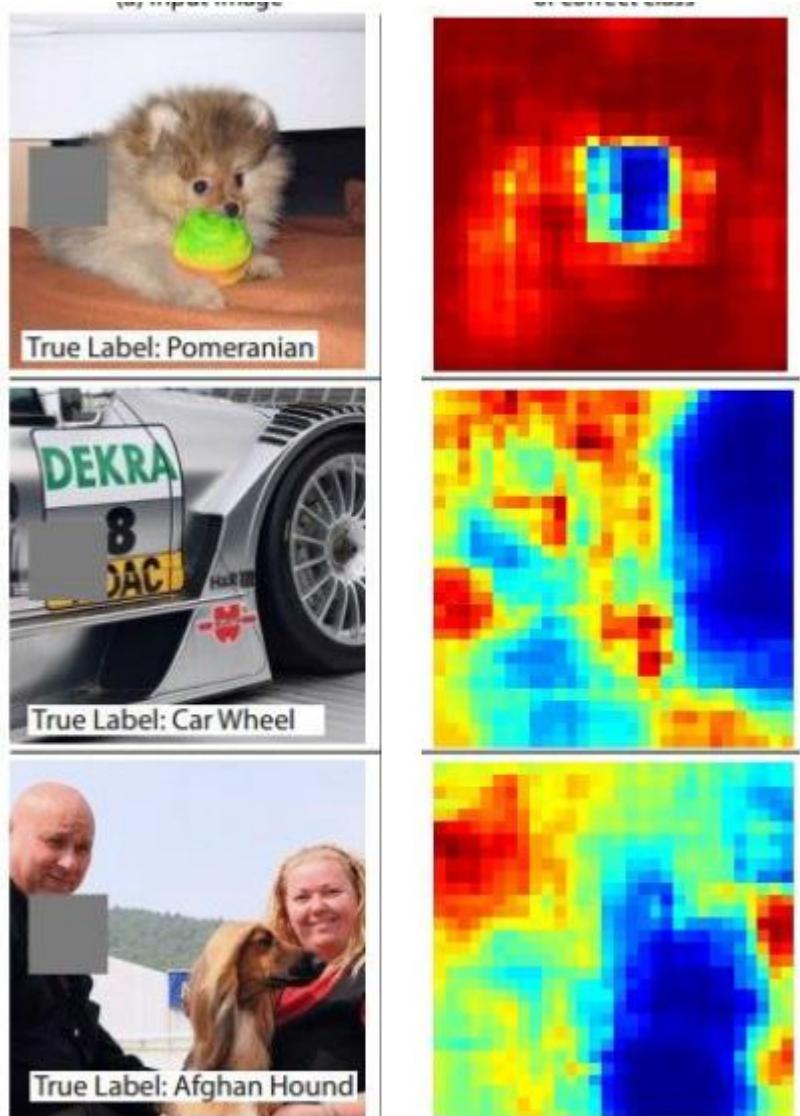


# Which pixels are important for the classification?

## Occlusion experiments

Occlude part of the image with a mask and check for each position of the mask how strongly the score for the correct class is changing.

Warning:  
Usefulness depends on application...



Occlusion experiments [\[Zeiler & Fergus 2013\]](#)

image credit: cs231n

# Which pixels are important for the classification?

## LIME: Local Interpretable Model-agnostic Explanations

### Idea:

- 1) perturb interpretable features of the instance – e.g. randomly delete super-pixels in an image and track as perturbation vector such as  $(0,1,1,0,\dots,1)=x$ .
- 2) Classify perturbed instance by your model, here a CNN, and track the achieved classification-score= $y$
- 3) Identify for which features/super-pixels the presence in the perturbed input version are important to get a high classification score (use RF or lasso for  $y \sim x$ )

<https://arxiv.org/abs/1602.04938>



(a) Husky classified as wolf



(b) Explanation

-> presence of snow was used to distinguish wolf and husky

-> Explain the CNN classification by showing instance-specific important features  
visualize important feature allows to judge the individual classification



(a) Original Image



(b) Explaining Electric guitar



(c) Explaining Acoustic guitar

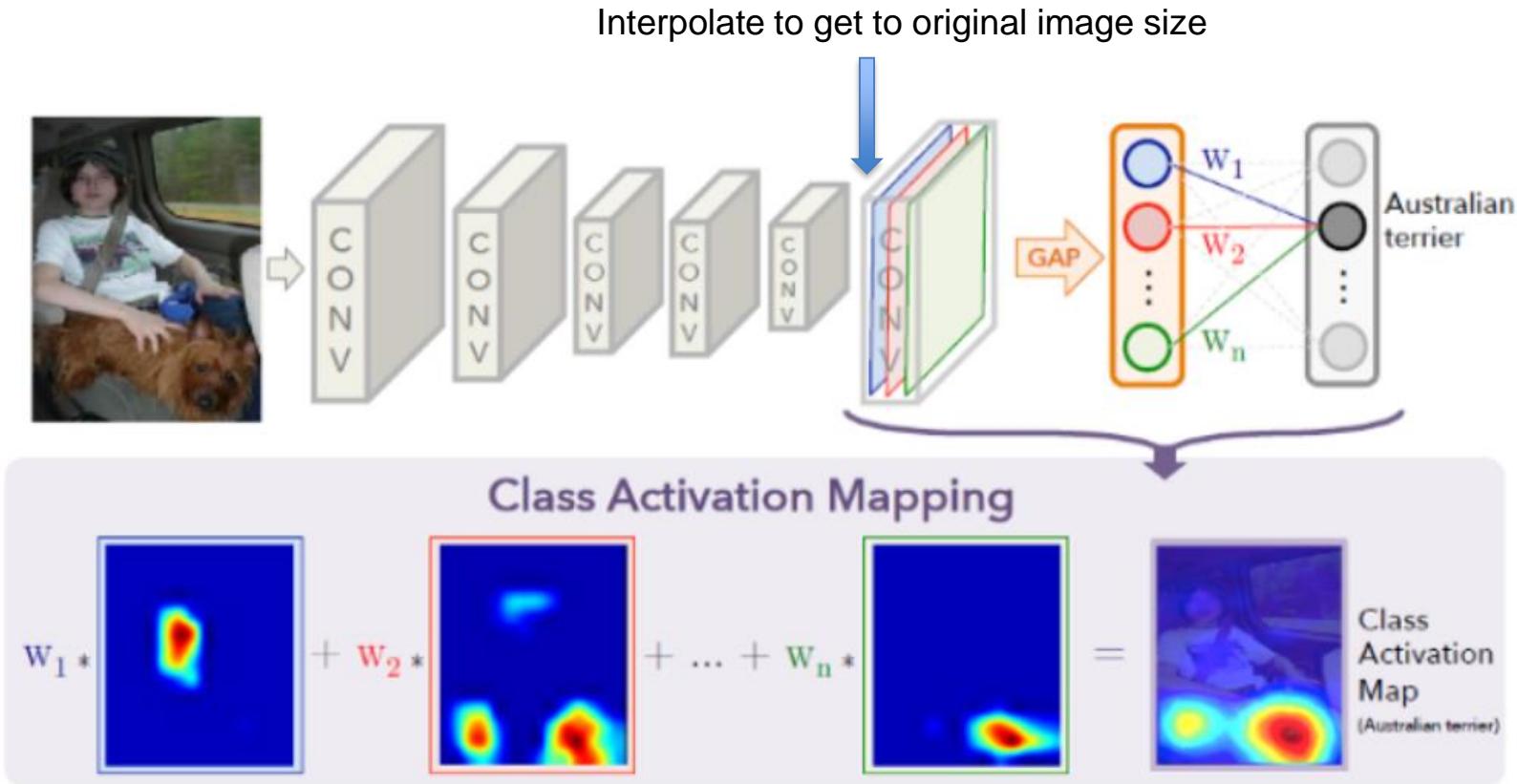


(d) Explaining Labrador

Tf, python code: <https://github.com/marcotcr/lime> -> image tutorial

<https://github.com/marcotcr/lime/blob/master/doc/notebooks/Tutorial%20-%20Image%20Classification%20Keras.ipynb>

# Class Activation Mapping (CAM)

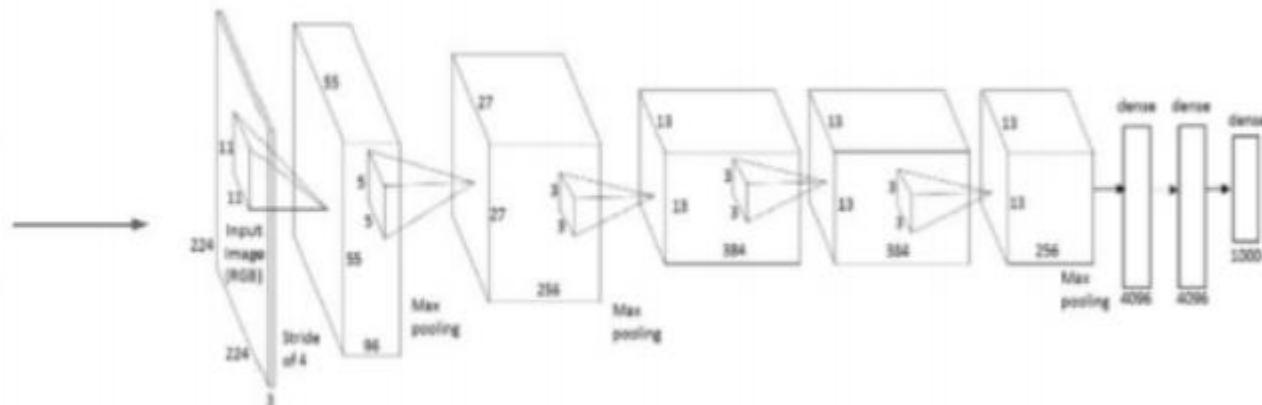


We compute the CAM by multiplying each feature map of the final convolution layer with the corresponding weight connected to the neuron of the winning class.

<https://arxiv.org/abs/1512.04150>

<https://towardsdatascience.com/class-activation-mapping-using-transfer-learning-of-resnet50-e8ca7cf657e>

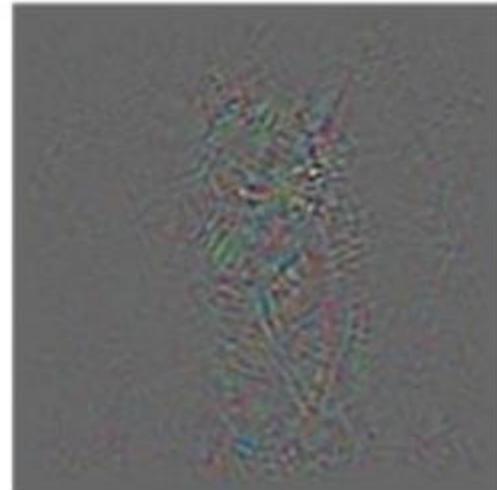
# Gradient Backpropagation



- 1) Do a forward pass with the image
- 2) Compute the gradient of the winning class neuron using backprop

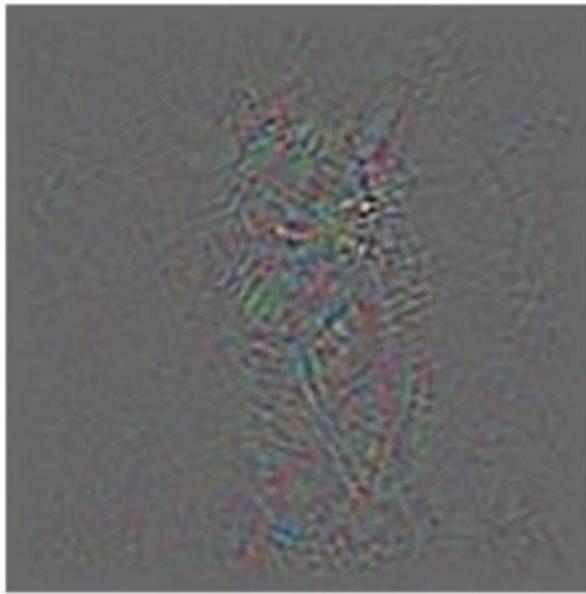
Which pixels would make the class neuron not turn on if they had been different?

In other words, for which inputs is  $\partial(\text{class} - \text{neuron})/\partial x_i$  large?

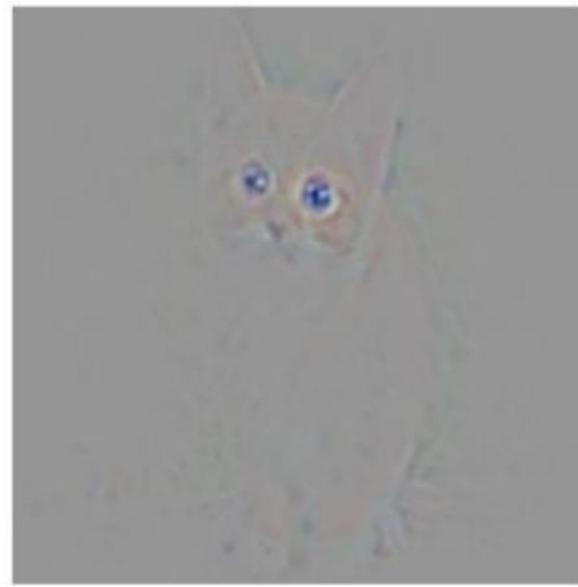


# Guided Gradient Backpropagation

- We are **only interested to see which image features the neuron detects**, not in what kind of stuff it doesn't detect
- So **only propagating positive gradients** (set all the negative gradients to 0)



Backprop



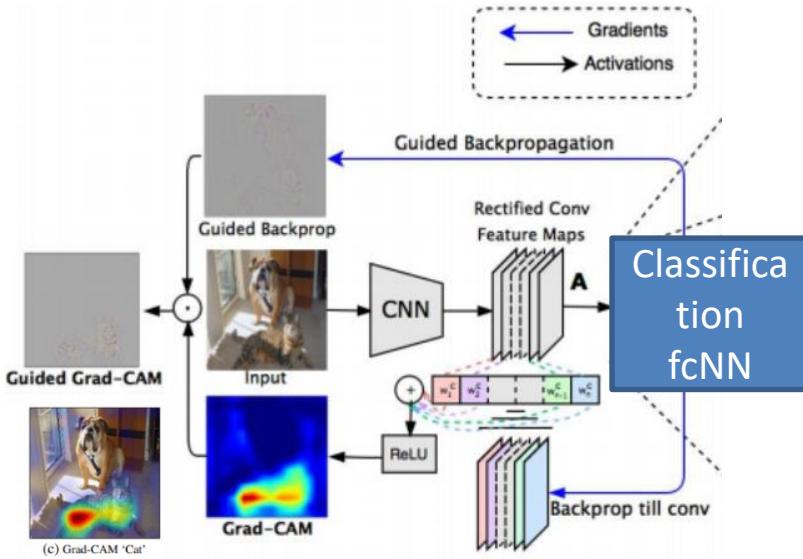
Guided Backprop

<https://www.cs.toronto.edu/~guerzhoy/321/lec/W07/HowConvNetsSee.pdf>

Code: see iNNvestigate colab NB [https://github.com/avciidp/explainable\\_ai/blob/master/iNNvestigate\\_fashion\\_light.ipynb](https://github.com/avciidp/explainable_ai/blob/master/iNNvestigate_fashion_light.ipynb)  
paper: <https://arxiv.org/abs/1808.04260>

# Grad-Cam: Gradient based Class Activation maps

Grad-Cam: Use Guided Backprop to scale contributions in CAMs



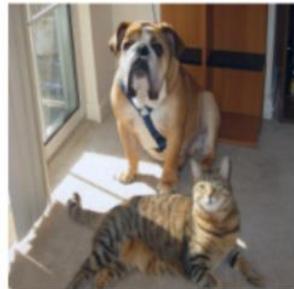
- Set gradient to zero for all classes except the predicted class
- Backpropagate this gradient to the last conv-layer  
-> get coarse grad-CAM localization
- Multiply coarse grad-CAM with the guided back-prop signal

<https://arxiv.org/abs/1610.02391>

Code: [https://keras.io/examples/vision/grad\\_cam/](https://keras.io/examples/vision/grad_cam/)

<https://www.pyimagesearch.com/2020/03/09/grad-cam-visualize-class-activation-maps-with-keras-tensorflow-and-deep-learning/>

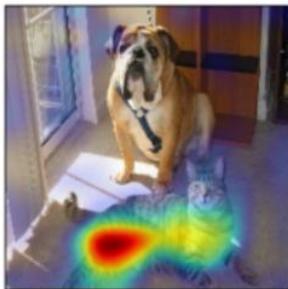
# What in the image was important for the prediction?



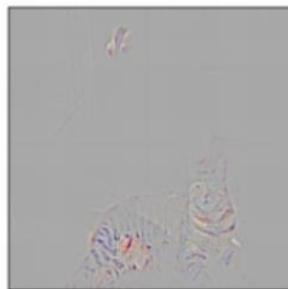
(a) Original Image



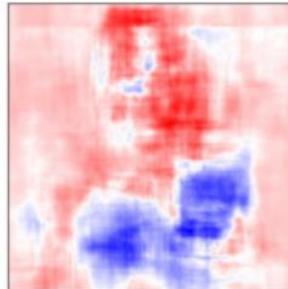
(b) Guided Backprop 'Cat'



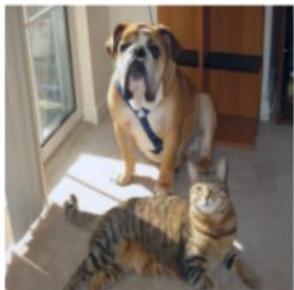
(c) Grad-CAM 'Cat'



(d) Guided Grad-CAM 'Cat'



(e) Occlusion map 'Cat'



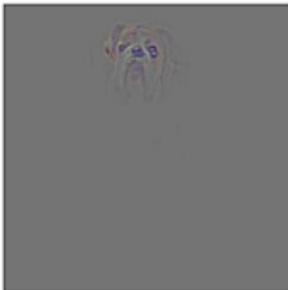
(g) Original Image



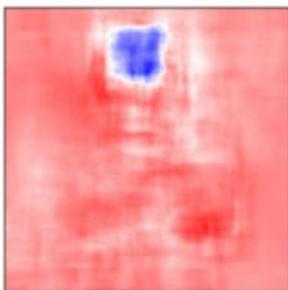
(h) Guided Backprop 'Dog'



(i) Grad-CAM 'Dog'



(j) Guided Grad-CAM 'Dog'

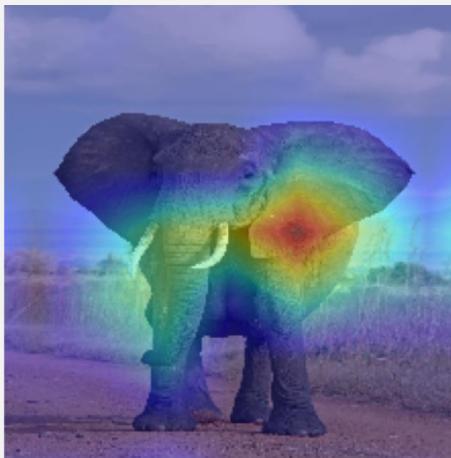


(k) Occlusion map 'Dog'

## Exercise:



Which part of the image was important for the prediction «African elephant»?  
Does the prediction change when a part of the image is occluded?



Work through the NB [08\\_gradcam\\_and\\_occlusion.ipynb](#)

# Summary

- **Use tricks of the trade when building a CNN**
  - Normalization of input data
  - Batchnorm
  - Dropout
  - Augmentation
  - Use challenge winning architectures
- **In case of few data**
  - Do augmentation during training of a CNN
  - Use shallow learner (e.g. RF) based on image features extracted via pretrained CNNs
  - Fine-tune a pretrained CNN on few data (transfer learning)
- **NNs are loosely inspired by the structure of the brain.**
  - When going deep the receptive field increases (~layer 5 sees whole input)
  - Deeper layer respond to more complex feature in the input
- **Shine light into the black-box (xAI) by**
  - Occlusion
  - gradCam
  - Lime