Zurich University of Applied Sciences

zh aw

CNNs part II

Beate Sick

sick@zhaw.ch

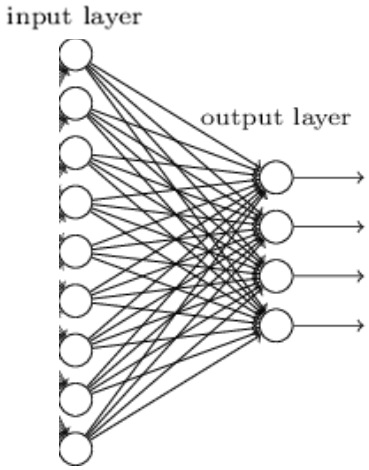
Remark: Much of the material has been developed together with Elvis Murina and Oliver Dürr

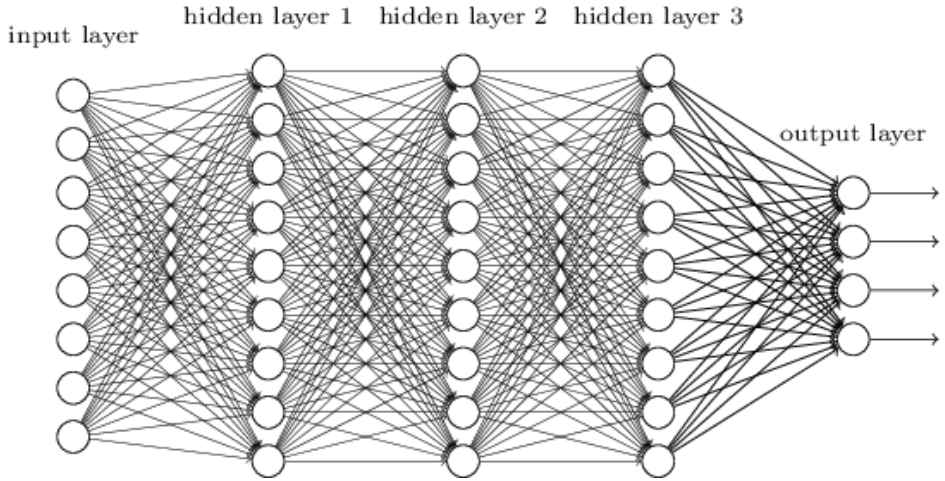
Topics

- Recap
- What to do in case of limited data?
  - Data augmentation
  - Transfer learning: use pre-trained CNNs and fine tune only last couple of layers
- Understand what a CNN has learned
  - Visualize the image patches that give rise for high activations of intermediate neurons
  - visualize image parts that are important for the assignment to a certain class
- Famous CNN architectures and tricks they make use of
  - LeNet, AlexNet, GoogleNet, VGG, Microsoft ResNet
- Causal and dilated 1D CNNs for time-ordered data

Discussed architectures NNs to CNNs

Fully connected Neural Networks (fcNN) without and with hidden layers:





Convolutional Neural Network:

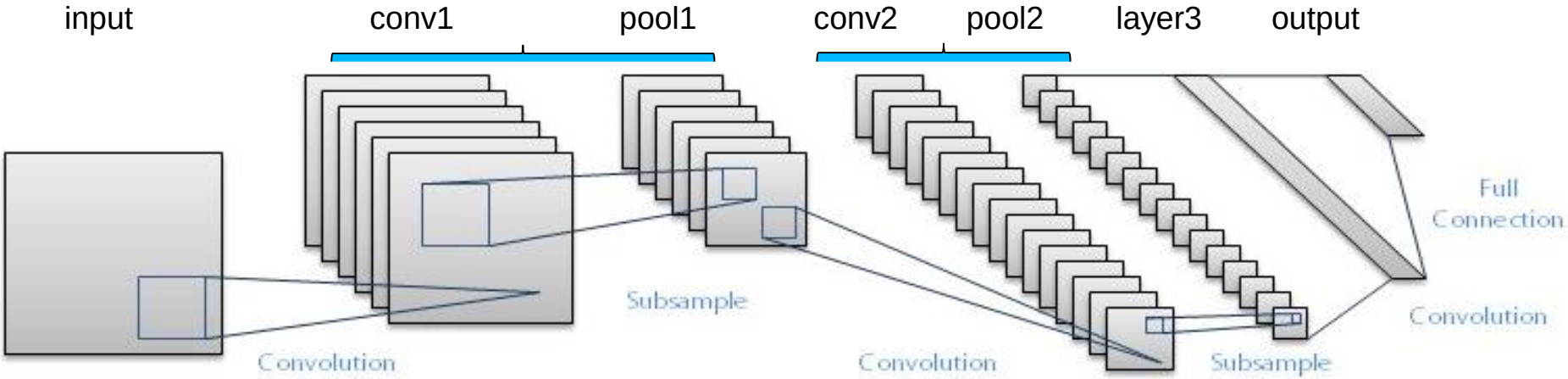
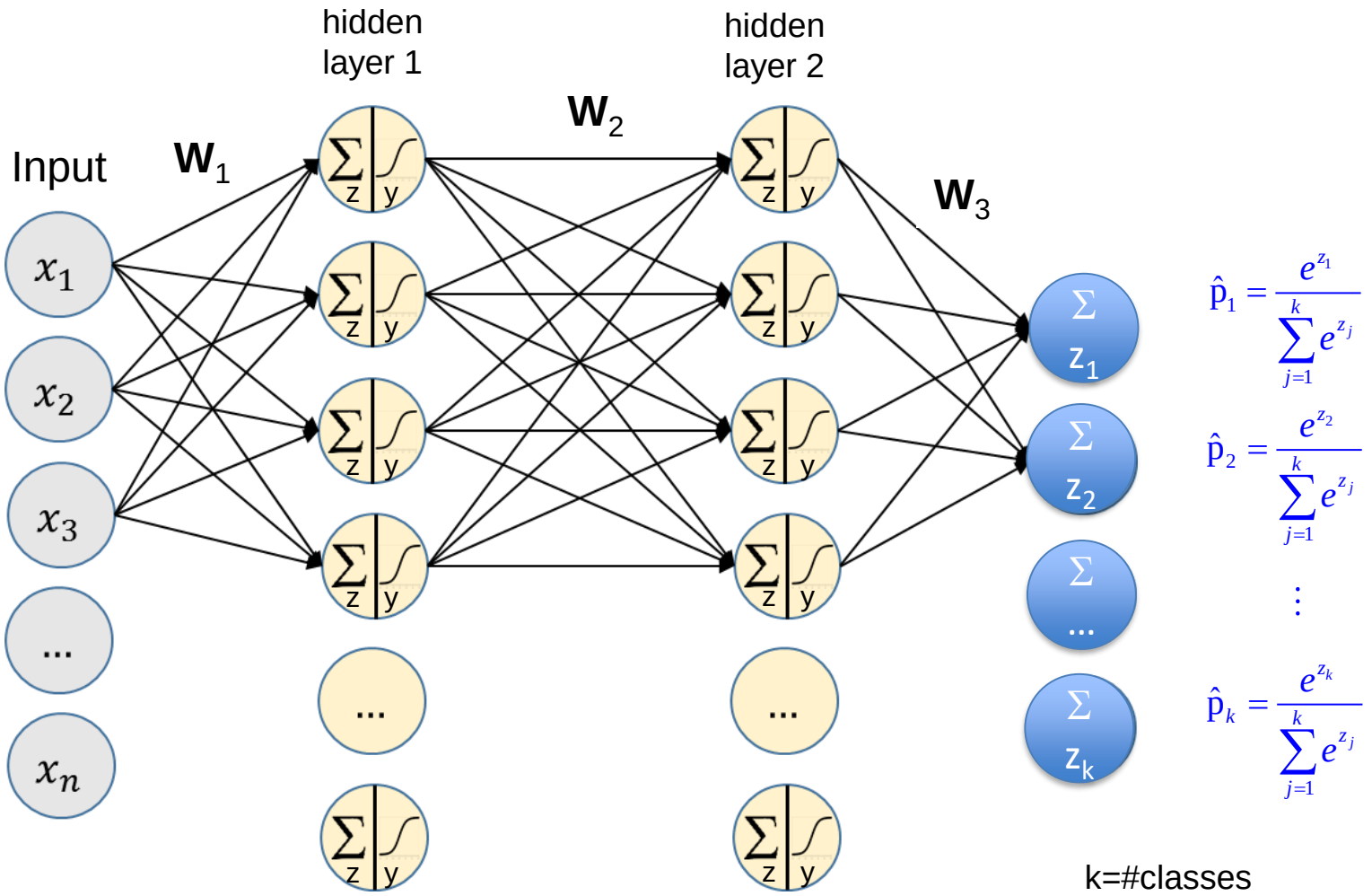


Image credits: http://neuralnetworksanddeeplearning.com/chap6.html, http://www.cemetech.net/projects/ee/scouter/thesis.pdf

A fully conneted neural networks with 2 hidden layers

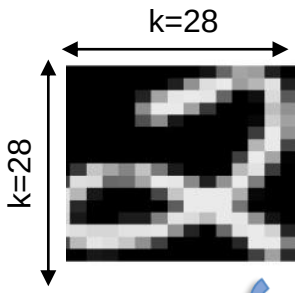


$$\hat{p}_1 = \frac{e^{z_1}}{\sum_{j=1}^{k} e^{z_j}} \qquad \hat{p}_2 = \frac{e^{z_2}}{\sum_{j=1}^{k} e^{z_j}} \qquad \hat{p}_k = \frac{e^{z_k}}{\sum_{j=1}^{k} e^{z_j}}$$

$$z = b + \sum_i (x_i \cdot w_i) \quad y = f(z)$$

Remark: weight values in the weight matrices that are learned during training.

What is going on in a 1 layer fully connected NN?



Flatten to vector with k^2 elements

input
$\mathbf{x}^t$

$$\begin{bmatrix} 24.0 \\ -5.1 \\ 12.2 \\ 89.9 \\ \vdots \\ 3.2 \end{bmatrix}$$

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

Score or logit
$\mathbf{z}^t$

$$\begin{bmatrix} 2.3 \\ 0.1 \end{bmatrix}$$

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

Softmax or probability
$\mathbf{p}=S(\mathbf{z})$

$$\begin{bmatrix} 0.9 \\ 0.1 \end{bmatrix}$$

1-hot labels
$\mathbf{y}$

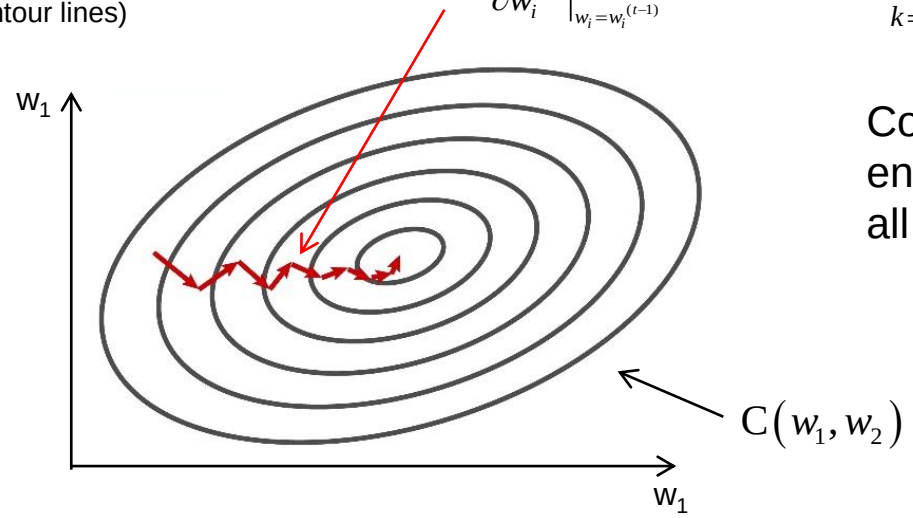
$$\begin{bmatrix} 1 \\ 0 \end{bmatrix}$$ "2" "1"

$$D(\mathbf{p},\mathbf{y})$$ cross-entropy

$$= -\sum_{k=1}^{2} y_k \cdot \log\left(p_k\right)$$

Take step in direction of descent gradient:
(the gradient is oriented orthogonal to contour lines)

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

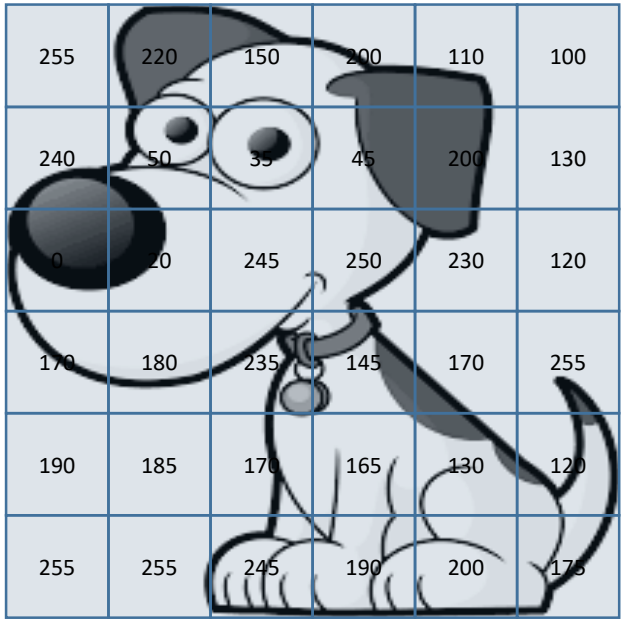


Cost C or Loss = cross-entropy averaged over all images in mini-batch

$$C = \frac{1}{N} \sum_i D(\mathbf{p}_i, \mathbf{y}_i)$$

Convolutional neural networks

Input image 6x6x1



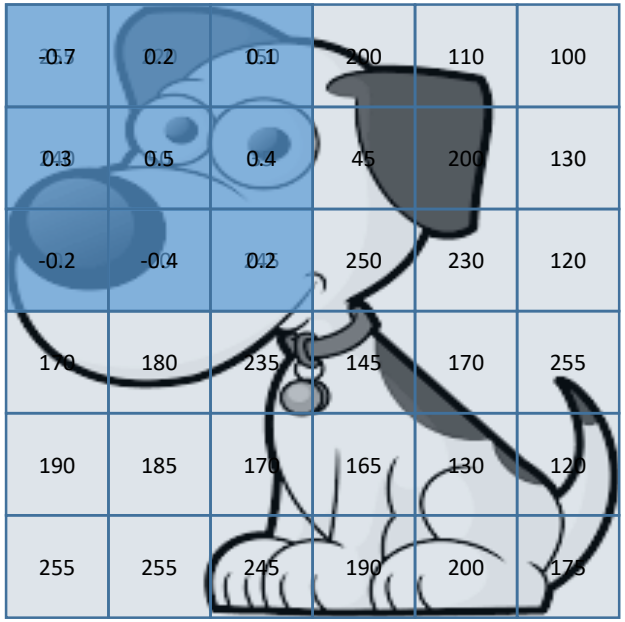
-0.7	0.2	0.1
0.3	0.5	0.4
-0.2	-0.4	0.2

3x3 filter

$$z_j = \sum_i \left(x_i \cdot w_{ij} \right)$$

Convolutional neural networks

Input image 6x6x1



Feature map
4x4x1

32.5

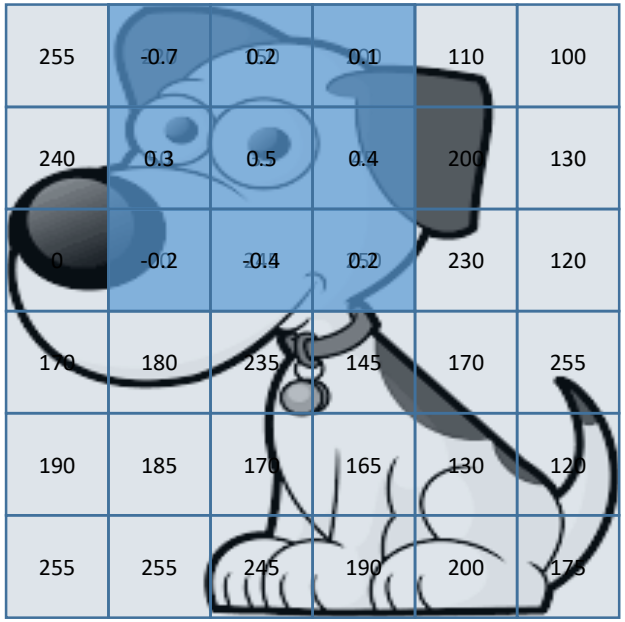
-0.7	0.2	0.1
0.3	0.5	0.4
-0.2	-0.4	0.2

3x3 filter

$$z_j = \sum_i \left(x_i \cdot w_{ij} \right)$$

Convolutional neural networks

Input image 6x6x1



Feature map
4x4x1

32.5	-105.5

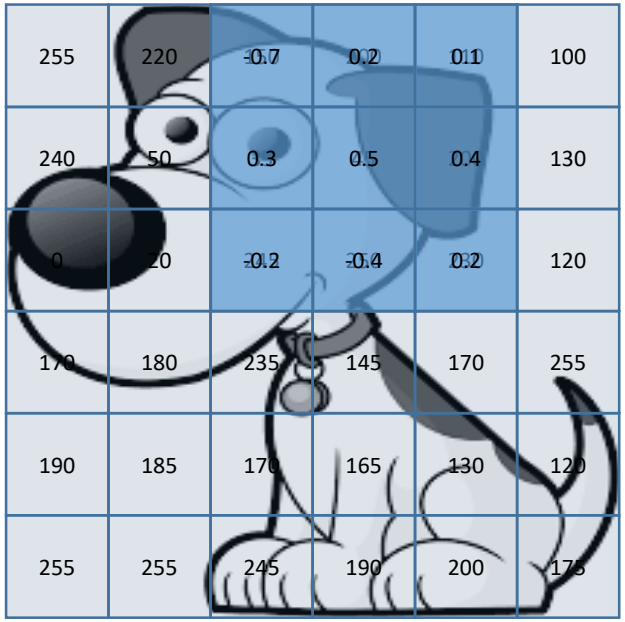
-0.7	0.2	0.1
0.3	0.5	0.4
-0.2	-0.4	0.2

3x3 filter

$$z_j = \sum_i \left(x_i \cdot w_{ij} \right)$$

Convolutional neural networks

Input image 6x6x1



Feature map 4x4x1

32.5	-105.5	185.5

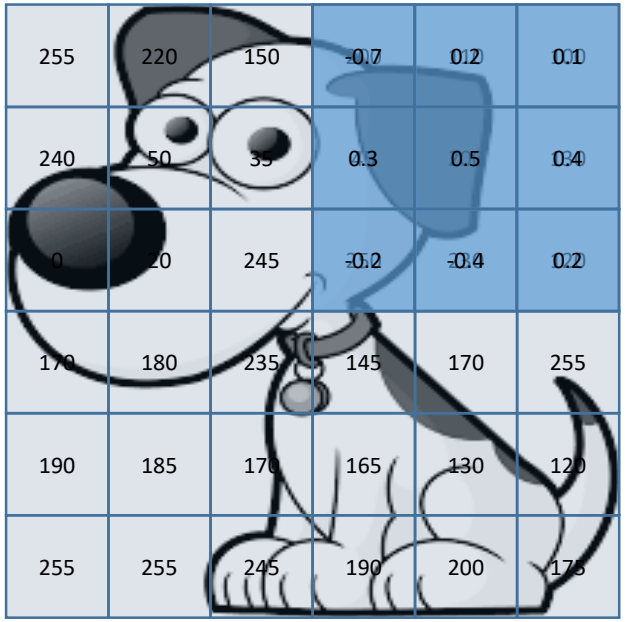
-0.7	0.2	0.1
0.3	0.5	0.4
-0.2	-0.4	0.2

3x3 filter

$$z_j = \sum_i \left(x_i \cdot w_{ij} \right)$$

Convolutional neural networks

Input image 6x6x1



Feature map 4x4x1

32.5	-105.5	185.5	54

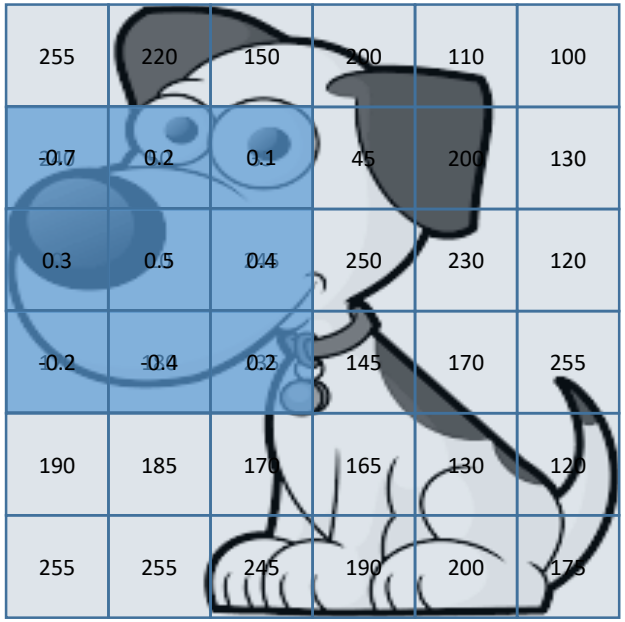
-0.7	0.2	0.1
0.3	0.5	0.4
-0.2	-0.4	0.2

3x3 filter

$$z_j = \sum_i \left(x_i \cdot w_{ij} \right)$$

Convolutional neural networks

Input image 6x6x1



Feature map
4x4x1

32.5	-105.5	185.5	54
-105.5			

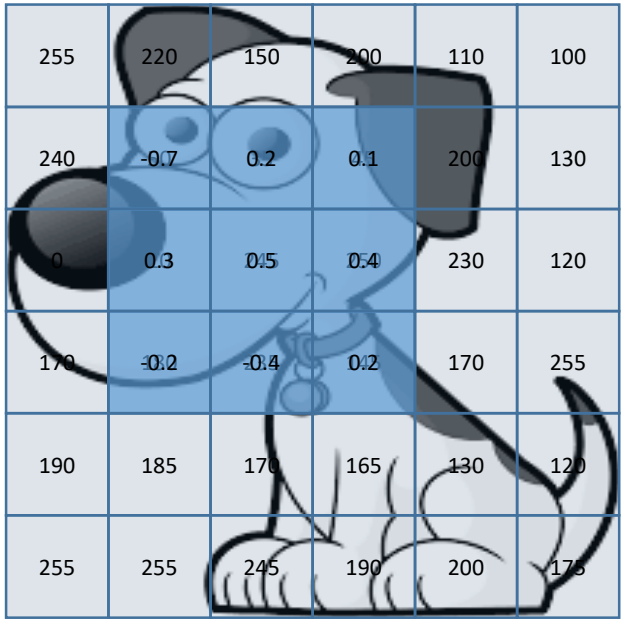
-0.7	0.2	0.1
0.3	0.5	0.4
-0.2	-0.4	0.2

3x3 filter

$$z_j = \sum_i \left(x_i \cdot w_{ij} \right)$$

Convolutional neural networks

Input image 6x6x1



Feature map 4x4x1

32.5	-105.5	185.5	54
-105.5	104		

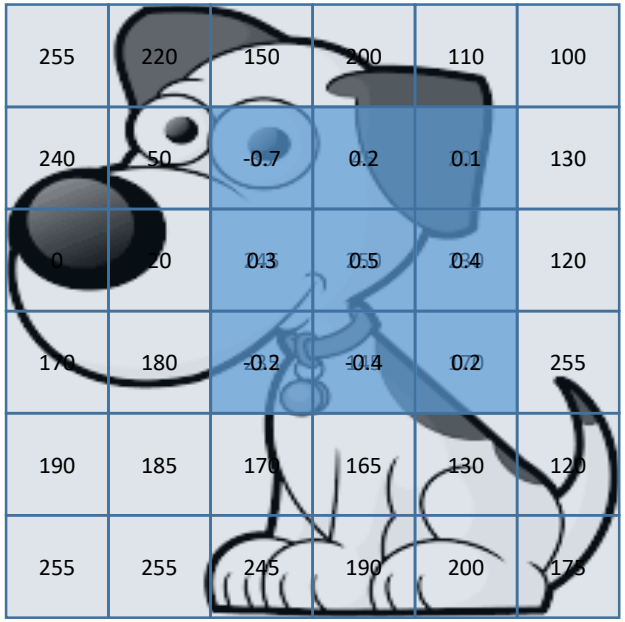
-0.7	0.2	0.1
0.3	0.5	0.4
-0.2	-0.4	0.2

3x3 filter

$$z_j = \sum_i \left(x_i \cdot w_{ij} \right)$$

Convolutional neural networks

Input image 6x6x1



Feature map 4x4x1

32.5	-105.5	185.5	54
-105.5	104	217.5	

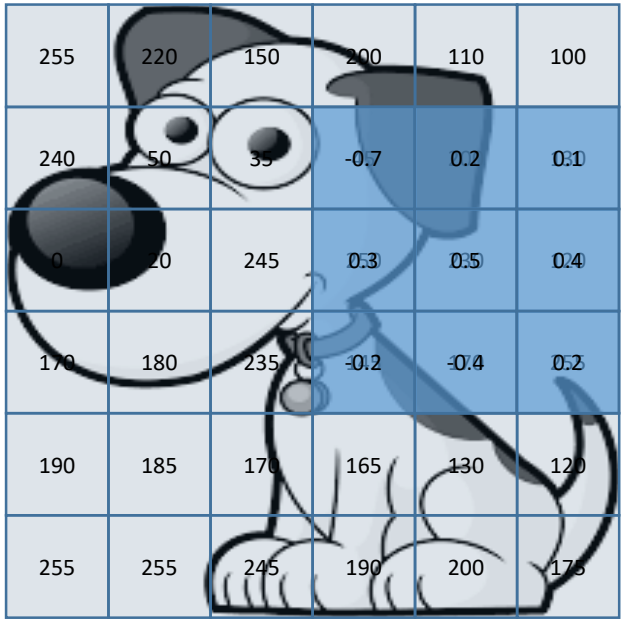
-0.7	0.2	0.1
0.3	0.5	0.4
-0.2	-0.4	0.2

3x3 filter

$$z_j = \sum_i \left(x_i \cdot w_{ij} \right)$$

Convolutional neural networks

Input image 6x6x1



Feature map
4x4x1

32.5	-105.5	185.5	54
-105.5	104	217.5	31

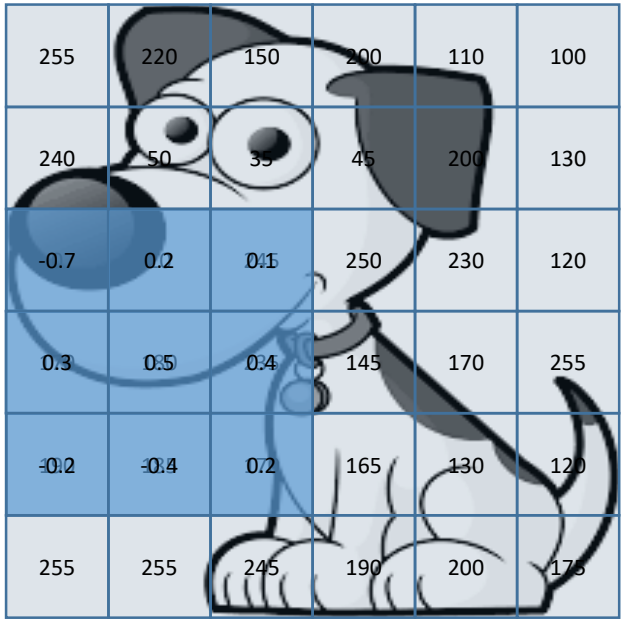
-0.7	0.2	0.1
0.3	0.5	0.4
-0.2	-0.4	0.2

3x3 filter

$$z_j = \sum_i \left(x_i \cdot w_{ij} \right)$$

Convolutional neural networks

Input image 6x6x1



Feature map
4x4x1

32.5	-105.5	185.5	54
-105.5	104	217.5	31
-44			

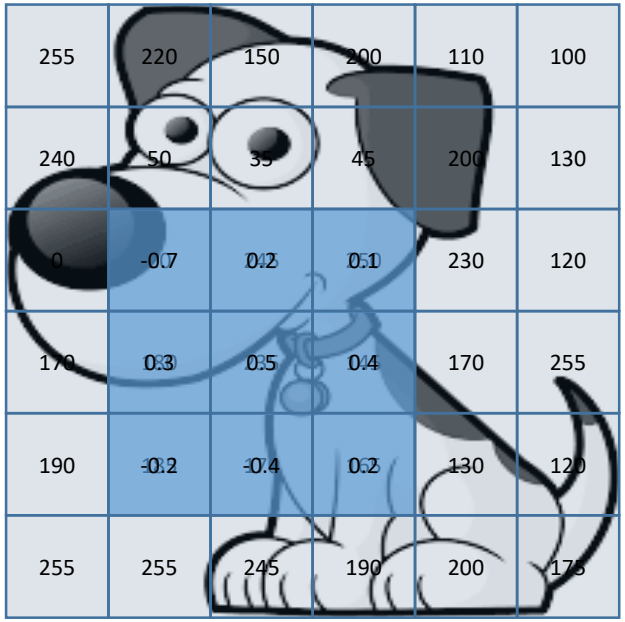
-0.7	0.2	0.1
0.3	0.5	0.4
-0.2	-0.4	0.2

3x3 filter

$$z_j = \sum_i \left(x_i \cdot w_{ij} \right)$$

Convolutional neural networks

Input image 6x6x1



Feature map
4x4x1

32.5	-105.5	185.5	54
-105.5	104	217.5	31
-44	224		

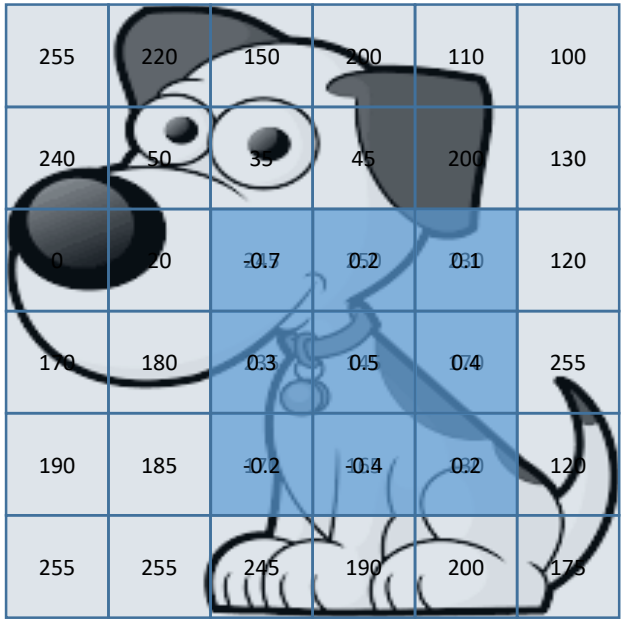
-0.7	0.2	0.1
0.3	0.5	0.4
-0.2	-0.4	0.2

3x3 filter

$$z_j = \sum_i \left(x_i \cdot w_{ij} \right)$$

Convolutional neural networks

Input image 6x6x1



Feature map
4x4x1

32.5	-105.5	185.5	54
-105.5	104	217.5	31
-44	224	38.5	

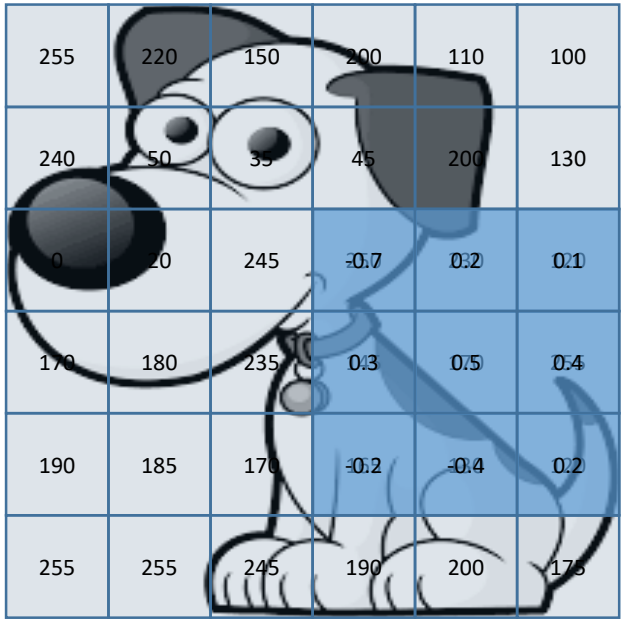
-0.7	0.2	0.1
0.3	0.5	0.4
-0.2	-0.4	0.2

3x3 filter

$$z_j = \sum_i \left(x_i \cdot w_{ij} \right)$$

Convolutional neural networks

Input image 6x6x1



Feature map 4x4x1

32.5	-105.5	185.5	54
-105.5	104	217.5	31
-44	224	38.5	-18

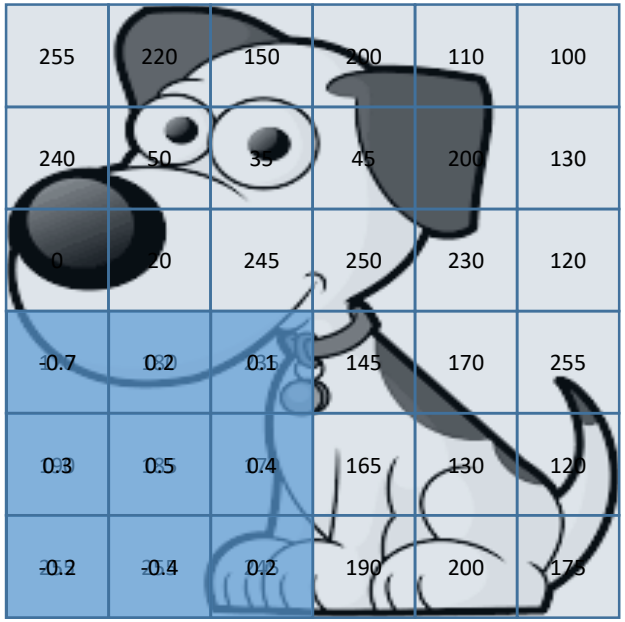
-0.7	0.2	0.1
0.3	0.5	0.4
-0.2	-0.4	0.2

3x3 filter

$$z_j = \sum_i \left(x_i \cdot w_{ij} \right)$$

Convolutional neural networks

Input image 6x6x1



Feature map
4x4x1

32.5	-105.5	185.5	54
-105.5	104	217.5	31
-44	224	38.5	-18
-60.5			

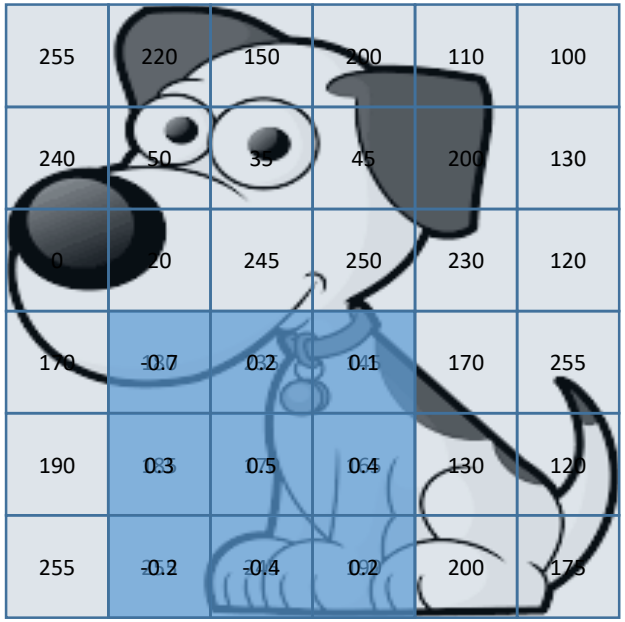
-0.7	0.2	0.1
0.3	0.5	0.4
-0.2	-0.4	0.2

3x3 filter

$$z_j = \sum_i \left(x_i \cdot w_{ij} \right)$$

Convolutional neural networks

Input image 6x6x1



Feature map 4x4x1

32.5	-105.5	185.5	54
-105.5	104	217.5	31
-44	224	38.5	-18
-60.5	213.5		

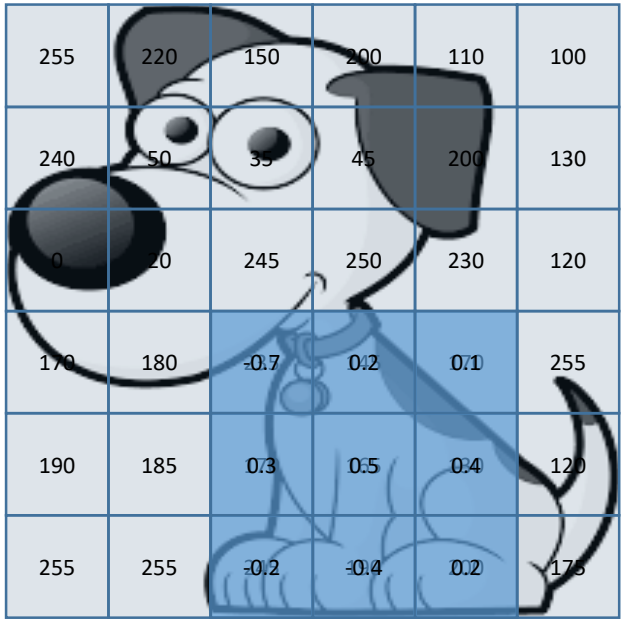
-0.7	0.2	0.1
0.3	0.5	0.4
-0.2	-0.4	0.2

3x3 filter

$$z_j = \sum_i \left(x_i \cdot w_{ij} \right)$$

Convolutional neural networks

Input image 6x6x1



Feature map 4x4x1

32.5	-105.5	185.5	54
-105.5	104	217.5	31
-44	224	38.5	-18
-60.5	213.5	52.5	

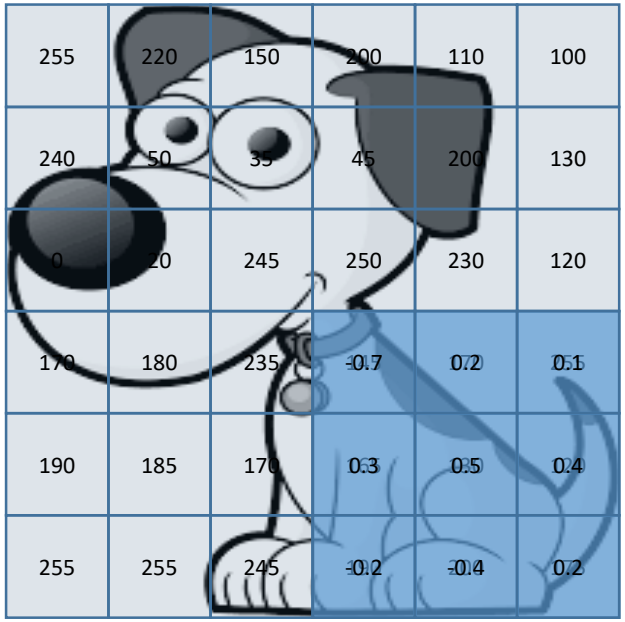
-0.7	0.2	0.1
0.3	0.5	0.4
-0.2	-0.4	0.2

3x3 filter

$$z_j = \sum_i \left(x_i \cdot w_{ij} \right)$$

Convolutional neural networks

Input image 6x6x1



Feature map
4x4x1

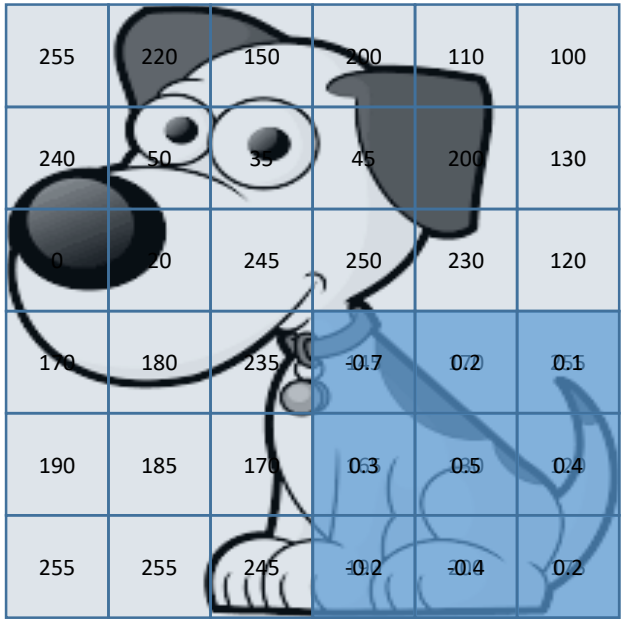
32.5	-105.5	185.5	54
-105.5	104	217.5	31
-44	224	38.5	-18
-60.5	213.5	52.5	37.5

-0.7	0.2	0.1
0.3	0.5	0.4
-0.2	-0.4	0.2

3x3 filter

Convolutional neural networks

Input image 6x6x1

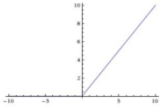


-0.7	0.2	0.1
0.3	0.5	0.4
-0.2	-0.4	0.2

3x3 filter

Feature map
4x4x1

32.5	-105.5	185.5	54
-105.5	104	217.5	31
-44	224	38.5	-18
-60.5	213.5	52.5	37.5

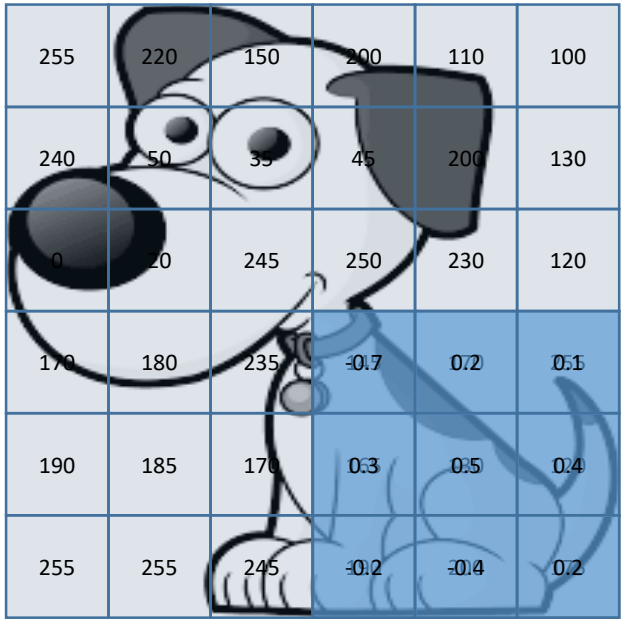


Relu

32.5	0	185.5	54
0	104	217.5	31
0	224	38.5	0
0	213.5	52.5	37.5

Convolutional neural networks

Input image 6x6x1

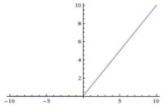


-0.7	0.2	0.1
0.3	0.5	0.4
-0.2	-0.4	0.2

3x3 filter

Feature map 4x4x1

32.5	-105.5	185.5	54
-105.5	104	217.5	31
-44	224	38.5	-18
-60.5	213.5	52.5	37.5



Relu

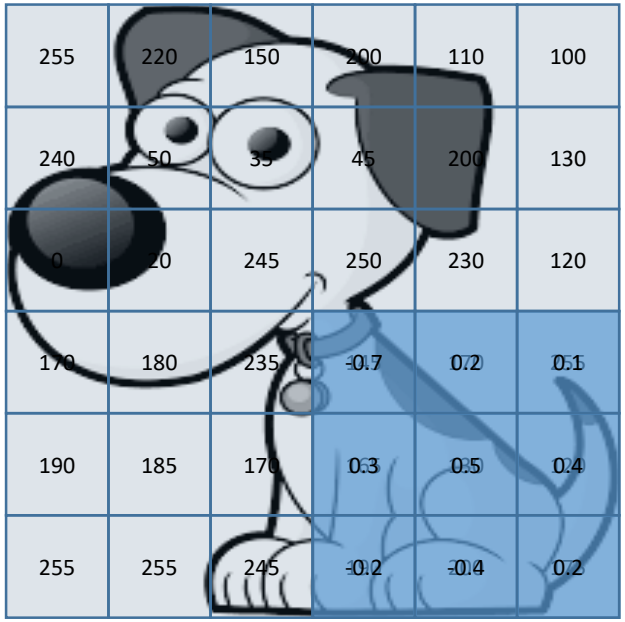
32.5	0	185.5	54
0	104	217.5	31
0	224	38.5	0
0	213.5	52.5	37.5

Maxpool (2x2x1)

104

Convolutional neural networks

Input image 6x6x1

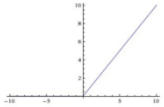


-0.7	0.2	0.1
0.3	0.5	0.4
-0.2	-0.4	0.2

3x3 filter

Feature map 4x4x1

32.5	-105.5	185.5	54
-105.5	104	217.5	31
-44	224	38.5	-18
-60.5	213.5	52.5	37.5



Relu

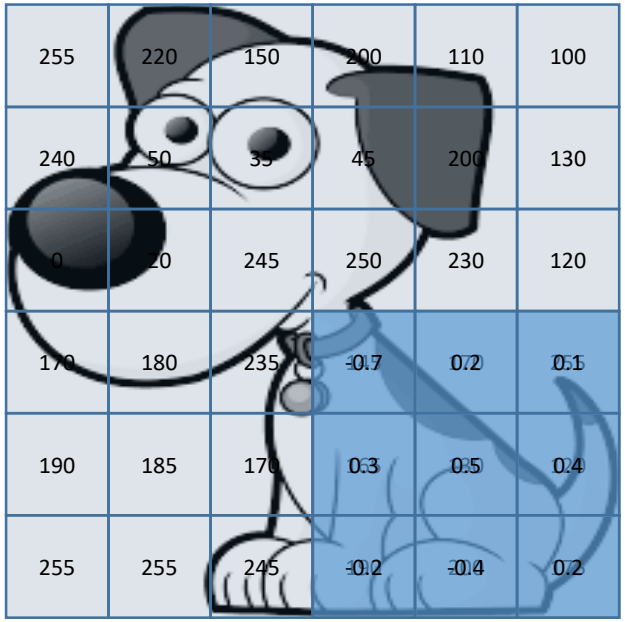
32.5	0	185.5	54
0	104	217.5	31
0	224	38.5	0
0	213.5	52.5	37.5

Maxpool (2x2x1)

104	217.5

Convolutional neural networks

Input image 6x6x1

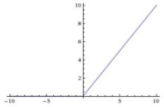


-0.7	0.2	0.1
0.3	0.5	0.4
-0.2	-0.4	0.2

3x3 filter

Feature map 4x4x1

32.5	-105.5	185.5	54
-105.5	104	217.5	31
-44	224	38.5	-18
-60.5	213.5	52.5	37.5



Relu

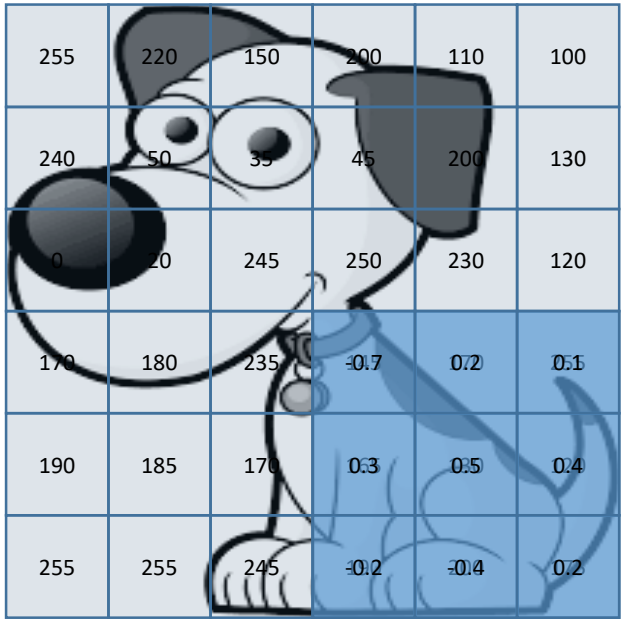
32.5	0	185.5	54
0	104	217.5	31
0	224	38.5	0
0	213.5	52.5	37.5

Maxpool (2x2x1)

104	217.5
224	

Convolutional neural networks

Input image 6x6x1

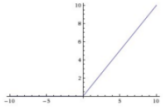


-0.7	0.2	0.1
0.3	0.5	0.4
-0.2	-0.4	0.2

3x3 filter

Feature map 4x4x1

32.5	-105.5	185.5	54
-105.5	104	217.5	31
-44	224	38.5	-18
-60.5	213.5	52.5	37.5



Relu

32.5	0	185.5	54
0	104	217.5	31
0	224	38.5	0
0	213.5	52.5	37.5

Maxpool (2x2x1)

104	217.5
224	52.5

One kernel or filter searches for specific local feature

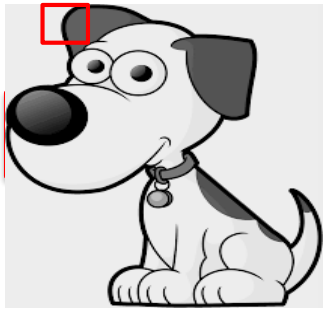


image patch

0	0	0	0	0	0	30
0	0	0	0	50	50	50
0	0	0	20	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0

Pixel representation of the receptive field

*

filter/kernel: curve detector

0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

Pixel representation of filter

=6600

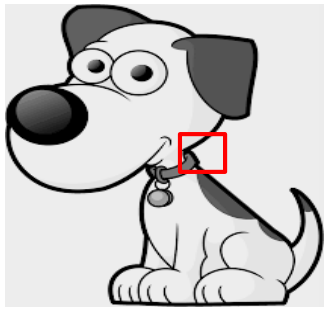


image patch

0	0	0	0	0	0	0
0	40	0	0	0	0	0
40	0	40	0	0	0	0
40	20	0	0	0	0	0
0	50	0	0	0	0	0
0	0	50	0	0	0	0
25	25	0	50	0	0	0

Pixel representation of receptive field

*

filter/kernel: curve detector

0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

Pixel representation of filter

=0

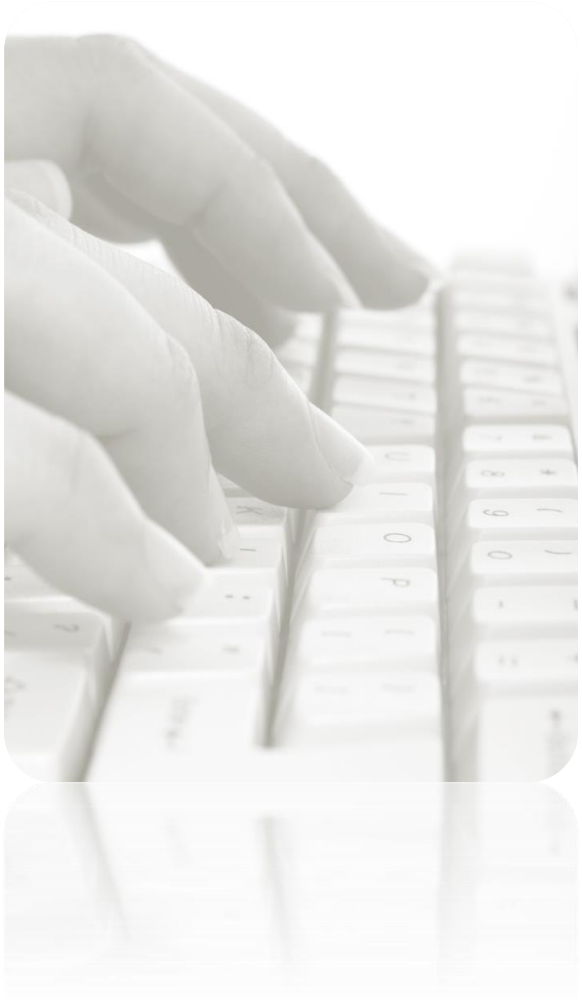
We get a large resulting value if the filter resembles the pattern in the image patch on which the filter was applied

→ convolution can be understood as correlation

credits: https://adeshpande3.github.io/adeshpande3.github.io/A-Beginner's-Guide-To-Understanding-Convolutional-Neural-Networks/

Looking back at home work

- Work through the instructions in 07 and 08 CNN Exercises **in day4** and use the ipython notebooks that are referred to.



exercises/07_CNN_MNIST a)

First we recall the design of the fc NN which performed so far best on MNIST when only keeping 2400 examples in the training data set (see below). With this NN we have reached ~91% accuracy on the validation data set. fcn_MNIST_keras

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 500)               392500
_________________________________________________________________
batch_normalization_1 (Batch (None, 500)               2000
_________________________________________________________________
dropout_1 (Dropout)          (None, 500)               0
_________________________________________________________________
activation_1 (Activation)    (None, 500)               0
_________________________________________________________________
dense_2 (Dense)              (None, 50)                25050
_________________________________________________________________
batch_normalization_2 (Batch (None, 50)                200
_________________________________________________________________
dropout_2 (Dropout)          (None, 50)                0
_________________________________________________________________
activation_2 (Activation)    (None, 50)                0
_________________________________________________________________
dense_3 (Dense)              (None, 10)                510
=================================================================
Total params: 420,260.0
Trainable params: 419,160.0
Non-trainable params: 1,100.0
_________________________________________________________________
```

28*28*500+500 = 392500

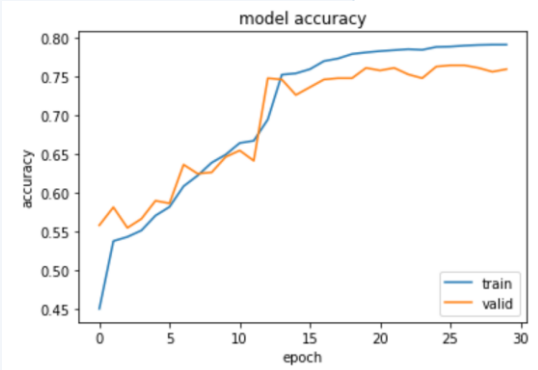
a) Where do we spend most learnable parameter? Can you explain the “Param #” of the dense_1 layer?
Remark: dense layer is the same as fully connected layer.

exercises/07_CNN_MNIST b)

b) Now we want to use our first CNN with only 1 convolutional and 1 dense layer:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 28, 28, 32)        320
_________________________________________________________________
activation_1 (Activation)    (None, 28, 28, 32)        0
_________________________________________________________________
flatten_1 (Flatten)          (None, 25088)             0
_________________________________________________________________
dense_1 (Dense)              (None, 10)                250890
_________________________________________________________________
activation_2 (Activation)    (None, 10)                0
=================================================================
Total params: 251,210
Trainable params: 251,210
Non-trainable params: 0
_________________________________________________________________
```

The CNN has much less weights than the fcNN, and most are in the fc part.
→ We expect less overfitting.



Ok - less overfitting, but worse accuracy (76%) compared to the fcNN (91%)

In which layer do we need to learn most parameter/weights?

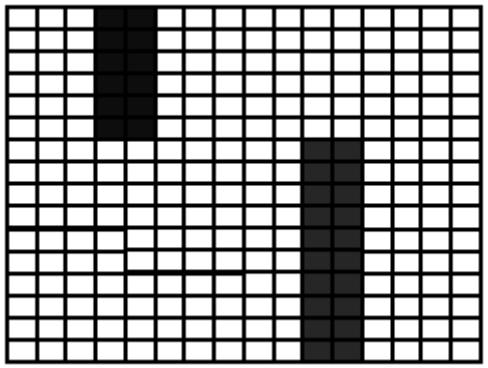
Do you expect with this cnn1 more or less overfitting then in the fc NN above? Why?

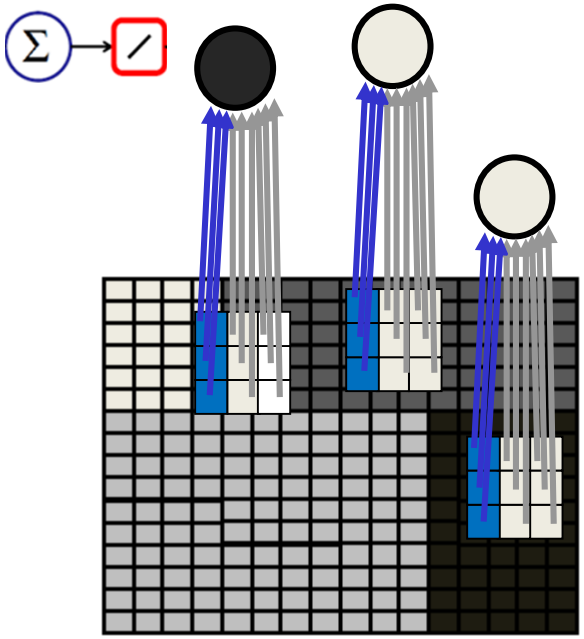
Please open the ipython-Notebook cnn1_mnist and try to understand the code and run the code.

Train the model without first standardizing the data which would be done in code cell 7 (cell is a comment here).
What is the accuracy on the validation set, now?
Can you explain, what happened?

exercises/07_CNN_MNIST b) cntd.
Standardizing data is more important in CNNs than fcNN

feature/activation map





input

Since we share weights in CNNs – only one filter is required per feature map - the weights in a filter should be appropriate for each patch of the input image, i.e. yielding inputs to the activation function that are party in their sweet spot.

To ensure that different image patches or even different pixels yield comparable ranges of values as input to the filter we need to standardize the input pixel-wise (we also restrict the variation to get small activations corresponding to uncertain classifications in the beginning of the training).

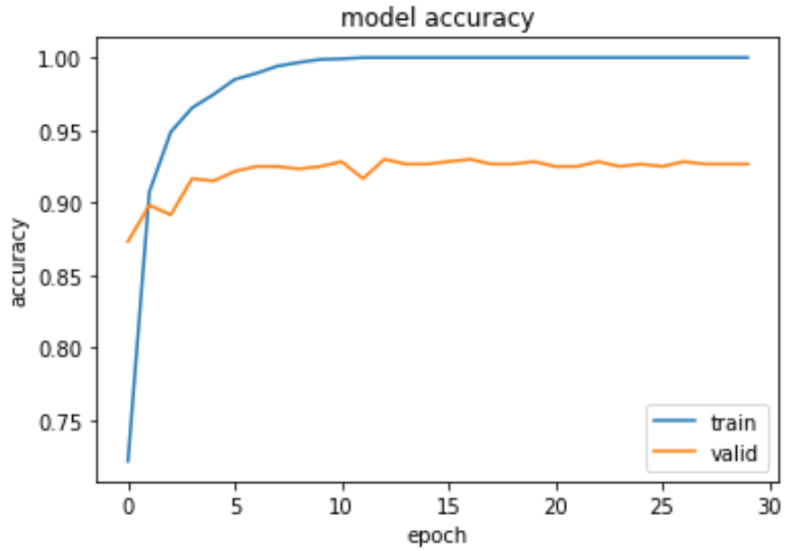
```
print(X_train.shape)
print(X_val.shape)
```

```
(4000, 28, 28, 1)
(1000, 28, 28, 1)
```

```
# here we center and standardize the data per pixel
# calculate mean, std over all training images at each pixel position
X_mean = np.mean( X_train, axis = 0)
X_std = np.std( X_train, axis = 0)

X_train = (X_train - X_mean ) / (X_std + 0.0001)
X_val = (X_val - X_mean ) / (X_std + 0.0001)
```

exercises/07_CNN_MNIST b) cntd.

Now train the model with standardizing the data by running the rest of the cells. How good is the accuracy on the validation and test set now?
Do you observe overfitting?
Describe how you check for overfitting and/or sketch the corresponding graph.

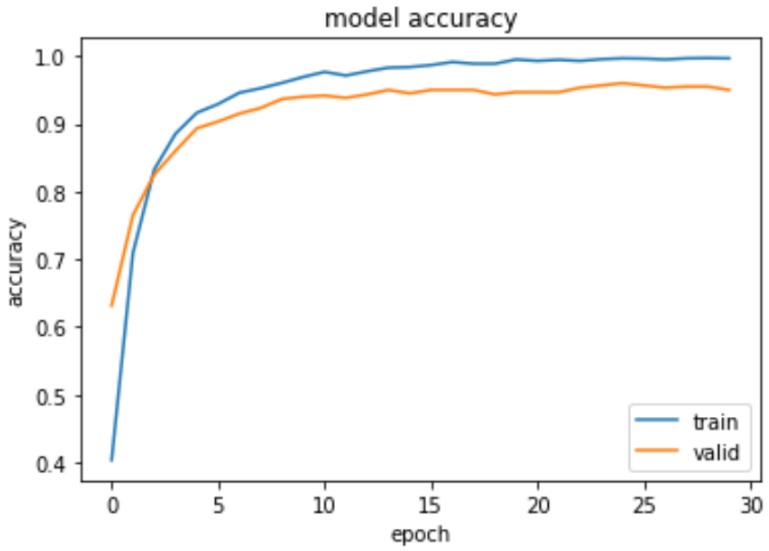


→ **We get now superior accuracy (92.5%) compared to fcNN (91%)**
  → **Standardizing helped a lot!**
→ **We observe quite strong overfitting.**
→ **Fighting overfitting:**
  → **More data**
  → **Simpler model, e.g. by regularization**
    → **Dropout**
    → **Batchnorm**
    → **…**

exercises/07_CNN_MNIST c)

c) Lets use more Convolutional Layers and also Dropout and Batchnorm.
Please open the ipython-Notebook cnn2_mnist.
Look for the position in the code which is marked by "# here is your code coming:" and add the missing layers - the missing layers are marked below.
How is the performance of cnn2?

```
### a deeper CNN model
name = 'cnn2'
model = Sequential()

model.add(Convolution2D(8,kernel_size,padding='same',input_shape=input_shape))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(Convolution2D(8, kernel_size,padding='same'))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=pool_size))

model.add(Convolution2D(16, kernel_size,padding='same'))
model.add(BatchNormalization())
model.add(Activation('relu'))

# here is your code comming:

model.add(Convolution2D(16,kernel_size,padding='same'))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=pool_size))

# end of your code

model.add(Flatten())#macht einen vektor aus dem output
model.add(Dense(40))
model.add(BatchNormalization())
model.add(Dropout(0.3))
model.add(Activation('relu'))
model.add(Dense(nb_classes))
model.add(Activation('softmax'))

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



→ **We get higher accuracy (95.2%)**
→ **Going deeper helped**

→ **We much lessoverfitting!**
→ **Dropout, Batchnorm helped**

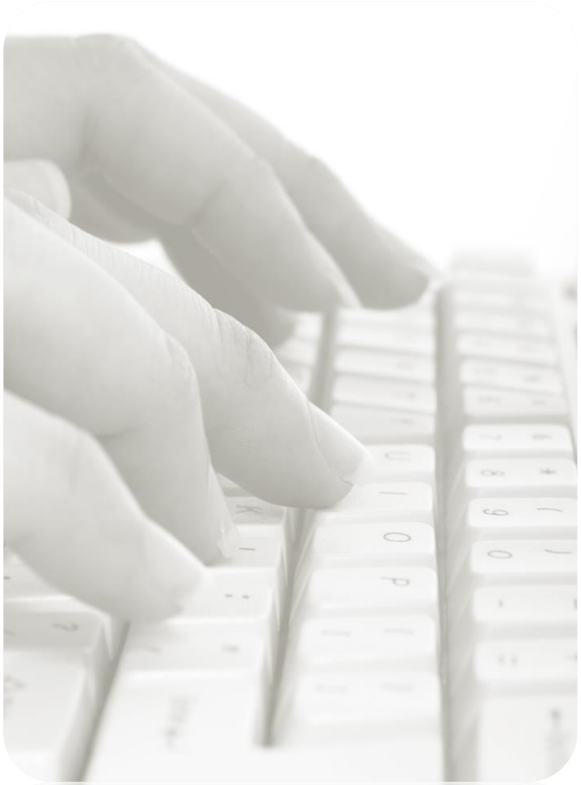
Exercise: Image based classification of celebrities

Task: Build and evaluate a fcNN and CNN for discrimination between 8 celebs.

For each of 8 celebrities you get 250 images in the training data set, 50 images in the validation data set and 50 images in test data set.

Example images:

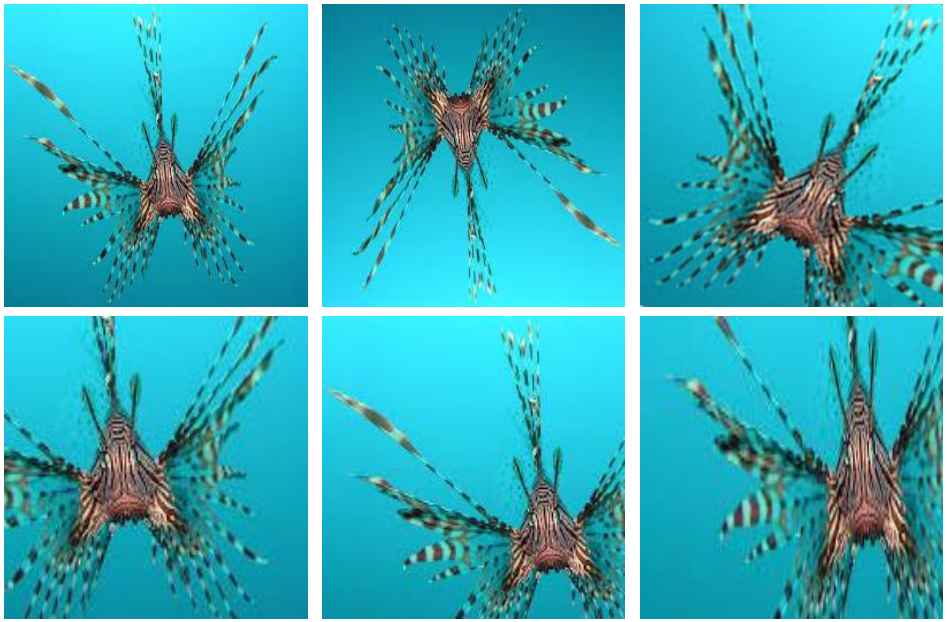
Label: Steve Jobs (entrepreneur)



What to do in case of limited data?

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

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



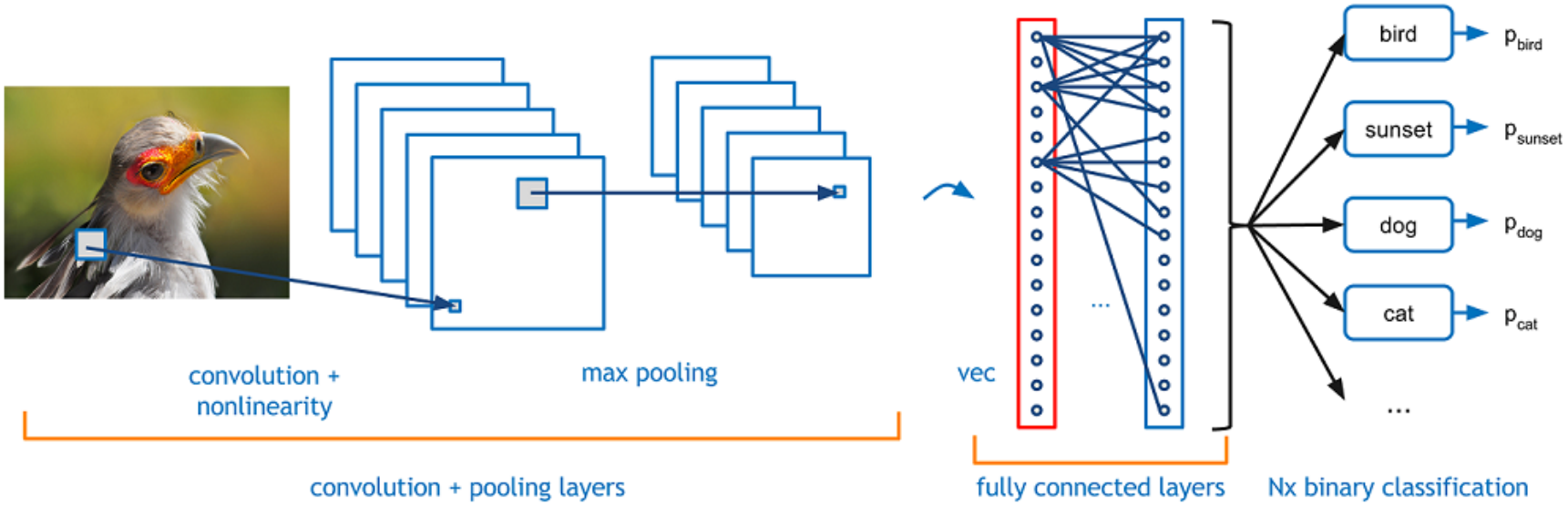
Data augmentation in Keras:

```
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]
)
```

Convolutional part in CNN as representation learning



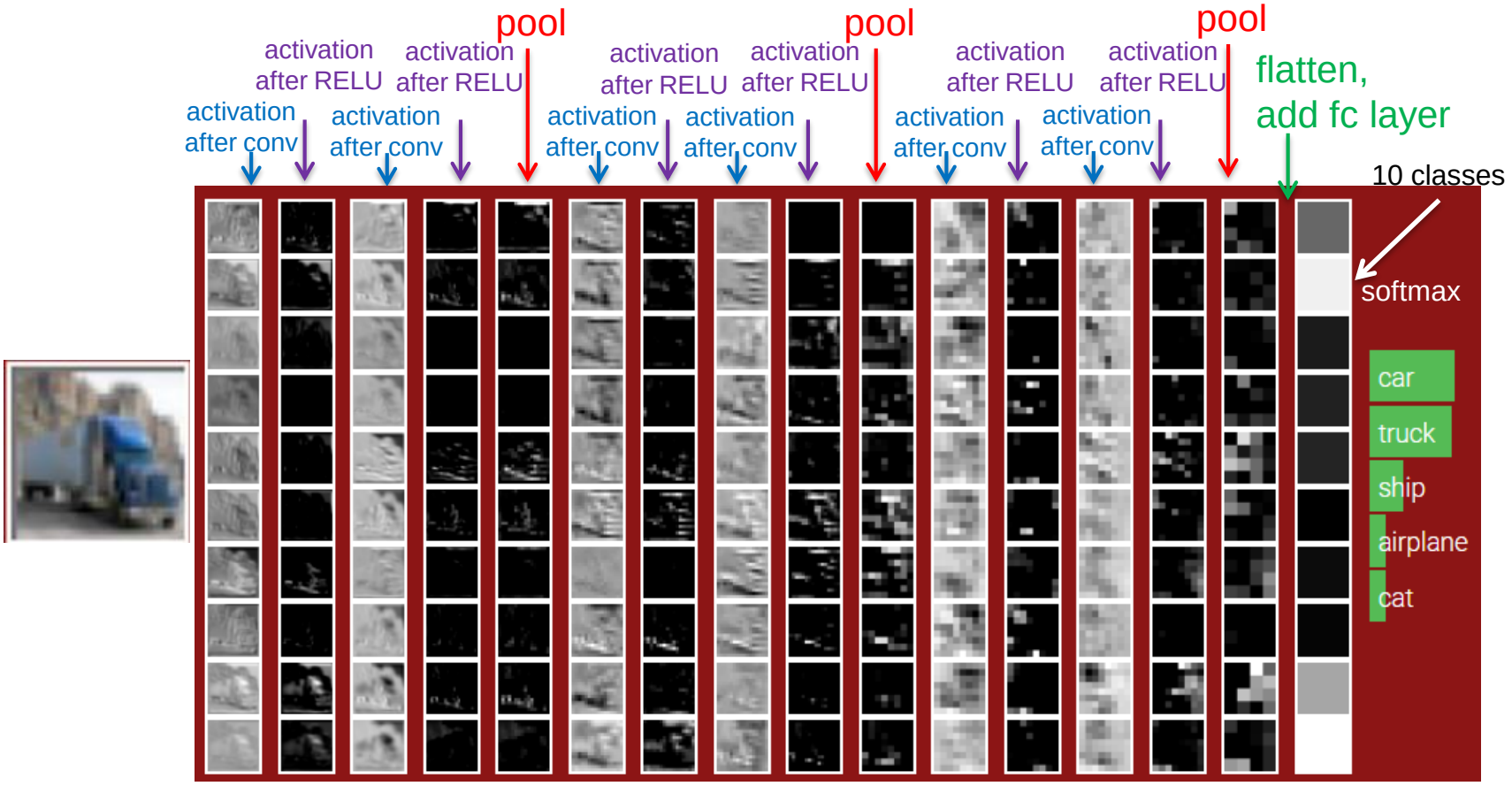
In a classical CNN we start with convolution layers and end with fc layers.

The task of the convolutional layers is to extract useful features from the image which might be appear at arbitrary positions in the image.

The task of the fc layer is to use these extracted features for classification.

Image credits:
https://adeshpande3.github.io/adeshpande3.github.io/A-Beginner's-Guide-To-Understanding-Convolutional-Neural-Networks/

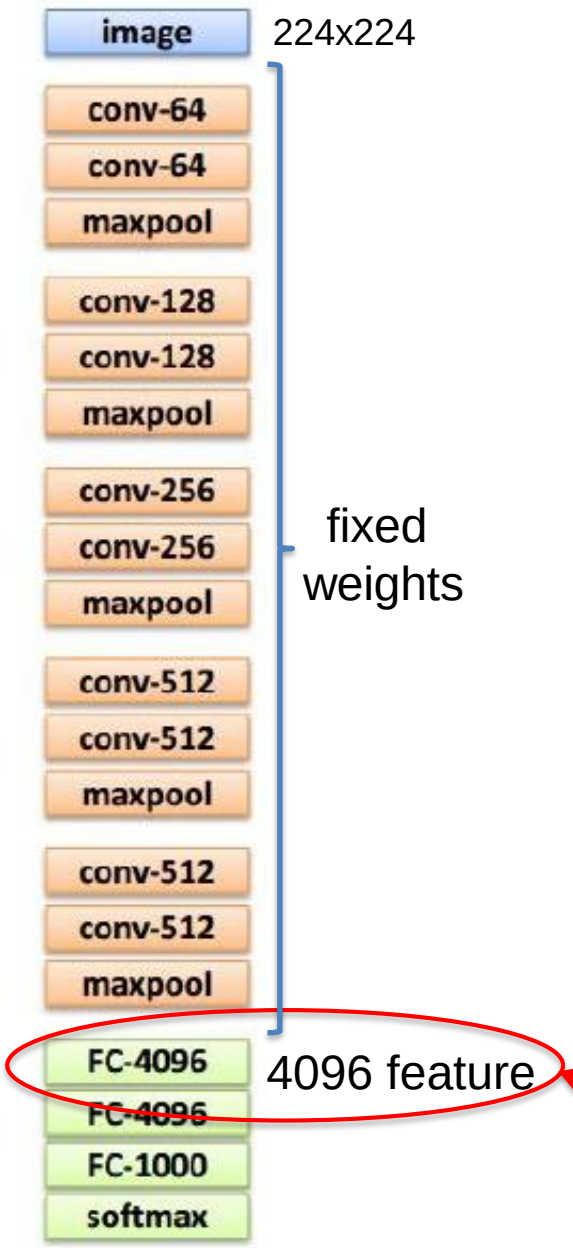
Activations in output-close layers can be used as new features



Activation maps give insight on the spatial positions where the filter pattern was found in the input one layer below (in higher layers activation maps have no easy interpretation) -> only the activation maps in the first hidden layer correspond directly to features of the input image.

http://cs231n.stanford.edu/

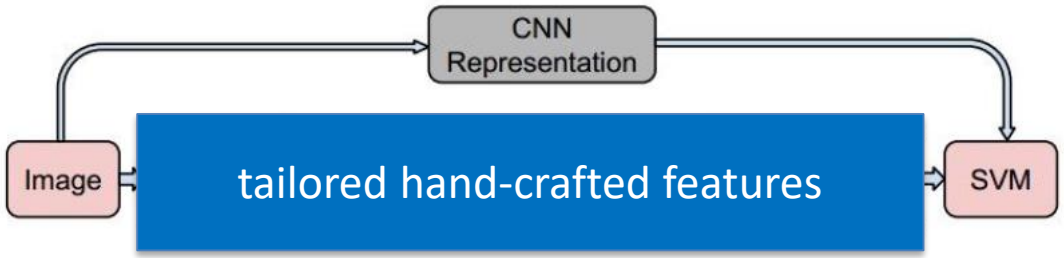
Use pre-trained CNNs for feature generation

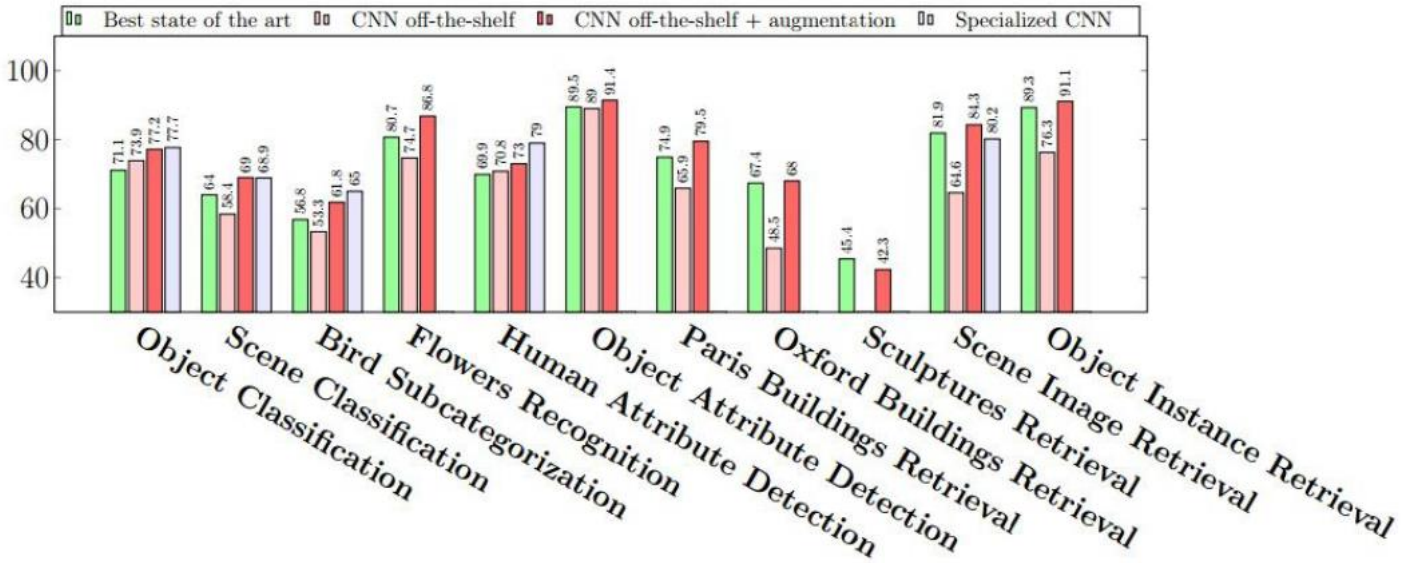


- Load a pre-trained CNN – e.g.g VGG16
- Resize image to required size (224x224 for VGG16)
- Rescaling of the pixel values to "VGG range"
- Do a forward pass and fetch 4096 features that are used as CNN representations, dump these features into a file on disk
- Use these CNN features as input to a simple classifier – e.g. fc NN, RF, SVM …
  (here it is easily possible to adapt to the new number of class labels)

Fetch this CNN feature vector for each image

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



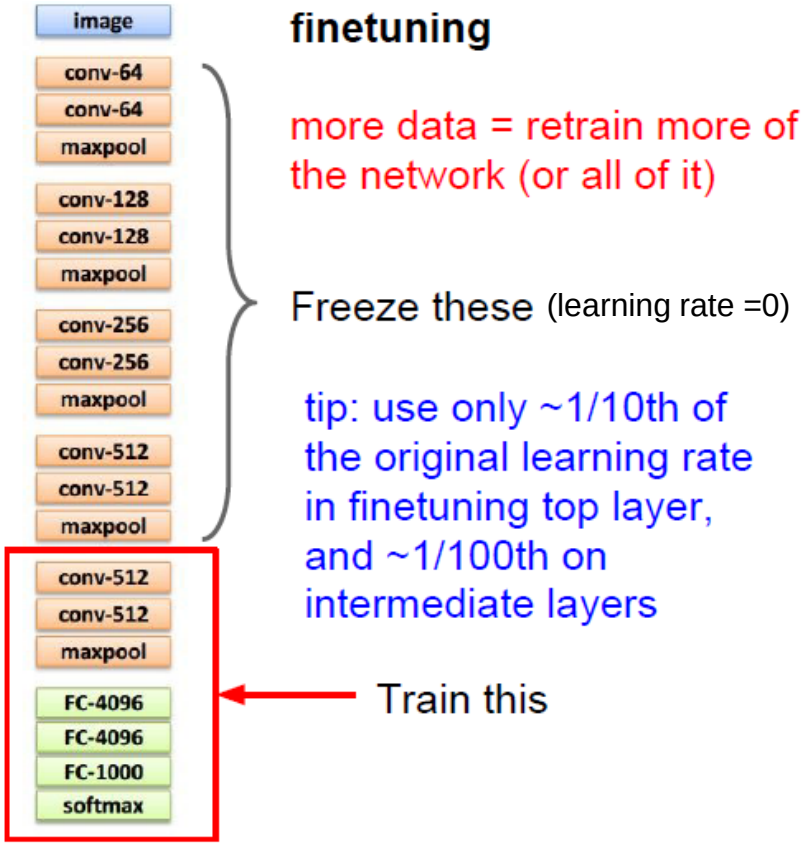


"Astonishingly, we report consistent superior results compared to the highly tuned state-of-the-art systems in all the visual classification tasks on various datasets."

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

Transfer learning beyond using off-shelf CNN feature

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



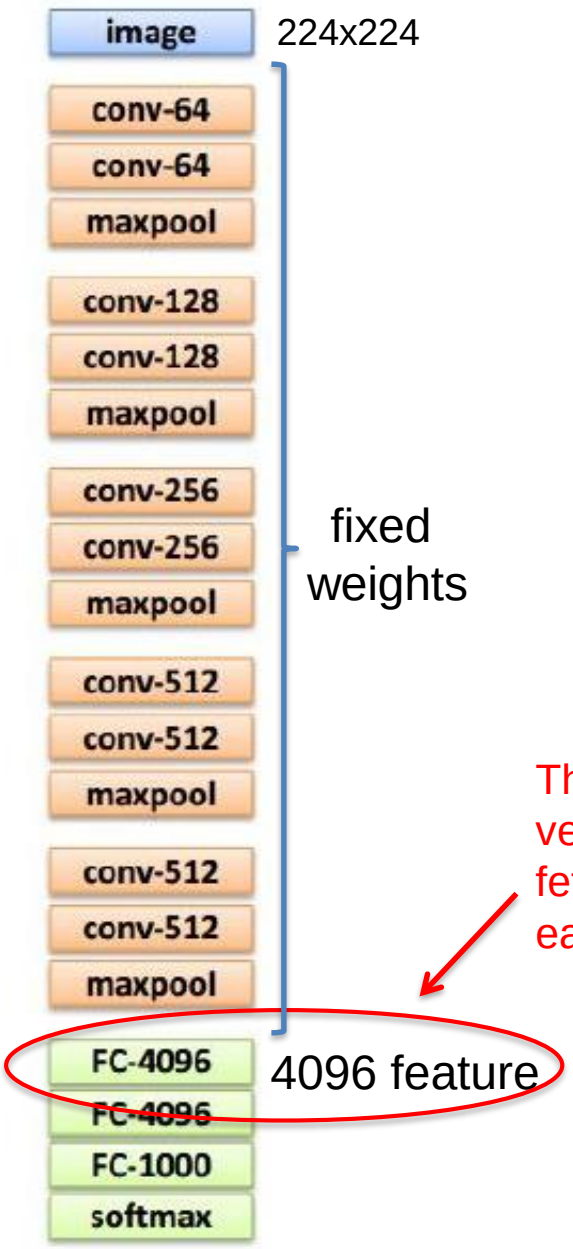
The strategy for fine-tuning depends on the size of the data set and the type of images:

	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

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

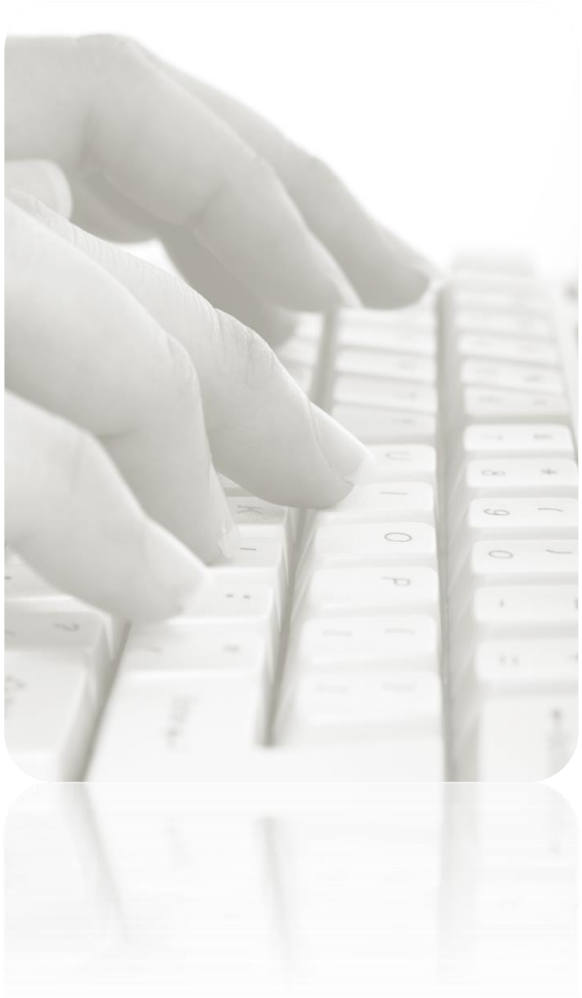
Slide credits (modified): cs231n

Exercise: Use CNN features for classification of 8 celebs



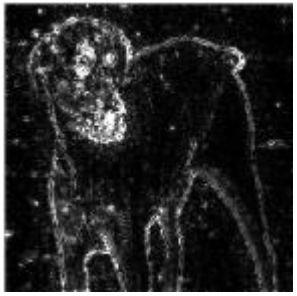
Work through exercises
'transfer_learning'

These CNN feature vectors have been fetched and stored for each image



What does the CNN look at?

Which pixels are important for the classification?
Saliency Maps



Determine the strength of the pixels influence on the correct class score:

Forward image through trained CNN.
Start from softmax neuron of correct class and set its gradient to 1. Back-propagate the gradient through the CNN.

Visualize the gradient that arrives at the image as 2D heatmap (each pixel intensity corresponds to the absolute value of the gradients at this position maximized over the channels).

Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps, Karen Simonyan, Andrea Vedaldi, Andrew Zisserman, 2014

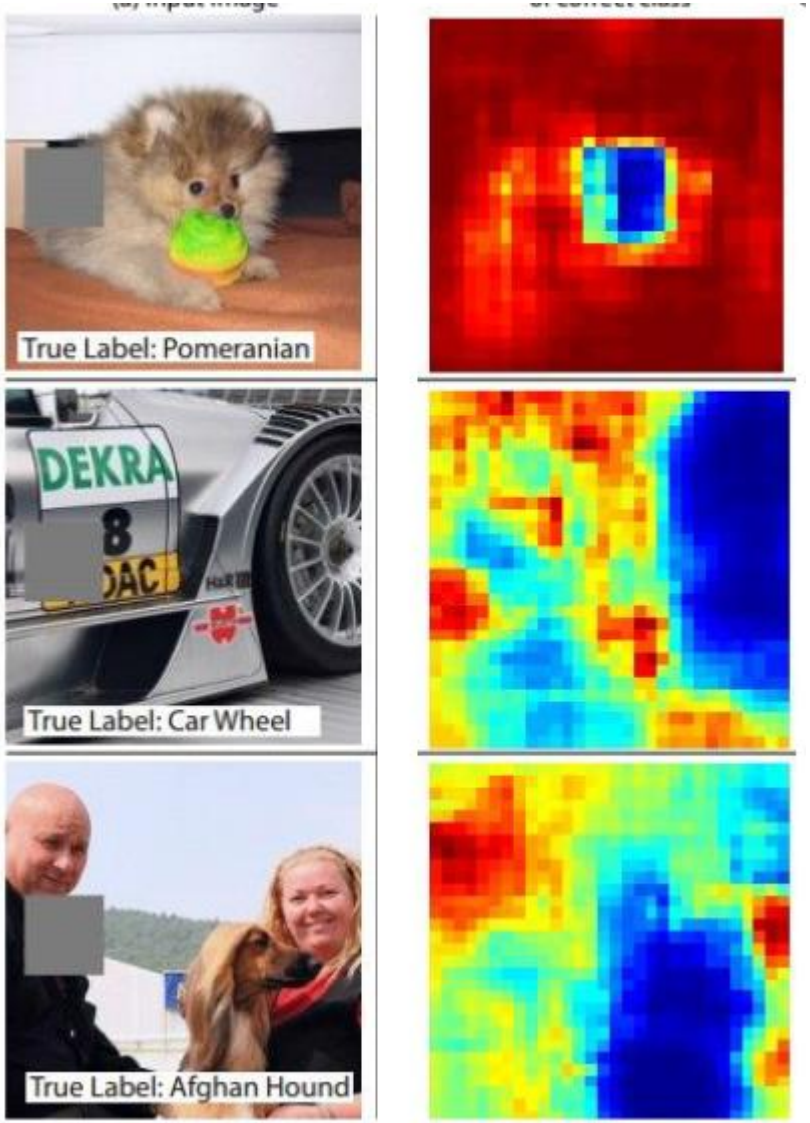
Example how to compute saliency with keras

image credit: cs231n

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,…,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~x)

(a) Husky classified as wolf

(b) Explanation

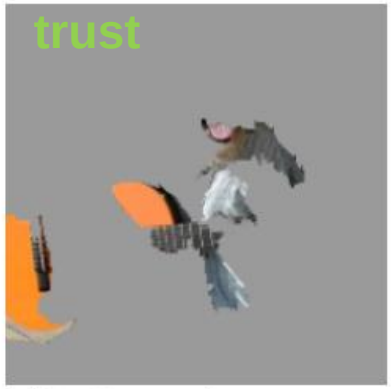
-> presence of snow was used to distinguish wolf and husky

-> 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*

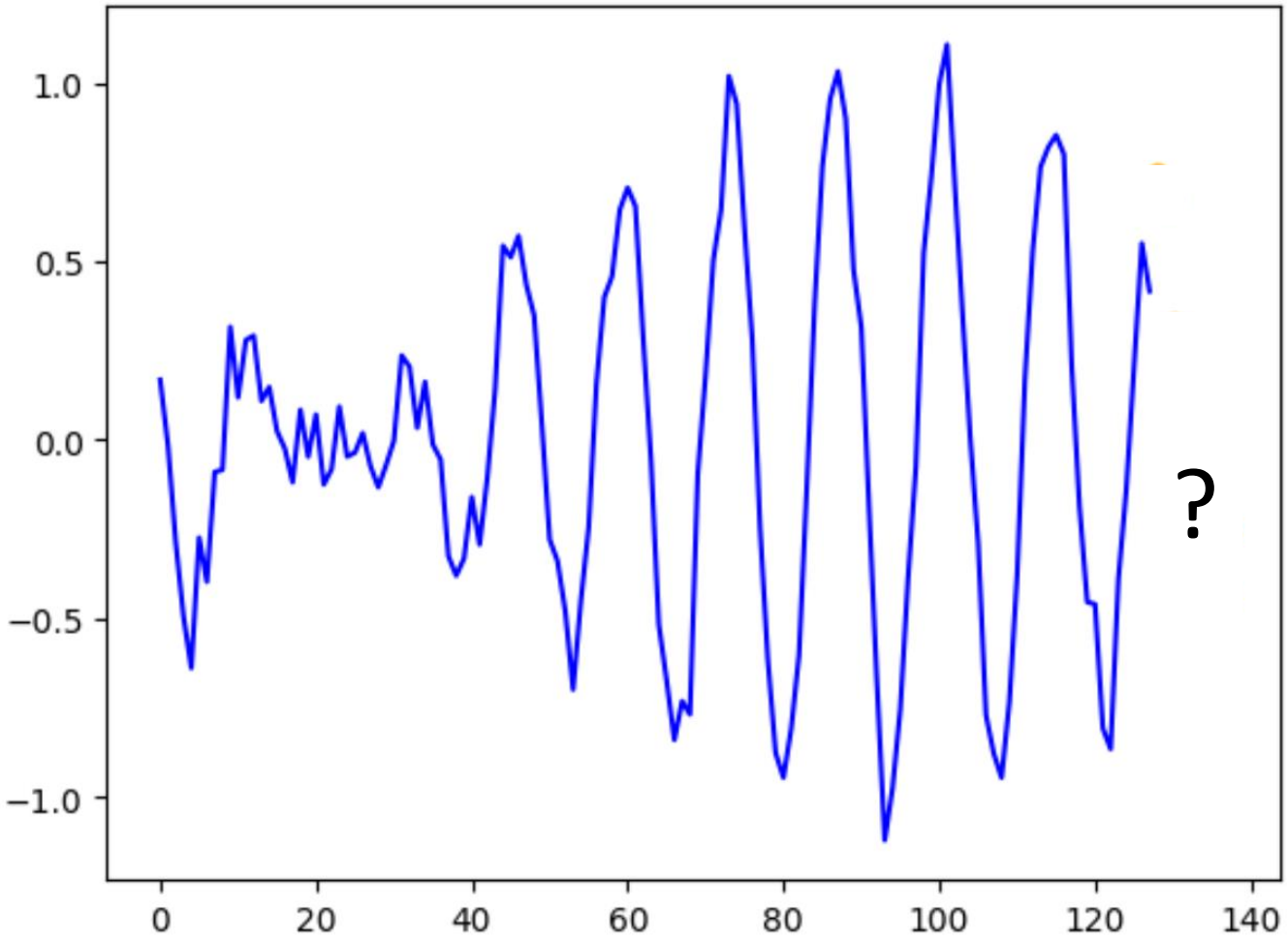
(d) Explaining *Labrador*

DOI: http://dx.doi.org/10.1145/2939672.2939778

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

1D CNNs for sequence data

How to make predictions based on a given time series?

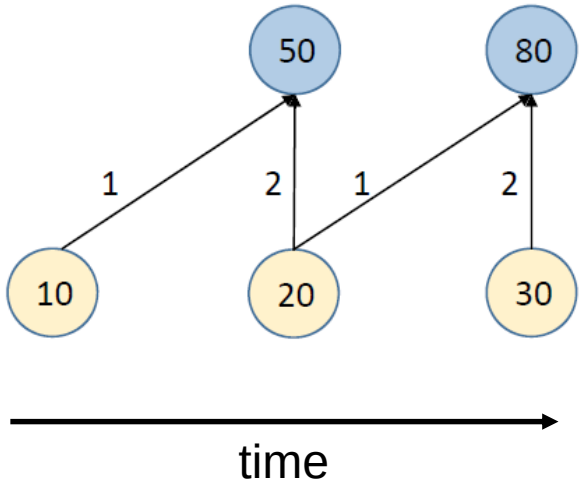


1D "causal" convolution for time-ordered data

Toy example:

Input X: 10,20,30

1D kernel of size 2: 1,2



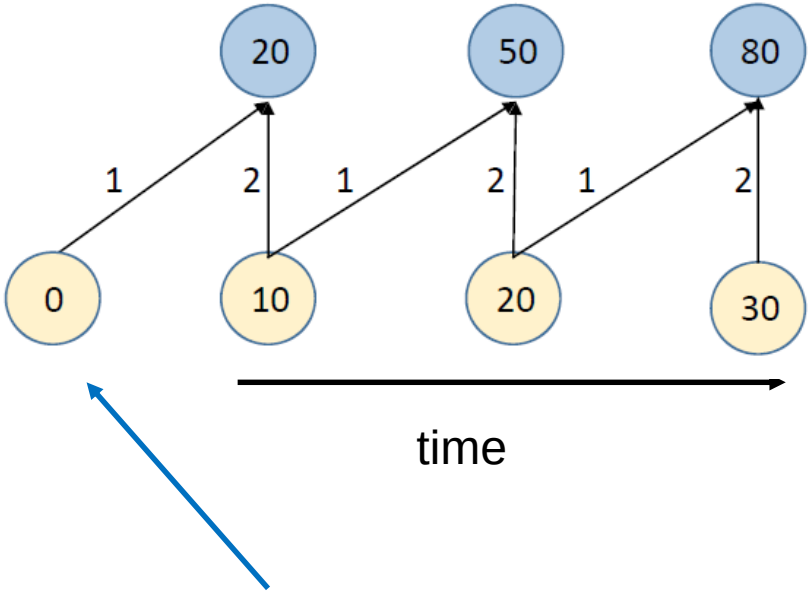
It's called "causal" networks, because the architecture ensured that only information from the past has an influence on the present and future.

Zero-padding in 1D "causal" CNNs

Toy example:

Input X: 10,20,30

1D kernel of size 2: 1,2

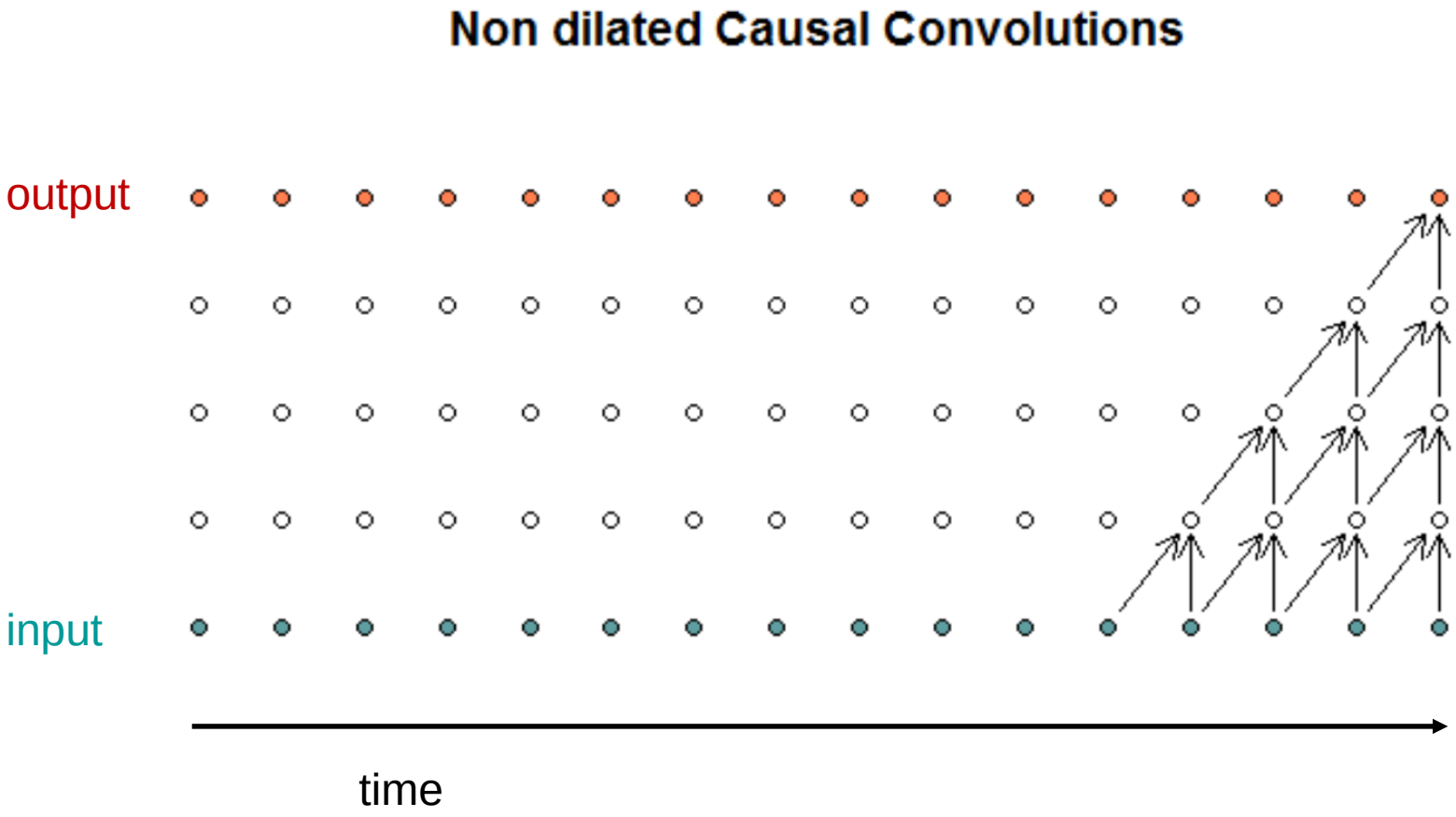


To make all layers the same size, a zero padding is added to the beginning of the input layers

1D "causal" convolution in Keras

```
model = Sequential()
model.add(Convolution1D(filters=1,
                        kernel_size=2,
                        padding='causal',
                        dilation_rate=1,
                        use_bias=False,
                        batch_input_shape=(None,3, 1)))
model.summary()
```

Stacking 1D "causal" convolutions without dilation



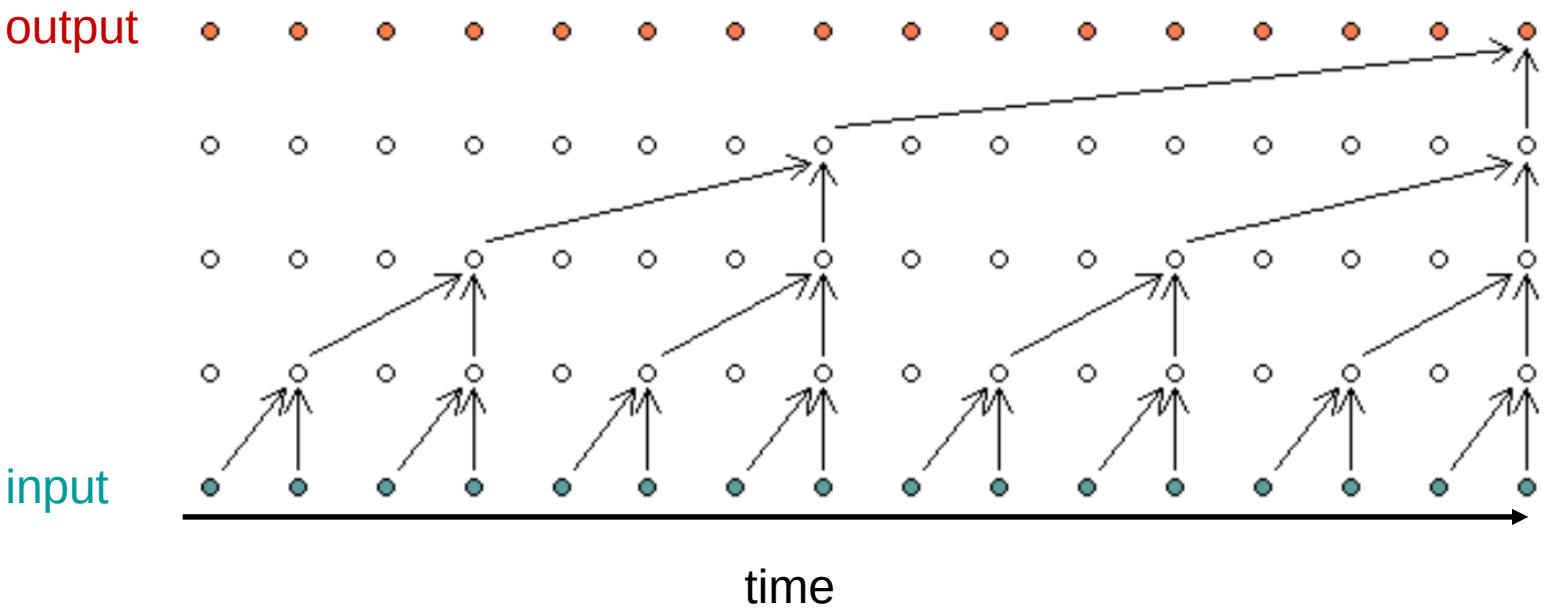
Stacking k causal 1D convolutions with kernel size 2 allows to look back k time-steps.

After 4 layers each neuron has a "memory" of 4 time-steps back in the past.

https://deepmind.com/blog/wavenet-generative-model-raw-audio/

Dilation allows to increase receptive field

To increase the memory of neurons in the output layer, you can use "dilated" convolutions:



After 4 layers each neuron has a "memory" of 15 time-steps back in the past.

https://deepmind.com/blog/wavenet-generative-model-raw-audio/

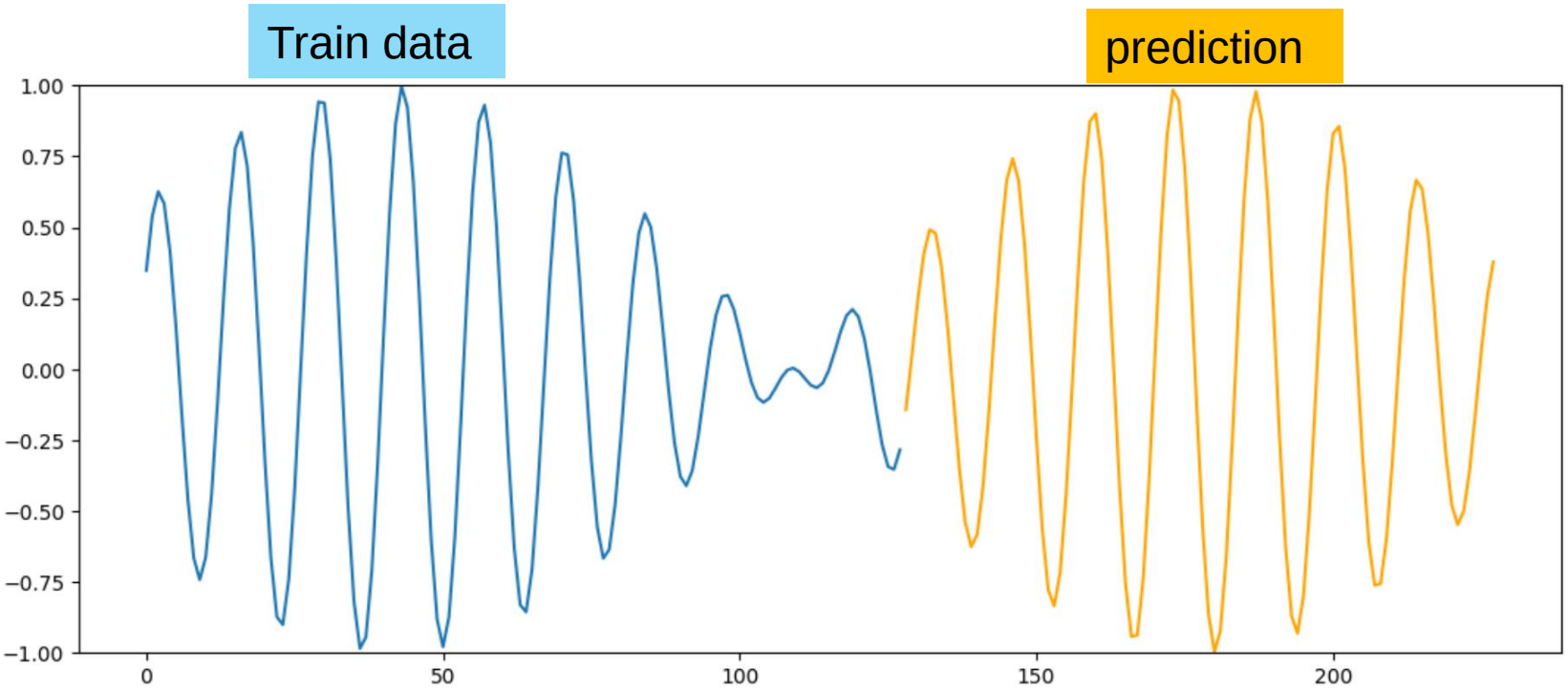
Dilated 1D causal convolution in Keras

To use time-dilated convolutions, simply use the argument rate dilation_rate=... in the Convolution1D layer.

```
X,Y = gen_data(noise=0)

modeldil = Sequential()
#<------ Just replaced this block
modeldil.add(Convolution1D(filters=32, kernel_size=ks, padding='causal', dilation_rate=1,
                           batch_input_shape=(None, None, 1)))
modeldil.add(Convolution1D(filters=32, kernel_size=ks, padding='causal', dilation_rate=2))
modeldil.add(Convolution1D(filters=32, kernel_size=ks, padding='causal', dilation_rate=4))
modeldil.add(Convolution1D(filters=32, kernel_size=ks, padding='causal', dilation_rate=8))
#<------ Just replaced this block

modeldil.add(Dense(1))
modeldil.add(Lambda(slice, arguments={'slice_length':look_ahead}))

modeldil.summary()

modeldil.compile(optimizer='adam',loss='mean_squared_error')

histdil = modeldil.fit(X[0:800], Y[0:800],
                       epochs=200,
                       batch_size=128,
                       validation_data=(X[800:1000],Y[800:1000]), verbose=0)
```

Dilated 1D causal CNNs help if long memory is needed

Dilated 1D CNNs can picked up the long-range time dependencies.



If you want to get a better understanding how 1D convolution work, you can go through the notebook at https://github.com/tensorchiefs/dl_book/blob/master/chapter_02/nb_ch02_04.ipynb.

Modern CNN architectures

Review of ImageNet winning CNN architectures

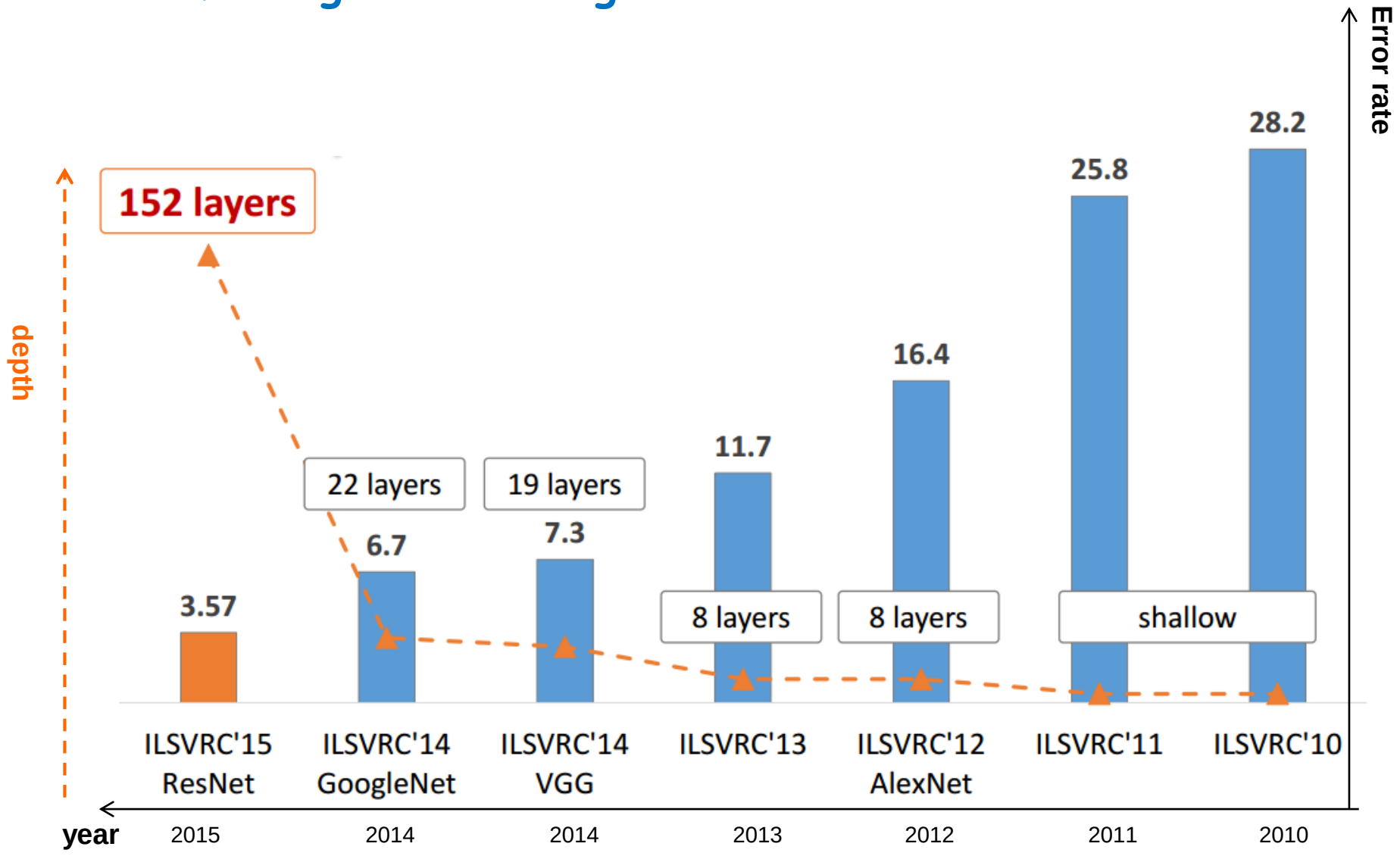


Image credits (modified): K.He, X.Zhang, S.Ren, J.Sun. “Deep Residual Learning for Image Recognition”. arXiv 2015

Going deeper is easy – or not?

The challenge is to design a network in which the gradient can reach all the layers of a network which might be dozens, or even hundreds of layers deep.

This was achieved by some recent improvements, such as ReLU and batch normalization, and by designing the architecture in a way which allows the gradient to reach deep layer, e.g. by additional skip connections.

LeNet-5 1998: first CNN for ZIP code recognition

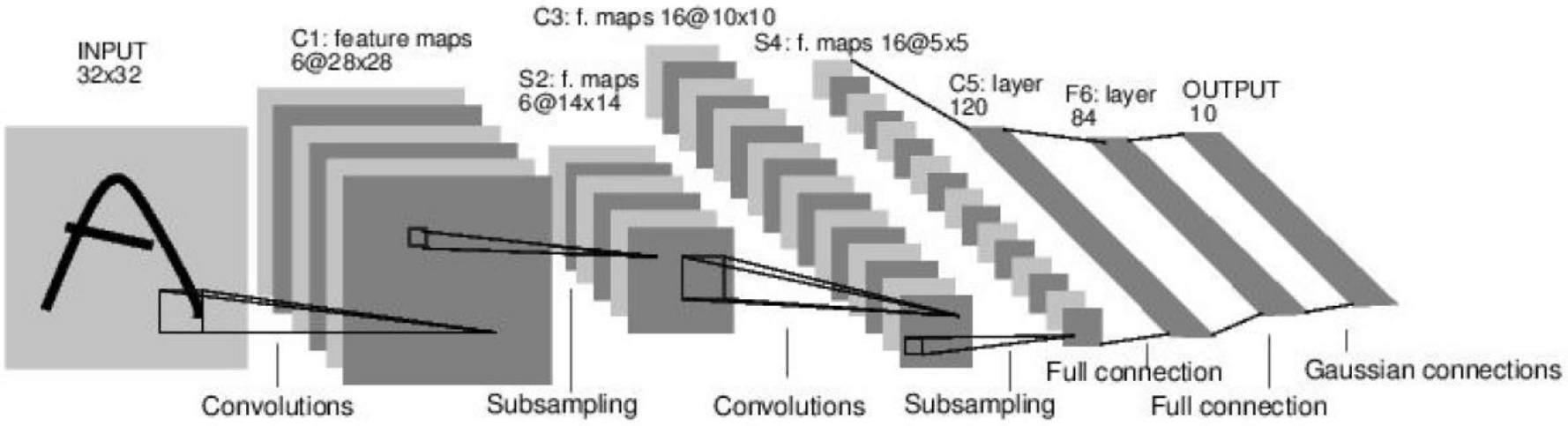
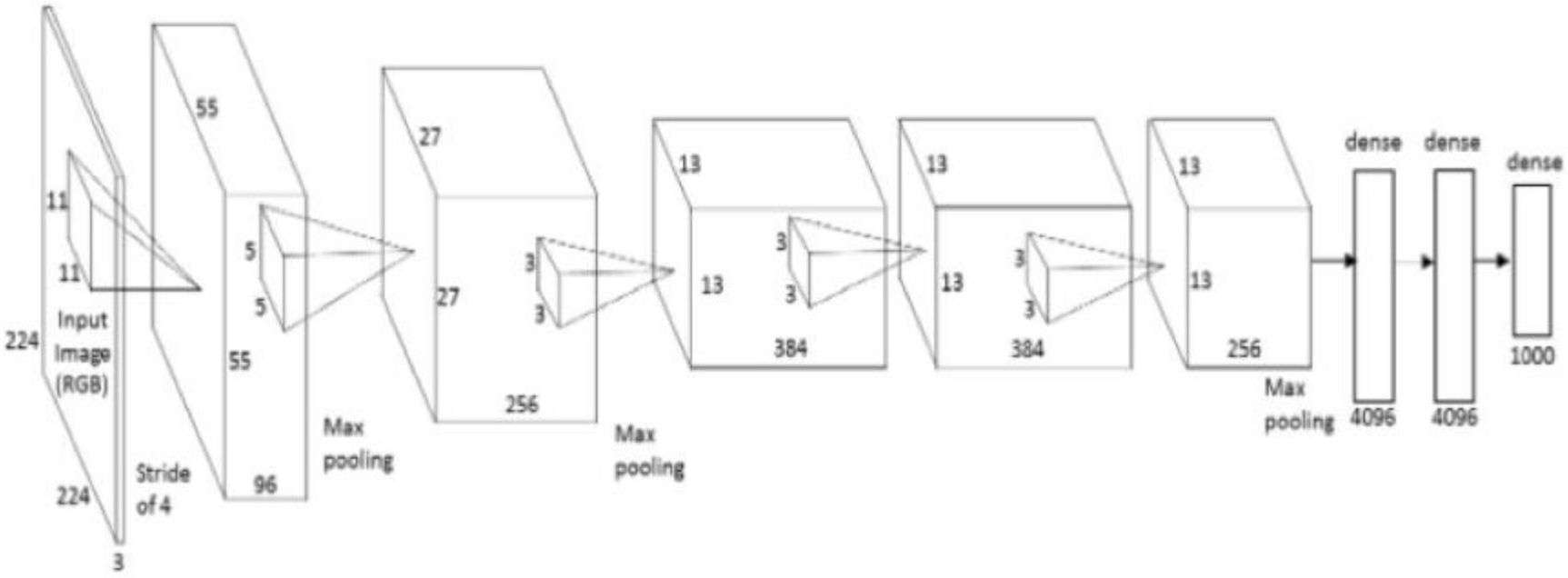


Image credits: http://yann.lecun.com/exdb/publis/pdf/lecun-98.pdf

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]

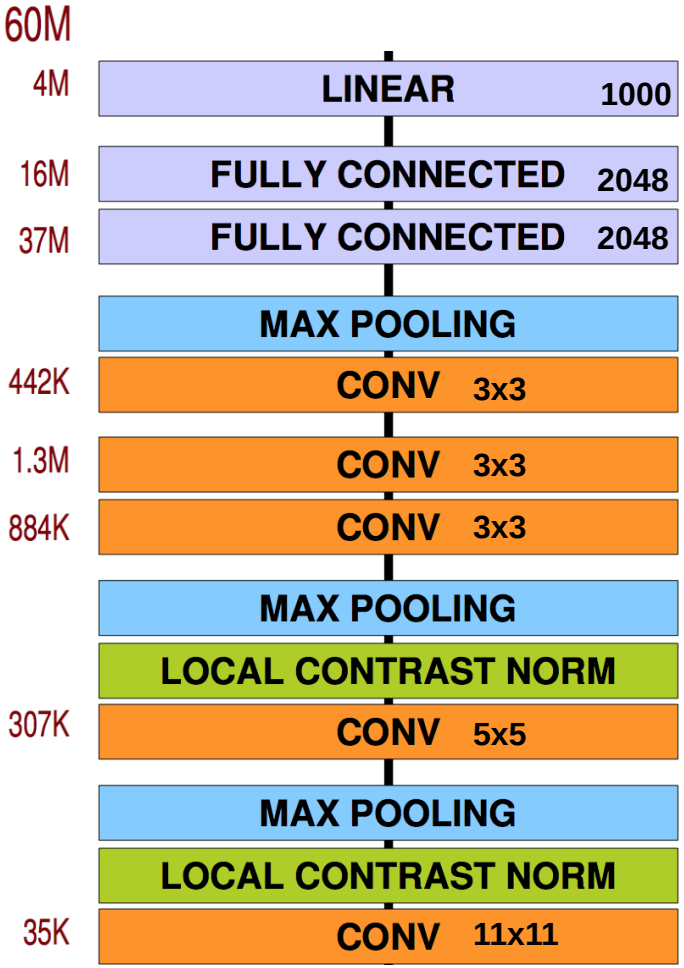
http://yann.lecun.com/exdb/lenet/index.html

"AlexNet", 2012 winner of the imageNet challenge



Architecture of AlexNet

"AlexNet", 2012 winner of the imageNet challenge



Seminal paper. 26.2% error → 16.5%

- Dropout
- ReLU instead of sigmoid
- Used data augmentation techniques

- 5 conv layer (filter: 11x11, 5x5, 3x3, 3x3, 3x3)

- Parallelisation on two GPUs
- Local Response Normalization (not used anymore)

The trend in modern CNN architectures goes to small filters

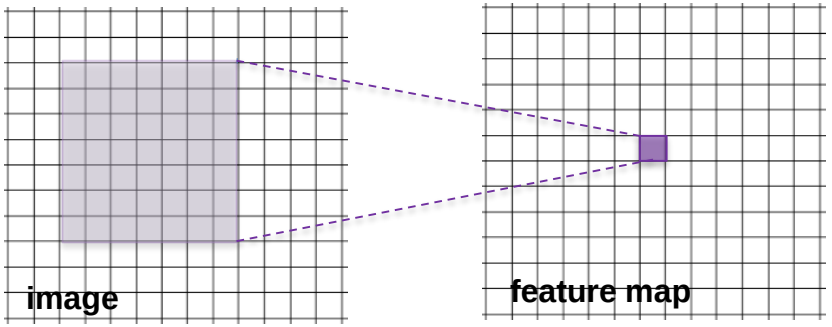
Why do modern architectures use very small filters?

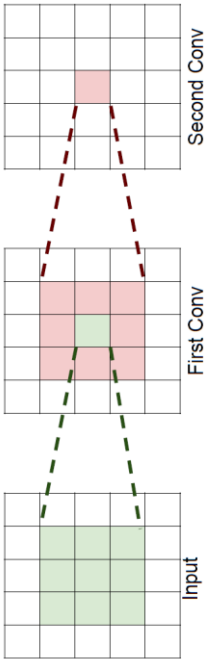
Determine the receptive field in the following situation:

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

49 weights

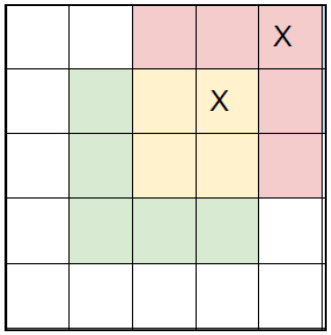
Answer1): 7x7





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

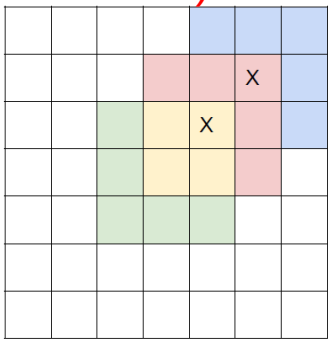
Answer 2): 5x5



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

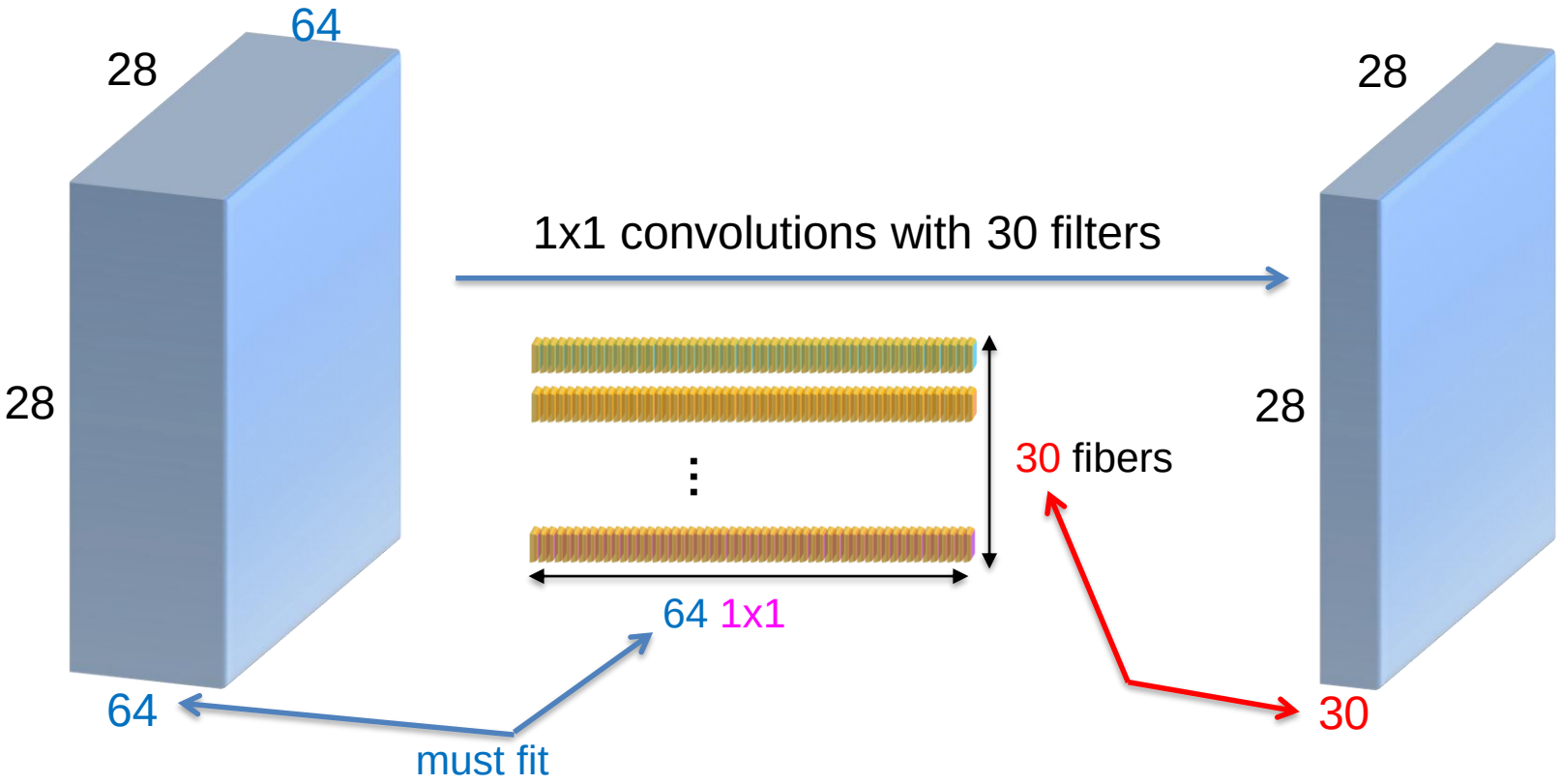
3*9=27 weights

Answer 3): 7x7



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

Go to the extreme: What is about filter size 1?
1x1 convolutions act only in depth dimension

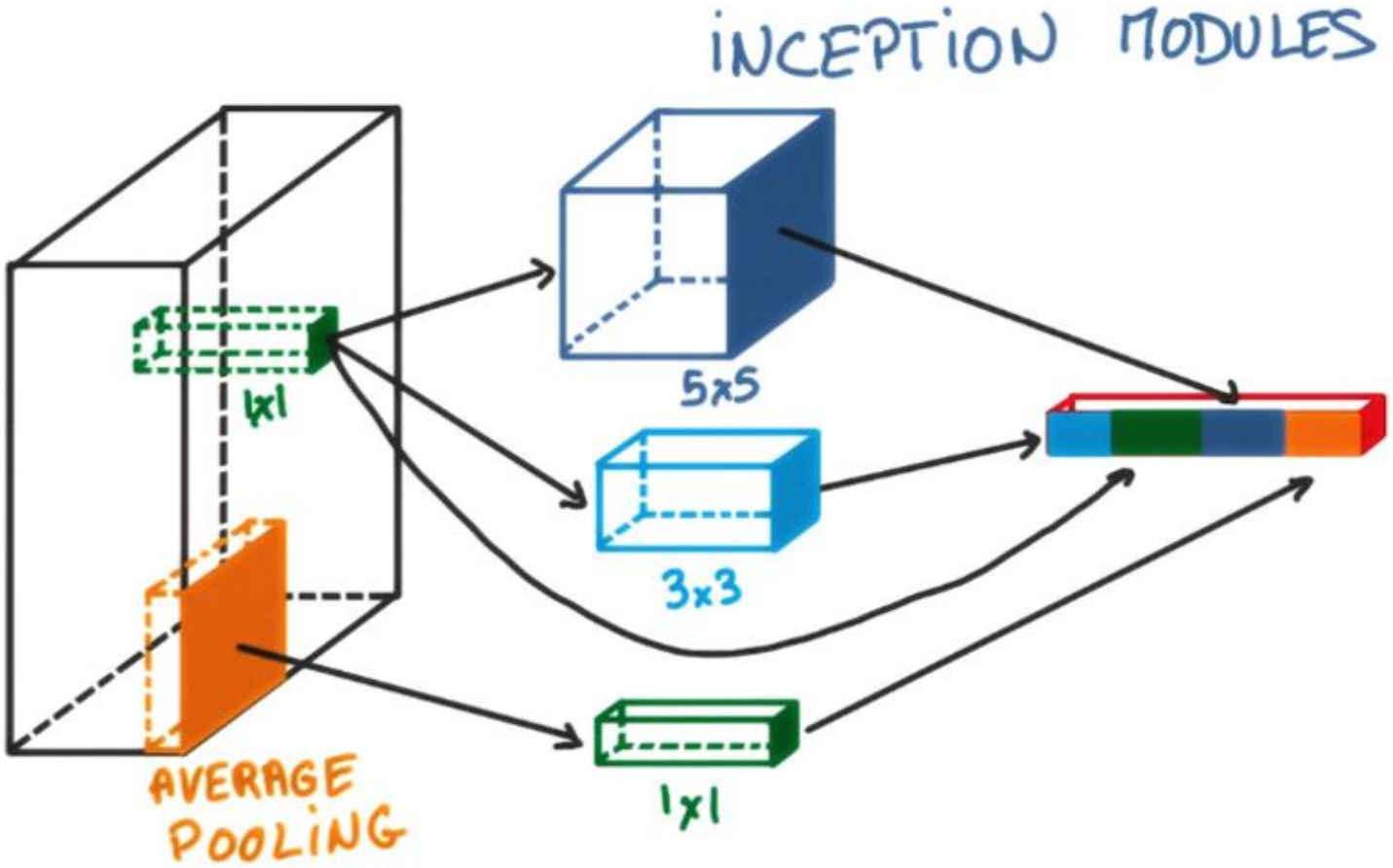


1x1 convolution act along a “fiber” in depth dimension across the channels.

→ efficient way to reduce/change the depth dimension

→ simultaneously introduce more non-linearity

The idea of inception modules



Between two layers just do several operations in parallel: pooling and 1x1 conv, and 3x3 and 5x5. “same”-conv and concatenate them together. Benefit: total number of parameters is small, yet performance better.

Slide (modified) credit: Google MOOC on tensorflow

How to concatenate tensors of different dimensions?

DepthConcat needs to make the tensor the same in all, but the depth dimension:

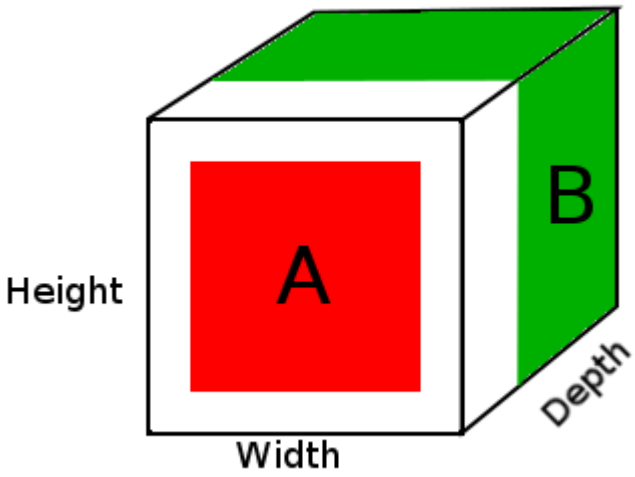
To deal with different output dimensions, the largest spatial dimension is select and zero-padding around the smaller dimensions is added.

Usually depth gets large!
Do 1x1 conv afterwards!

Pseudo code for an example:

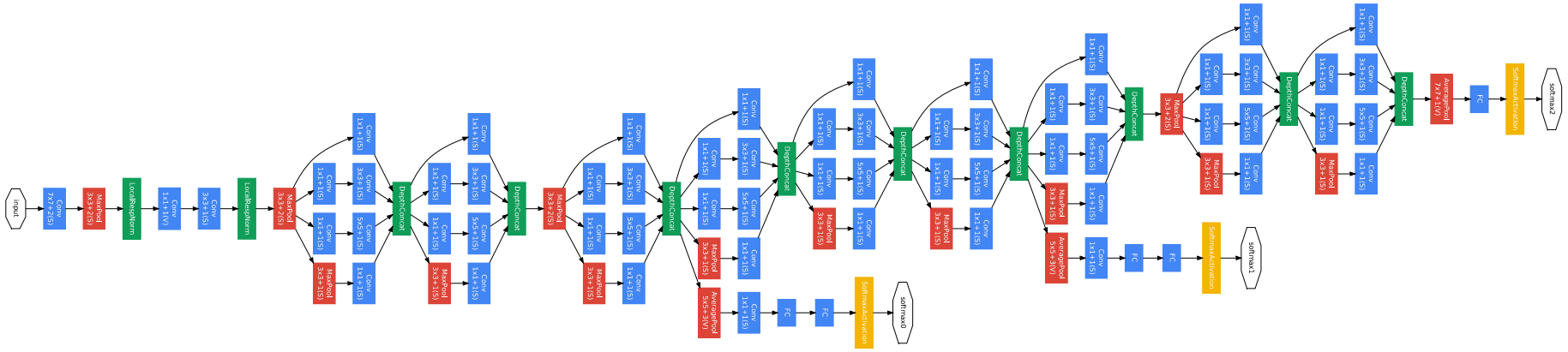
```
A = tensor of size (14, 14, 2)
B = tensor of size (16, 16, 3)
result = DepthConcat([A, B])
```

where result with have a
height of 16,
width of 16 and a
depth of 5 (2 + 3)

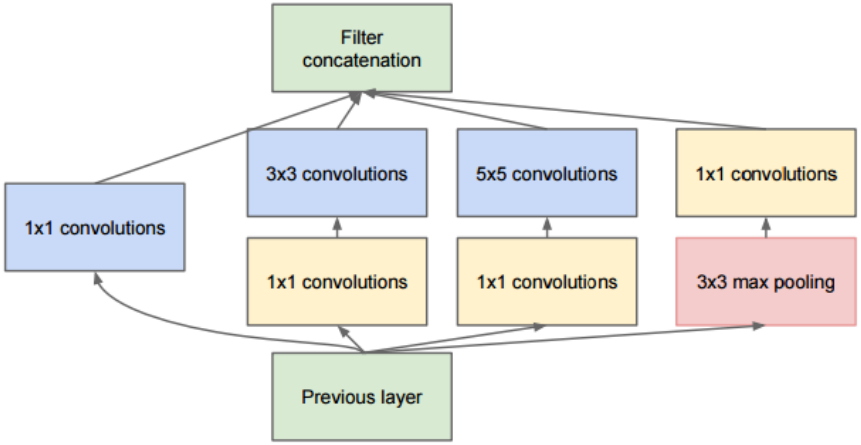


See: http://stats.stackexchange.com/questions/184823/how-does-the-depthconcat-operation-in-going-deeper-with-convolutions-work

Winning architecture (GoogLeNet, 2014)



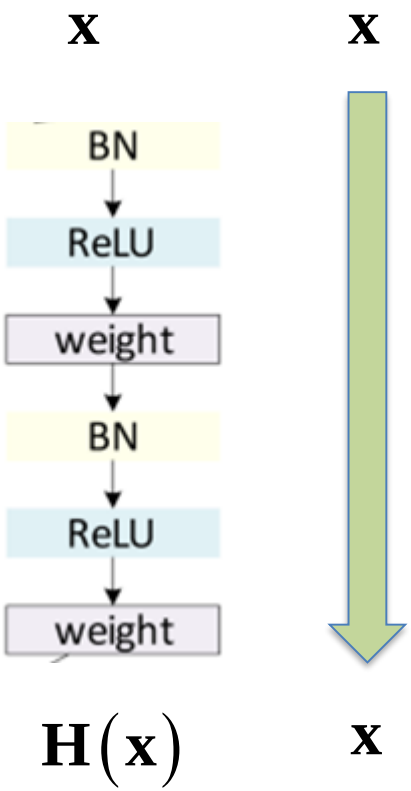
The inception module (convolutions and maxpooling)



Few parameters, hard to train.
Comments see here

Going deeper with convolution http://arxiv.org/abs/1409.4842

Highway Networks: providing a highway for the gradient



Idea: Use nonlinear transform T to determine how much of the output **y** is produced by H or the identity mapping. Technically we do that by:

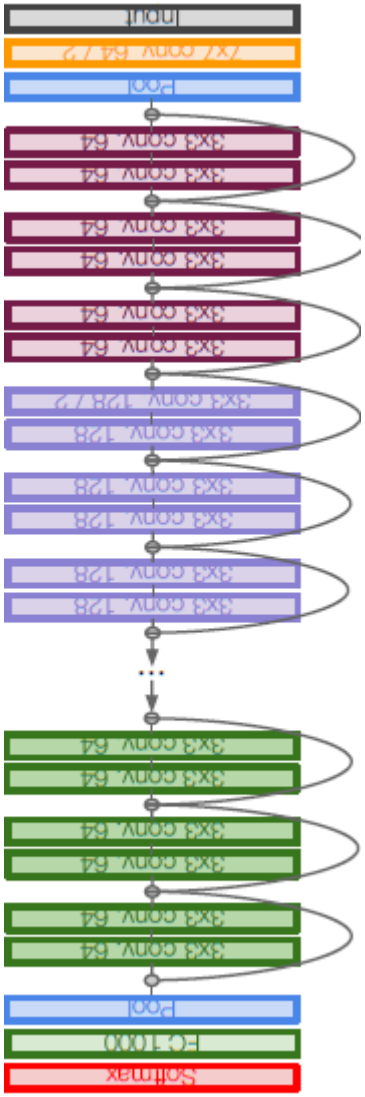
$$\mathbf{y} = H(\mathbf{x}, \mathbf{W_H}) \cdot T(\mathbf{x}, \mathbf{W_T}) + \mathbf{x} \cdot (1 - T(\mathbf{x}, \mathbf{W_T}))$$

Special case:

$$\mathbf{y} = \begin{cases} \mathbf{x}, & \text{if } T(\mathbf{x}, \mathbf{W_T}) = \mathbf{0} \\ H(\mathbf{x}, \mathbf{W_H}), & \text{if } T(\mathbf{x}, \mathbf{W_T}) = \mathbf{1} \end{cases}$$

This opens a highway for the gradient:

$$\frac{d\mathbf{y}}{d\mathbf{x}} = \begin{cases} \mathbf{I}, & \text{if } T(\mathbf{x}, \mathbf{W_T}) = \mathbf{0} \\ H'(\mathbf{x}, \mathbf{W_H}), & \text{if } T(\mathbf{x}, \mathbf{W_T}) = \mathbf{1} \end{cases}$$



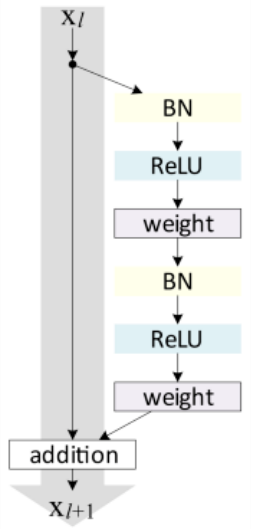
Srivastava, Greff, Schmidhuber 2014, “Highway Networks” https://arxiv.org/pdf/1505.00387.pdf

"ResNet" from Microsoft 2015 winner of imageNet

152 layers

ResNet basic design (VGG-style)

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



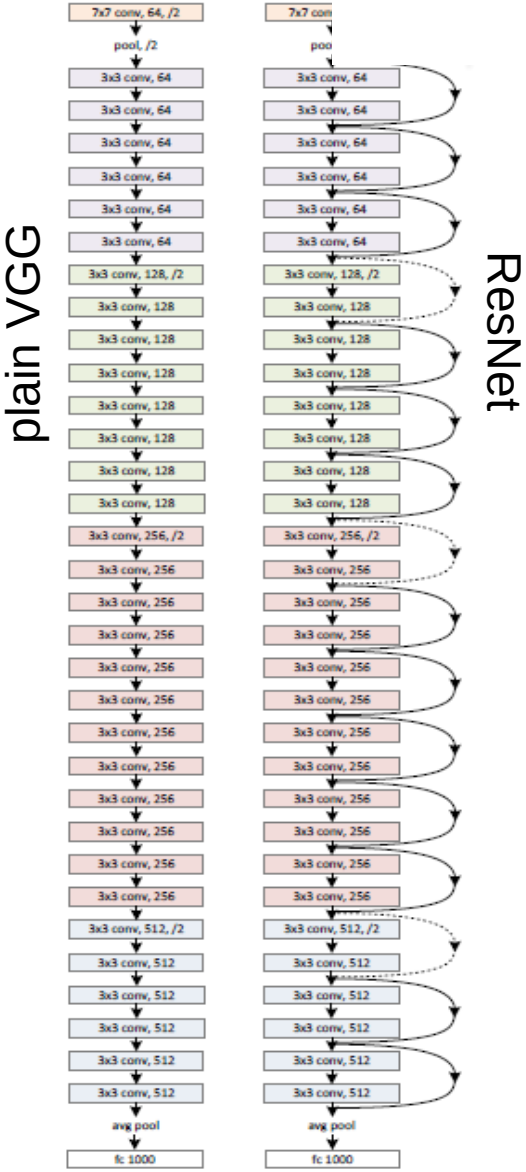
$F(x_l)$

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

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

- 2nd 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-trainined model is available

http://arxiv.org/abs/1409.1556