Zurich University of Applied Sciences

CNNs part II

Beate Sick

Institut für Datenanalyse und Prozessdesign

Zürcher Hochschule für Angewandte Wissenschaften

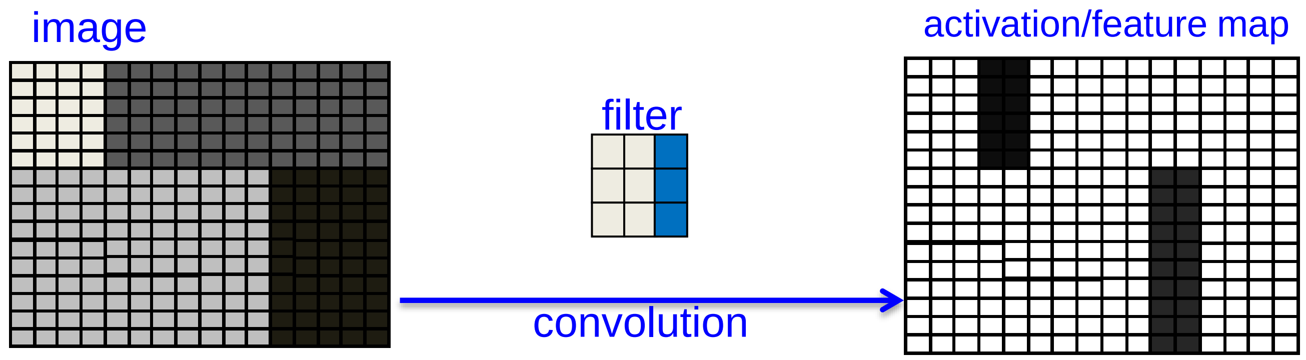
sick@zhaw.ch

Topics

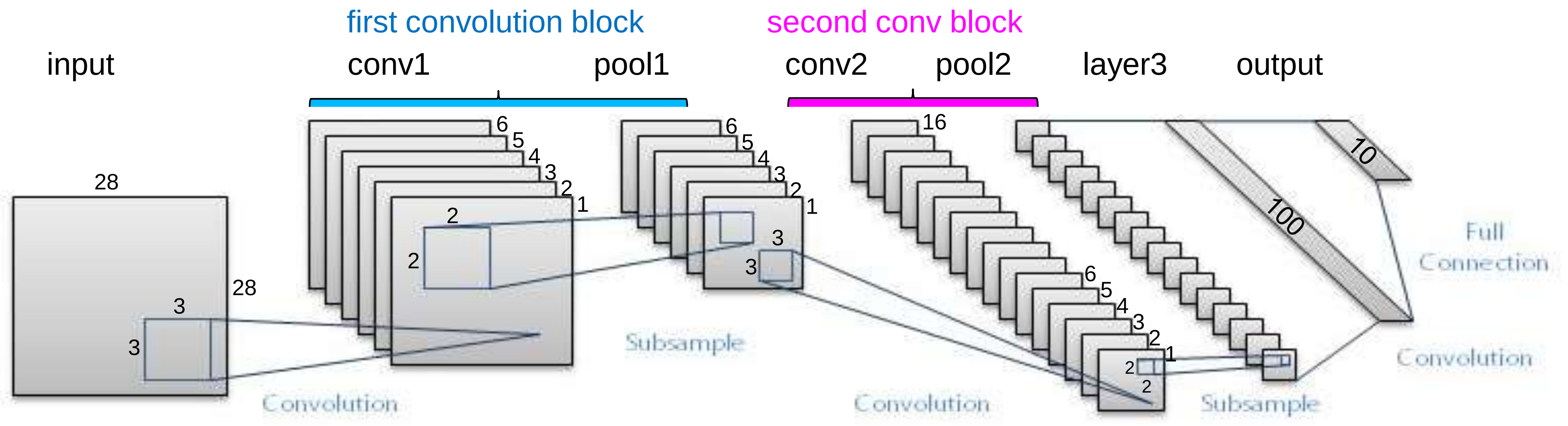
- Recap basics of CNNs
- Important ingredients to get a well performing (simple) CNN
  - Weight initialization, choice of activation function, rescaling input (recap)
  - Dropout, batch-normalization, choice of optimizer
- What to do in case of limited data?
  - Data augmentation
  - Transfer learning: 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

Recap: Why do we want to use convolutional layer?

- We want not to rely on rigid templates (like in fully connected NN)
- We want to exploit the information that is contained in the neighborhood structure of pixels in a image.
- We want to learn hierarchical and semantic features
- We want to capture relative positions of the features in the image and not depend on absolute positions of the objects in the image
- We share weights for one filter extracting the activation of the same feature at each positions of the input volume.

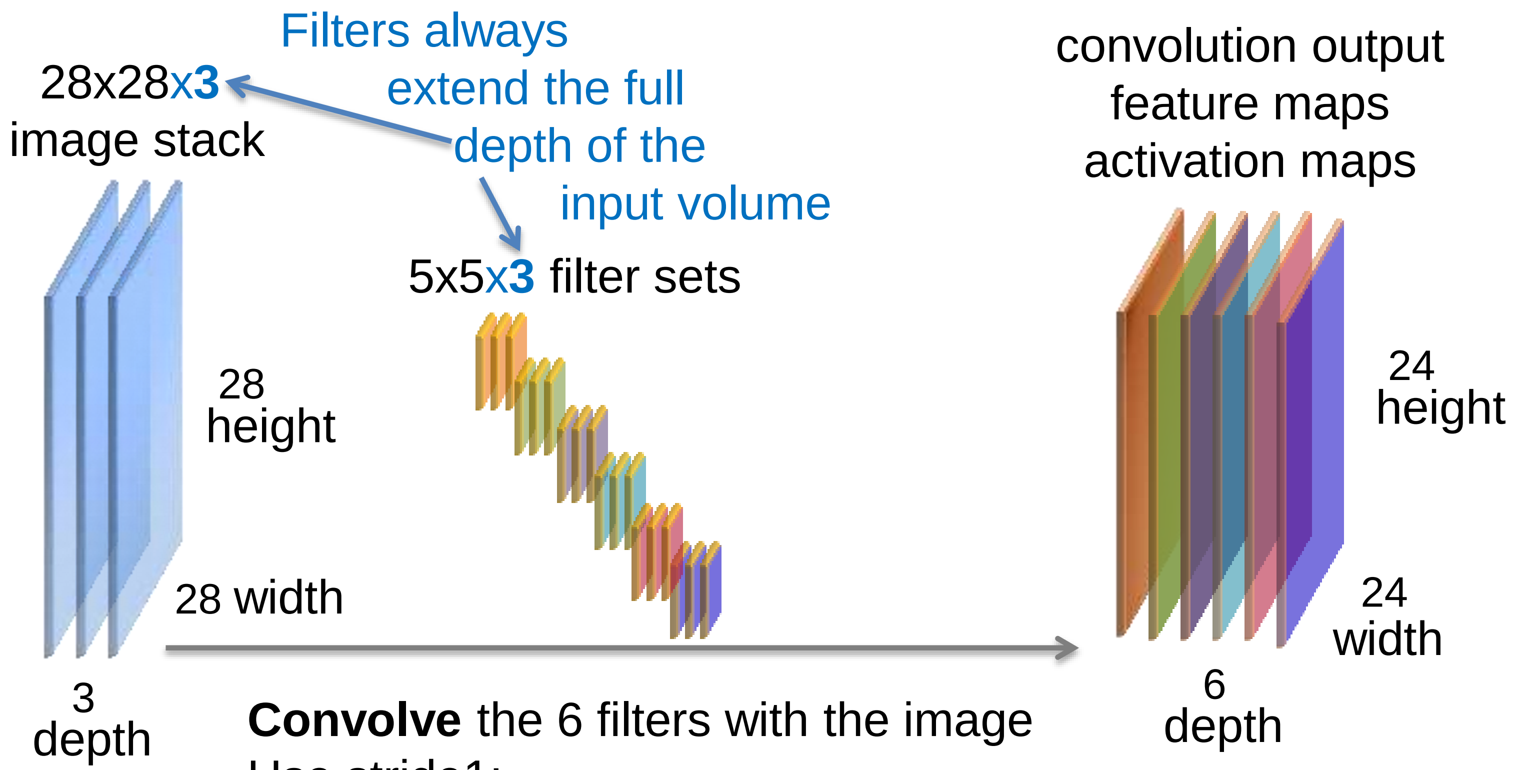


Architecture of a Convolutional Neural Network



```
from keras import layers
from keras import models
model = models.Sequential()
model.add(layers.Conv2D(6, (3, 3), activation='relu', input_shape=(28, 28, 1)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(16, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten())
model.add(layers.Dense(100, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
```

A convolution block (recap)

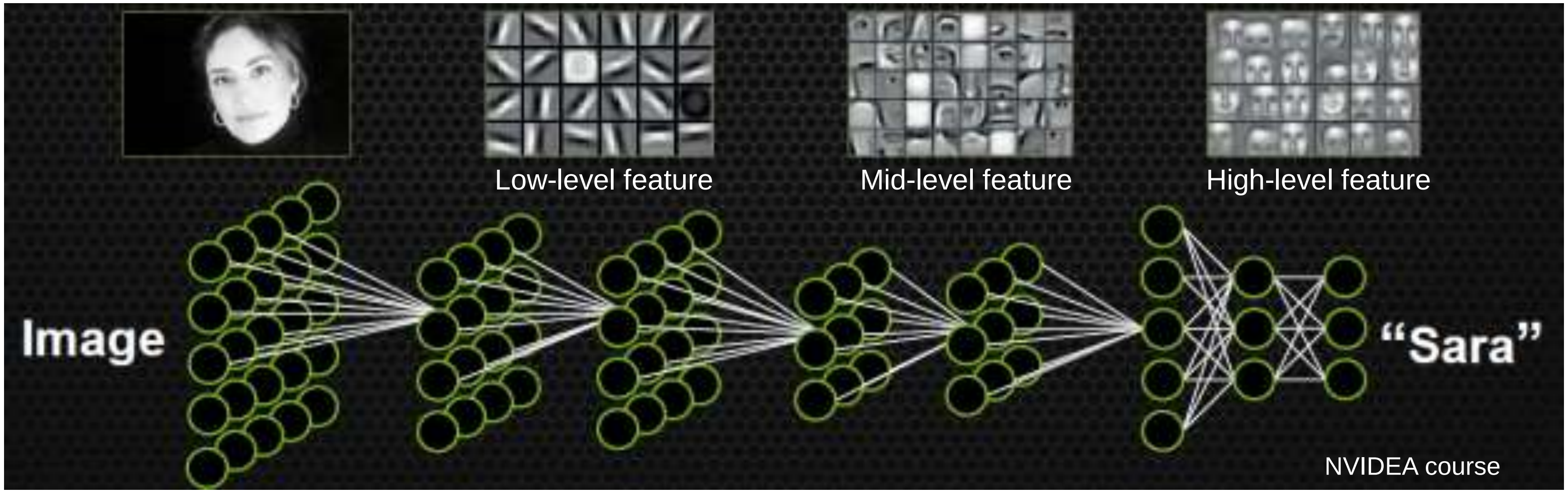


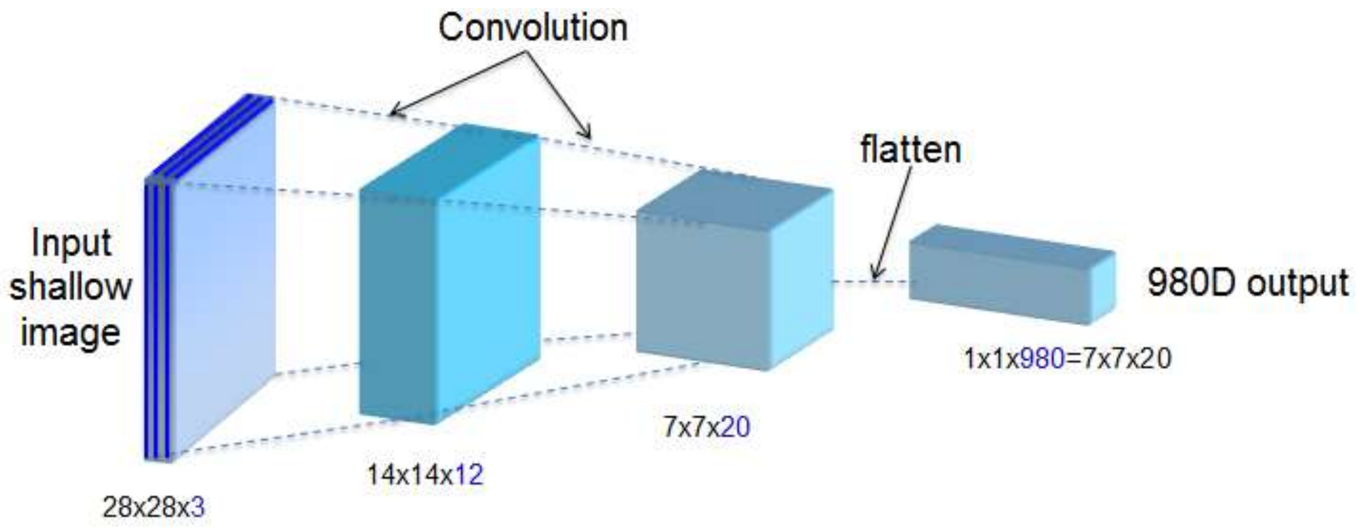
Convolve the 6 filters with the image
Use stride1:,
computing dot products at each position"

```
In Keras we do not need to specify the depth of the filter
(since Keras can deduce it):
```

```
model = Sequential()
model.add(Convolution2D(6, (5, 5), input_shape=(28, 28, 3)))
```

Goal of a CNN: learn hierarchical abstract feature





How to get a well performing CNN?

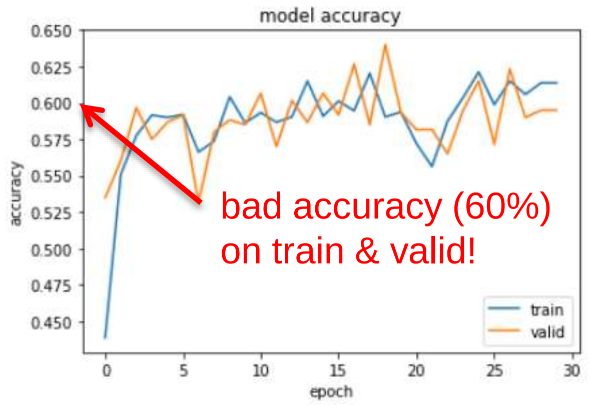
- What to do that the network can learn good weights?
  - How to initialize the weights: small, random numbers are preferred
  - How to let the gradients flow: activation functions (ReLu), preprocessing, batch-norm…
  - How to update weights: learning rate, choice of optimizer …
- What kind of image data do we need?
  - Only limited data preprocessing is needed: scaling and resizing images
  - How to detect and fight overfitting: learning curves, data augmentation, dropout…
  - What to do if data is limited: transfer learning and fine-tuning pre-trained nets
- What does a CNN learn?
  - What a CNN looks at: receptive field, image pattern that activate neurons, lime…
- Architecture matters
  - Learn from winning architectures

Why scaling the input?

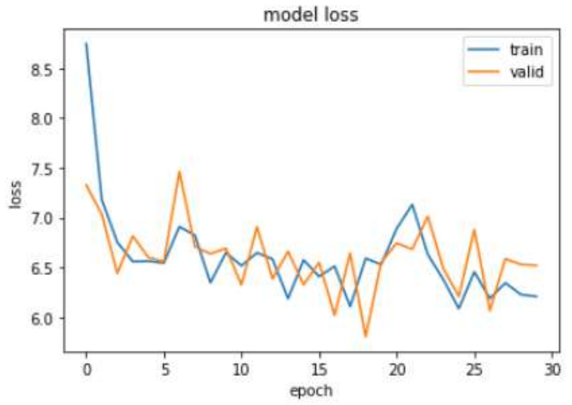
Is preprocessing required? Lessons learned from 07_cnn1

Results on MNIST subset using cnn1 with only 1 conv block 32 filter

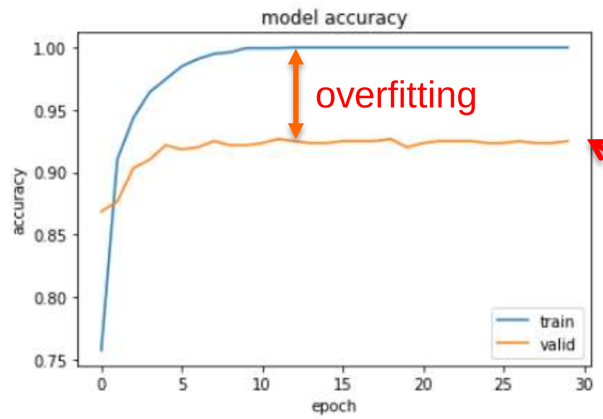
w/o standardizing the data



bad accuracy (60%) on train & valid!

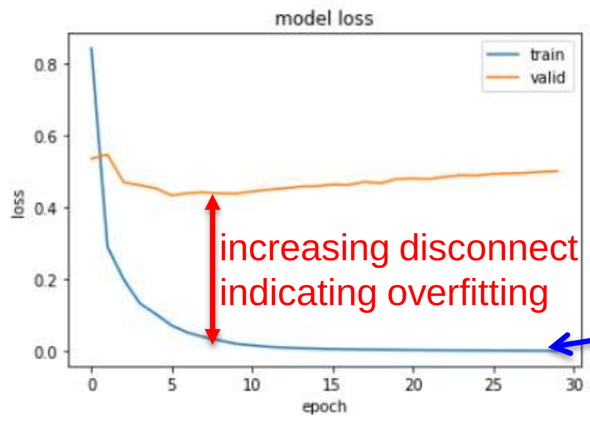


with standardizing the data



overfitting

The accuracy of cnn1 is with 92% still not much better than our fc NN!

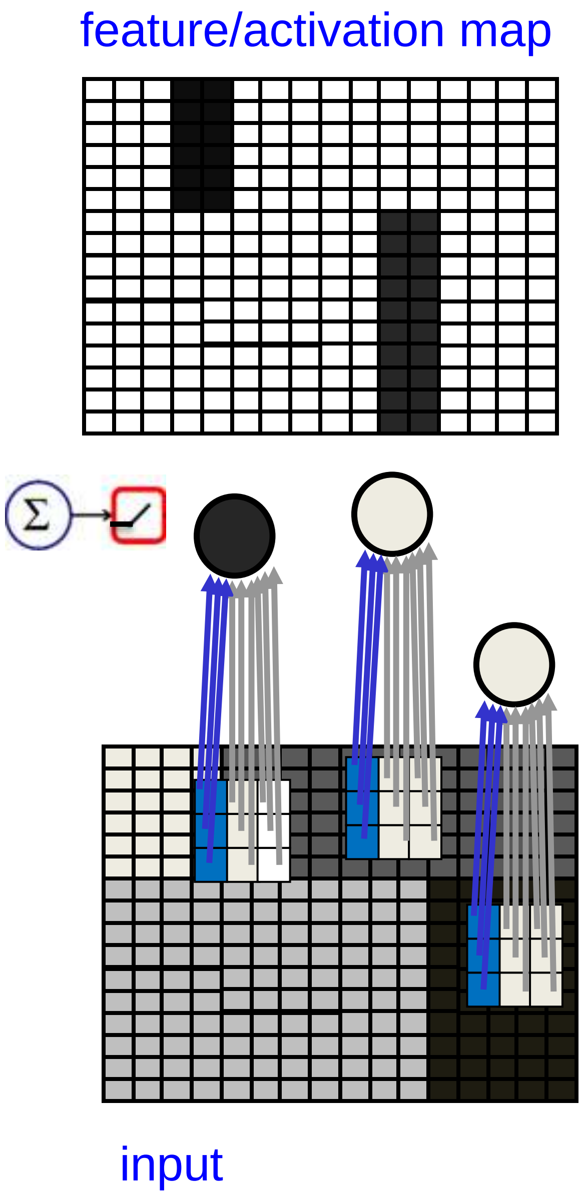


increasing disconnect indicating overfitting

A learning NN can easily memorize any (fixed) training data set and reach loss=0

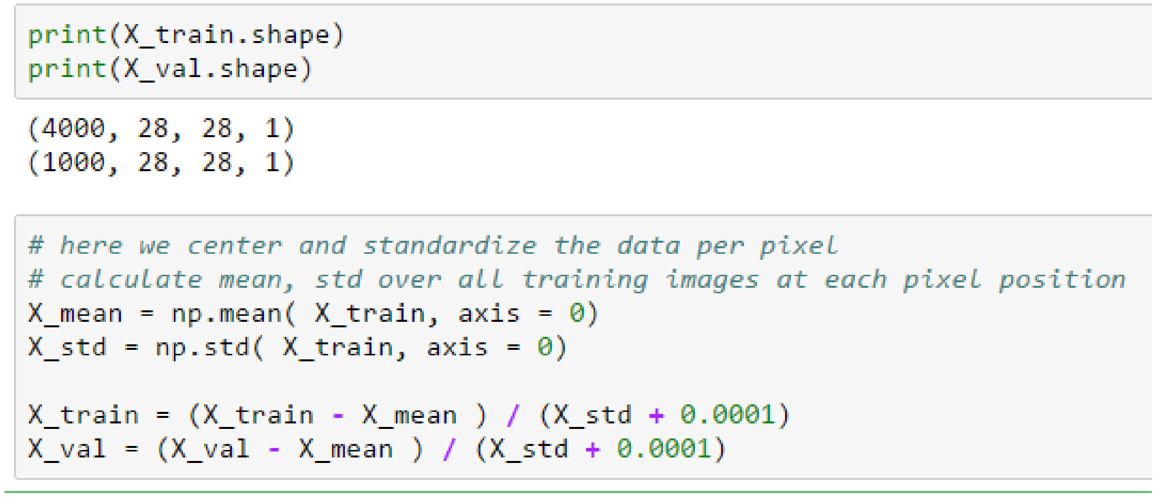
Only after standardizing the input data the CNN could be trained properly!

Why can standardizing get important for CNNs?



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

→ standardize the input pixel-wise



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

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

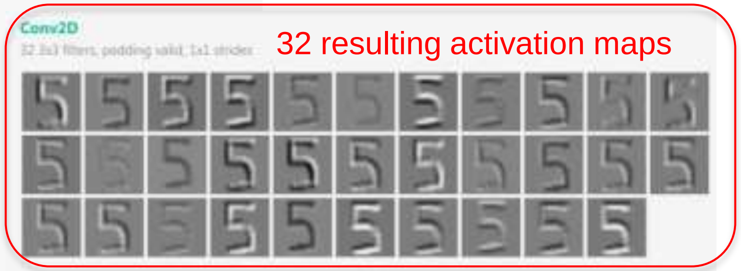
```
# here we center and standardize the data per pixel
# calculate mean, std over all training images at each pixel position
X_mean = np.mean( X_train, axis = 0)
X_std = np.std( X_train, axis = 0)

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

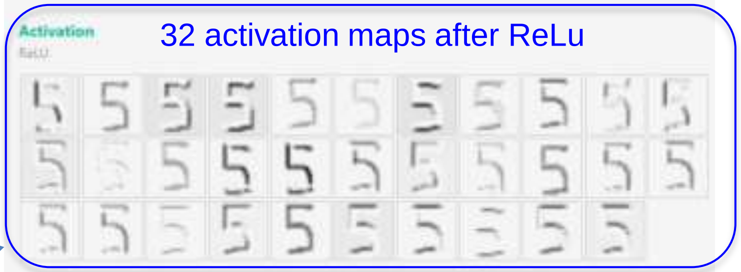
Why is our small cnn1 not outperforming a fc NN?

```
name = 'cnn1'
model = Sequential()

model.add(Convolution2D(32, (3, 3), padding='same', input_shape=(28, 28, 1)))
model.add(Activation('relu'))

model.add(Flatten())

model.add(Dense(10))
model.add(Activation('softmax'))

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

With 3x3 filters we can only detect simple and very local features like edges!

fc NN w/o hidden layer

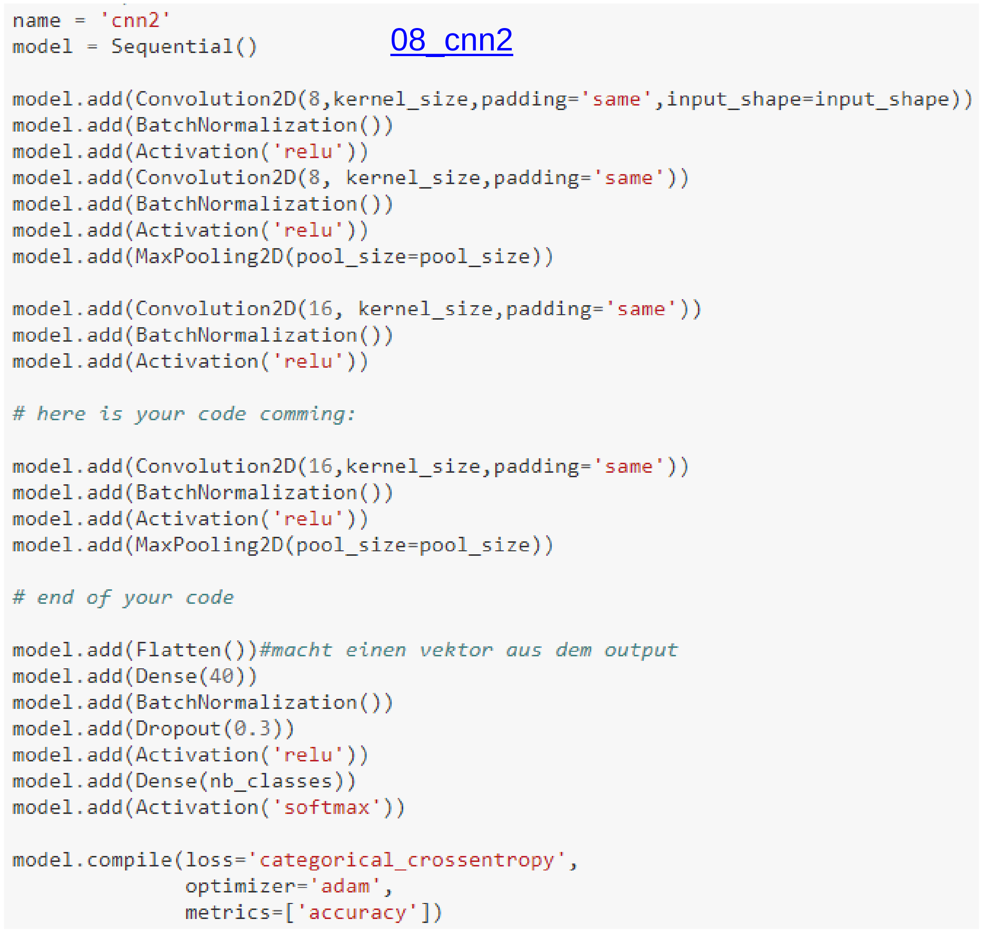




cnn live in a browser

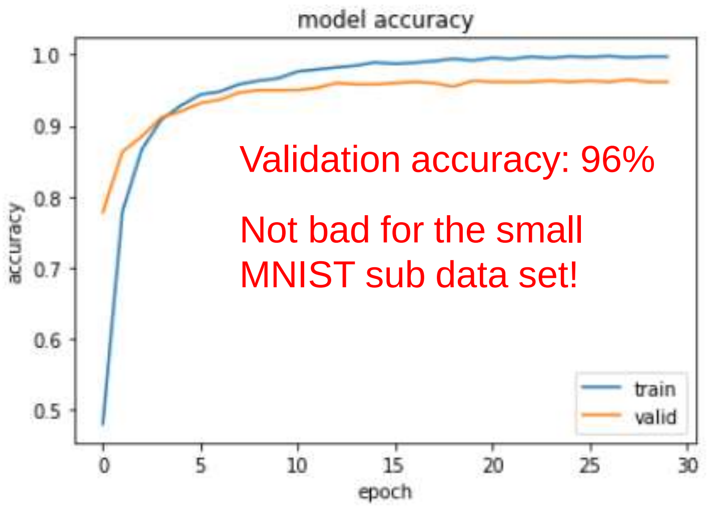
The first convolution block turns the 1-channel image into a 32 block of activation maps (still 5-looking with emphasized edges of different orientations). The following fc NN part has to find 10 rigid templates for the 32 channel image representation for the possible outputs 0,1,…9 → little improvement vs a fc NN. However: MNIST pattern can be quite good detected using rigid patterns ;-)

For better features we need to go deeper



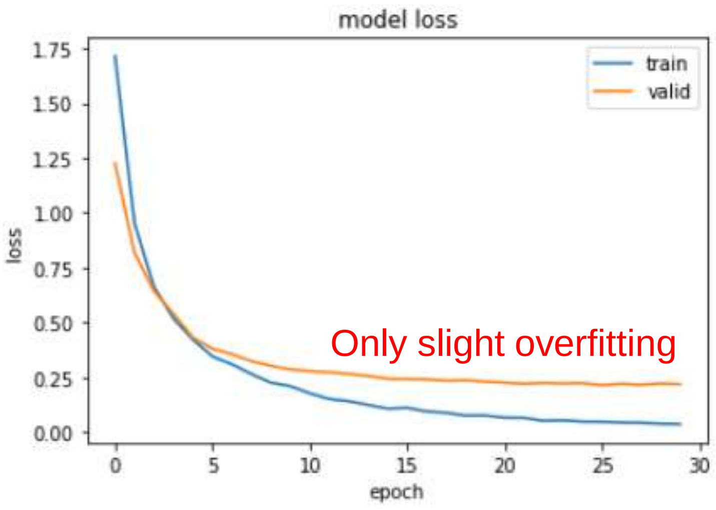
08_cnn2

```
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'])
```



Validation accuracy: 96%

Not bad for the small MNIST sub data set!

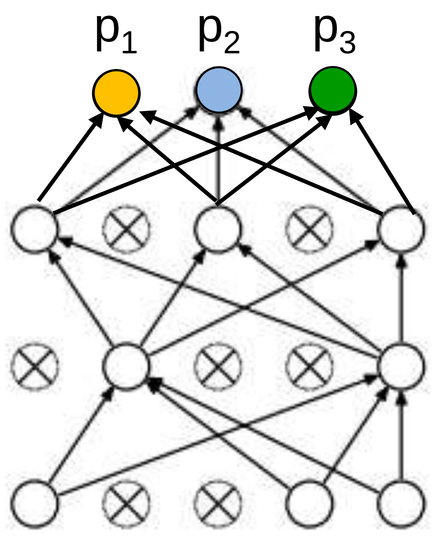


Only slight overfitting

This cnn2 allows to construct more complex hierarchical features, but has less parameters (~36k) than cnn1 (~251k weights) since the parameter-consuming fc layer in the end is smaller.

Why dropout?

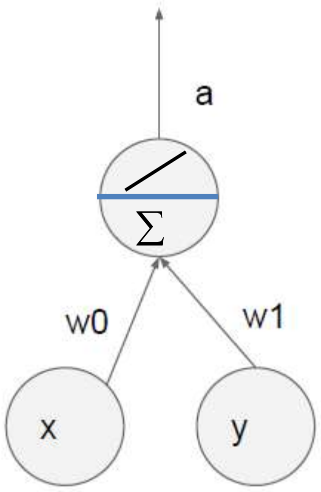
Dropout helps to fight overfitting



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-trained NN are kind of NN ensemble averages



0	0
0	y
x	0
x	y

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 = w0*x + w1*y**

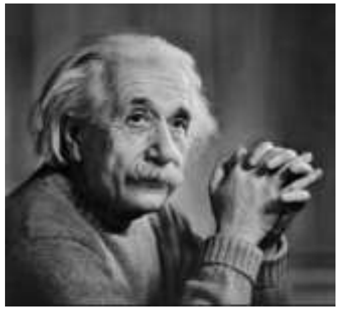
during training with dropout probability 0.5:

$$
\begin{aligned}
E[a] = \tfrac{1}{4} * (&w0*0 + w1*0 \;+ \\
&w0*0 + w1*y \;+ \\
&w0*x + w1*0 \;+ \\
&w0*x + w1*y) \\
= \tfrac{1}{4} &* (2\, w0*x + 2\, w1*y) \\
= \tfrac{1}{2} &* \mathbf{(w0*x + w1*y)}
\end{aligned}
$$

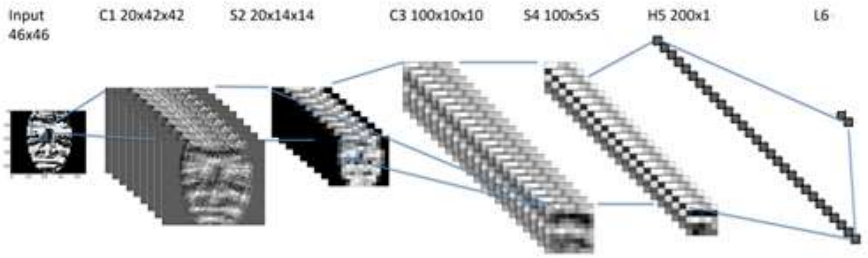
=> To get same expected output in training and test time, we reduce the weights during test time by multiplying them by the dropout probability p=0.5

Another intution: Why “dropout” can be a good idea

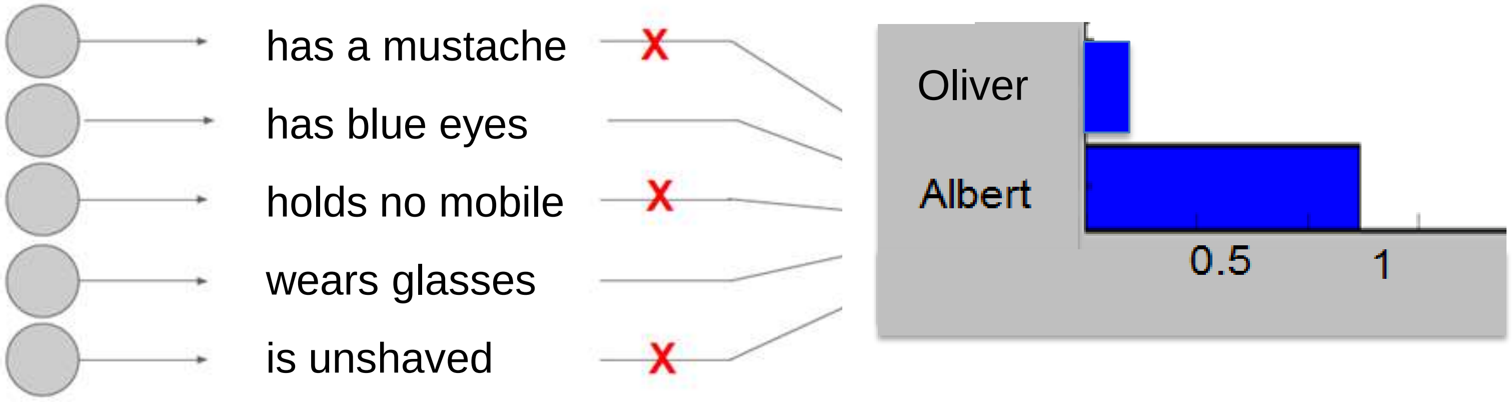
The training data consists of many different pictures of Oliver and Einstein



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

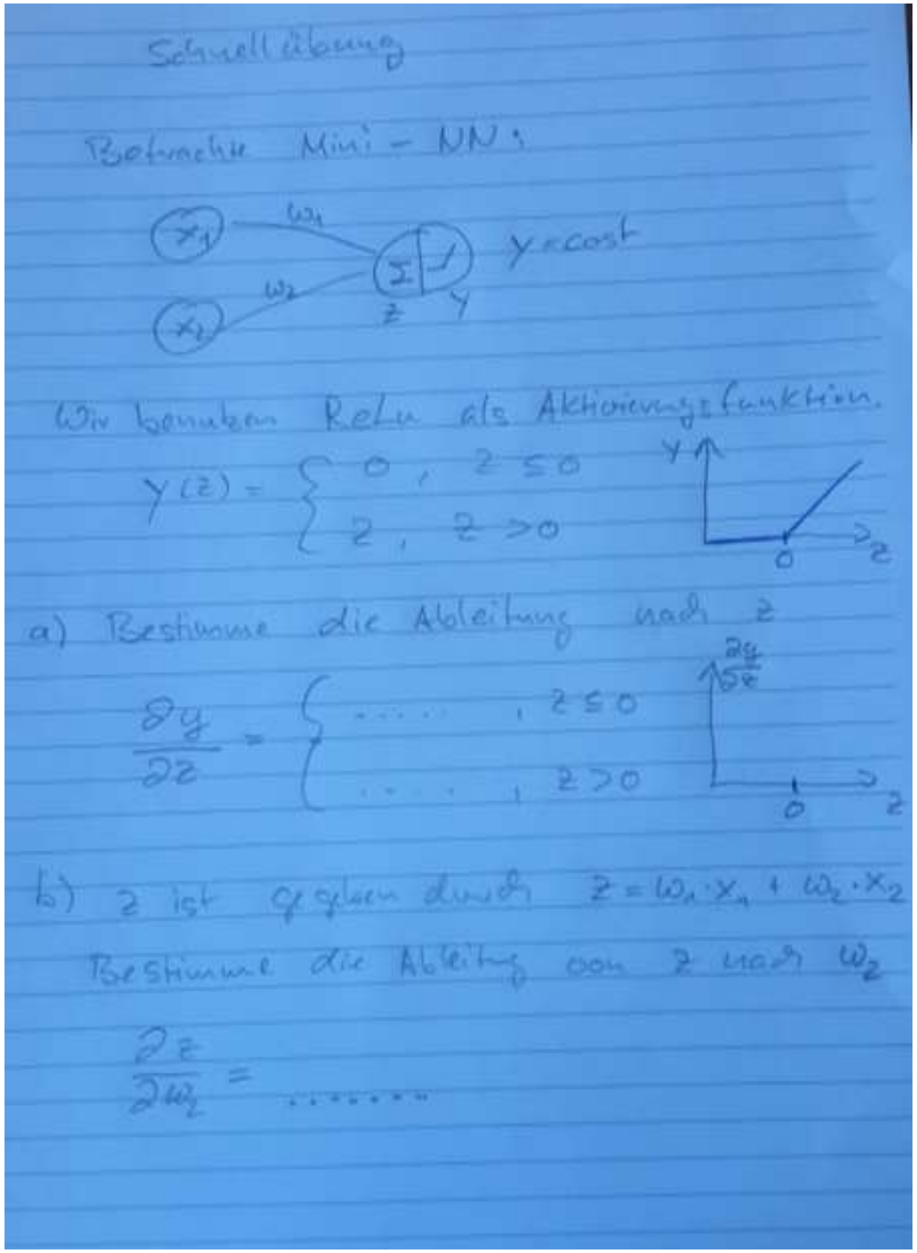


Dropout forces the network to learn redundant and independent features



Why batchnorm?

Gradient propagation when using ReLu



Schnellübung

Betrachte Mini – NN:

x_1, x_2, w_1, w_2, Σ, z, y

$y \approx cost$

Wir benutzen ReLu als Aktivierungsfunktion.

$$y(z) = \begin{cases} 0, & z \le 0 \\ z, & z > 0 \end{cases}$$

y, 0, z

a) Bestimme die Ableitung nach z

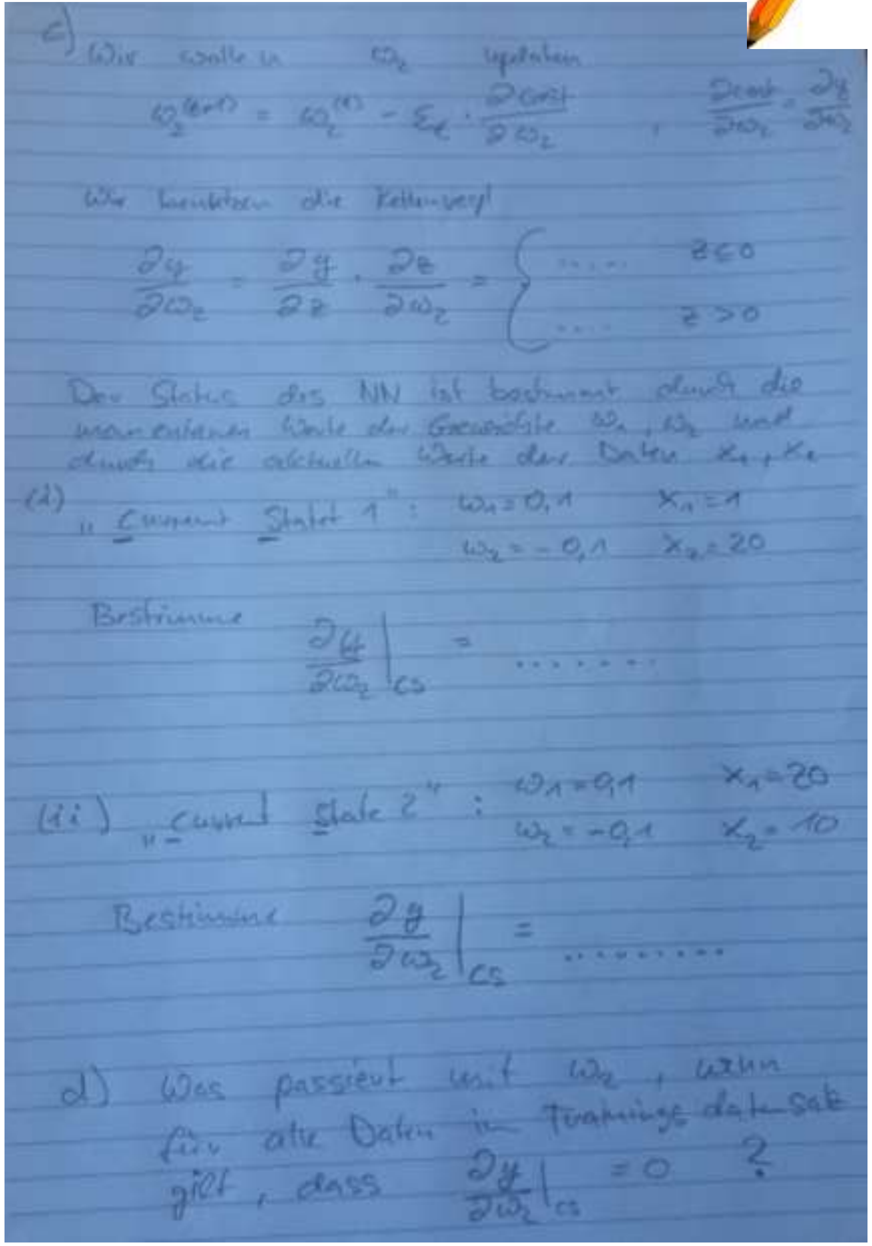
$$\frac{\partial y}{\partial z} = \begin{cases} \ldots\ldots, & z \le 0 \\ \ldots\ldots, & z > 0 \end{cases}$$

$\frac{\partial y}{\partial z}$, 0, z

b) z ist gegeben durch $z = w_1 \cdot x_1 + w_2 \cdot x_2$

Bestimme die Ableitung von z nach w_2

$$\frac{\partial z}{\partial w_2} = \ldots\ldots\ldots$$



c) Wir wollen w_2 updaten

$$w_2^{(t+1)} = w_2^{(t)} - \varepsilon_t \cdot \frac{\partial cost}{\partial w_2}, \quad \frac{\partial cost}{\partial w_2} = \frac{\partial y}{\partial w_2}$$

Wir benutzen die Kettenregel

$$\frac{\partial y}{\partial w_2} = \frac{\partial y}{\partial z} \cdot \frac{\partial z}{\partial w_2} = \begin{cases} \ldots\ldots & z \le 0 \\ \ldots\ldots & z > 0 \end{cases}$$

Der Status des NN ist bestimmt durch die momentanen Werte der Gewichte w_1, w_2 und durch die aktuellen Werte der Daten x_1, x_2

(i) „Current State 1": $w_1 = 0{,}1$ $x_1 = 1$
$w_2 = -0{,}1$ $x_2 = 20$

Bestimme $\left.\frac{\partial y}{\partial w_2}\right|_{CS} = \ldots\ldots\ldots$

(ii) „Current State 2": $w_1 = 0{,}1$ $x_1 = 20$
$w_2 = -0{,}1$ $x_2 = 10$

Bestimme $\left.\frac{\partial y}{\partial w_2}\right|_{CS} = \ldots\ldots\ldots$

d) Was passiert mit w_2, wenn für alle Daten im Trainings-datensatz gilt, dass $\left.\frac{\partial y}{\partial w_2}\right|_{CS} = 0$?

Gradient propagation when using ReLu

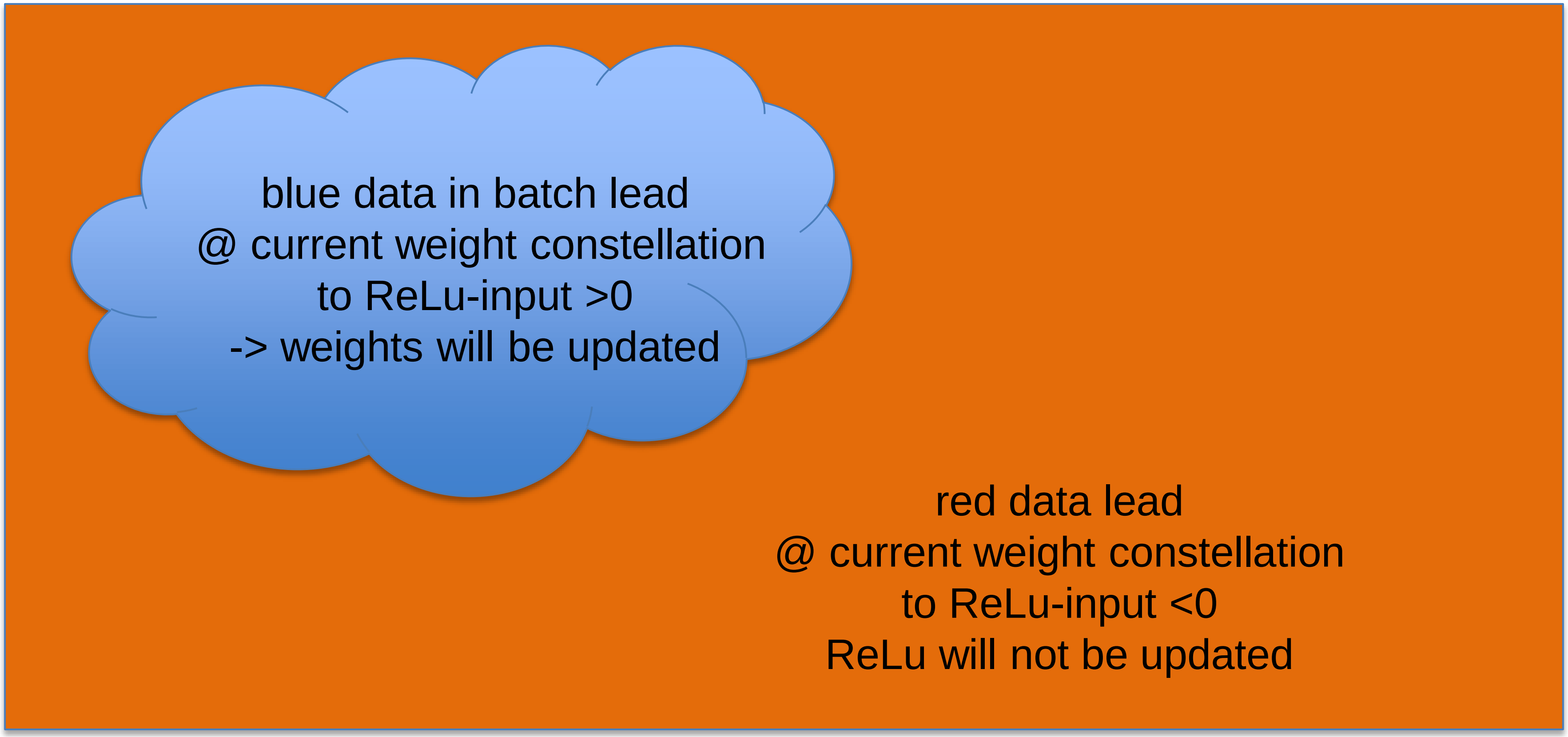
Schnellübung

c) Wir wollen ω_2 updaten

∂cost ∂cost ∂y

Why do we need batch-normalization?

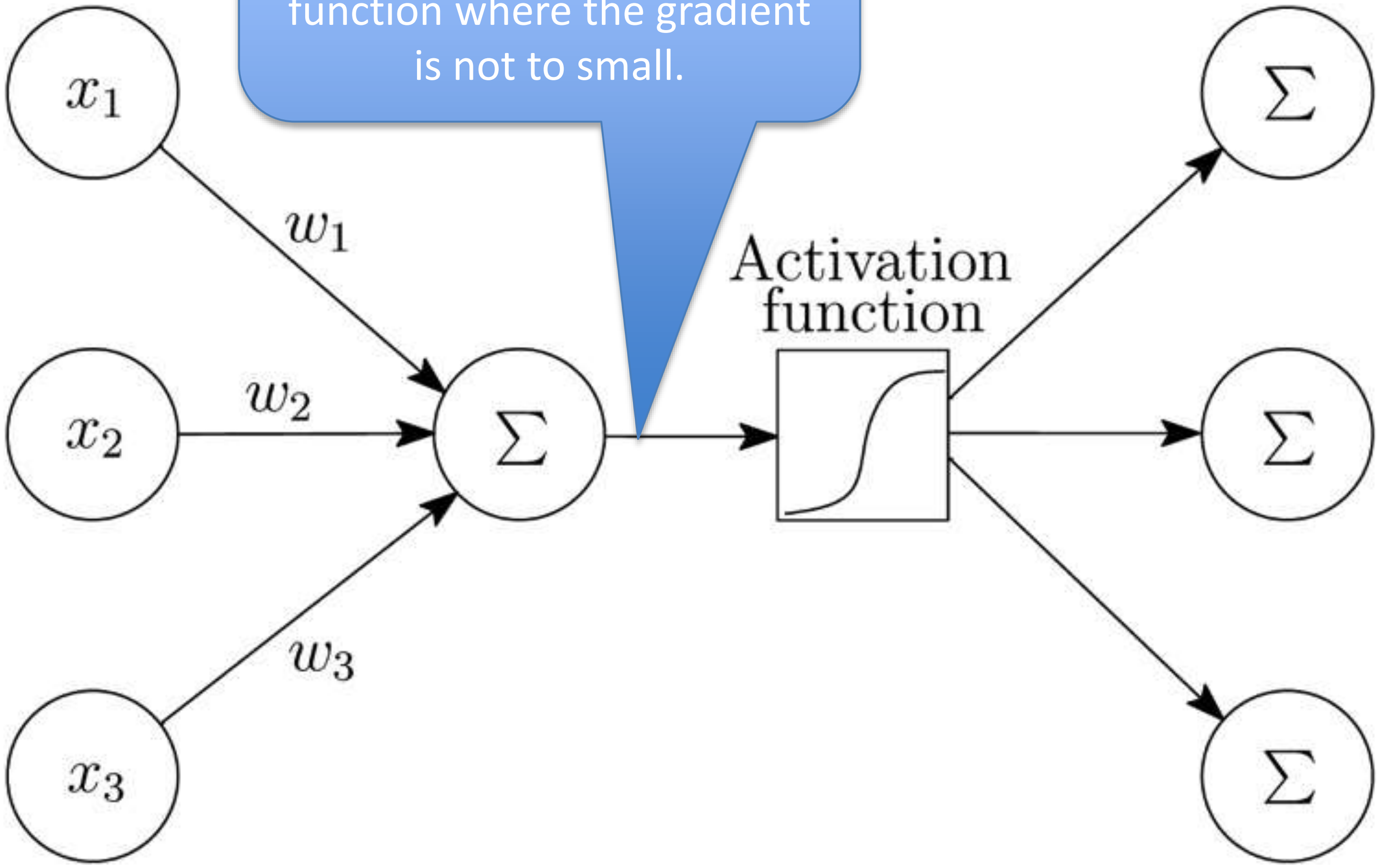
We have seen (Schnellübung) that there is no way to train=update-weights when the data and weight constellation is such that for **all** data the input to the ReLu is <0 ! Then we have a dead ReLu and the NN to left (direction input) is not trainable.



Depending on the data values, it can happen that there is no blue cloud and the NN is not trainable. This risk can be reduced when working with BN (or without BN with good initialization and small learning rates).

What is the idea of Batch-Normalization (BN)

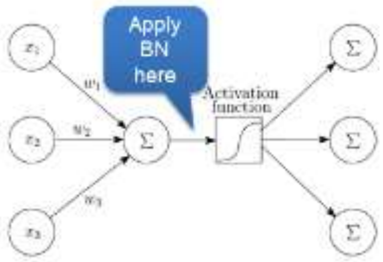
BN rescales the signal allowing to shift it into the region of the activation function where the gradient is not to small.



Batch Normalization

- Idea: Allow before each activation (non-linear transformation) to standardize the ingoing signal, but also allow to learn redo (partly) the standardization if it is not beneficial.

A BN layer performs a 2-step procedure with α **and** β **as learnable** parameter:



Step 1: $$\hat{x} = \frac{x - avg_{batch}(x)}{stdev_{batch}(x) + \epsilon}$$

$avg(\hat{x}) = 0$

$\text{stdev}(\hat{x}) = 1$

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

α learned

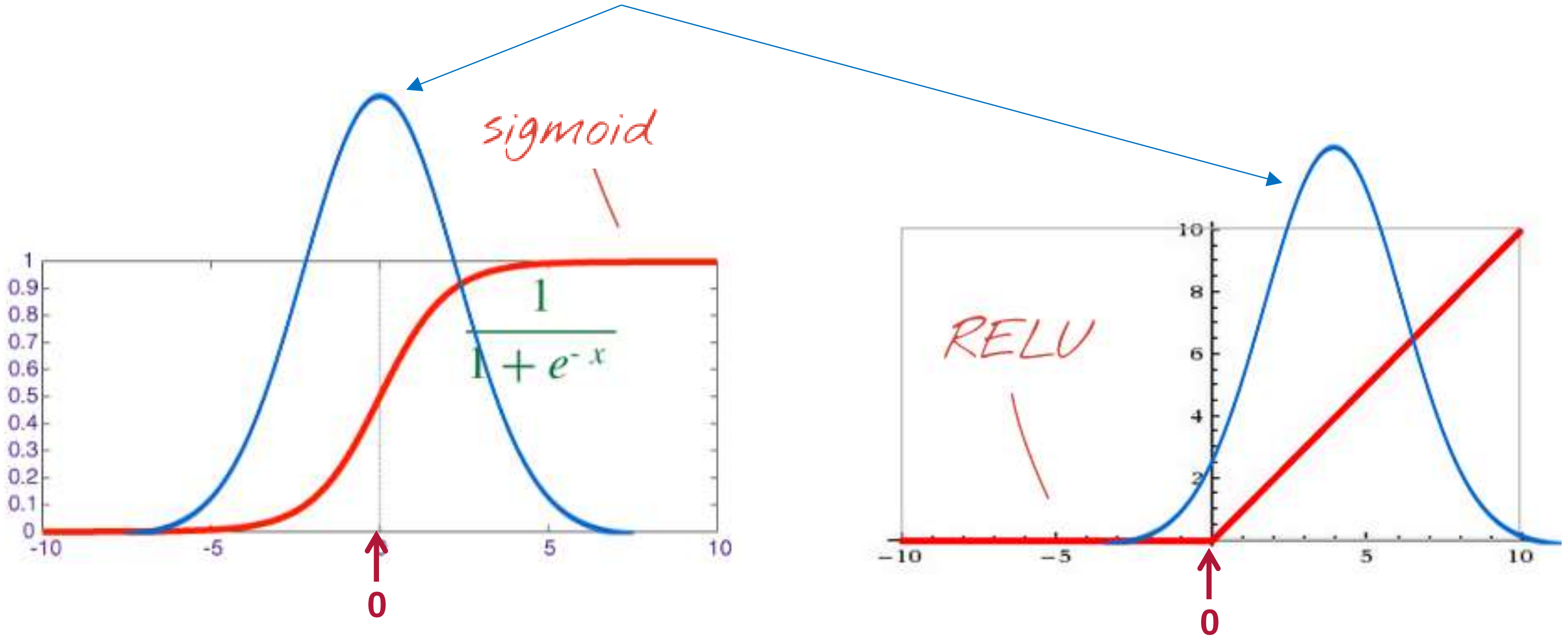
β learned

The learned parameters α and β determine how strictly the standardization is done. If the learned α=stdev(**x**) and the learned β=avg(**x**), then the standardization performed in step 1 is undone (for $\varepsilon \approx 0$) 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
- In the linear transformation the biases can be dropped (step 2 takes care of the shift)
- In case of ReLu only the shift β in steps 2 need to be learned (α can be dropped)

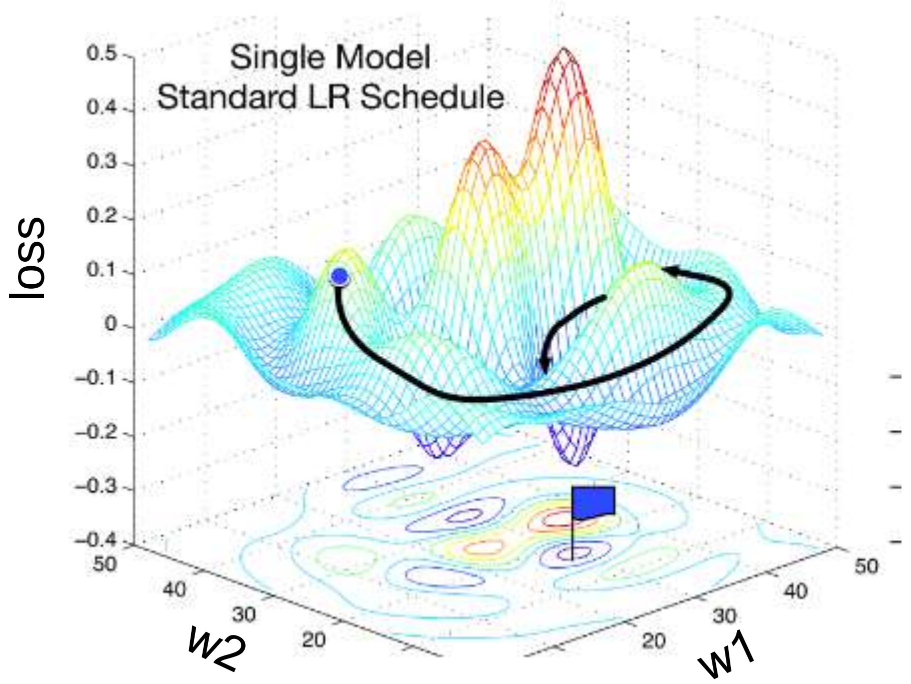
Image credits: Martin Gorner:
https://docs.google.com/presentation/d/e/2PACX-1vRouwj_3cYsmLrNNI3Uq5gv5-hYp_QFdeoan2GlxKgIZRSejozruAbVV0IMXBoPsINB7Jw92vJo2EAM/pub?slide=id.g187d73109b_1_2921

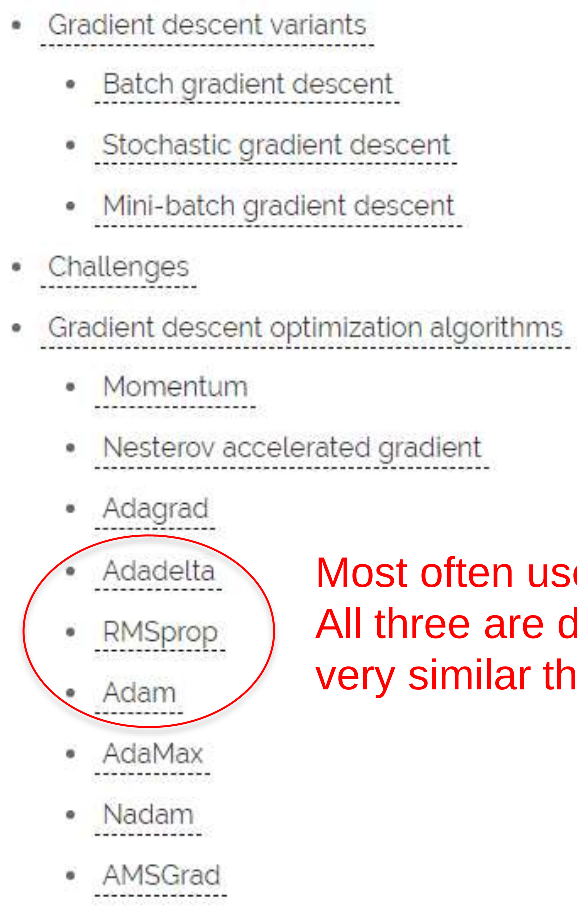
Coffee
Break

Which optimizer?

Choice of optimizers:
How to find the weight constellation of the minimum cost

For an overview of optimizers check out keras documentation and the nice blog of Sebastian Ruder http://ruder.io/deep-learning-optimization-2017/





- Gradient descent variants
  - Batch gradient descent
  - Stochastic gradient descent
  - Mini-batch gradient descent
- Challenges
- Gradient descent optimization algorithms
  - Momentum
  - Nesterov accelerated gradient
  - Adagrad
  - Adadelta
  - RMSprop
  - Adam
  - AdaMax
  - Nadam
  - AMSGrad

Most often used!
All three are doing very similar things

Good news: In DL all minima are good!

Things observed but not well understood:

- All minima are “similar” good
- DL models generalize well despite p>>n
- Also sharp minima are good in DL with ReLu
- loss landscape depends on parametrization (e.g. BN)

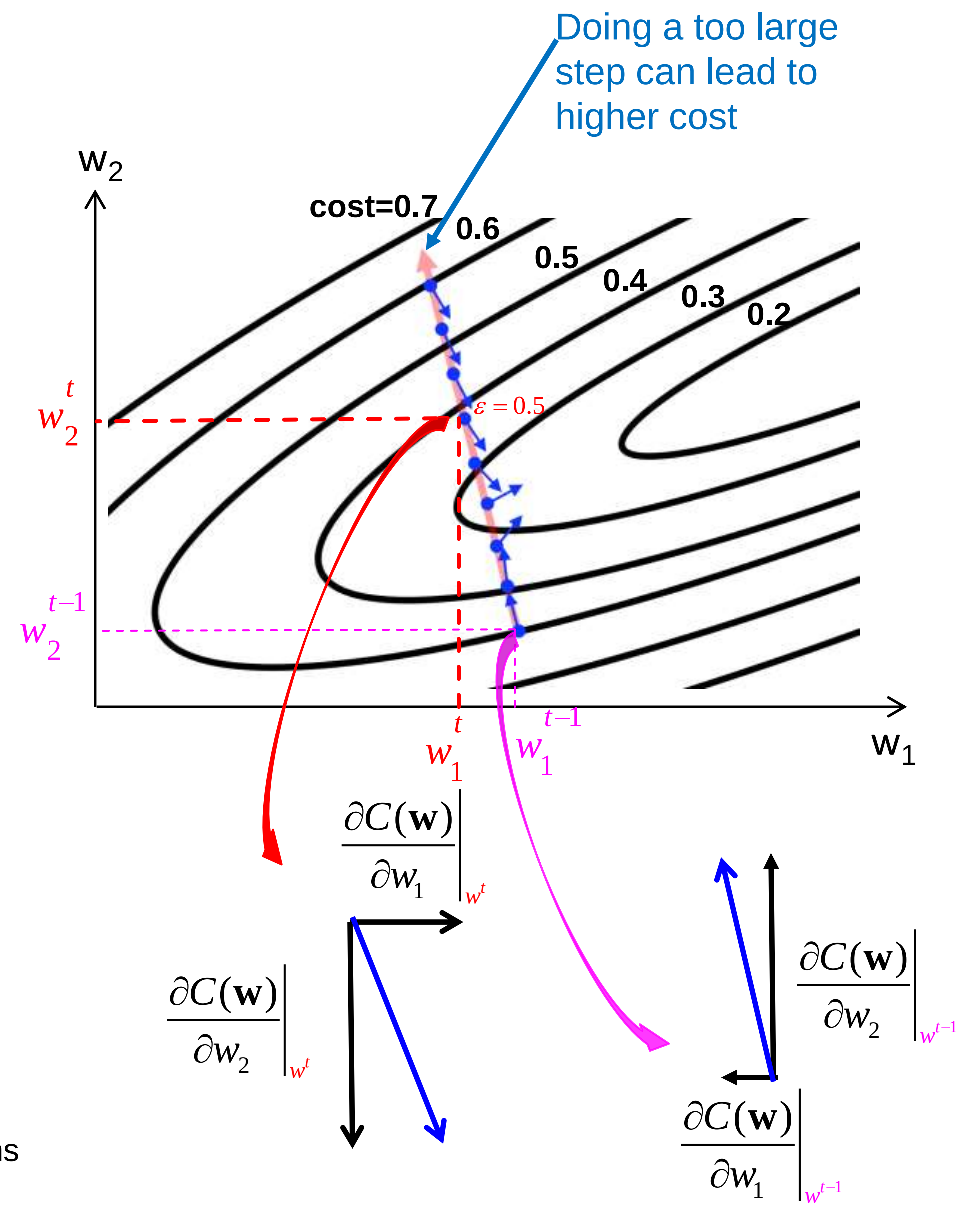
Bengio 2017 paper on loss landscape, flat and sharp minima, and generalization

One standing hypothesis that is … that the flatness of minima of the loss function found by stochastic gradient based methods results in good generalization. This paper argues that most notions of flatness are problematic for deep models and can not be directly applied to explain generalization. Specifically, when focusing on deep networks with rectifier units, we can … build equivalent models corresponding to arbitrarily sharper minima… without affecting its generalization properties.

Gradient Descent (GD) and stochastic GD (SGD) method: Getting the learning rate right is most important!

Take step in direction of descent gradient

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

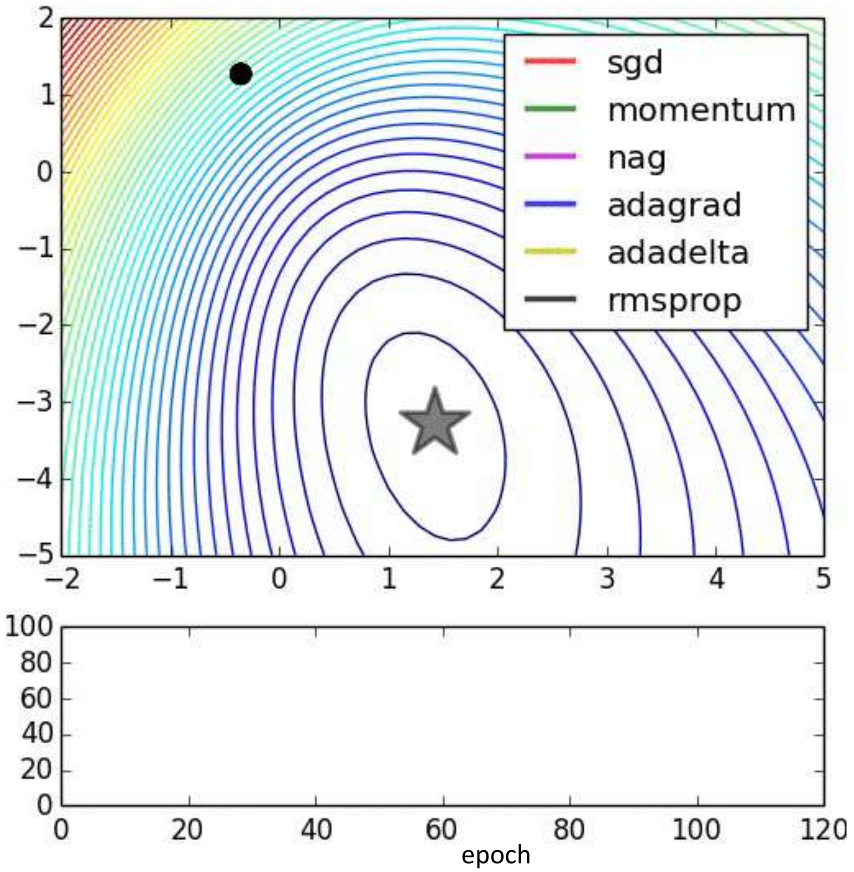


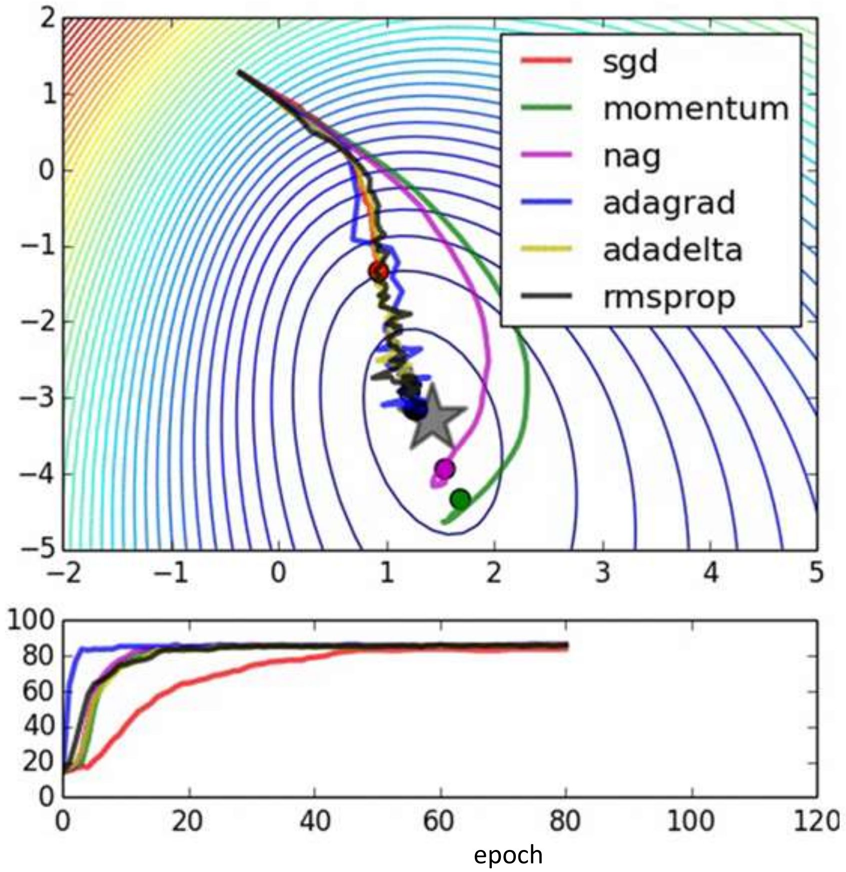
Doing a too large step can lead to higher cost
w_2
cost=0.7
0.6
0.5
0.4
0.3
0.2
$\varepsilon = 0.5$
w_2^t
w_2^{t-1}
w_1^t
w_1^{t-1}
w_1
$\left.\frac{\partial C(\mathbf{w})}{\partial w_1}\right|_{w^t}$
$\left.\frac{\partial C(\mathbf{w})}{\partial w_2}\right|_{w^t}$
$\left.\frac{\partial C(\mathbf{w})}{\partial w_2}\right|_{w^{t-1}}$
$\left.\frac{\partial C(\mathbf{w})}{\partial w_1}\right|_{w^{t-1}}$

From GD to SGD:
Instead of using all data to determine the cost at one weight constellation we use a "random" mini batch of the data.

Remark:
GD is guaranteed to find the optimum of a convex loss functions if learning rate is adapted (shrinked) in the course of training.

What is wrong with good old gradient descent?





http://ruder.io/optimizing-gradient-descent/index.html#fn:6

Comparison of a few optimization methods (animation by Alec Radford).
The star denotes the global minimum on the error surface.

Notice that stochastic gradient descent (SGD) is the slowest method to converge!

RMSprop: decreases step sizes in oscillating directions

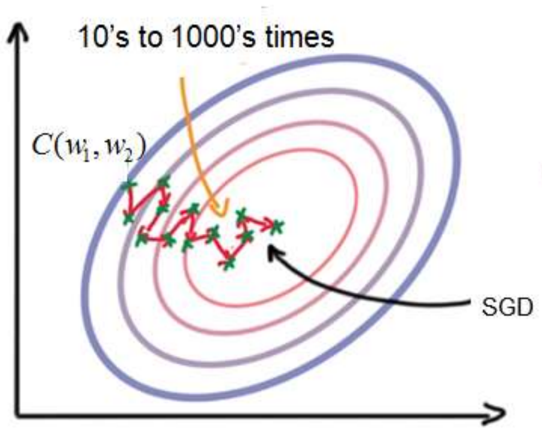
RMSprop adapts the learning rates $\varepsilon^{(t)}$ for each weight w_i based on the past couple of gradients that have been computed for this weight w_i .

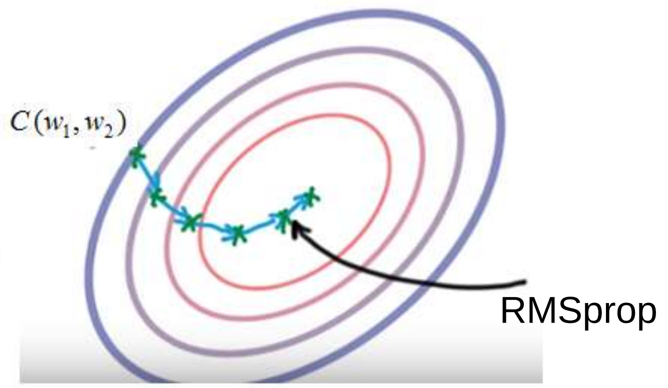
$$\text{SGD: } w_i^{(t)} = w_i^{(t-1)} - \varepsilon^{(t)} \cdot g_i^{(t)}$$

$$\text{with gradient } \quad g_i^{(t)} = \left.\frac{\partial C(\mathbf{w})}{\partial w_i}\right|_{w_i = w_i^{(t-1)}}$$

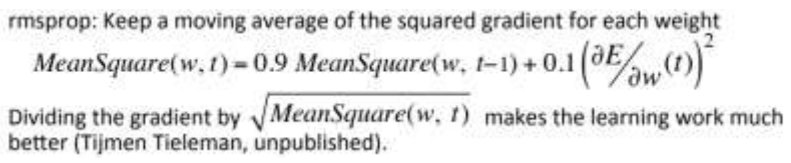
$$\text{RMSprop: } w_i^{(t)} = w_i^{(t-1)} - \frac{\varepsilon^{(t)}}{\mu_i^{(t)}} \cdot g_i^{(t)}$$

μ_i^t gets large if either the current gradient of this weight is large or the exponentially decaying average of past squared gradients of this weight were large.





RMSprop is almost identical to AdaDelta and was indepently proposed by Geoff Hinton in [Lecture 6e of his Coursera Class](#) Hinton suggests to be set the weight of past squared gradients to 0.9 and use for the learning rate 0.001 as default.



rmsprop: Keep a moving average of the squared gradient for each weight

$$MeanSquare(w,t) = 0.9\ MeanSquare(w,\ t-1) + 0.1\left(\partial E/_{\partial w}(t)\right)^2$$

Dividing the gradient by $\sqrt{MeanSquare(w,\ t)}$ makes the learning work much better (Tijmen Tieleman, unpublished).

AdaDelta: decreases step sizes in oscillating directions

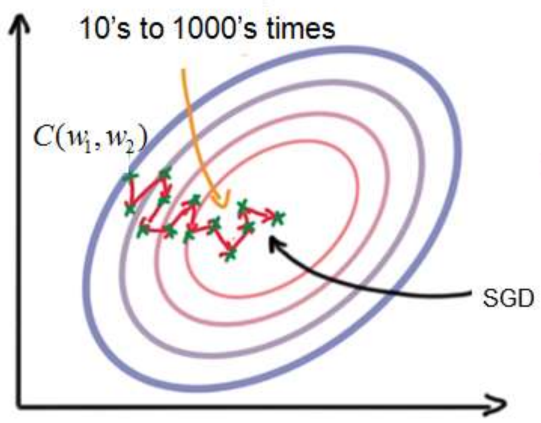
AdaDelta also adapts the learning rates $\varepsilon^{(t)}$ for each weight w_i based on the past couple of gradients that have been computed for this weight w_i .

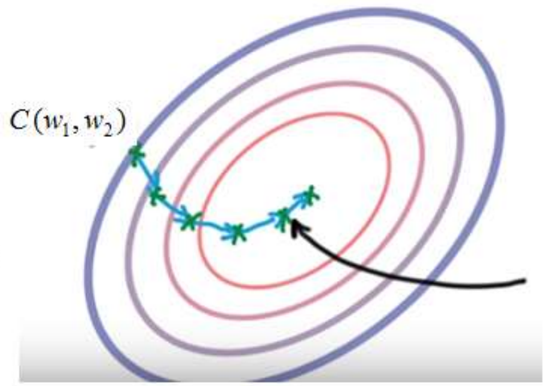
$$\text{SGD: } w_i^{(t)} = w_i^{(t-1)} - \varepsilon^{(t)} \cdot g_i^{(t)}$$

$$\text{with gradient } \quad g_i^{(t)} = \left.\frac{\partial C(\mathbf{w})}{\partial w_i}\right|_{w_i = w_i^{(t-1)}}$$

$$\text{AdaDelta: } w_i^{(t)} = w_i^{(t-1)} - \frac{\varepsilon^{(t)}}{\mu_i^{(t)}} \cdot g_i^{(t)}$$

μ_i^t gets large if either the current gradient of this weight is large or the exponentially decaying average of past squared gradients (oscillation) of this weight were large.





AdaDelta

M. Zeiler (2012): ADADELTA: An Adaptive Learning Rate Method https://arxiv.org/abs/1212.5701

Adam: Adaptive Gradients using momentum principle

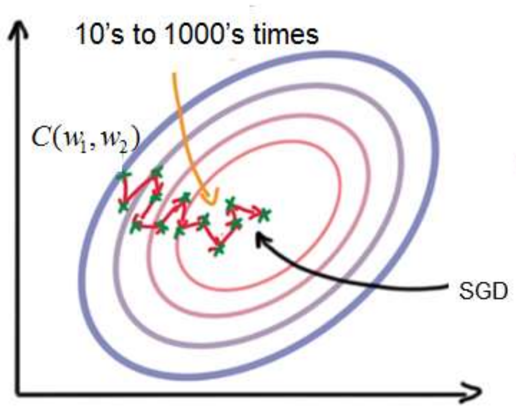
Adam (like RMSprop or AdaDelta) adapts the learning rates $\varepsilon^{(t)}$ for each weight w_i based on the past couple of gradients that have been computed for this weight w_i .

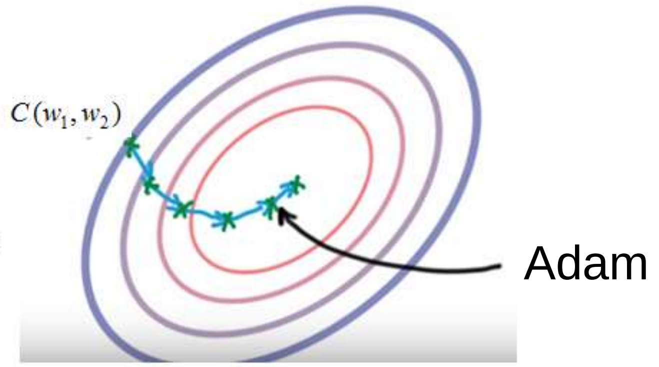
$$\text{SGD: } w_i^{(t)} = w_i^{(t-1)} - \varepsilon^{(t)} \cdot g_i^{(t)}$$

$$\text{with gradient} \quad g_i^{(t)} = \left.\frac{\partial C(\mathbf{w})}{\partial w_i}\right|_{w_i = w_i^{(t-1)}}$$

$$\text{Adam: } w_i^{(t)} = w_i^{(t-1)} - \frac{\varepsilon^{(t)}}{\mu_i^{(t)}} \cdot g_i^{(t)}$$

μ_i^t in Adam depends not only on the decaying average of past squared gradients but also on the exponentially decaying average of past gradients.





Adam Adaptive Moment Estimation , where the first moment (mean velocity) is given by average of past gradients and the second moment (uncentered variance, acceleration, oscillation) is given by average of past squared gradients

Kingma, D. P., & Ba, J. L. (2015). Adam: a Method for Stochastic Optimization. International Conference on Learning Representations, 1–13 https://arxiv.org/abs/1412.6980

Challenge: Who got the best CNN and which architecture and tricks were used?

Now you get the test set:

For a given image from the internet, decide which out of 8 celebrities is on the image.

Example images:

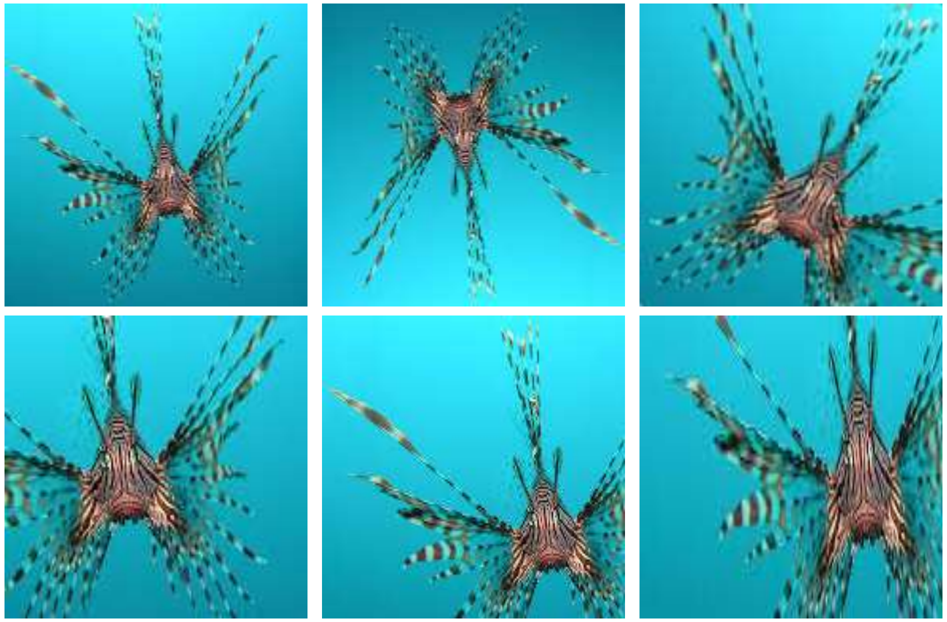
Label: Steve Jobs (entrepreneur)

Label: Emma Stone (actress)

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
- ….



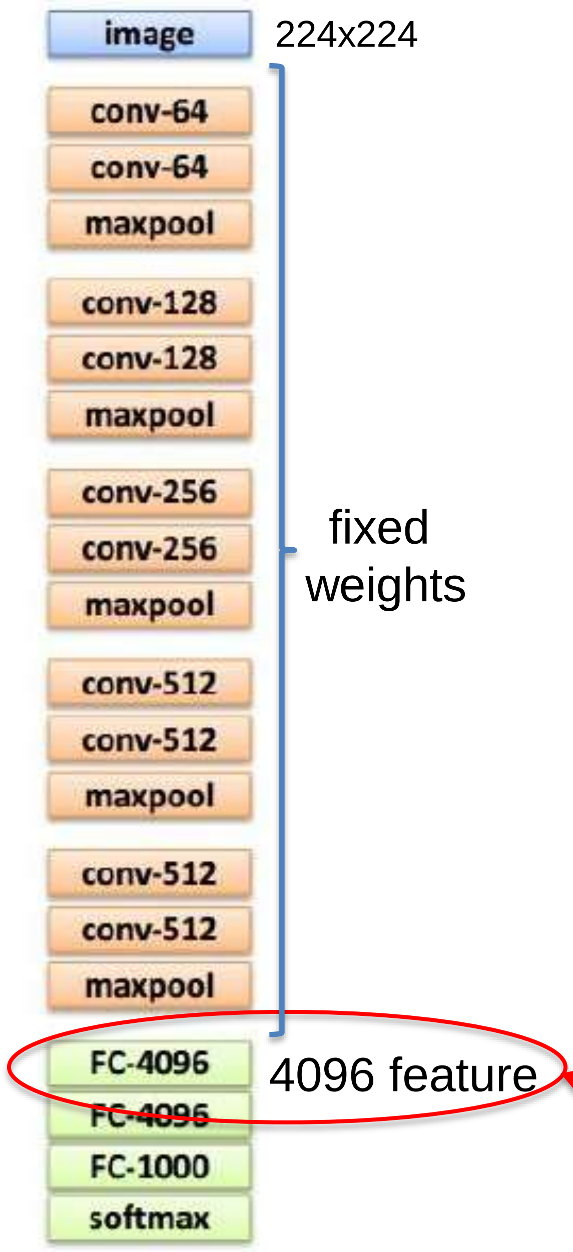
Data augmentation in Keras:

```
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]
)
```

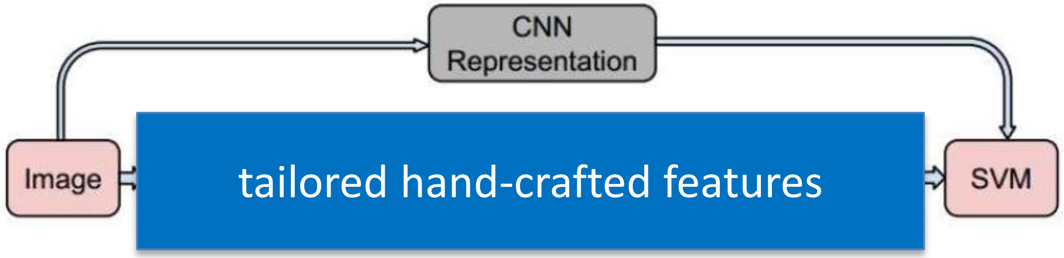
Use pre-trained CNNs for feature generation

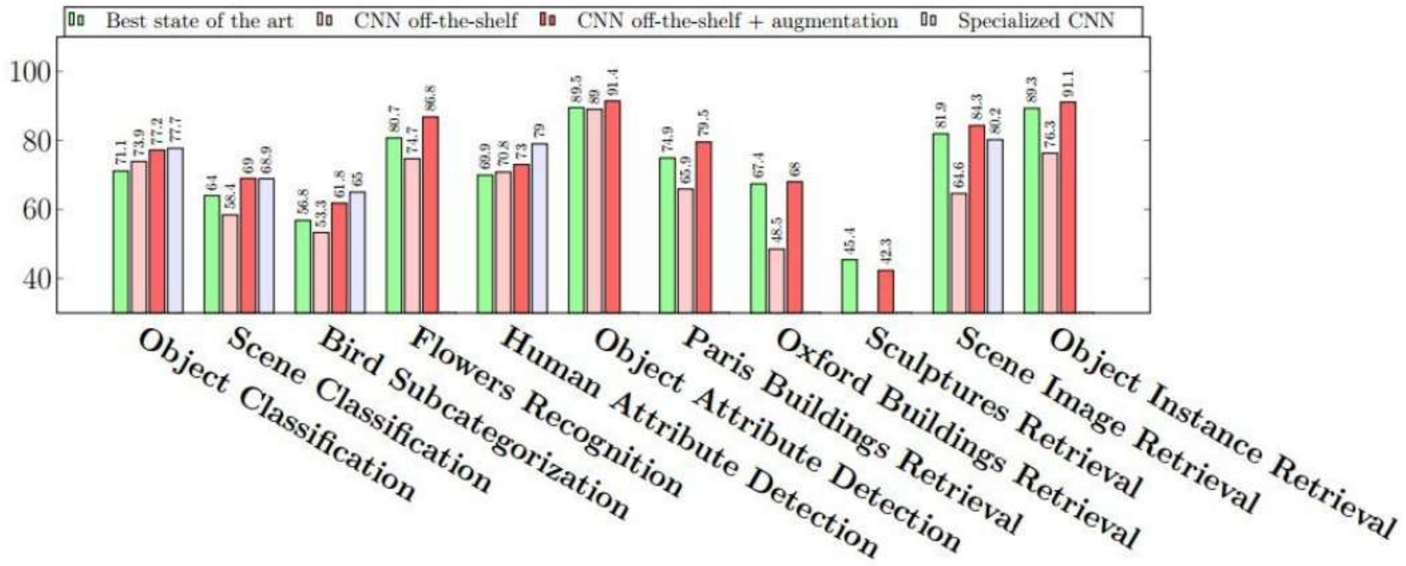


- Load a pre-trained CNN – e.g.g VGG16
- Resize image to required size (224x224 for VGG16)
- Rescaling of the pixel values to “VGG range”
- Do a forward pass and fetch 4096 features that are used as CNN representations, dump these features into a file on disk
- Use these CNN features as input to a simple 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)

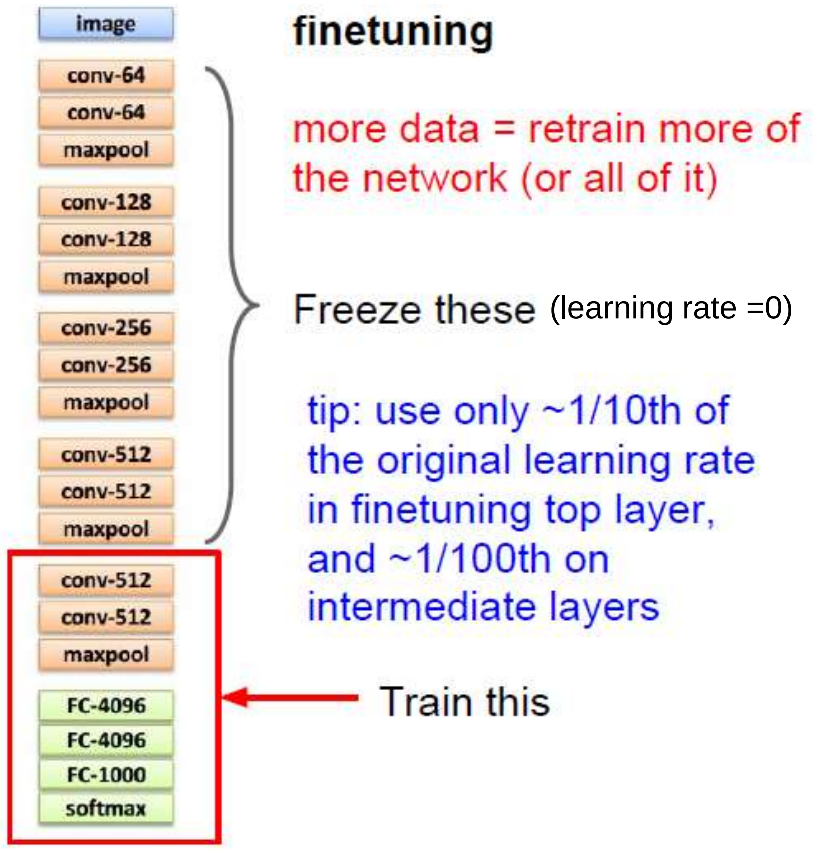


image
conv-64
conv-64
maxpool
conv-128
conv-128
maxpool
conv-256
conv-256
maxpool
conv-512
conv-512
maxpool
conv-512
conv-512
maxpool
FC-4096
FC-4096
FC-1000
softmax
finetuning
more data = retrain more of the network (or all of it)
Freeze these (learning rate =0)
tip: use only ~1/10th of the original learning rate in finetuning top layer, and ~1/100th on intermediate layers
Train this

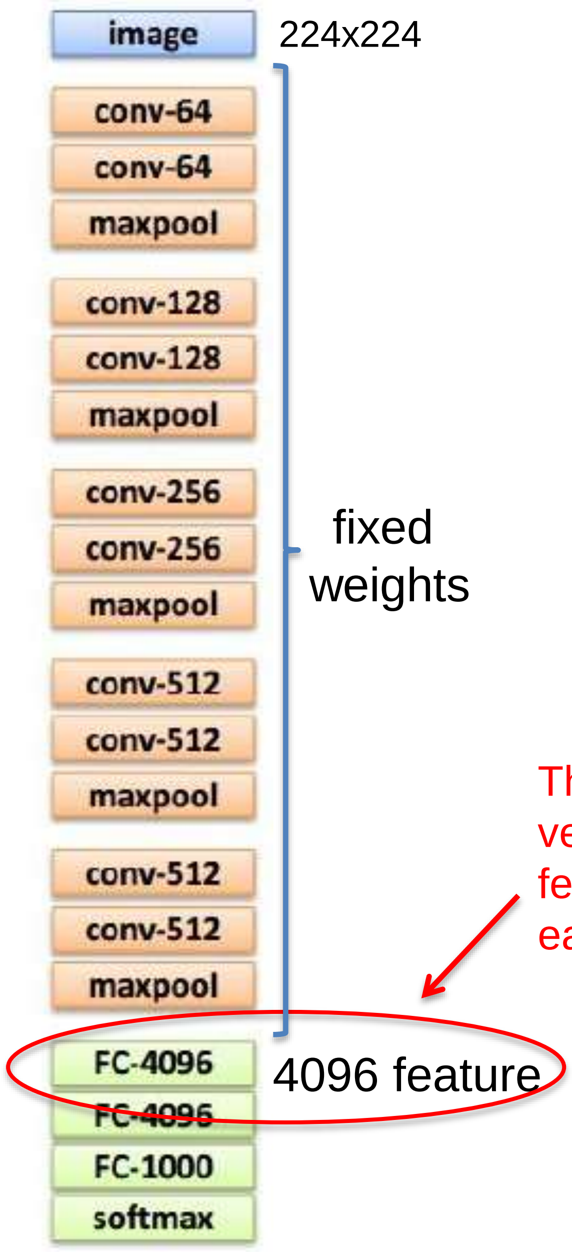
The strategy for fine-tuning depends on the size of the data set and the type of images:

	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

Coffee
Break

What does the CNN look at?

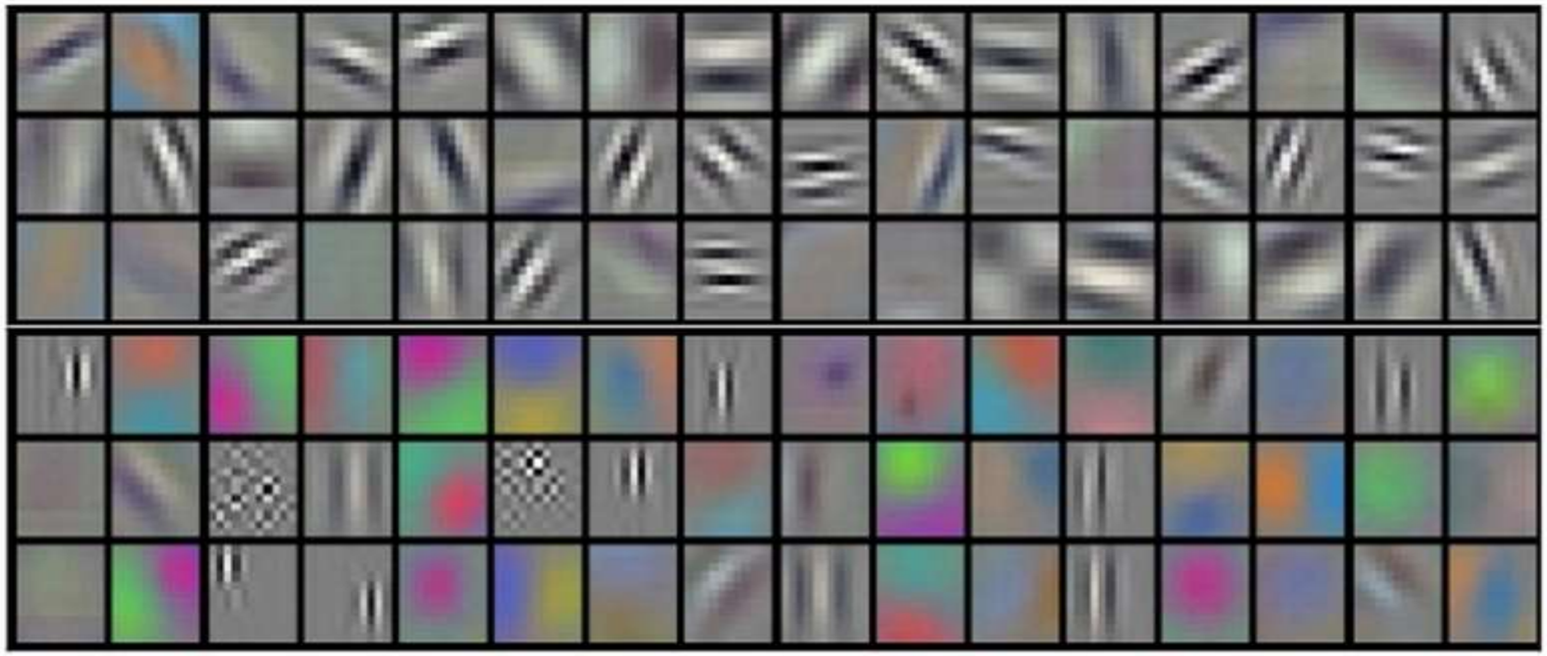
How to understand a CNN model?

- **Visualize filter weights (mainly useful in first layer)**
- **Visualize activation patterns that correspond to a specific input image in different layers and activation maps**
- Receptive field
- Show images or image patches that lead to maximal activations in a certain activation map
- Synthesize images that lead to maximal activations in a certain activation map
- Occlusion Experiments, Saliency Maps, LIME

Example of learned filters

96 filters in first layer

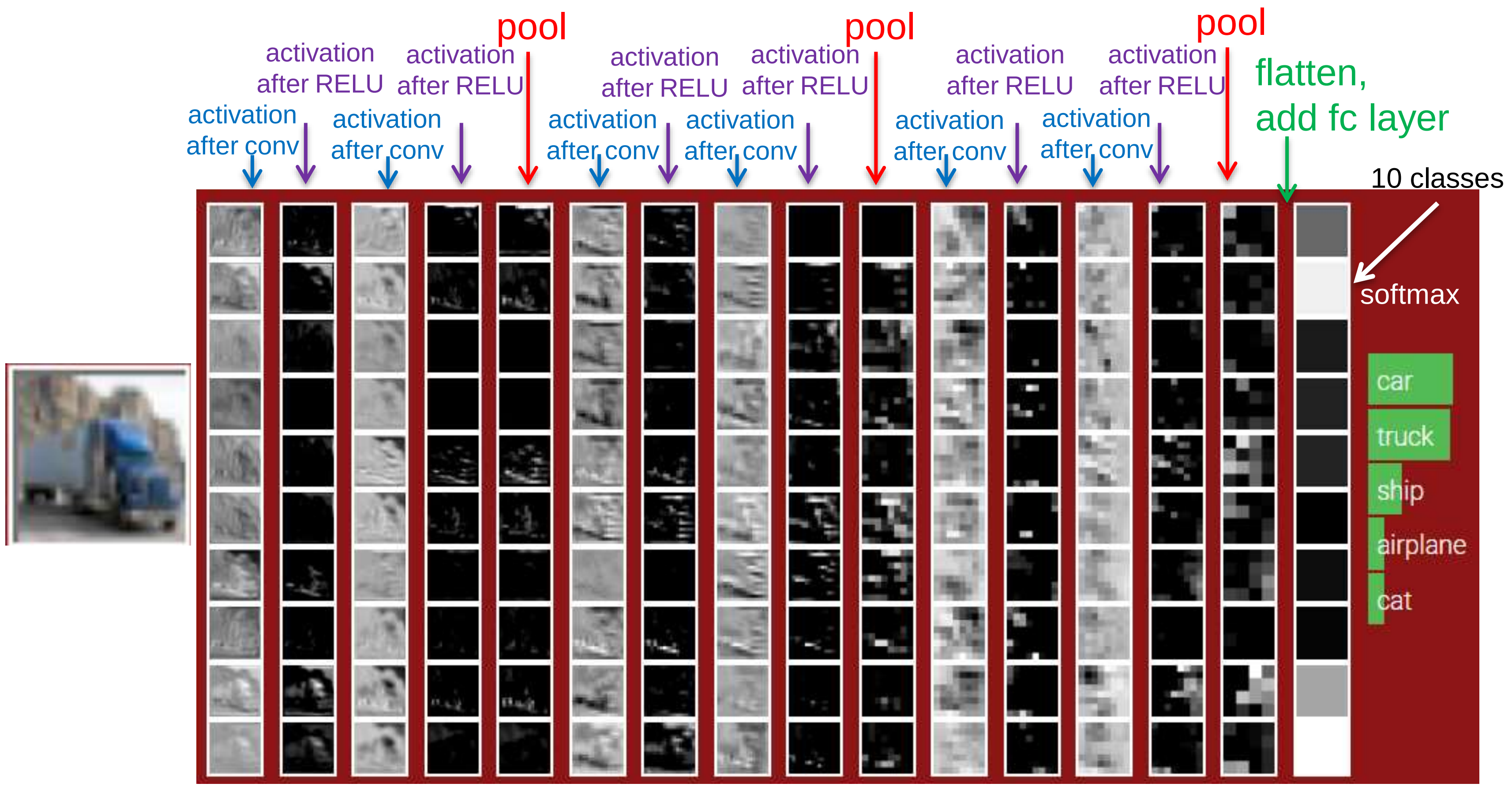
Here the input had 3 channels and the used filters had the dimension 11 × 11 ×3 which nowadays is unusual large.



Examples of 96 11x11x3 filters learned (taken from Krizhevsky et al. 2012).
Looks pretty much like old fashioned handcrafted filters to detect edges and simple patterns.

The filters show the detected patterns in the input which is for the first layer the original image. For all other layers the filter pattern do not directly correspond to patterns in the input image to the CNN.

Appearance of activation/feature maps in different layers



http://cs231n.stanford.edu/

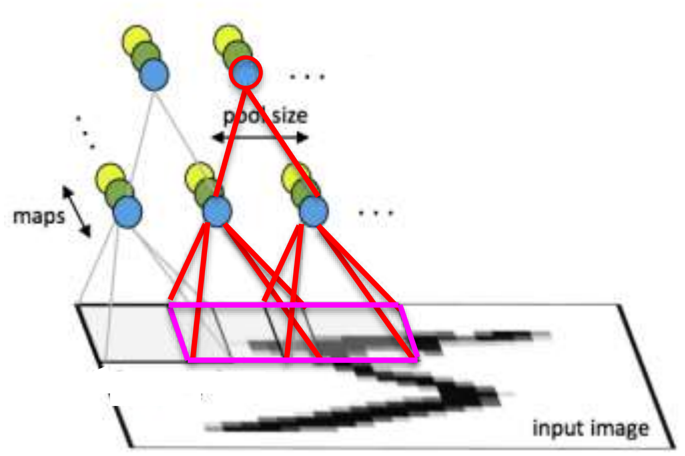
Activation maps can be seen as a transformed version of the input image.
A CNN aims to learn such new data representations (features) that help for the classification (only the activation maps in the first hidden layer correspond directly to features of the input image).

How can a CNN model be visualized?

- Visualize filter weights (mainly useful in first layer)
- Visualize activation patterns that correspond to a specific input image in different layers and activation maps
- Receptive field
- Show images or image patches that lead to maximal activations in a certain activation map
- Synthesize images that lead to maximal activations in a certain activation map
- Occlusion Experiments, Saliency Maps, LIME

The receptive field

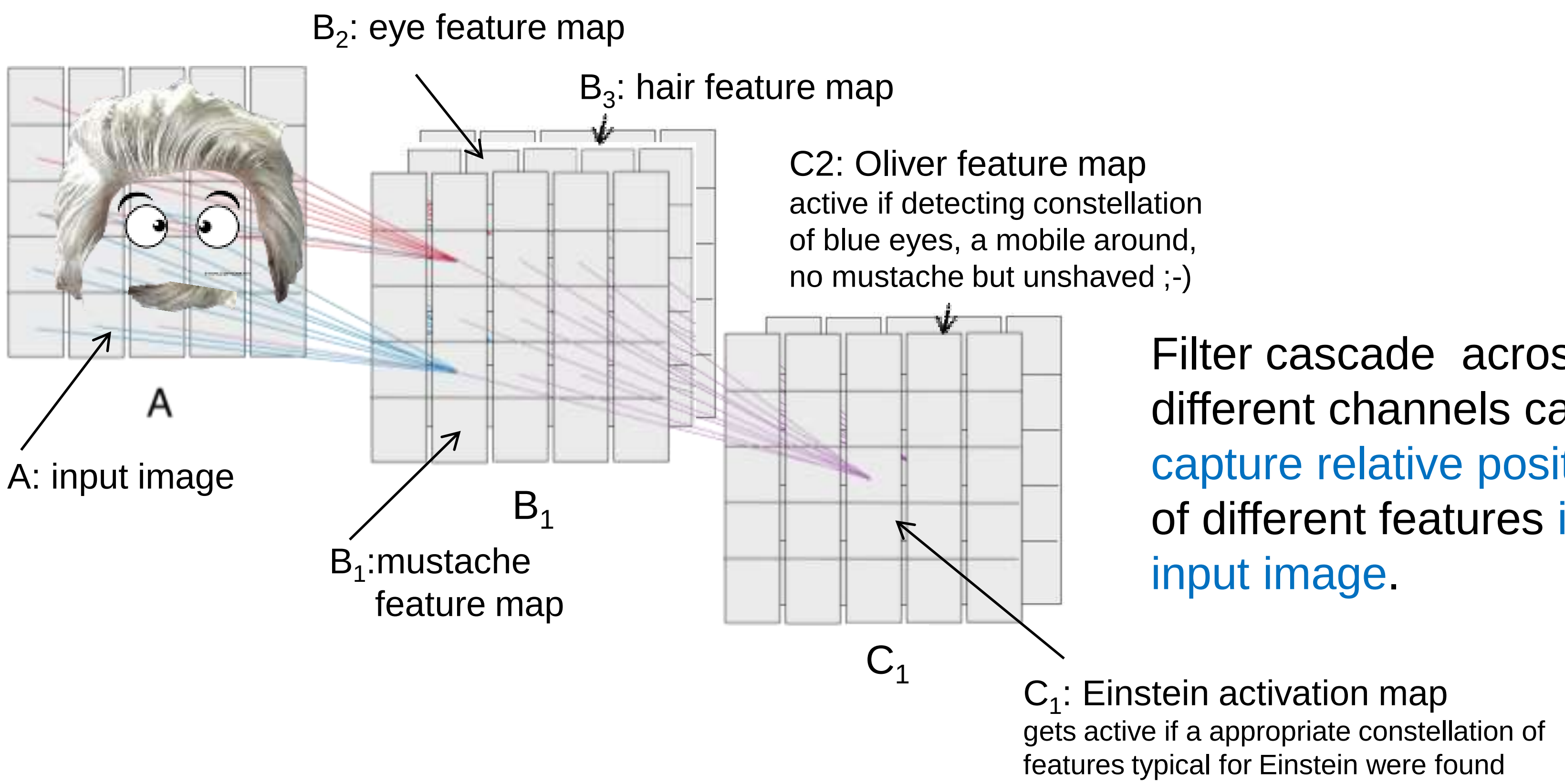
Receptive field of a neuron refers either to the area of the ingoing activation maps from the previous layer to which that neuron is connected or to the area in the original input image of the CNN that impacts the value of this neuron.



A neuron in the pooled layer has a larger receptive field of the original image than a neuron before pooling.

For each neuron we can determine the connected area in the input from the previous layer(s). This area in the input image is called receptive field.

A simplified view: hierarchy of features

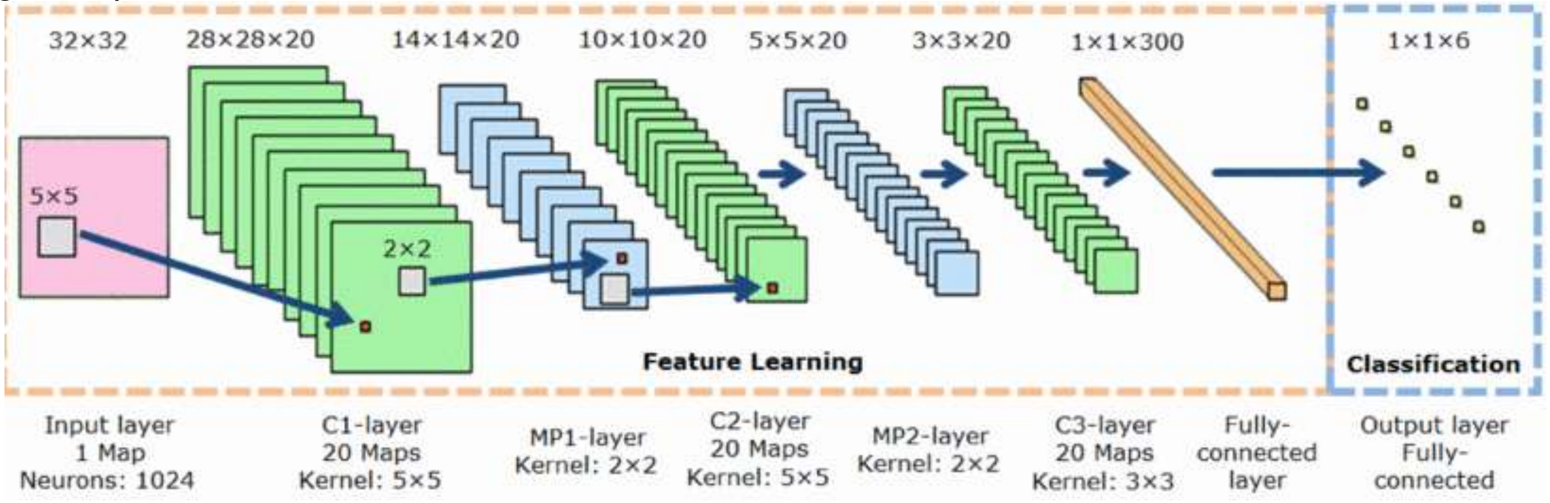

B_2: eye feature map
B_3: hair feature map
C2: Oliver feature map
active if detecting constellation of blue eyes, a mobile around, no mustache but unshaved ;-)
A
A: input image
B_1
B_1:mustache feature map
C_1
C_1: Einstein activation map
gets active if a appropriate constellation of features typical for Einstein were found

Filter cascade across different channels can capture relative position of different features in input image.

The receptive field is growing from layer to layer

The receptive field of a neuron is the area in the original input image that impact the value of this neuron – “that can be seen by this neuron”.

Receptive field in original input 5x5 6x6 14*14 16x16 24x24 32x32



Neurons from feature maps in higher layers have a larger receptive field than neurons sitting in feature maps of lower layers.

Code to determine size of receptive field: http://stackoverflow.com/questions/35582521/how-to-calculate-receptive-field-size

Figure credits: http://pubs.sciepub.com/ajme/2/7/9/figs

Visualize patches that maximally activate neurons

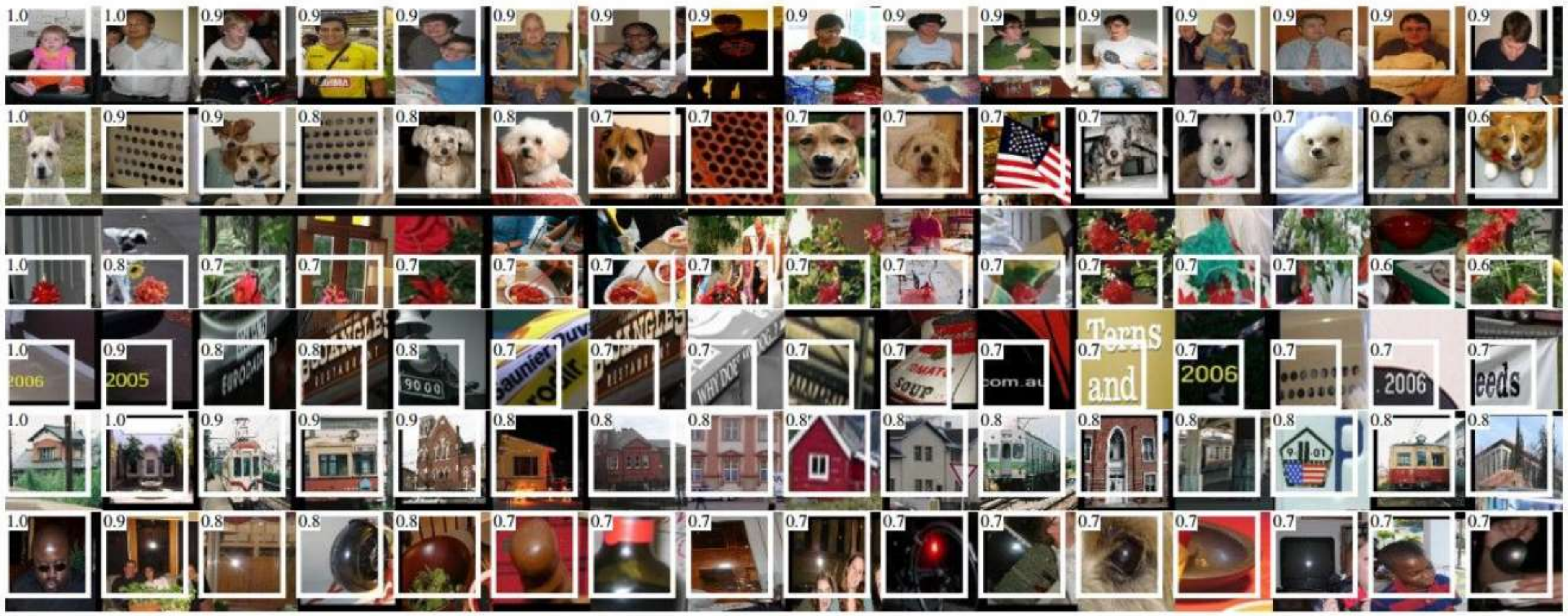
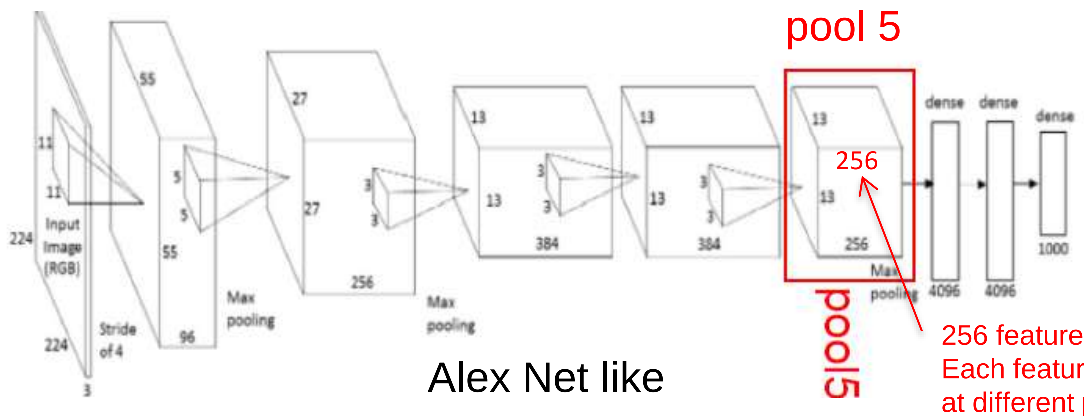


Figure 4: Top regions for six pool$_5$ units. Receptive fields and activation values are drawn in white. Some units are aligned to concepts, such as people (row 1) or text (4). Other units capture texture and material properties, such as dot arrays (2) and specular reflections (6).

http://cs231n.github.io/understanding-cnn/



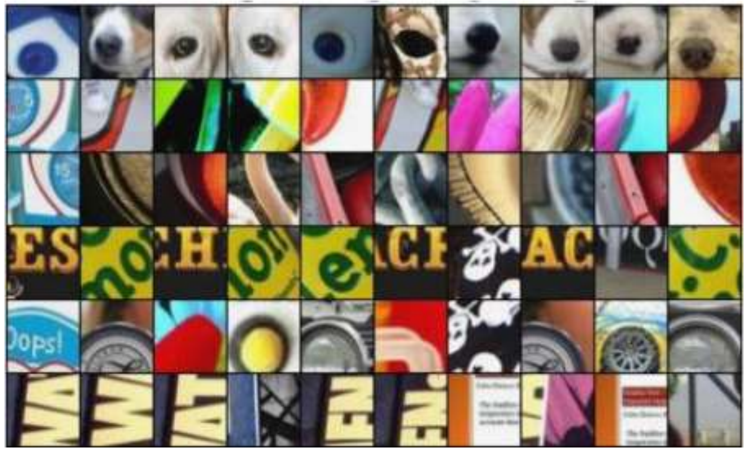
Alex Net like

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

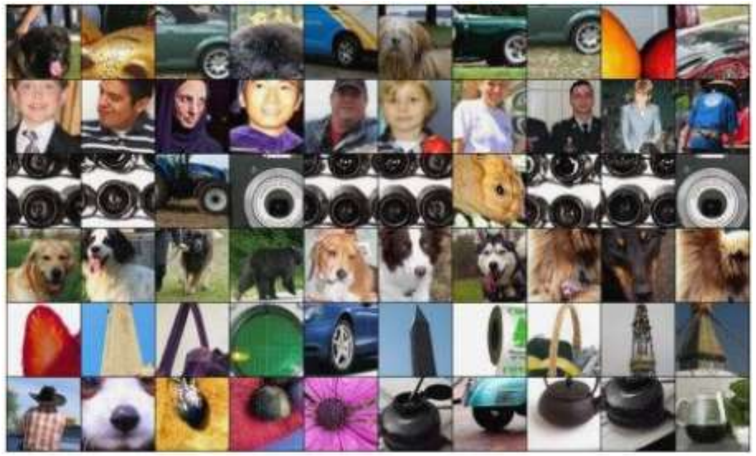
256 feature maps generated by 256 different 3x3 filters. Each feature map consists of equivalent neurons looking at different positions of the input volume.

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

10 images from data set exciting the 6 first neurons/filters in **conv6** most



10 images from data set exciting the 6 first neurons/filters in **conv9** most



Alternative approach: Construct a synthetic image giving rise to an high mean activation in a selected neuron-type/feature-map of a selected layer of the CNN.

image credit: cs231n

Optimize image for a high activation of a selected neuron type

Use a trained CNN with fixed weights.
Which content should the image have to get a high activation in this feature map?
We want to optimize the pixel values!

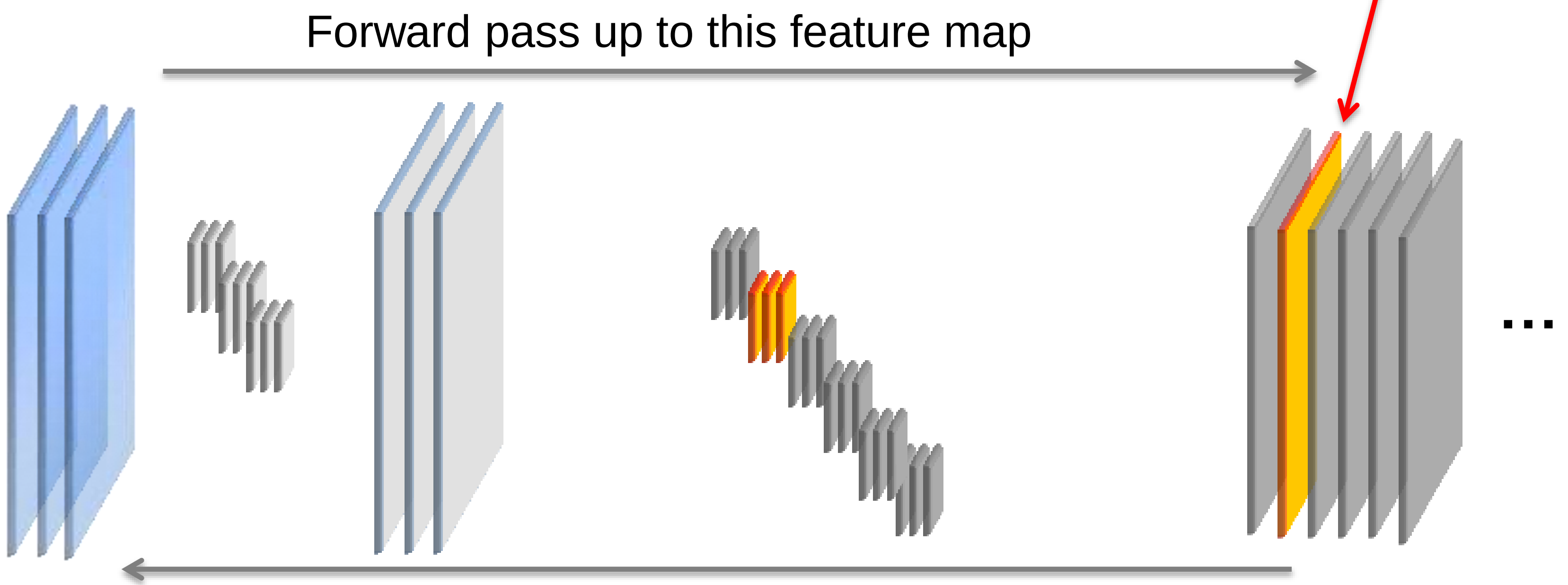


image
initialize with random values

Gradient of mean activation A in the selected feature map with respect to the pixel values in the image.

pixel update

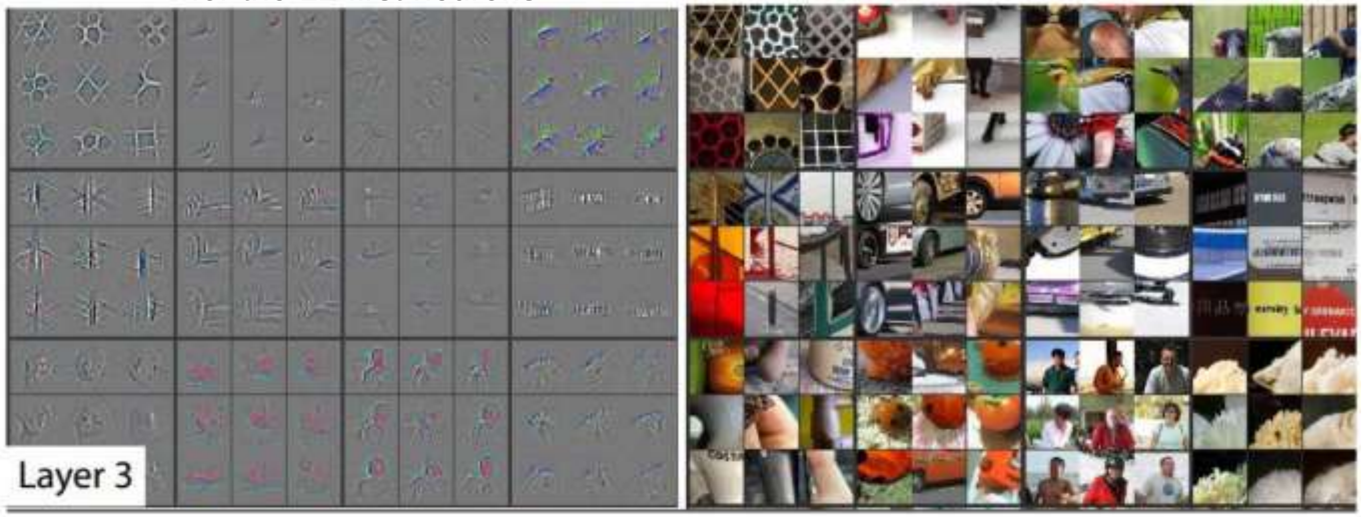
$$p_i^{(t)} = p_i^{(t-1)} + \varepsilon^{(t)} \left. \frac{\partial \mathrm{A}(\mathbf{p})}{\partial p_i} \right|_{p_i = p_i^{(t-1)}}$$

Set the gradient of neurons in this feature map to 1 and gradients of all other maps to zero. Since all neurons in one feature map use the same filter, they will fire for the same image pattern.

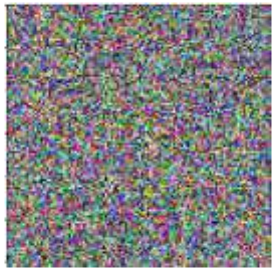
What kind of image (patches) excites a certain neuron corresponding to a large activation in a feature map?

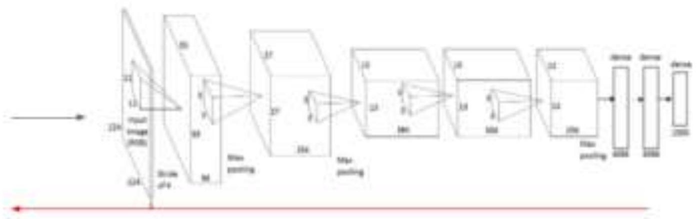
9 synthetic images, optimized for the 12 first neurons

9 images from data set exciting the 12 first neurons/filters most



Layer 3





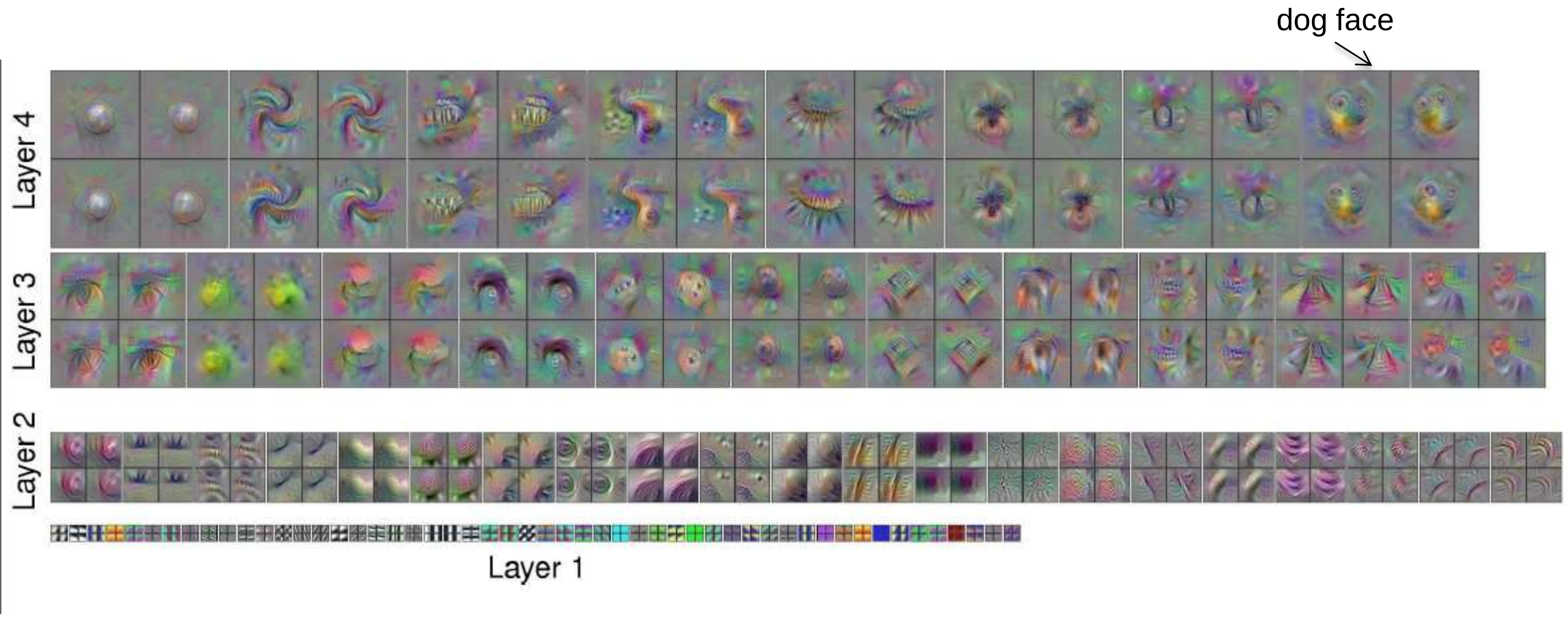
1) Initialize image with random grey values
2) Set the gradient to this neuron =1 all others=0
3) Do a small “image update” (constrained to be smooth)
4) Forward the image through net – go back to 2)

Use keras to synthesize image patch that activates a certain feature map

image credit: cs231n

What kind of synthesized image (patch) excites the neurons in layer1, layer2, layer3, layer4 ?

Kernels from higher layers have a larger receptive field and are excited by more complex image content.



It is more instructive to visualize image-parts of input images that excite certain filter at deeper layers of a CNN instead of the filter pattern or observed activation maps.

image credit: Stanford course cs231n

Optimize image for a high class probability

Use a trained CNN with fixed weights.
Which content should the image have to get a high probability for class k?
We want to optimize the pixel values!

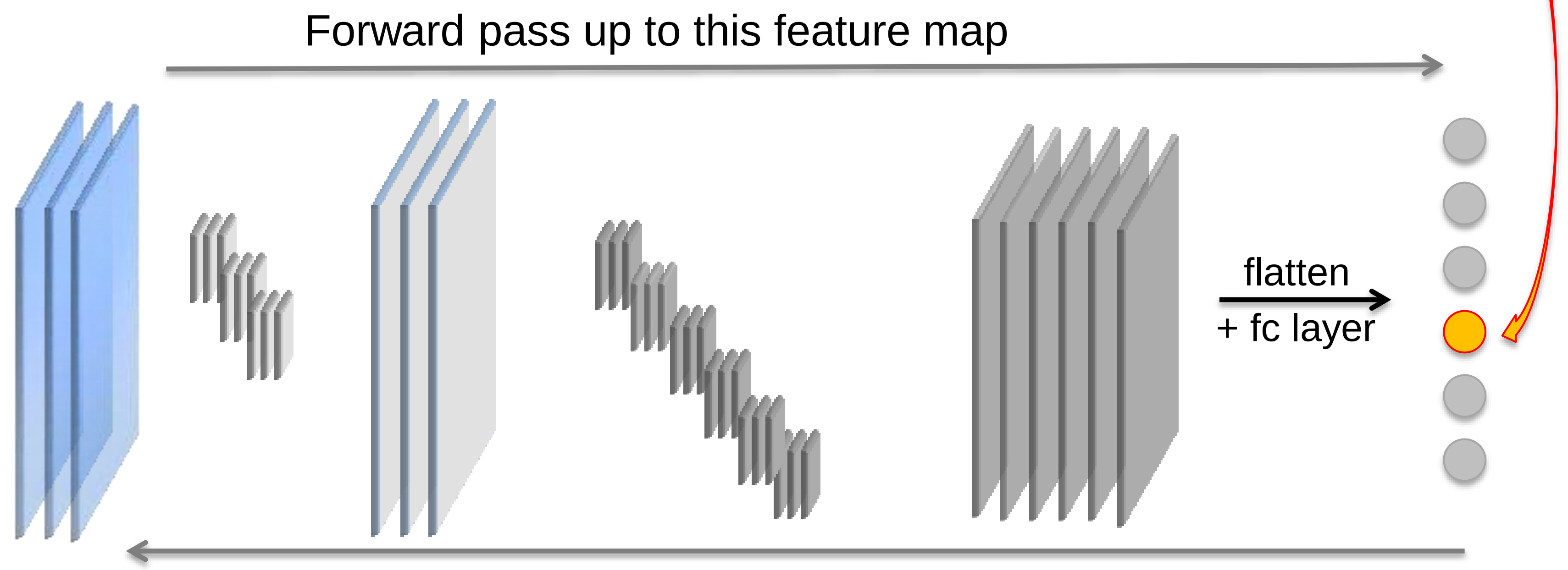


image
initialize with random values

Gradient of activation A in the selected softmax neuron with respect to the pixel values in the image.

Set the gradient of neuron for class k to one and all other gradients to zero.

pixel update

$$p_i^{(t)} = p_i^{(t-1)} + \varepsilon^{(t)} \left. \frac{\partial \mathrm{A}(\mathbf{p})}{\partial p_i} \right|_{p_i = p_i^{(t-1)}}$$

Optimize image for a high class probability

Synthetic images that maximize the probability for a selected class label when using a CNN with fixed weights that were trained on imageNet data.

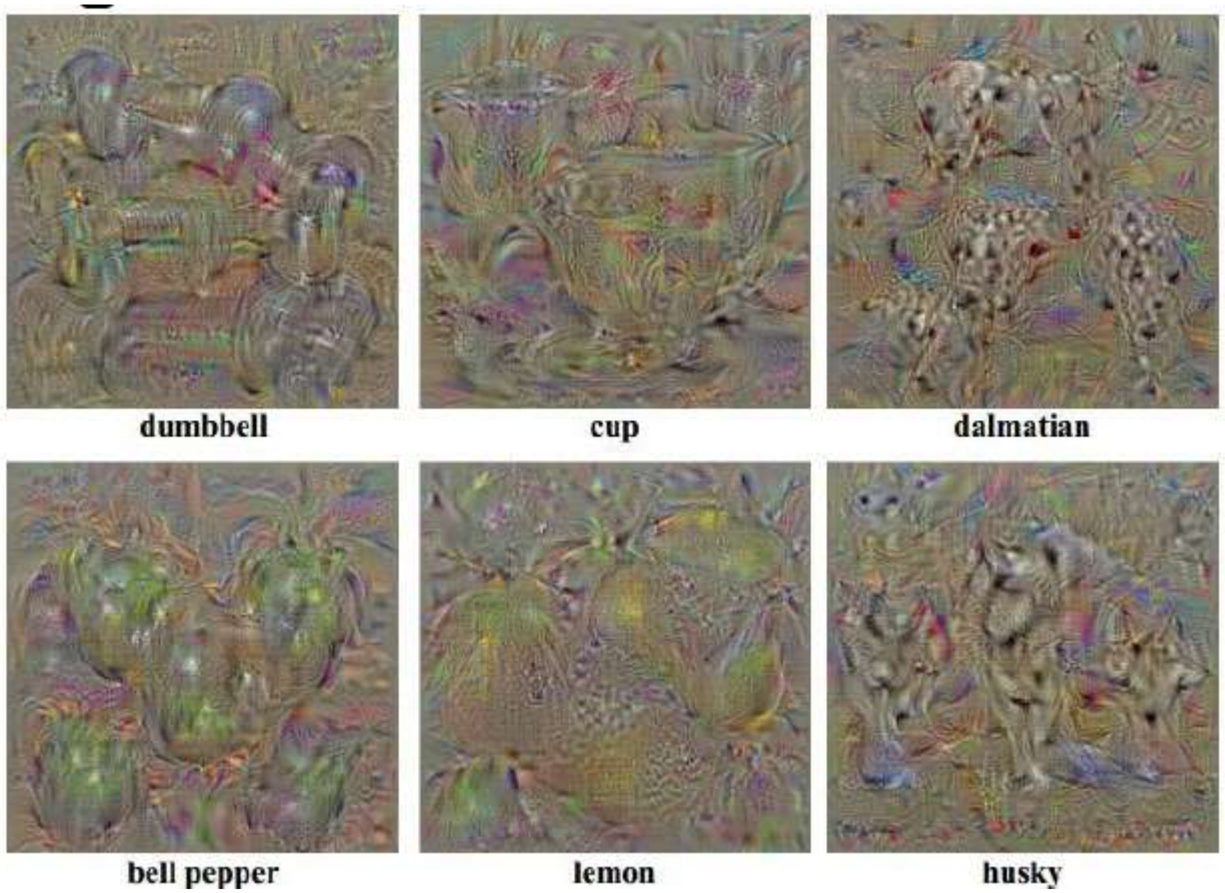
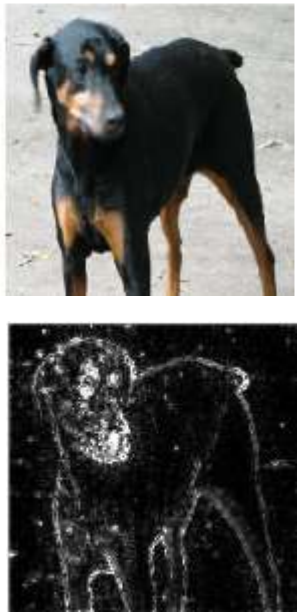


image credit: cs231n

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

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

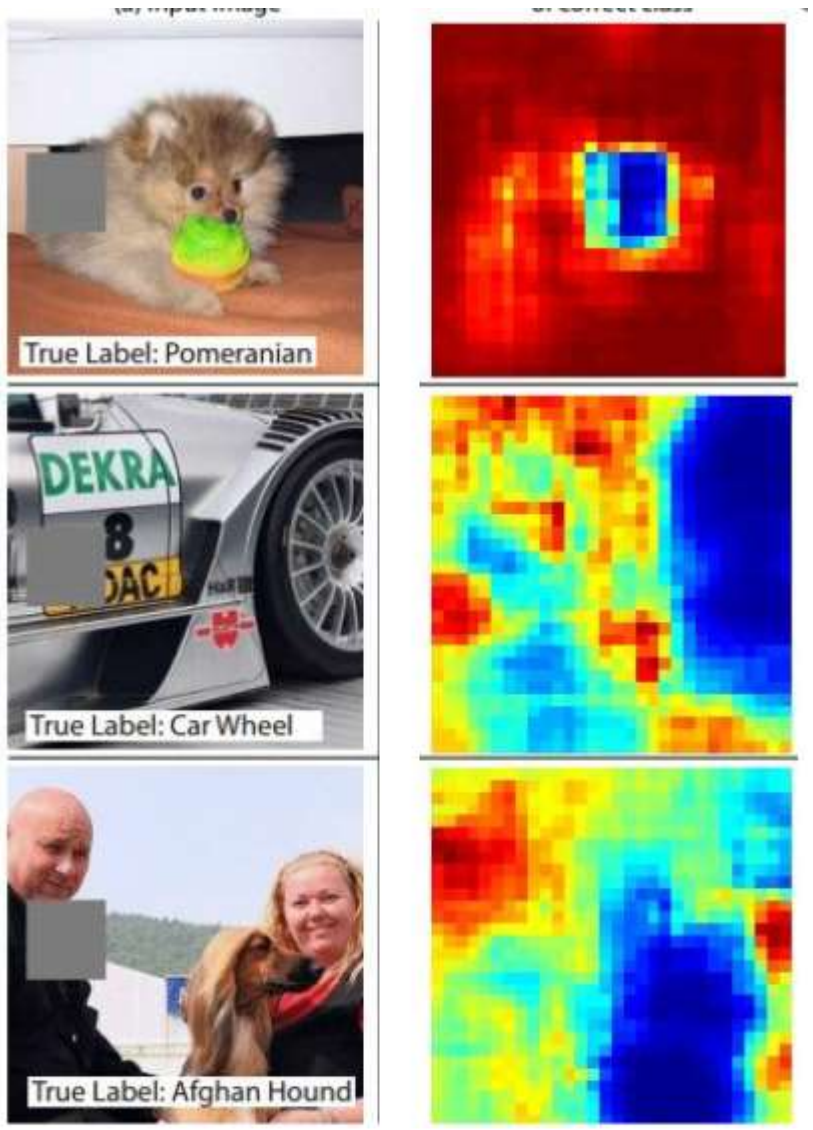
Example how to compute saliency with keras

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…



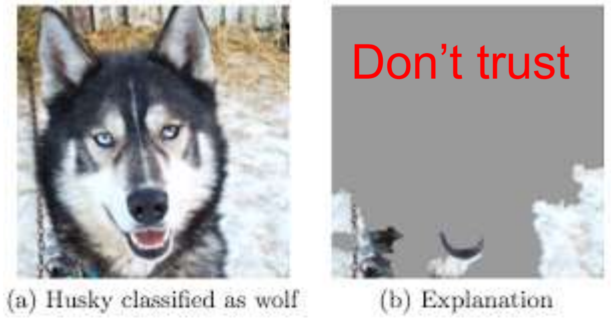
Occlusion experiments [Zeiler & Fergus 2013]

image credit: cs231n

Which pixels are important for the classification?
LIME: Local Interpretable Model-agnostic Explanations

Idea:

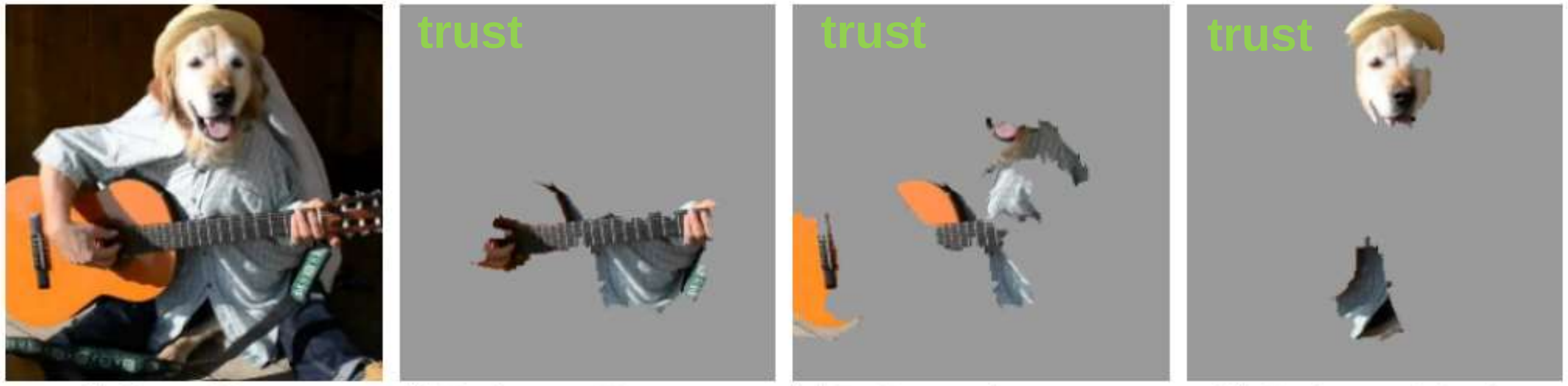
1) perturb interpretable features of the instance – e.g. randomly delete super-pixels in an image and track as perturbation vector such as (0,1,1,0,…,1)=x.
2) Classify perturbed instance by your model, here a CNN, and track the achieved classification-score=y
3) Identify for which features/super-pixels 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*