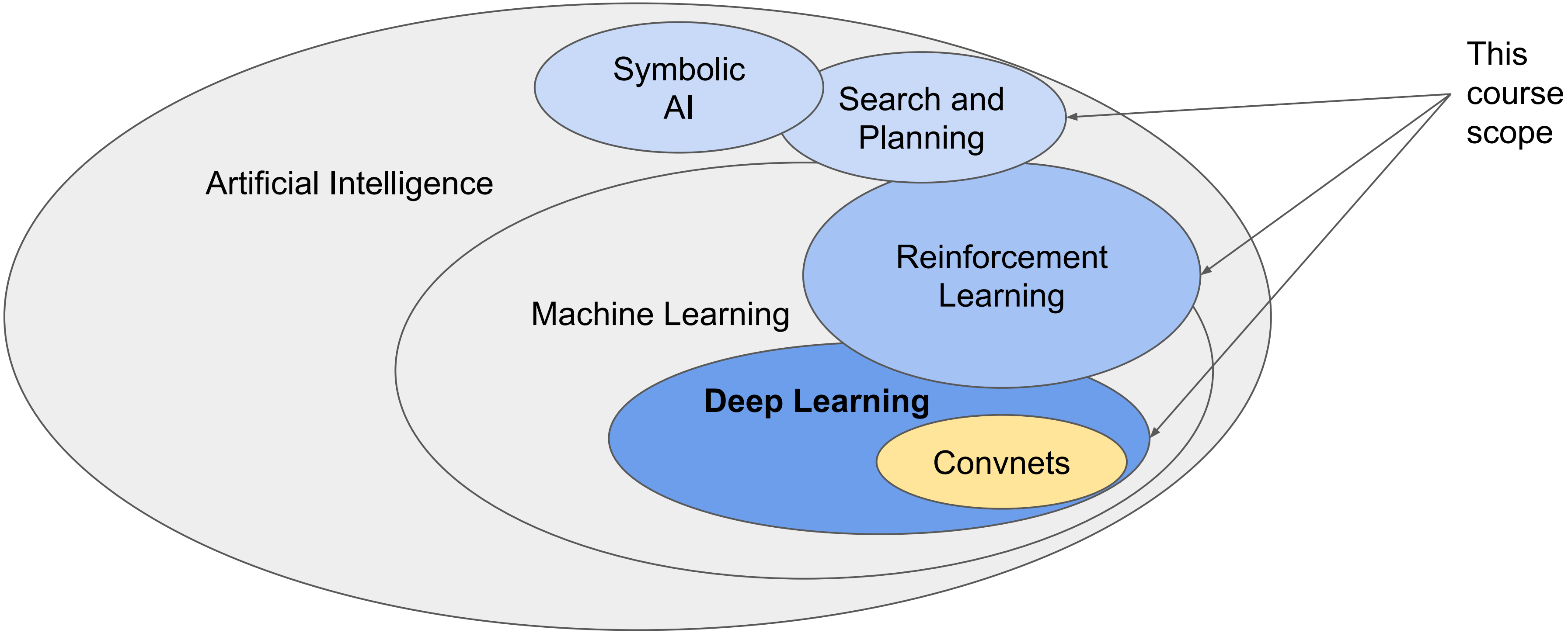
IT00DP82-3007 Artificial Intelligence and Machine Learning

peter.hjort@metropolia.fi

Machine Learning in Artificial Intelligence (AI) landscape



Computer vision

Computer vision is one of the fields where significant progress has been made during last ~20 years.

Typical problems in computer vision deal with:

Image classification: classify the image to one (or more) known categories. For example ImageNet data set has the images classified to 1000 categories (and some have multiple subcategories etc). (By the way: is 1000 much or little or just right?)

Object detection: detect interesting objects from a given image, and, for example, draw bounding boxes and classify them.

Face recognition: find the closest known match for a given face image.

Dense network and image classification

We have already done this with MNIST data set, what’s the problem?

MNIST data set is rather simple. Data sets with more classes and/or with more complex images are more difficult.

If we use a dense MNIST example network to classify into 10 categories:

- 64x64x3 image: 1,241,025 trainable parameters
- 256x256x3: 19,666,460
- 1024x1024x3: 314,578,460

(And this was with a small model)

Dense network and image classification

In a dense network every output from a layer is connected to every neuron in the next layer. Also, the spatial relationships between pixel values are lost (pixel matrix is turned into a vector before it is fed to the input layer).

But does that make sense when dealing with images?

- Perhaps in case of images **spatial** relationships are important
- Perhaps images have more **structure** than just pixels → hierarchy of features
- Perhaps it makes sense to take a **more local look** at pictures → some features of the image might get repeated in other locations

Better have something a bit more clever?

Image classification and hierarchy

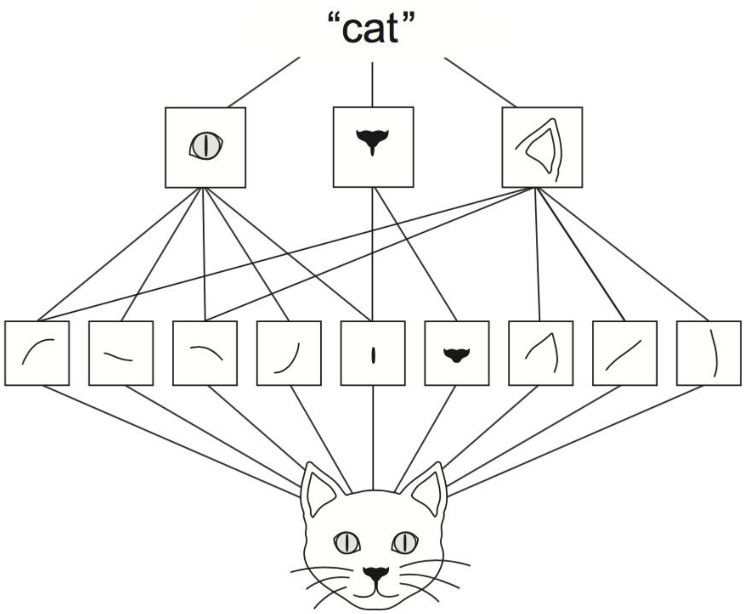
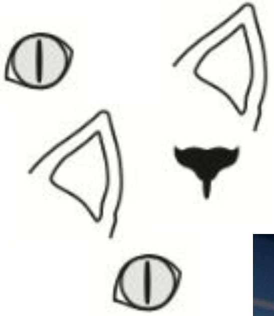


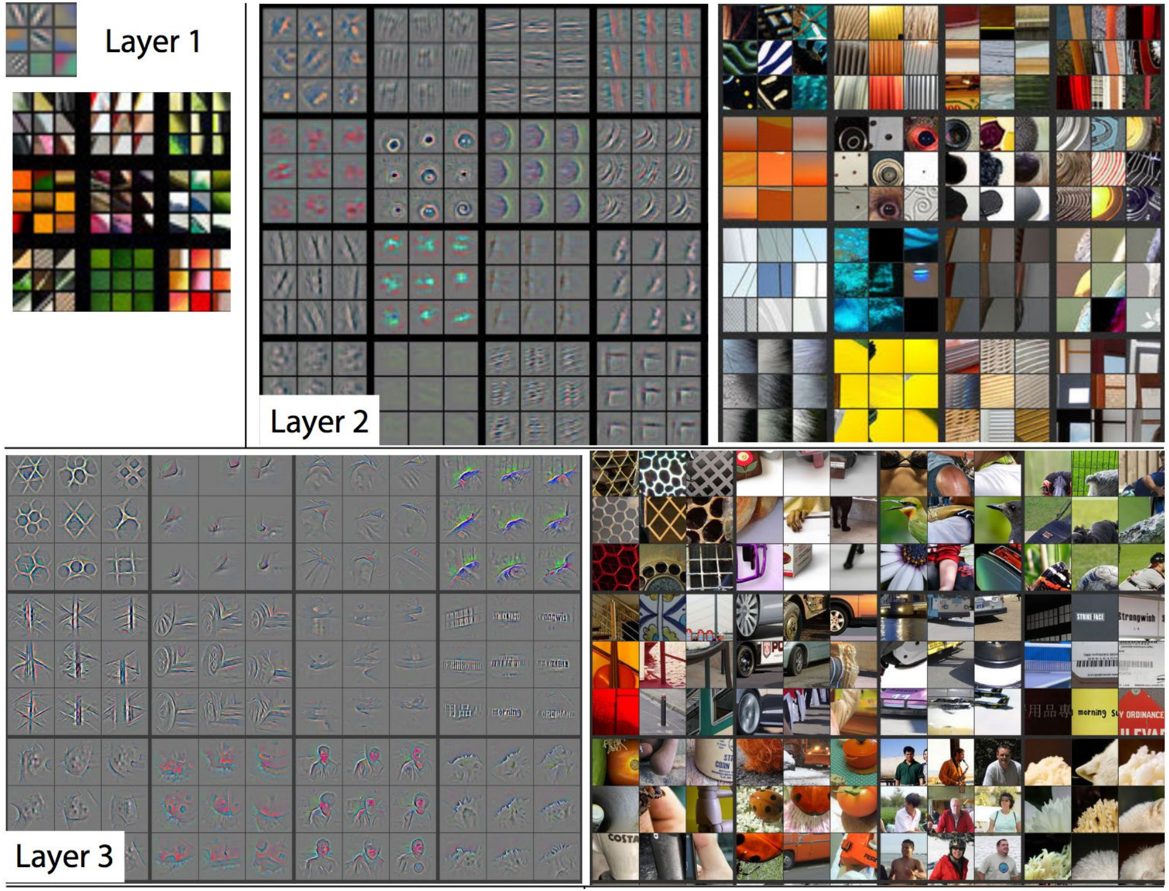
Figure 5.2 The visual world forms a spatial hierarchy of visual modules: hyperlocal edges combine into local objects such as eyes or ears, which combine into high-level concepts such as "cat."

From Chollet, chapter 5.1.1



Is this a cat ?

Image classification and hierarchy with filters



From Zeiler and Fergus “Visualizing and Understanding Convolutional Networks” (https://arxiv.org/abs/1311.2901)

Convolutional networks or ConvNets

A convolutional net has a **prior** for image-related inputs. The prior comes from the use of convolution and pooling operations on image data - network is not as free in combining the weights as in dense case. Instead, same patterns will be recognised in different parts of the image - translation invariance.

The use of convolutions has brought image classification accuracies in some problems to very high level, even surpassing human accuracy.

There is a range of choices for the overall structure of a convnet that can be used as basis. Research on ConvNets goes on, and new network architectures are being suggested.

2D convolution

Element-wise multiplication and sum:
1*0+1*3+1*6+0*0+0*4+0*7+(-1)*5+(-1)*1+(-1)*1

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

6x6x1 image

*

1	0	-1
1	0	-1
1	0	-1

3x3 **filter**, or **kernel**

Filter size is sometimes called *receptive field*

→

2	7	4	-7
13	17	-3	-6
13	14	-5	-5
12	8	-5	-1

(6-3+1) x (6-3+1) x 1 result

What is happening in 2D convolution

9	9	9	9	0	0	0	0
9	9	9	9	0	0	0	0
9	9	9	9	0	0	0	0
9	9	9	9	0	0	0	0
9	9	9	9	0	0	0	0
9	9	9	9	0	0	0	0
9	9	9	9	0	0	0	0
9	9	9	9	0	0	0	0

lighter pixels

darker pixels

*

1	0	-1
1	0	-1
1	0	-1

0	0	27	27	0	0
0	0	27	27	0	0
0	0	27	27	0	0
0	0	27	27	0	0
0	0	27	27	0	0
0	0	27	27	0	0

This particular convolution filter/kernel seems to detect vertical edges.

Details - Padding

Convolution shrinks the height and width dimensions of tensor. If no shrinking is desired, tensor can be padded with zeros before convolution.

- No padding: `'valid'` in Keras
- Pad to make input & output dimensions same: `'same'` in Keras
- Other values are possible, too (but not used much)

padding = 1

0	0	0	0	0	0
0	2	4	2	4	0
0	1	1	2	1	0
0	1	4	3	3	0
0	3	2	1	2	0
0	0	0	0	0	0

*

1	0	-1
1	0	-1
1	0	-1

-5	-1	0	4
-9	-3	2	7
-7	-1	1	6
-6	0	1	4

Details - Stride

of positions the convolution filter moves at one step.

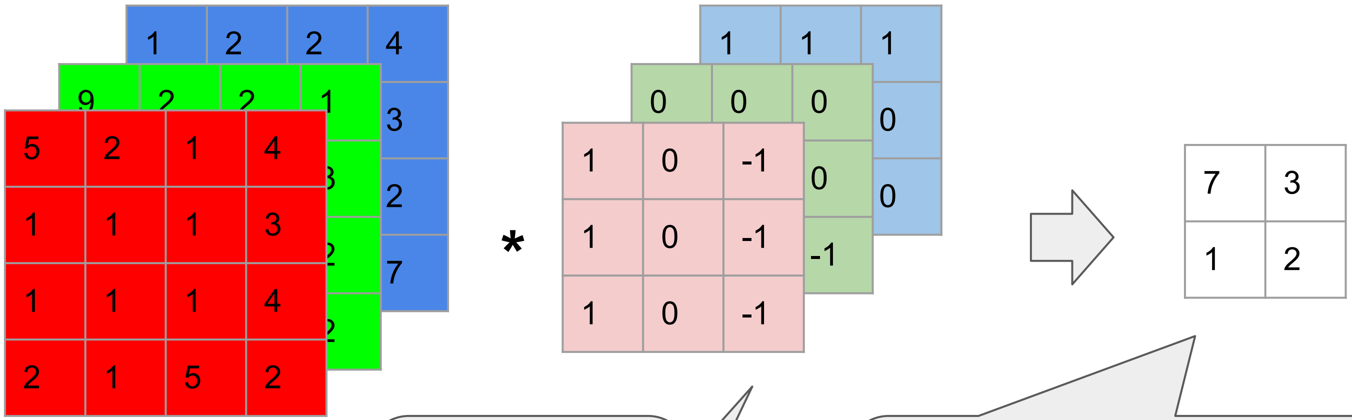
- Previous examples had stride = 1, so we were moving in one step at a time
- Stride = 2 - move 2 positions at a time:

$$\begin{bmatrix} 5 & -1 & 0 & 4 \\ -9 & -3 & 2 & 7 \\ -1 & -1 & 1 & 6 \\ -6 & 0 & 1 & 4 \end{bmatrix} * \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix} \Rightarrow \begin{bmatrix} -4 & 2 \\ -7 & 2 \end{bmatrix}$$

- Strides are more often used in pooling operations (wait for a couple of slides)

RGB image convolution

Think of RGB image as 3-dimensional box, where channels are in depth direction:

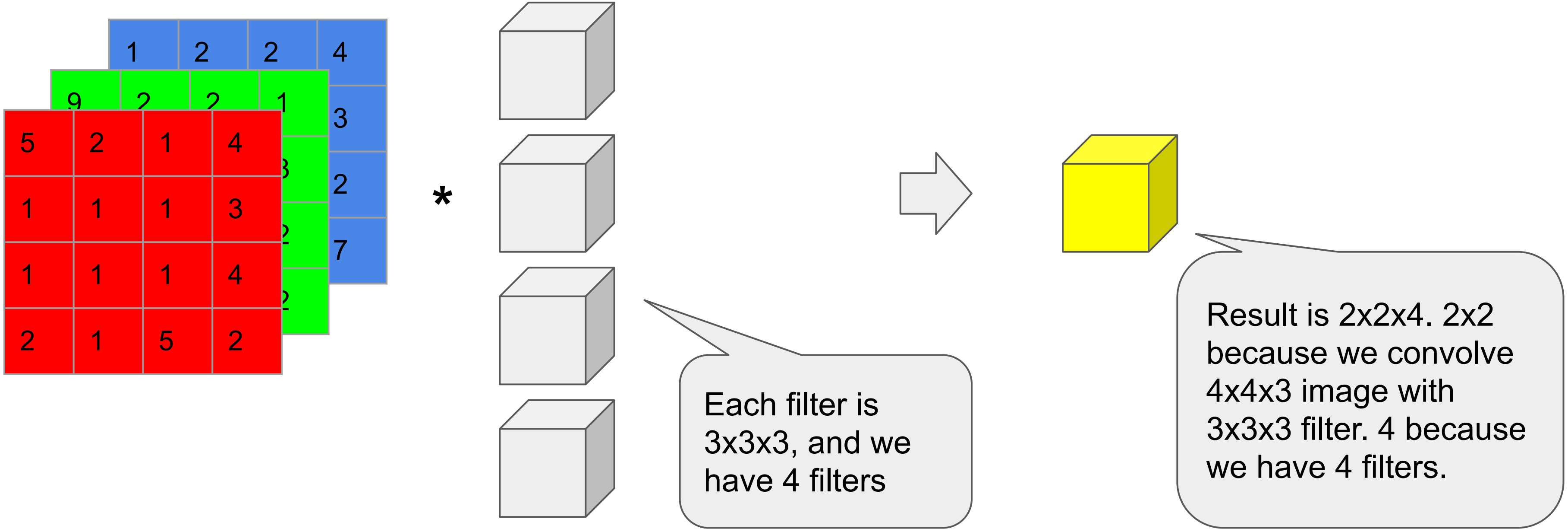


Filter must have the same depth as the sample.

Upper left corner 7 is the result of doing convolution in upper left corner of red channel with light red filter, green with light green filter, blue with light blue filter, and summing all.

Convolution with multiple filters

Usually more than one filter is used in convolution. Each filter is **applied separately**, and **results are stacked in depth dimension**:



Where do the convolution filter values come?

They are **learnable parameters** (or **weights**)!

We are not designing the filters beforehand, like in signal processing, but **learn the values in filters during training**. This gives the network ability to **learn whatever filters needed to minimize the loss function**.

Filter parameters get learned once, and are then used in all spatial positions (height & width) of the image - parameter sharing.

A typical convolutional network architecture network that is tasked to do classification is divided into two parts:

- convolutional part where **an internal representation** of the image is found
- classification part which outputs probability vector that contains probabilities for all categories (this is quite like the network we have created before)

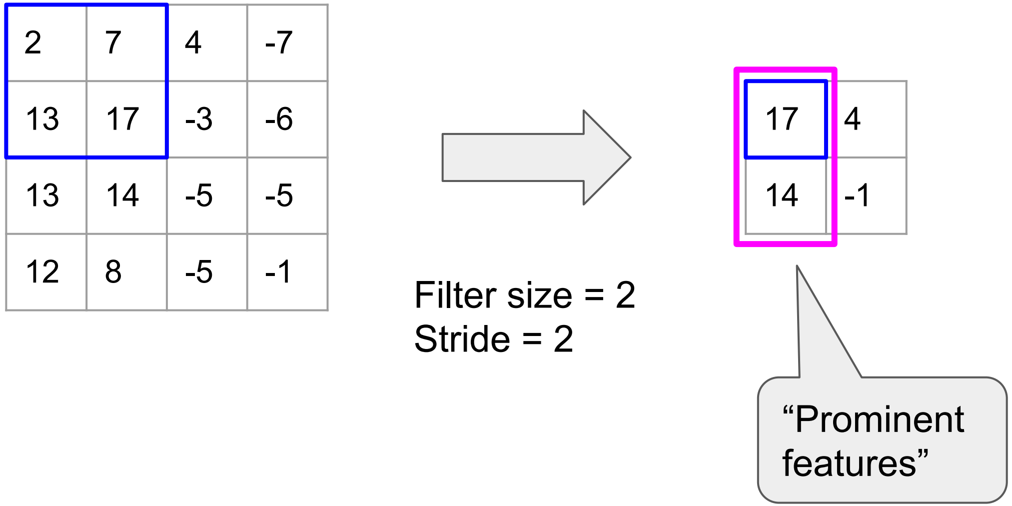
of parameters/weights in convolutional layer in Keras

```
model.add(Conv2D(filters=6,
                 kernel_size=5,
                 strides=1,
                 use_bias=True,
                 padding='valid',
                 input_shape=(28, 28, 3,),
                 activation='relu'))
```

((5 x 5 x 3) + 1) * 6 = 456 (450 if `use_bias` parameter in `Conv2D` is set to `False`)

```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 24, 24, 6)         456
_________________________________________________________________
```

Max pooling



Another option: average pooling (not very common)

Max pooling is done independently for all channels. With max pooling there are no parameters to learn.

Typical convolution network structure

One or more groups of one or more convolution layers, followed by pooling layer

```
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=(28,28,1))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
```

Group of dense layers, followed by softmax output layer

```
model.add(Dense(128, activation='relu'))
model.add(Dense(num_classes, activation='softmax'))
```

Flattening layer

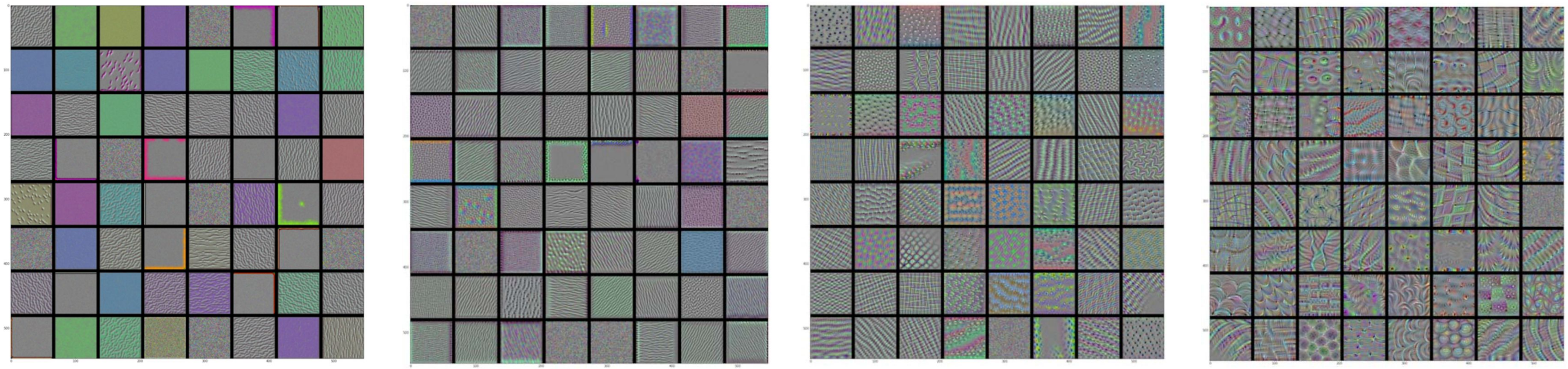
In the previous example the output of `MaxPooling2D` layer has shape `(12,12,64)`. How do we feed this to a `Dense` layer?

Flatten it:

```
model.add(Flatten())
```

This will output shape `(9216)` (12*12*64)

What do convnets “see” - filter patterns



Low to high layer

From “Deep Learning with Python by Francois Chollet

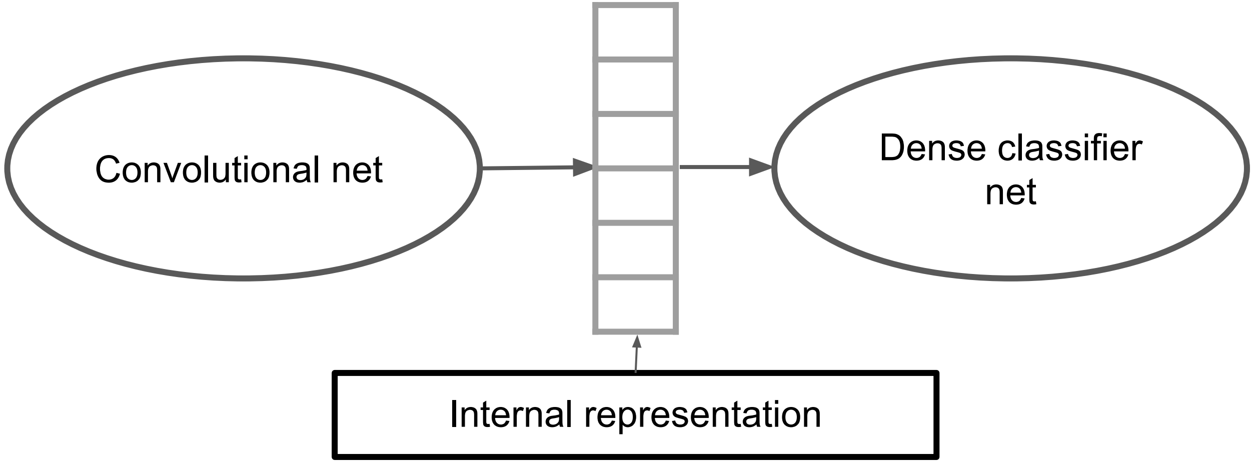
What do convnets “see” - class activation heatmap

From “Deep Learning with Python by Francois Chollet

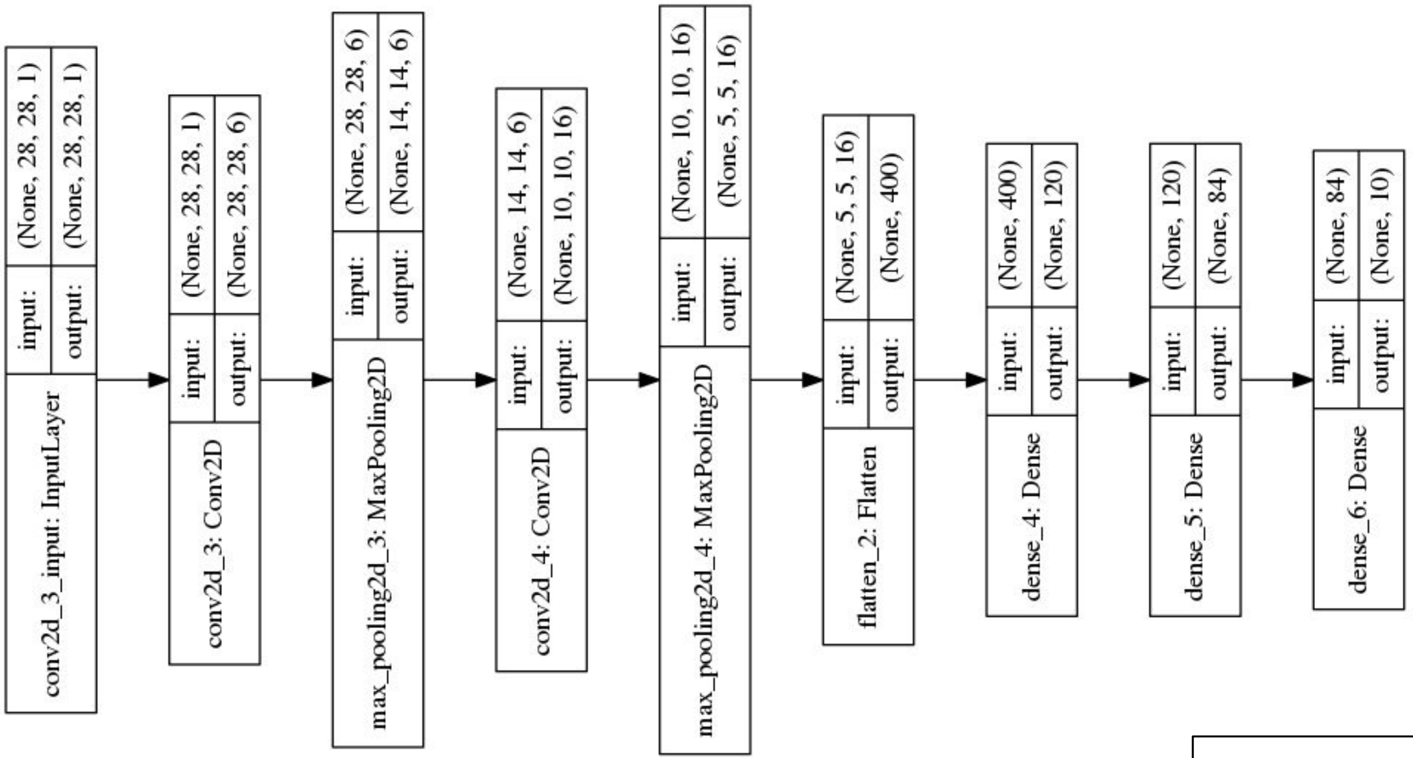
Example network architectures

We'll take a look at some example networks that have been used for solving image classification problem.

The networks share a common structure: there is a convolutional network (convolution and pooling layers), followed by a dense network for producing classification predictions.

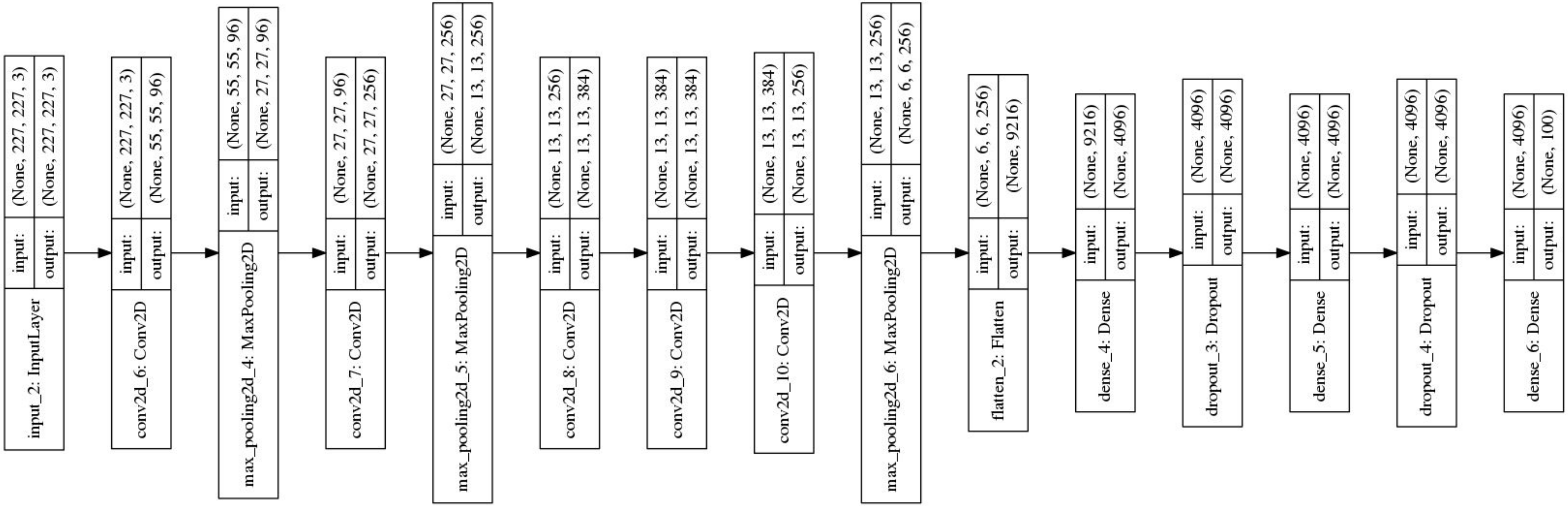


Example network: LeNet-5 (1998) (simplified)



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

Example network: AlexNet (2012) (simplified)



https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

VGG-16 (2015)

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Table 2: **Number of parameters** (in millions).

Network	A,A-LRN	B	C	D	E
Number of parameters	133	133	134	138	144

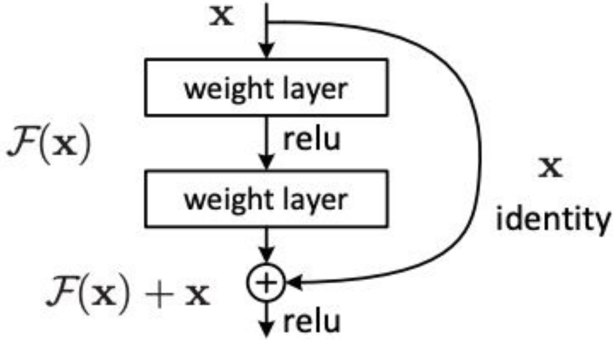
From https://arxiv.org/pdf/1409.1556.pdf

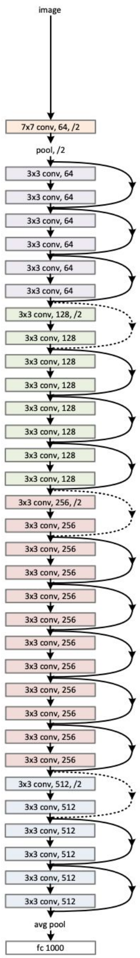
Residual connections

In **ResNet** the network is very deep (up to 152 layers) - gives ability to learn very wide family of functions, but vanishing gradients are a real problem.

Solution: skip connection - add an identity connection that “short-circuits” the conv layer. This way derivative information “leaks” to the earlier layers in the network in back propagation.

But how to implement this?





Why do convolutional nets work for image-related tasks?

- The patterns convolutional nets recognise are **translation-invariant** - after learning to recognise a pattern in one position in the image, it is recognised in all positions (think about the filter, or kernel, or **receptive field moving over the image**)

- The patterns learned by a convolutional net are hierarchical - in first layer low-level patterns are learned, but subsequent layers learn patterns based on the simpler patterns found in previous layer.

Data augmentation

Too few samples to train the model? Image classifier models need a respectable amount of data to train them with.

Don’t worry (or do) - but let’s create fake data! Well not completely fake, but for image data, for example, data from real images by random modifications:

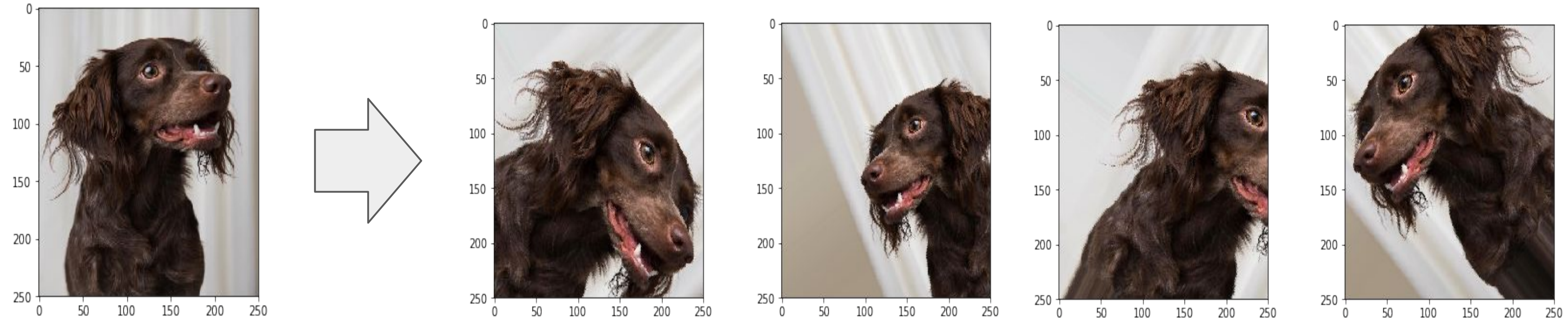
- Rotation
- Zooming
- Flipping
- (and more, see https://keras.io/preprocessing/image/ and Chollet 5.2.5)

Data augmentation with `ImageDataGenerator`

```
from keras.preprocessing.image import ImageDataGenerator

datagen = ImageDataGenerator(
      rotation_range=60,
      zoom_range=0.2,
      width_shift_range=0.4,
      height_shift_range=0.2,
      horizontal_flip=True,
      fill_mode='nearest')
```

Note: values exaggerated for demonstration purposes. Use smaller values in real life.



Preprocessing and generators

Keras `preprocessing.image` has `ImageDataGenerator` that can be used for:

- Reading from disk and preprocessing data in batches
- Making transformations to augment data

```
train_datagen = ImageDataGenerator(rescale=1./255)
train_generator = train_datagen.flow_from_directory(train_dir,
                          target_size=(150, 150)
                          batch_size=20,
                          class_mode='binary')
```

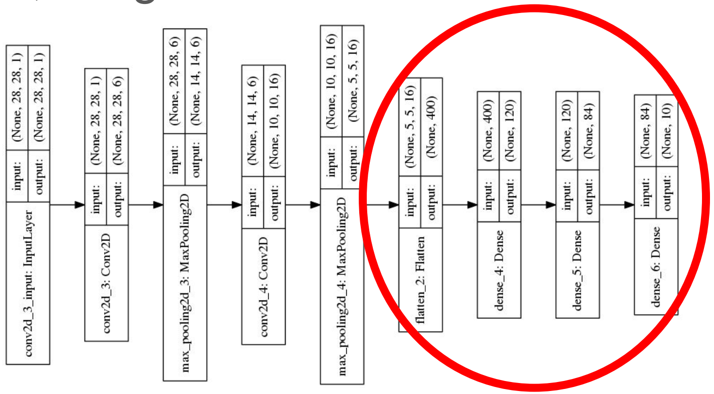
```
history = model.fit_generator(
      train_generator,
      steps_per_epoch=100,
      epochs=30,
      validation_data=validation_generator,
      validation_steps=50)
```

One epoch is 20 * 100 training samples

Transfer learning - using pre-trained convnet

For an image classification task use a pre-trained convolutional network as the basis:

- Assumption is that the convolutional part of a pre-trained network has learned general enough lower-level features when it was trained for the classification task
- The classification (dense) part of the network is specialised in the original classification task, so ignore that and train a new classifier in its place



Loading pre-trained network

```
from keras.applications import VGG16

conv_base = VGG16(weights='imagenet',
                  include_top=False,
                  input_shape=(150, 150, 3))
```

Load VGG16 model trained with ImageNet data set and ignore the dense classification layers.

Options for the next step:

- run own training samples through `conv_base` and save to file and train classifier on top of that, or
- extend the model with dense layers for classification - in this case data augmentation can be used

See https://keras.io/guides/transfer_learning/

Extend model with classification layers

```
from keras import models
from keras import layers

model = models.Sequential()
model.add(conv_base)
model.add(layers.Flatten())
model.add(layers.Dense(256, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

A loaded model can be used just like a layer in building a new model.

This is a dog vs. cat example - binary classification so `sigmoid` is used. For multinomial, last layer would have `softmax`.

```
conv_base.trainable = False
model.compile(loss='binary_crossentropy',
              optimizer=optimizers.RMSprop(lr=2e-5),
              metrics=['acc'])
```

Before training, freeze the pre-trained part and compile the model.

Using pre-trained convnet - fine-tuning

Could some part of the convolutional base be included in training for the target data set? Perhaps upper layers in convolutional part would contribute in more accurately classifying target data.

So, let’s unfreeze the top of convolutional part, and train it together with the top layer that was trained in “extend the model” thread. (This to avoid too large loss values getting propagated during training).

```
conv_base.trainable = True

for layer in conv_base.layers:
    if layer.name == 'block5_conv1':
        set_trainable = True
    if set_trainable:
        layer.trainable = True
    else:
        layer.trainable = False
model.compile(loss='binary_crossentropy',
    optimizer=optimizers.RMSprop(lr=2e-5),
    metrics=['acc'])
```

Test accuracies of dog vs. cat classification (from Chollett)

Standalone model, no augmentation: 70-72%

Standalone model with augmentation: 82%

Re-use ImageNet model, no augmentation (all samples mapped to convolutional base results): 90%

Re-use ImageNet model with augmentation: 96%

Re-use ImageNet model with fine tuning: 97%

See https://www.tensorflow.org/tutorials/images/transfer_learning

Example on using the internal representation - Autoencoder

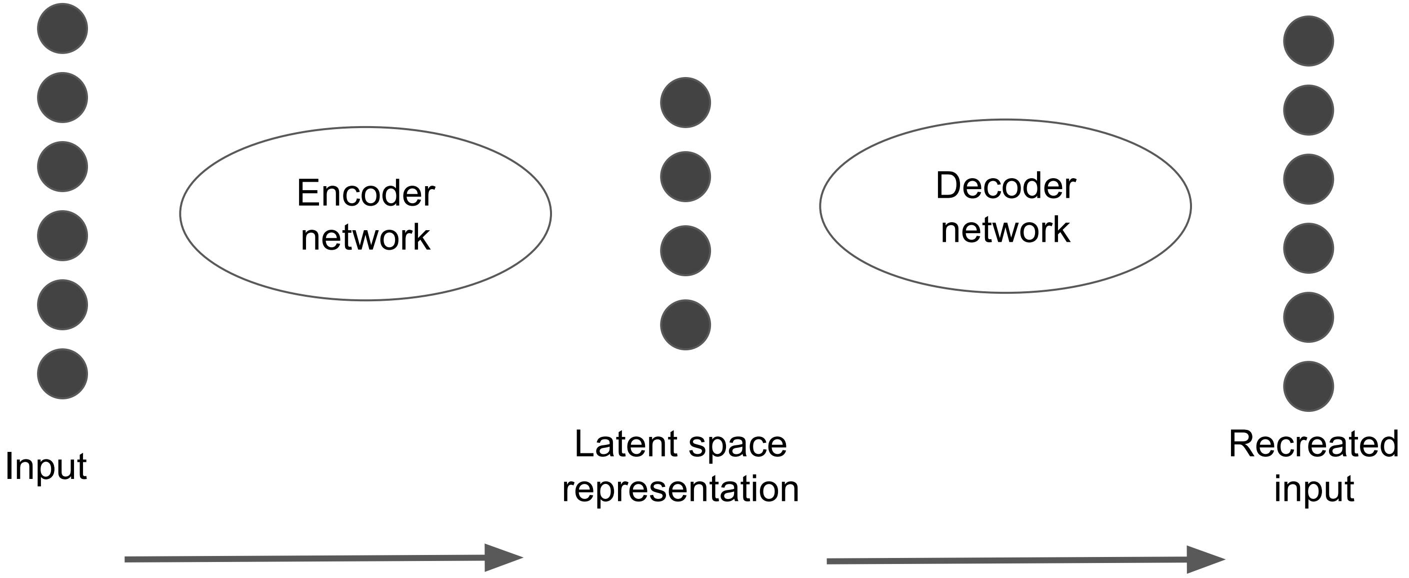
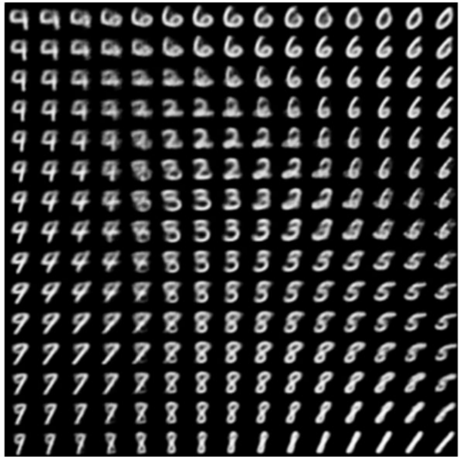


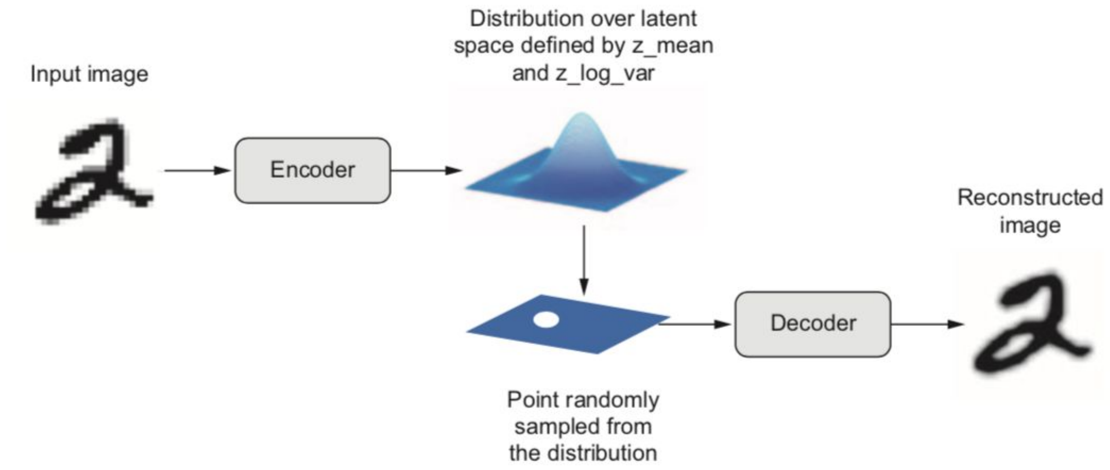
Image generation - variational autoencoder

Variational autoencoder takes the autoencoder idea further by

- Forcing the encoded (latent) space to be more **structured**
- Making the latent space more **continuous** - the space should not contain “gaps” that don’t have a mapping to reasonable image, and when moving a small distance in the latent space does not cause the mapped image to be drastically different



Variational autoencoder architecture



For Keras implementation, see 8.4-generating-images-with-vaes in book examples github.

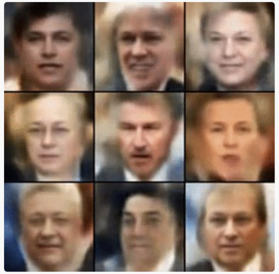
Both the encoder and decoder are convnets that are trained with the same idea as an autoencoder - reconstructed image should match input image.

Variational: the encoded image representation fed into decoder is sampled from a point defined by the normal distribution with parameters z_mean and z_var. Even if the sample is not exactly at z_mean the target is to recreate the original image. This creates potentially useful structure in the latent space (which the plain autoencoder does not have).

Variational autoencoder and concept vectors

For some applications it is possible to identify concept vectors in the latent space. Concept vectors are added/subtracted from the latent representation and image in original space recreated →

Add (or remove) “smile” vector to/from images



Create fictional faces

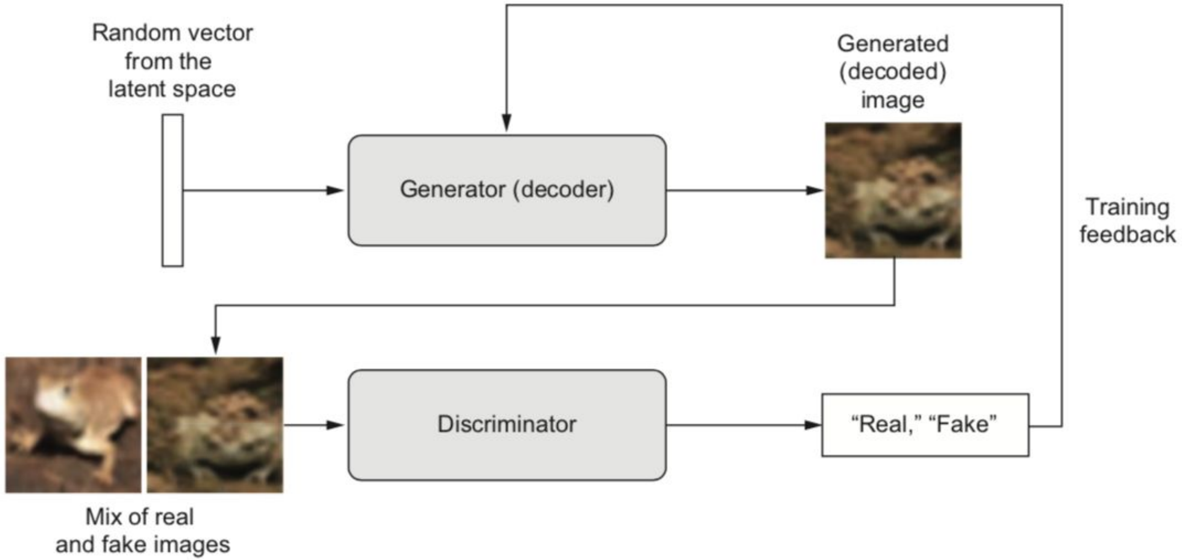
GAN (Generative Adversarial Network)

Same target as in VAEs (Variational Autoencoders) - learn a useful latent space (typically for images).

Two components:

- **Generative network**: generate a synthetic image based on random point in the latent space - aiming to learn to generate more and more real-looking images
- **Discriminator network (adversary or teacher)**: predict whether a given image is real (is taken from training set) or fake (is generated) - learn to detect fakes better and better

GAN architecture



Generated face faces based on CelebA images dataset. See https://research.nvidia.com/publication/2017-10_Progressive-Growing-of