MSE | MASTER OF SCIENCE IN ENGINEERING

Deep Learning

TSM-DeLearn

10. CNN3 - Keras fun API - Transfer learning - Autoencoders

Jean Hennebert
Martin Melchior

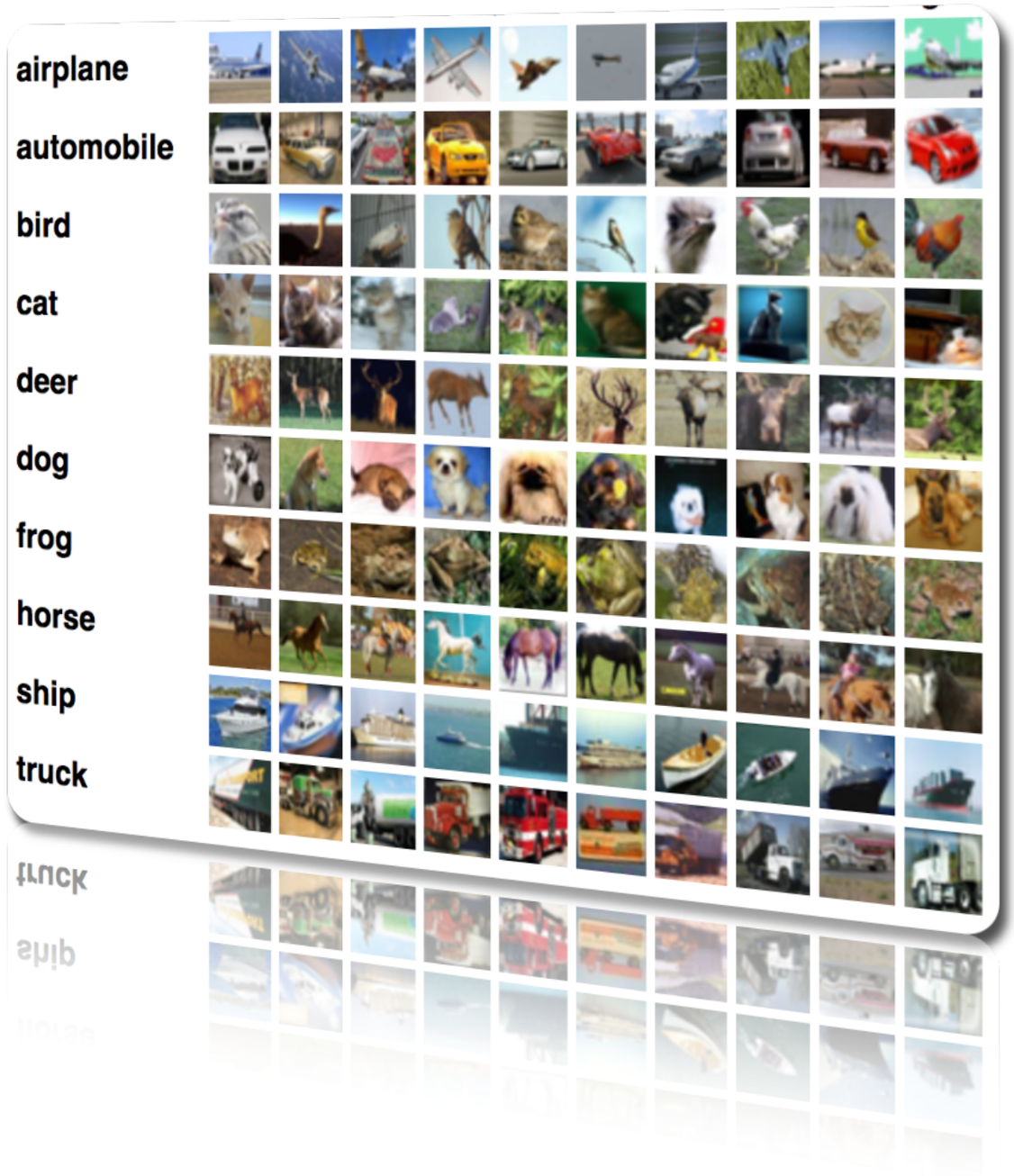
Plan

1. CNN2 recaps
2. Keras Functional API
3. Transfer learning
4. Autoencoders
5. PW

CNN2 recaps

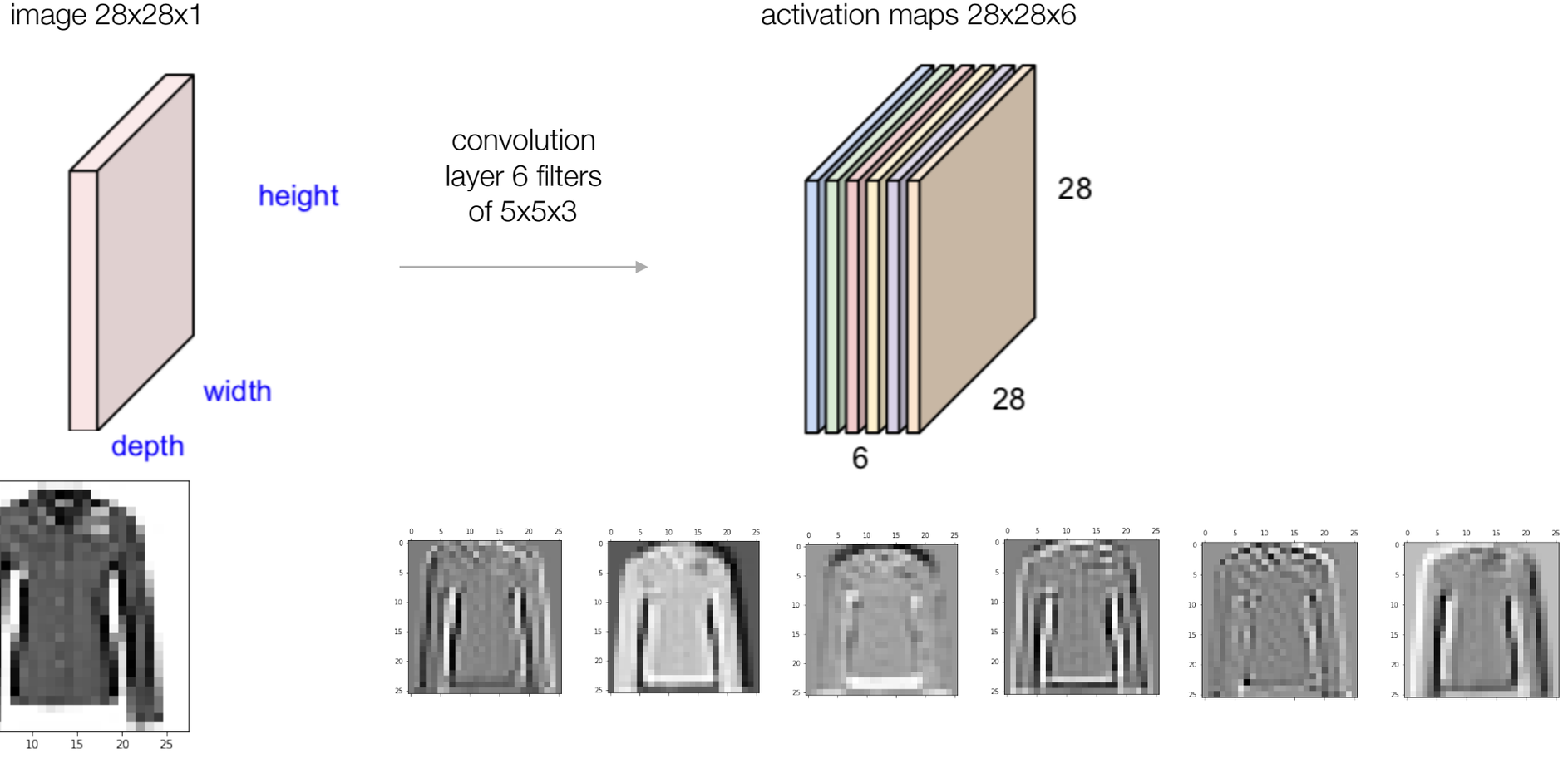
Recaps from last week

CIFAR10 data augmentation leaderboard

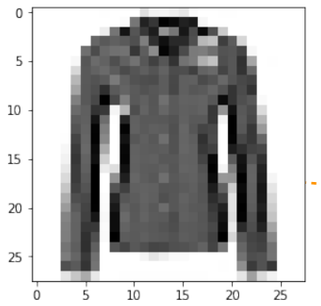


Visualisation - activation maps

- First strategy: **visualise the activation maps**
  - For a given conv layer, consider each filter independent and visualise the corresponding activation map



Visualisation - activation maps



Extract the output of the 1st layer from the CNN model.

Rebuild a new Keras Model with the input layer and the output of the first layer.

The network expects a batch of images so we need to reshape the input image into a batch of 1.

Do the forward pass and plot the activations for 6 of the filters.

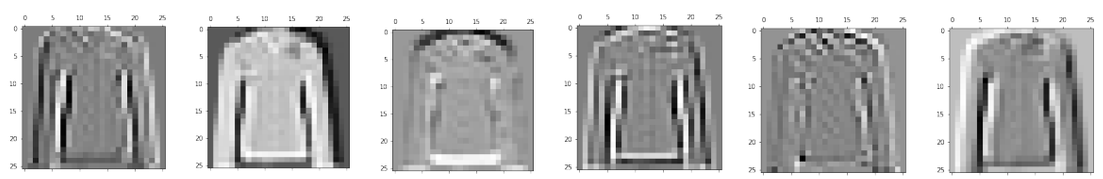
```
test_im1 = X_train[500]
plt.imshow(test_im1.reshape(28,28), cmap='Greys',
           interpolation='none')
plt.show()
```

```
from keras import models
# extracts the output of the first layer
layer_1st_conv = cnn1.layers[0].output
```

```
# creates a model able to return these outputs, given an input
activation_model = models.Model(inputs=cnn1.input,
                                outputs=layer_1st_conv)
```

```
# we need to reshape the image (28,28,1) into (1,28,28,1)
# as the network expect a batch of images as input
test_im1 = test_im1.reshape(1,28,28,1)
```

```
# returns the first layer activation
first_layer_activation = activation_model.predict(test_im1)

# display 6 of the activations of the 1st conv layer
plt.matshow(first_layer_activation[0, :, :, 0], cmap='Greys')
plt.matshow(first_layer_activation[0, :, :, 1], cmap='Greys')
plt.matshow(first_layer_activation[0, :, :, 2], cmap='Greys')
plt.matshow(first_layer_activation[0, :, :, 3], cmap='Greys')
plt.matshow(first_layer_activation[0, :, :, 4], cmap='Greys')
plt.matshow(first_layer_activation[0, :, :, 5], cmap='Greys')
```

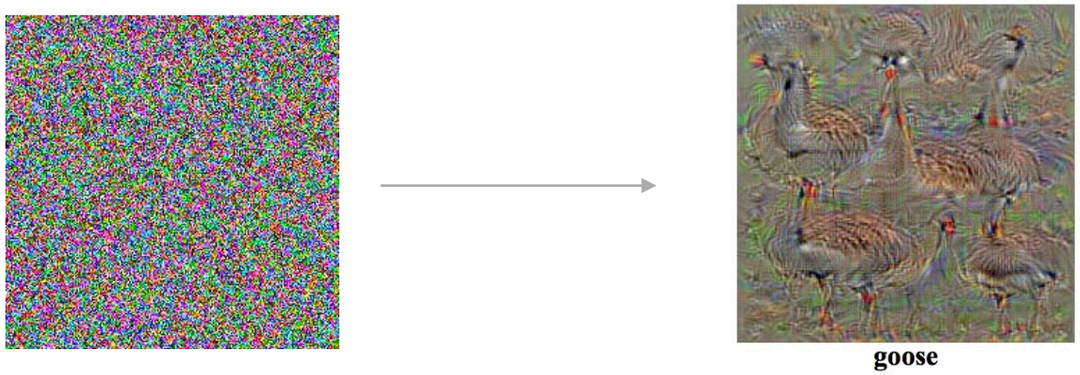


Visualisation - inputs with max activations

- Second strategy: find **input images that maximise the activation** of a given neuron
  1. Start with a random image $\mathbf{x}$
  2. Forward: compute the activation the neuron $a_{i,j}(\mathbf{x})$
  3. Backward: perform the backprob of the gradient of $a_{i,j}(\mathbf{x})$ up to the pixel values: $\frac{\partial a_{i,j}(\mathbf{x})}{\partial \mathbf{x}}$
  5. This gradient tells us how to change the pixel values to increase the activation of the neuron. We can then apply an update rule in the form of a gradient ascent:

$$x \leftarrow x + \alpha \frac{\partial a_{i,j}(\mathbf{x})}{\partial \mathbf{x}} + \text{Regularisation Term}$$

  6. Iterate in 2 until convergence

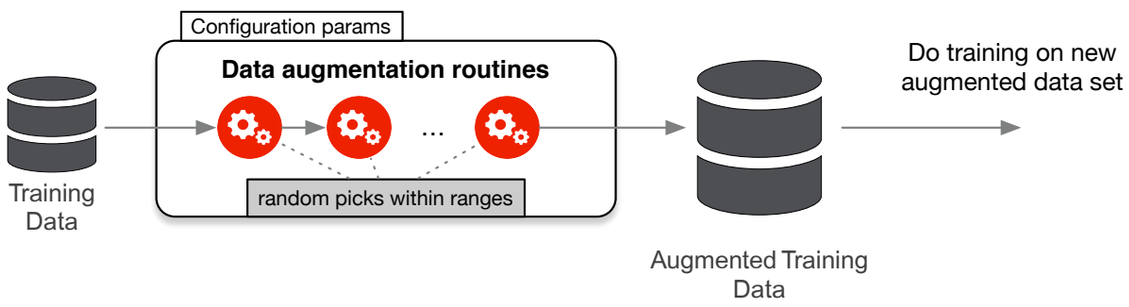


The **regularisation** term impeach pixels to be at extreme values. It kind of pressure to converge to natural looking images.
See e.g. http://yosinski.com/deepvis

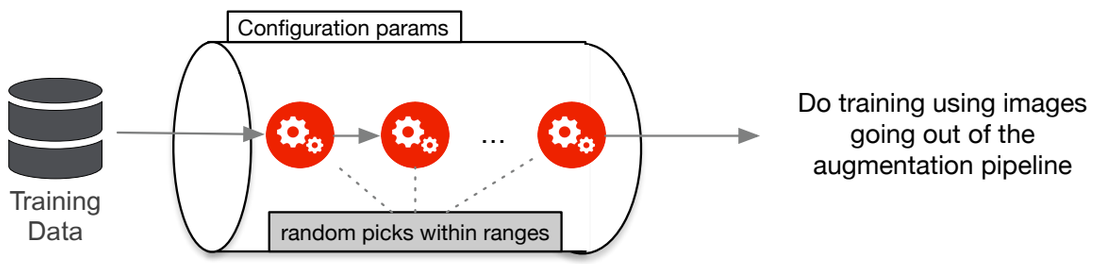
Data augmentation - implementation strategies

- **Data augmentation** takes the approach of generating artificially more training data from existing training samples.

Data augmentation - pre-augmentation

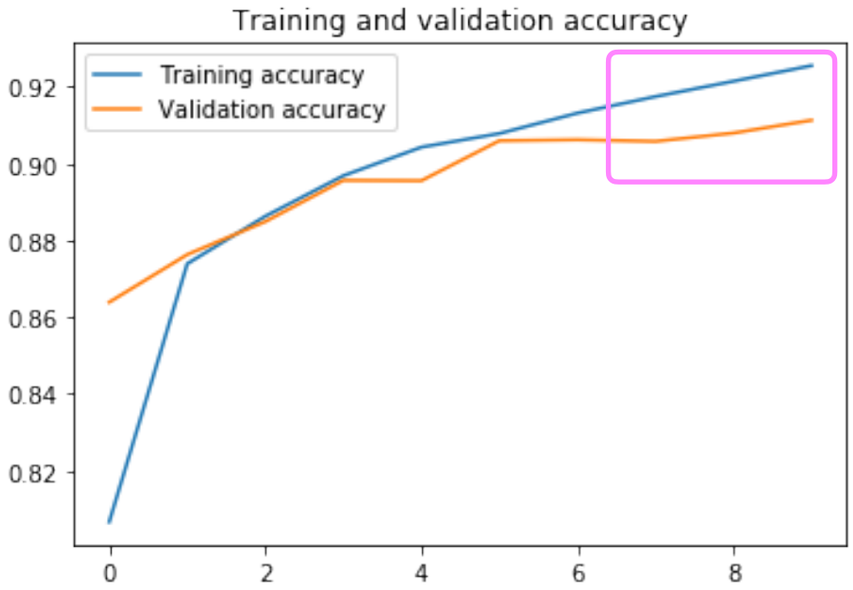


Data augmentation - online augmentation



CNN for FashionMNIST - data augmentation

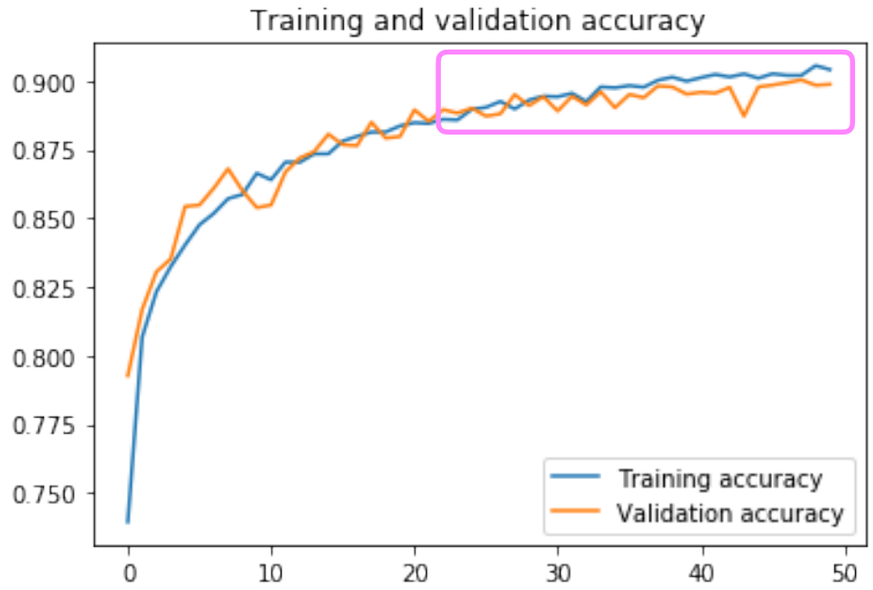
Without data augmentation




```
score1 = cnn1.evaluate(X_test, y_test, verbose=0)
print('Test loss:', score1[0])
print('Test accuracy:', score1[1])
```

```
Test loss: 0.2484699842095375
Test accuracy: 0.9104
```

With data augmentation



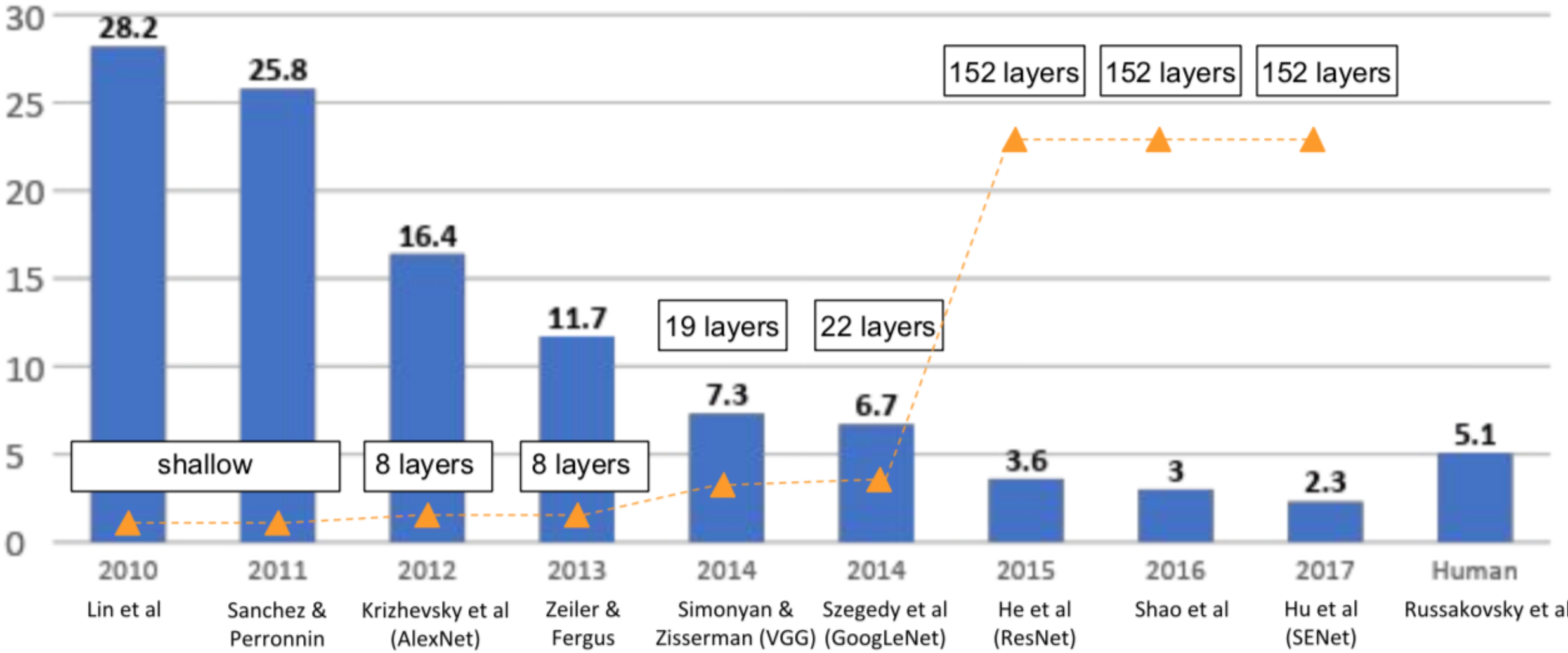

```
score1 = cnn1.evaluate(X_test, y_test, verbose=1)
print('Test loss:', score1[0])
print('Test accuracy:', score1[1])
```

```
10000/10000 [==============================] - 1s 90us/step
Test loss: 0.225622302877903
Test accuracy: 0.92
```

- We may observe that with data augmentation:
  - Training shows less difference between training set and validation set - less prone to overfitting
  - Overall improvement is not so high: from 91.0% to 92.0%
    - This is probably due to the “stability” of the images of FashionMNIST

CIFAR10 - data augmentation in-class leaderboard

ImageNet Large Scale Visual Recognition Challenge (ILSVRC) - winners

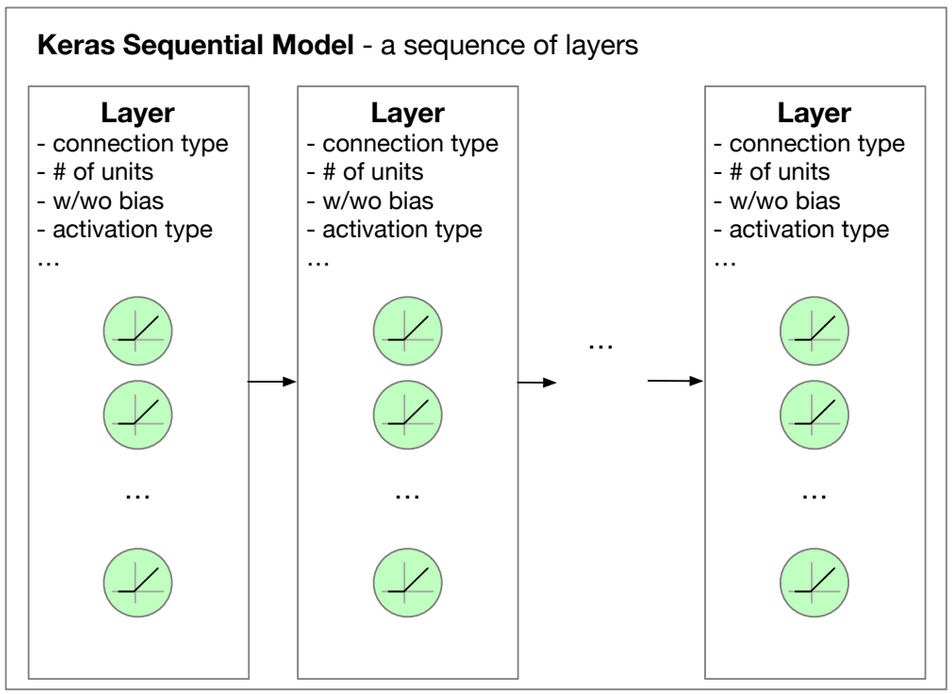


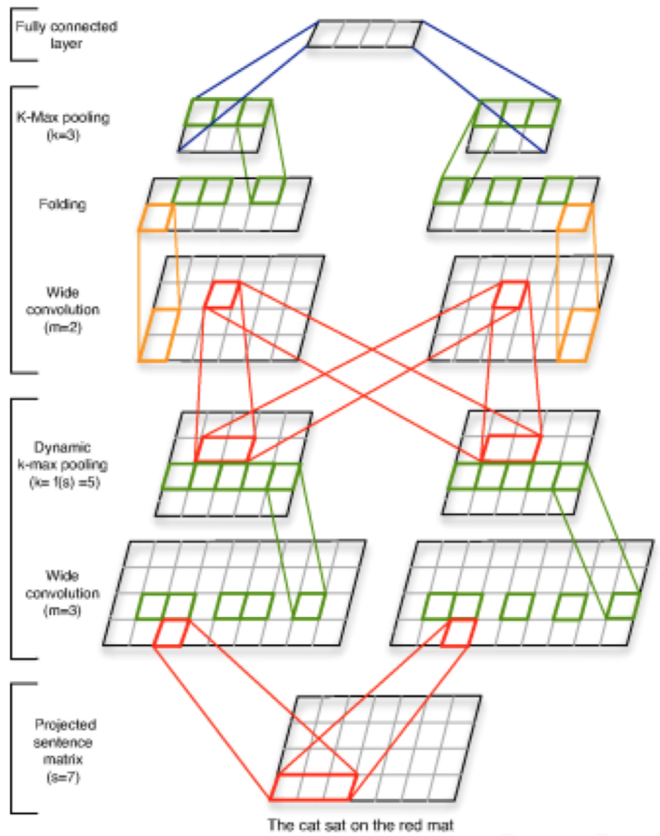
The functional API of Keras

RECAP - Keras Models

- The **sequential** model corresponds to a regular stack of layers
- The **functional** API is used for non sequential architectures

- The functional API allows for
  - Multiple path in computational graph through layer sharing
  - Multiple inputs, multiple outputs



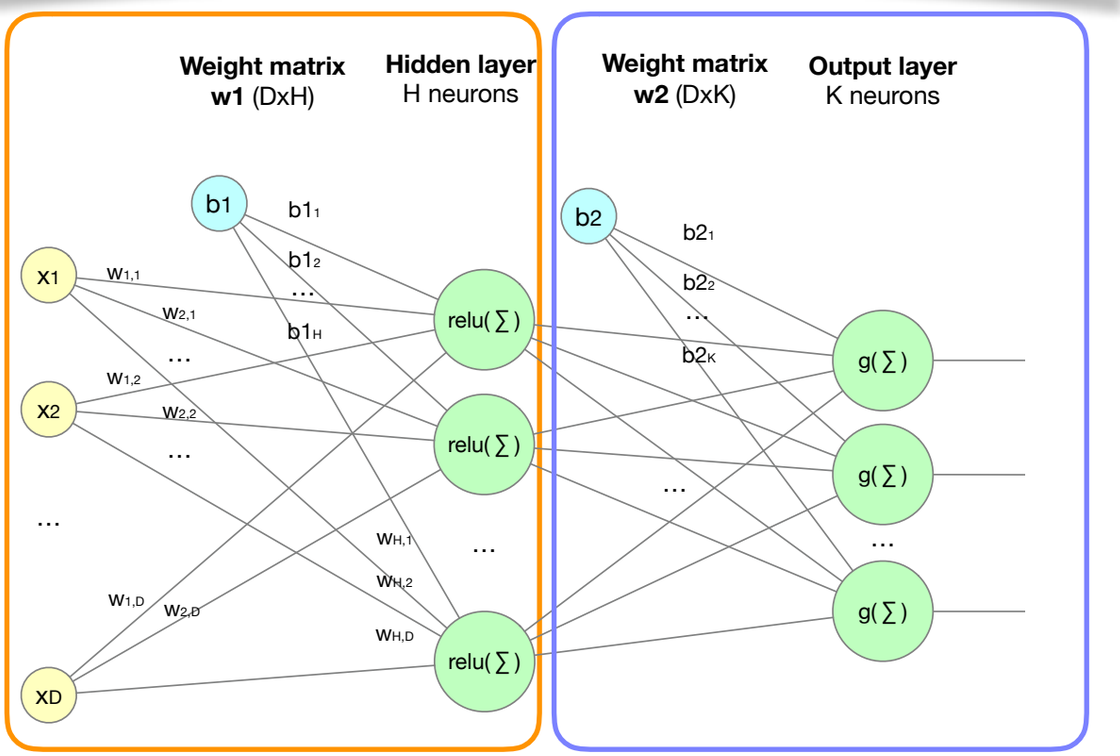


RECAP: MLP with Keras sequential model

In the Sequential model of Keras, layers are added one after the other. In this example, after declaring the model as Sequential(), we add a first fully connected layer.

The "input layer" is automatically created by the sequential model, assuming input is connected to the first layer. In the functional model, we will have to declare it.

We then add a second fully connected layer. The last added layer is assumed to be the "output" layer of the model.




```
H = 300                 # number of neurons
E = 10                  # number of epochs
B = 128                 # batch size
D = X_train.shape[1]    # dimension of input sample - 784 for MNIST

model = Sequential()
model.add(Dense(H, input_shape=(D,), activation='relu'))
model.add(Dense(n_classes, activation='sigmoid'))

model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 300)               235500
_________________________________________________________________
dense_2 (Dense)              (None, 10)                3010
=================================================================
Total params: 238,510
Trainable params: 238,510
Non-trainable params: 0
_________________________________________________________________
```

Keras functional API - examples

- MLP
- CNN
- Shared input layer
- Shared feature extraction layer
- Multiple input
- Multiple output

- All examples in the following slides done on MNIST

Keras functional API - MLP

Unlike the Sequential model, you must create and define a standalone **Input** layer that specifies the shape of input data. The input layer takes a shape argument that is a tuple that indicates the dimensionality of the input data. When input data is one-dimensional, such as for a MLP, the shape must explicitly leave room for the shape of the mini-batch size used when splitting the data when training the network. Therefore, the shape tuple is always defined with a hanging last dimension when the input is one-dimensional (2,)

The layers in the model are connected **pairwise**. This is done by specifying where the input comes from when defining each new layer. A callable syntax is used (see next slide), such that after the layer is created, the layer from which the input to the current layer comes from is specified.

A **Model** is declared with its inputs and its outputs that are both tensors defining the inputs and outputs of the computational graph.

Train & eval done as usual

```
loss_test, metric_test = model2.evaluate(X_test, Y_test)
print('Test loss:', loss_test)
print('Test accuracy:', metric_test)
```

```
10000/10000 [==============================] - 0s 18us/step
Test loss: 0.07016694105217758
Test accuracy: 0.9801
```

```
H = 300                 # number of neurons
D = X_train.shape[1]    # dimension of input - 784 for MNIST

#Keras sequential model
model1 = Sequential()
model1.add(Dense(H, input_shape=(D,), activation='relu'))
model1.add(Dense(n_classes, activation='softmax'))
model1.summary()

#Keras functional API
visible = Input(shape=(D,)) # func api, input is declared
hidden1 = Dense(H, activation='relu')(visible)
output = Dense(n_classes, activation='softmax')(hidden1)
model2 = Model(inputs=visible, outputs=output)
model2.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_23 (Dense)             (None, 300)               235500
_________________________________________________________________
dense_24 (Dense)             (None, 10)                3010
=================================================================
Total params: 238,510
Trainable params: 238,510
Non-trainable params: 0
_________________________________________________________________
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_4 (InputLayer)         (None, 784)               0
_________________________________________________________________
dense_25 (Dense)             (None, 300)               235500
_________________________________________________________________
dense_26 (Dense)             (None, 10)                3010
=================================================================
Total params: 238,510
Trainable params: 238,510
Non-trainable params: 0
```

Keras functional API - Python callable syntax

- Keras has opted to use a callable syntax in Python for the inner layers of the computational graph
  - It might be confusing at first so here are some explanations

- Objects are **callable** when the class definition includes a `__call__` method. It allows to “call” the object as we would “call” a method with the syntax `obj()`.

- See example next page.

Python callable syntax - example

Definition of the class `Adder` with a constructor that takes optionally an initial value, stored in class member `memory`

Definition of the method `__call__` which is here adding some value `x` to `memory`.

Here the object `add1`, instance of class `Adder` is now "callable" with syntax `add1(value)`.

The constructor returns the object which is directly callable. Here

```
add2 = Adder(1)(2)
```

is equivalent to

```
add2 = Adder(1)
add2(2)
```

```
class Adder:

    def __init__(self, x = 0):
        self.__memory = x

    def get(self):
        return self.__memory

    def __call__(self, x):
        self.__memory += x
        return self
```

```
add1 = Adder(1)
add1(2)
add1(3)
add1(4)
print(add1.get())
```

```
10
```

```
add2 = Adder(1)(2)
add2(3)
add2(4)
print(add2.get())
```

```
10
```

```
add3 = Adder(1)(2)(3)(4)
print(add3.get())
```

```
10
```

Keras functional API - CNN

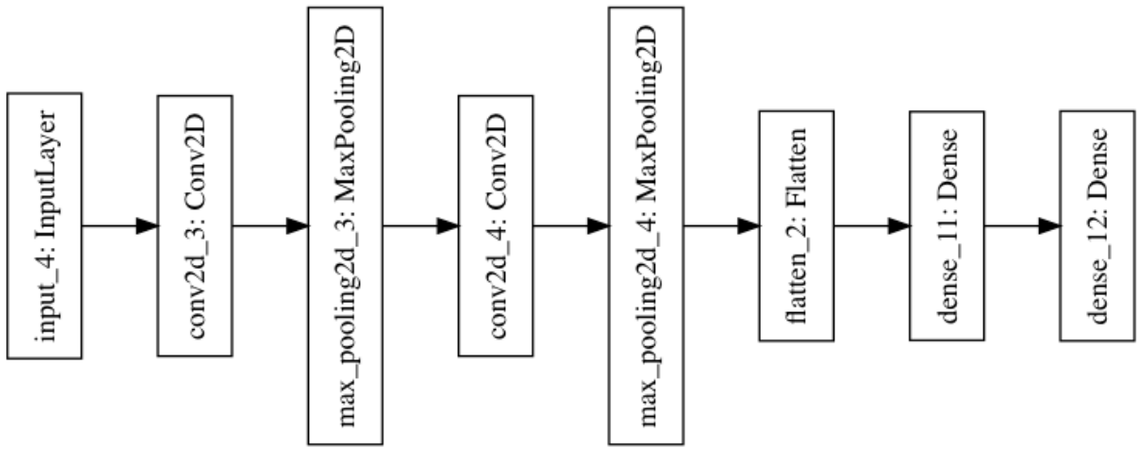
The model receives 28×28 images as input with only one channel (e.g. grey level MNIST data).

We then define two CONV-POOL layers as feature extractors.

We then have a Flatten layer followed by a fully connected layer to interpret the features and fully connected output layer with a softmax activation for ten-class predictions.

The model is then defined by specifying the inputs and the outputs.

```
# CNN - Keras functional API
from keras.utils import plot_model
from keras.models import Model
from keras.layers import Input
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers.convolutional import Conv2D
from keras.layers.pooling import MaxPooling2D

visible = Input(shape=(28,28,1))
conv1 = Conv2D(32, kernel_size=3, activation='relu')(visible)
pool1 = MaxPooling2D(pool_size=(2, 2))(conv1)
conv2 = Conv2D(32, kernel_size=3, activation='relu')(pool1)
pool2 = MaxPooling2D(pool_size=(2, 2))(conv2)
flat = Flatten()(pool2)
hidden1 = Dense(100, activation='relu')(flat)
output = Dense(10, activation='softmax')(hidden1)
model3 = Model(inputs=visible, outputs=output)
# summarize layers
print(model3.summary())
# plot graph
plot_model(model3, to_file='convolutional_neural_network.png')
```

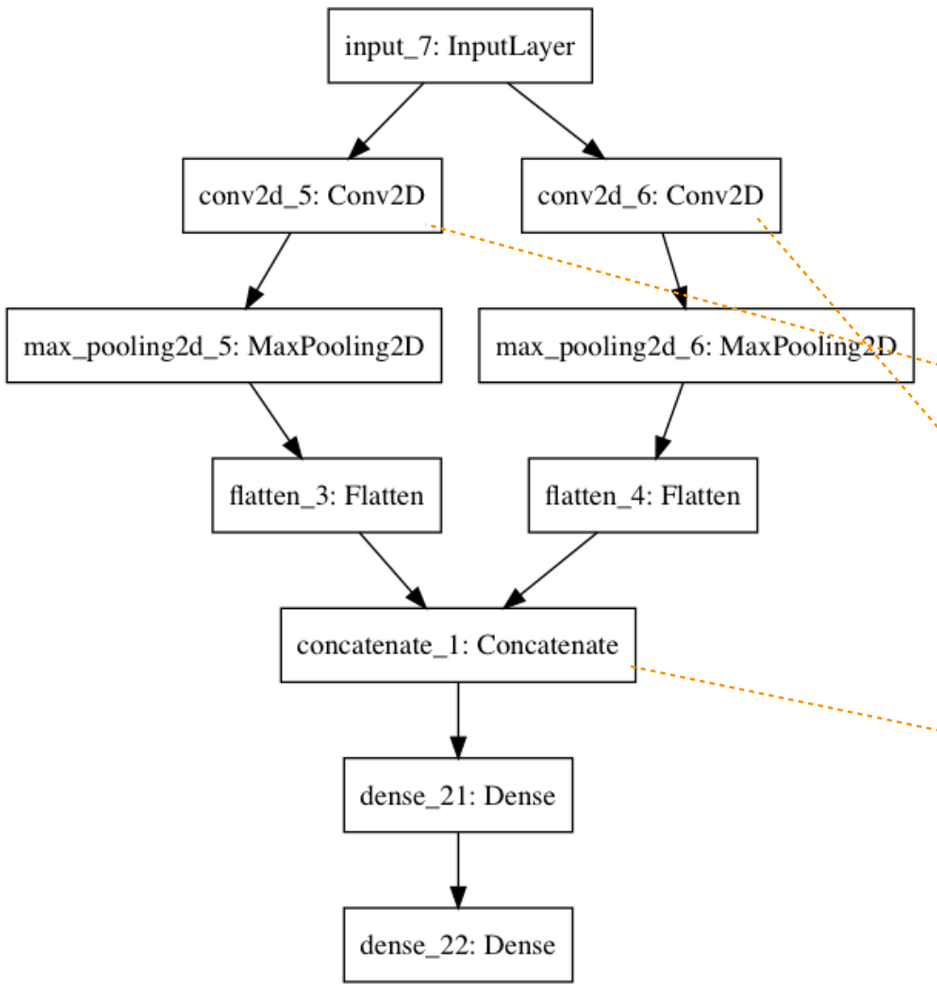


The plot_model function allows to generate an image representing the layers. It relies on a GraphViz tool that can be tricky to install: https://www.graphviz.org

```
loss_test, metric_test = model3.evaluate(X_test, Y_test)
print('Test loss:', loss_test)
print('Test accuracy:', metric_test)
```

```
10000/10000 [==============================] - 1s 88us/step
Test loss: 0.03386159673221523
Test accuracy: 0.9898
```

Keras functional API - CNN multiple path

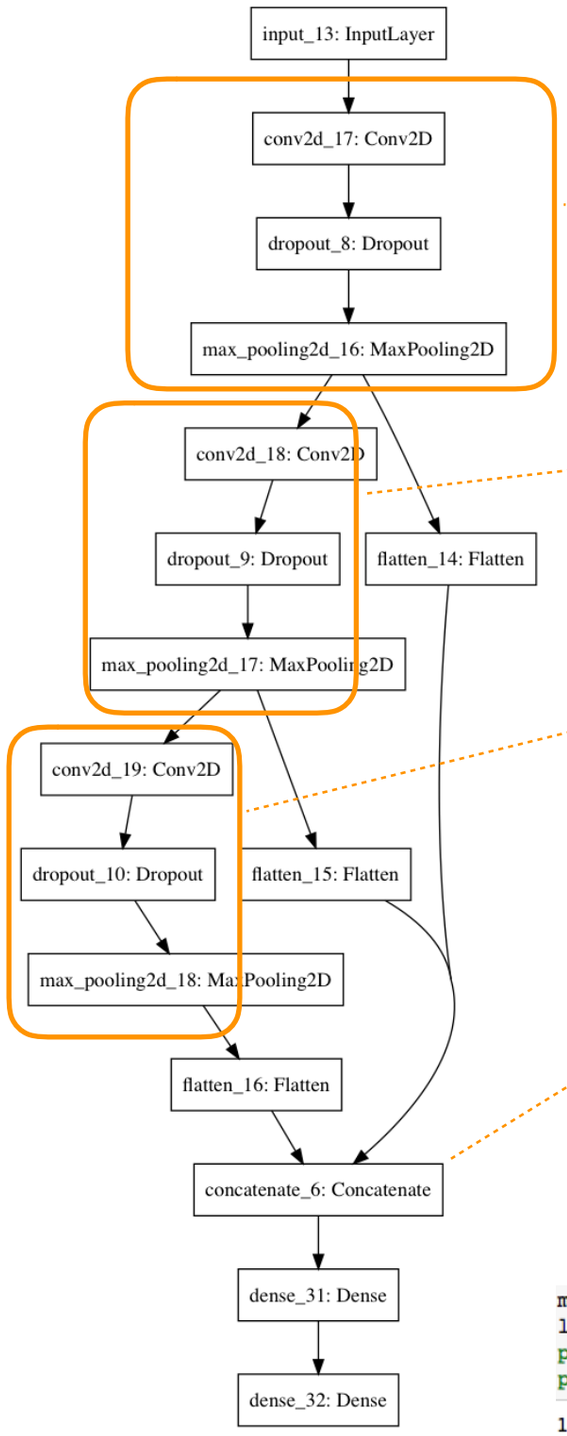



```
# Shared Input Layer
from keras.utils import plot_model
from keras.models import Model
from keras.layers import Input
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers.convolutional import Conv2D
from keras.layers.pooling import MaxPooling2D
from keras.layers.merge import concatenate

# input layer
visible = Input(shape=(28,28,1))
# first feature extractor
conv1 = Conv2D(32, kernel_size=3, activation='relu')(visible)
pool1 = MaxPooling2D(pool_size=(2, 2))(conv1)
flat1 = Flatten()(pool1)
# second feature extractor
conv2 = Conv2D(32, kernel_size=5, activation='relu')(visible)
pool2 = MaxPooling2D(pool_size=(2, 2))(conv2)
flat2 = Flatten()(pool2)
# merge feature extractors
merge = concatenate([flat1, flat2])
# interpretation layer
hidden1 = Dense(100, activation='relu')(merge)
# prediction output
output = Dense(10, activation='softmax')(hidden1)
model4 = Model(inputs=visible, outputs=output)
# summarize layers
print(model4.summary())
# plot graph
plot_model(model4, to_file='shared_input_layer.png')
```

```
loss_test, metric_test = model4.evaluate(X_test, Y_test)
print('Test loss:', loss_test)
print('Test accuracy:', metric_test)
```

```
10000/10000 [==============================] - 2s 171us/step
Test loss: 0.04740242167646156
Test accuracy: 0.9888
```

Keras functional API - CNN multiple features

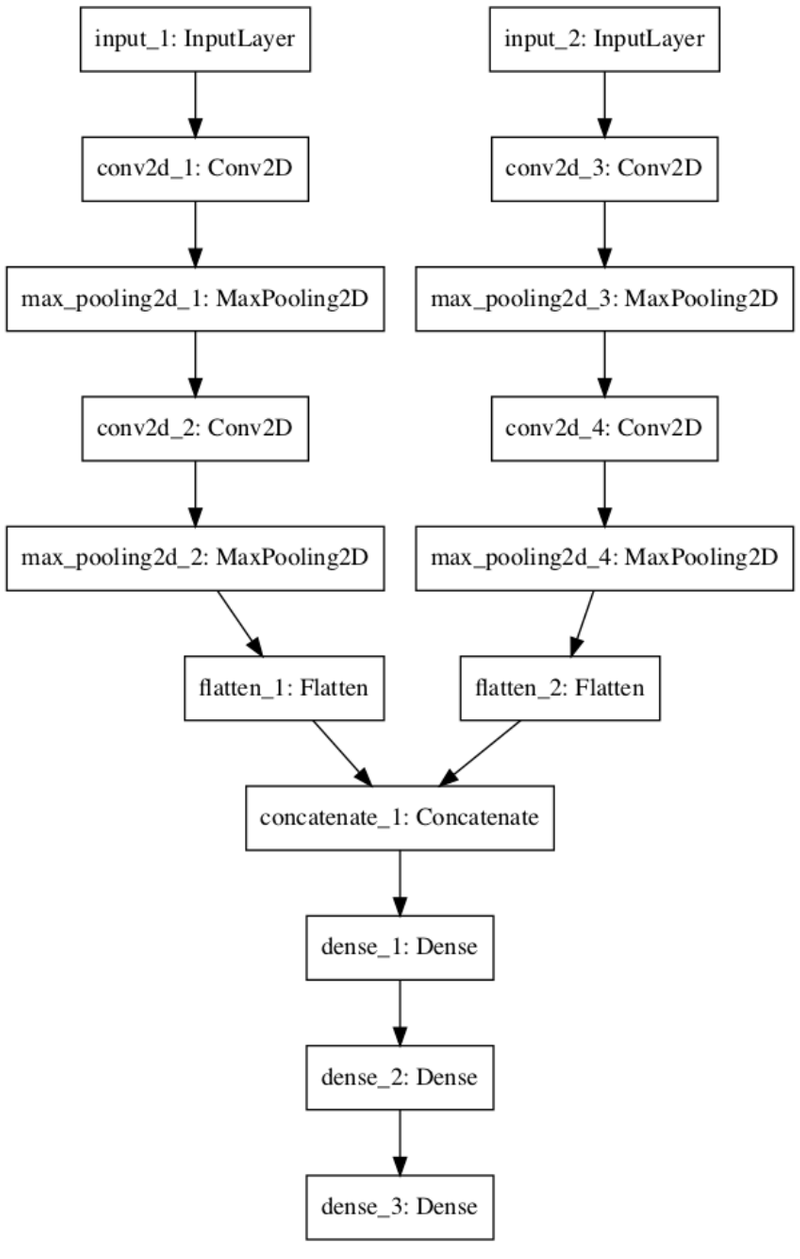



```
# input layer
visible = Input(shape=(28,28,1))
# first feature extractor
conv1 = Conv2D(32, kernel_size=3, activation='relu')(visible)
drop1 = Dropout(0.2)(conv1)
pool1 = MaxPooling2D(pool_size=(2, 2))(drop1)
flat1 = Flatten()(pool1)
# second feature extractor
conv2 = Conv2D(32, kernel_size=3, activation='relu')(pool1)
drop2 = Dropout(0.2)(conv2)
pool2 = MaxPooling2D(pool_size=(2, 2))(drop2)
flat2 = Flatten()(pool2)
# third feature extractor
conv3 = Conv2D(32, kernel_size=3, activation='relu')(pool2)
drop3 = Dropout(0.2)(conv3)
pool3 = MaxPooling2D(pool_size=(2, 2))(drop3)
flat3 = Flatten()(pool3)
# merge feature extractors
merge = concatenate([flat1, flat2, flat3])
# interpretation layer
hidden1 = Dense(100, activation='relu')(merge)
# prediction output
output = Dense(10, activation='softmax')(hidden1)
model5 = Model(inputs=visible, outputs=output)
# summarize layers
print(model5.summary())
# plot graph
plot_model(model5, to_file='shared_input_layer_multi_feat.png')
```

```
model5.load_weights(filepath = 'model-008.h5')
loss_test, metric_test = model5.evaluate(X_test, Y_test)
print('Test loss:', loss_test)
print('Test accuracy:', metric_test)
```

```
10000/10000 [==============================] - 1s 133us/step
Test loss: 0.029344594429839343
Test accuracy: 0.9911
```

Keras functional API - multiple inputs



- Such graphs are useful to merge different modalities
  - multiple images of the same object, e.g. extracted from movie
  - stereo speech recording
  - different parameters of the acquisition device

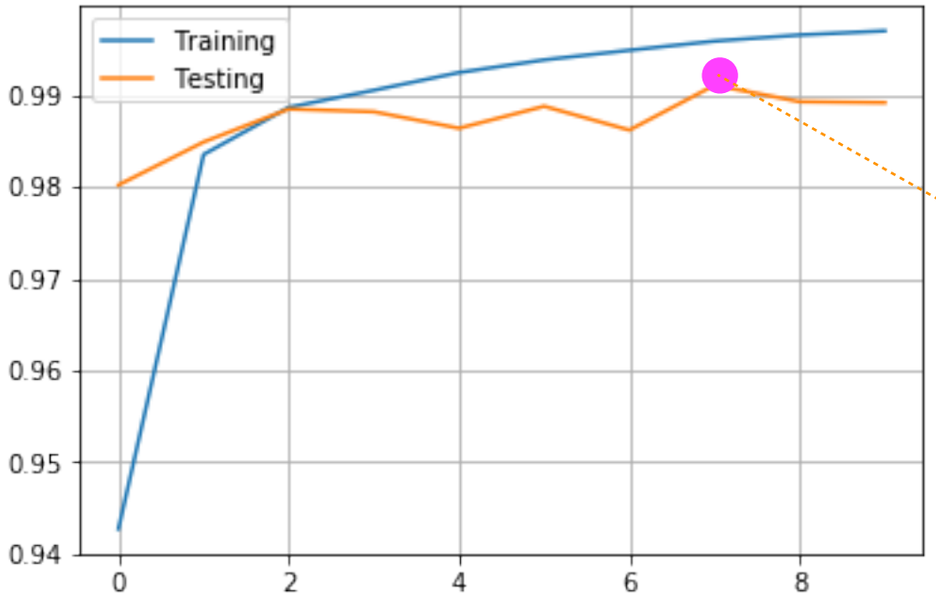
Keras functional API - callbacks

- https://keras.io/callbacks/
- Callback are functions to be applied at given stages of the training procedure. You can use callbacks to get a view on internal states and statistics of the model during training.
- Some are automatically applied in the fit() method such as BaseLogger and History.
- Below an example of ModelCheckpoint() that allows to save intermediary models during training

In ModelCheckPoint(), we define the filename, what to monitor and tell the callback to save only the best model determined automatically according to the monitored value - in this case the accuracy on the test set.

Declare callbacks in fit().

```
B = 128
E = 10
checkpoint = ModelCheckpoint('model-{epoch:03d}.h5', verbose=1,
                             monitor='val_acc',save_best_only=True,
                             mode='auto')
model5.compile(loss='categorical_crossentropy', optimizer='rmsprop',
               metrics=['accuracy'])
log = model5.fit(X_train, Y_train, batch_size=B, epochs=E,
                 verbose=1, validation_data=(X_test, Y_test),
                 callbacks=[checkpoint])
```



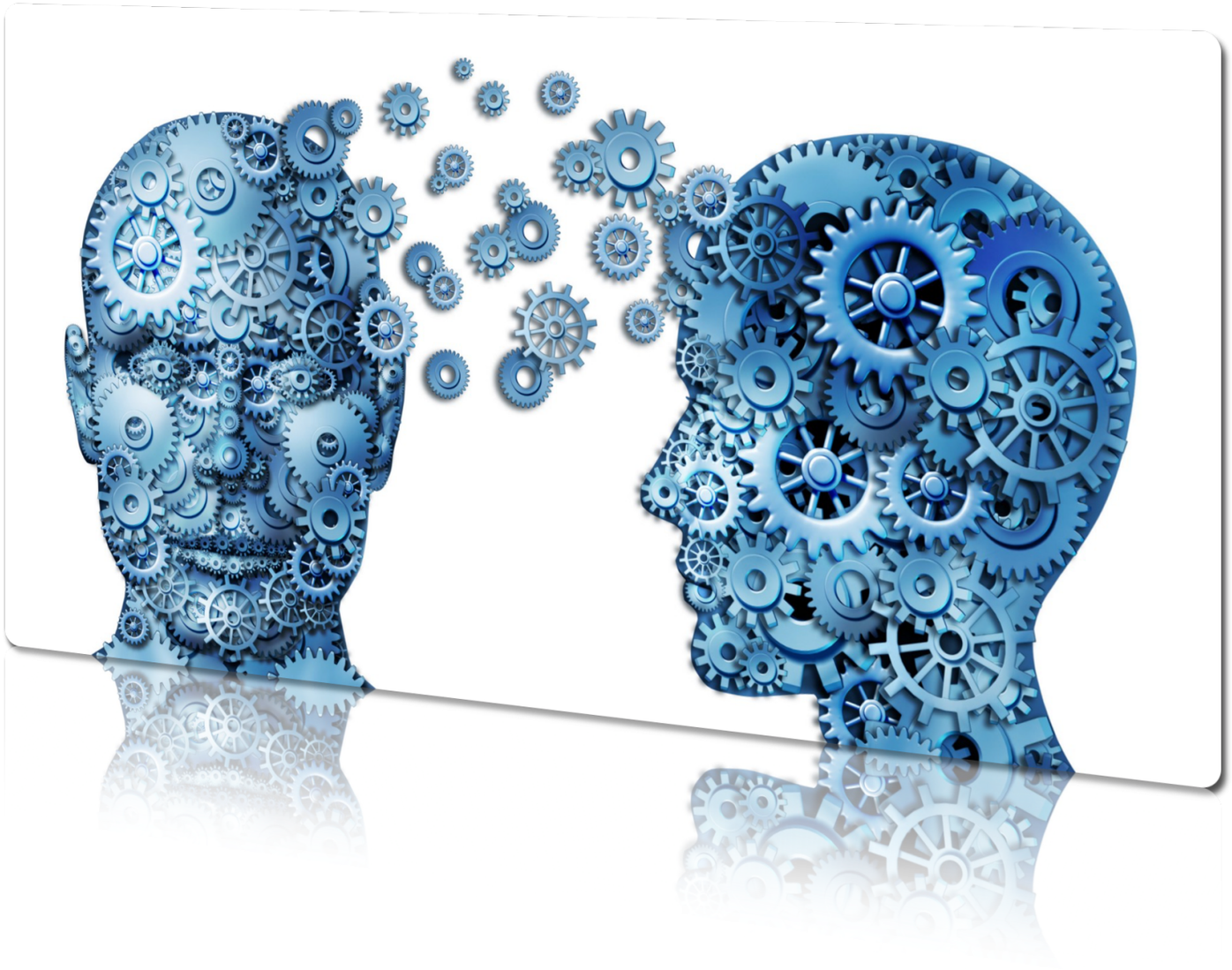

```
model5.load_weights(filepath = 'model-008.h5')
loss_test, metric_test = model5.evaluate(X_test, Y_test)
print('Test loss:', loss_test)
print('Test accuracy:', metric_test)
```

```
10000/10000 [==============================] - 1s 133us/step
Test loss: 0.029344594429839343
Test accuracy: 0.9911
```

Keras functional API - best practices

- **Use Consistent Variable Names**. Use same variable name for input (visible) and output layers (output), hidden layers (hidden1, hidden2). It will help to connect things together correctly.
- **Review Layer Summary**. Always print the model summary and review the layer outputs to ensure that the model was connected together as you expected.
- **Review Graph Plots**. Create a plot of the model graph and review it to ensure that everything was put together as you intended.
- **Name the layers**. You can assign names to layers that are used when reviewing summaries and plots of the model graph. For example: Dense(1, name='hidden1').
- **Use Callbacks.** You should use callbacks to save the (best) models from epochs to epochs as a safety against interrupt and overfitting.

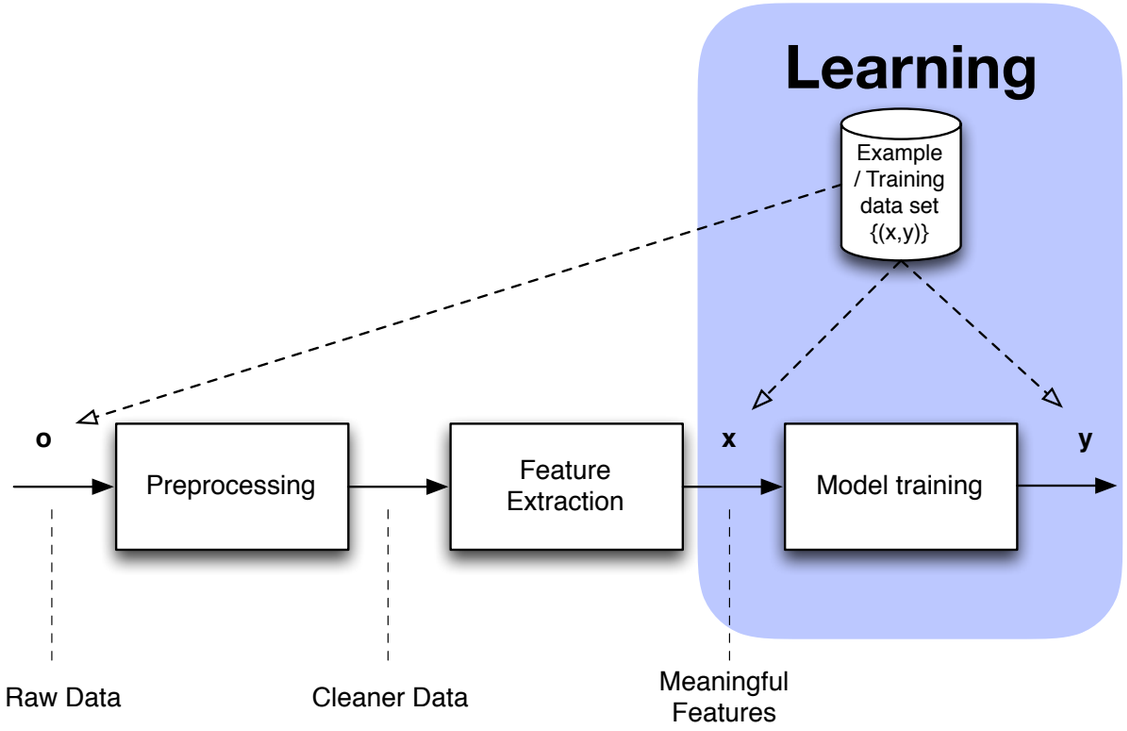
Transfer learning

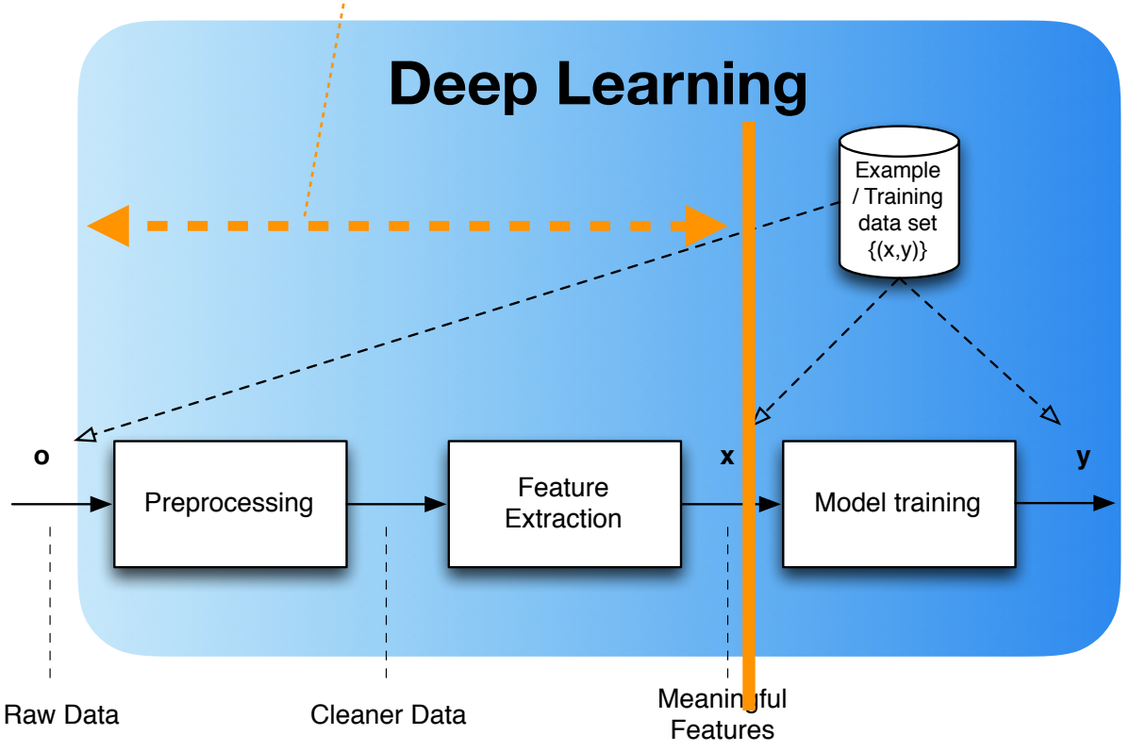


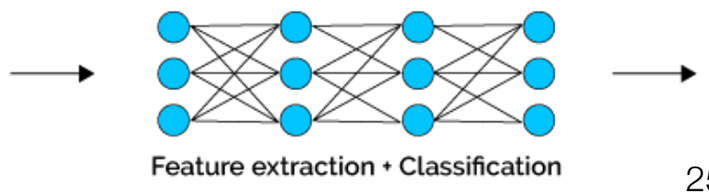
RECAP - Convolution layers are extracting features

- The filter parameters are learnt through the training process. The whole system is learning not only to classify but also to extract features.

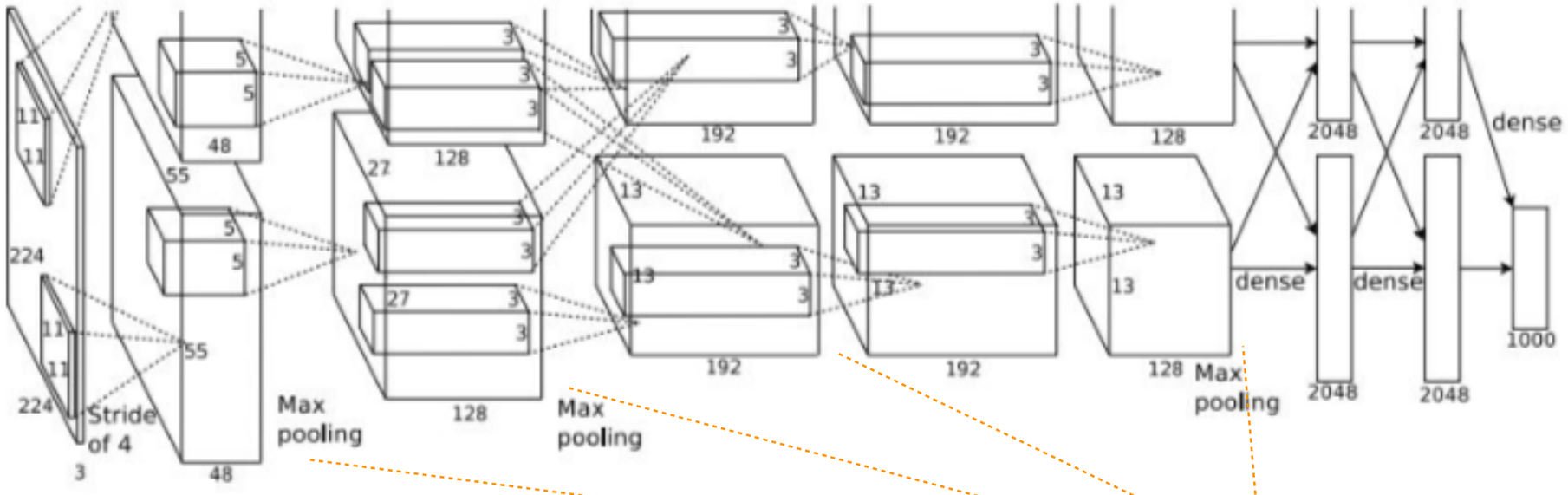
Transfer-learning:
attempt to re-use this part







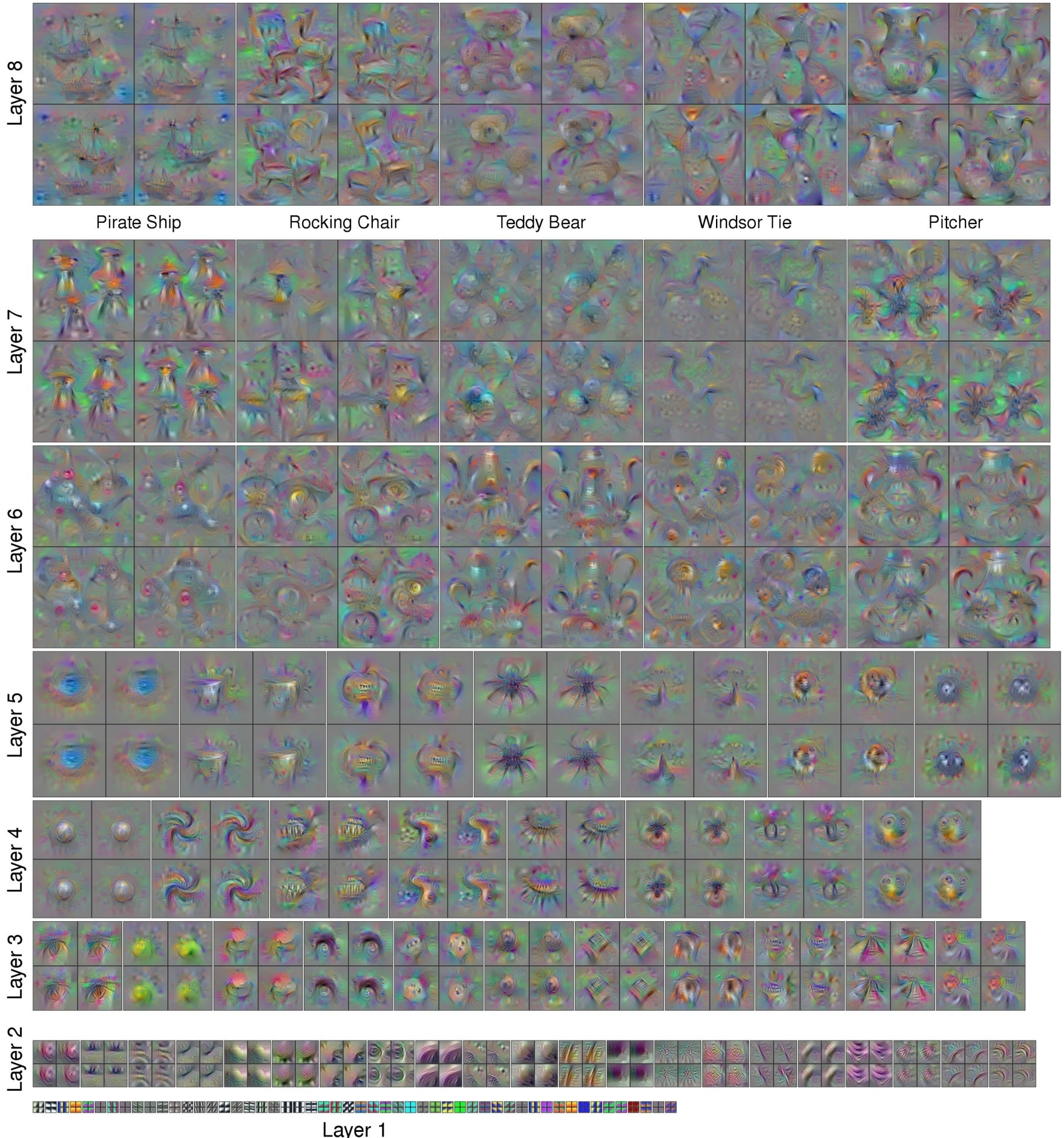
RECAP - A hierarchy of features



- Above is the AlexNet architecture [2012]
- The deeper the network:
  - The bigger are the number of filters in the conv layers
  - The smaller are the activation maps due to max pooling
- Intuitively:
  - Early layers will extract lower-level features
  - Late layers will extract higher-level features

If trained on "**broad**" types of inputs, the layers of the network will act as a "**generic**" feature extractor

Source: http://yosinski.com/deepvis



Transfer learning - definition

- **Transfer learning** is about using knowledge learned from tasks for which a lot of labelled data is available in settings where only little labelled data is available.

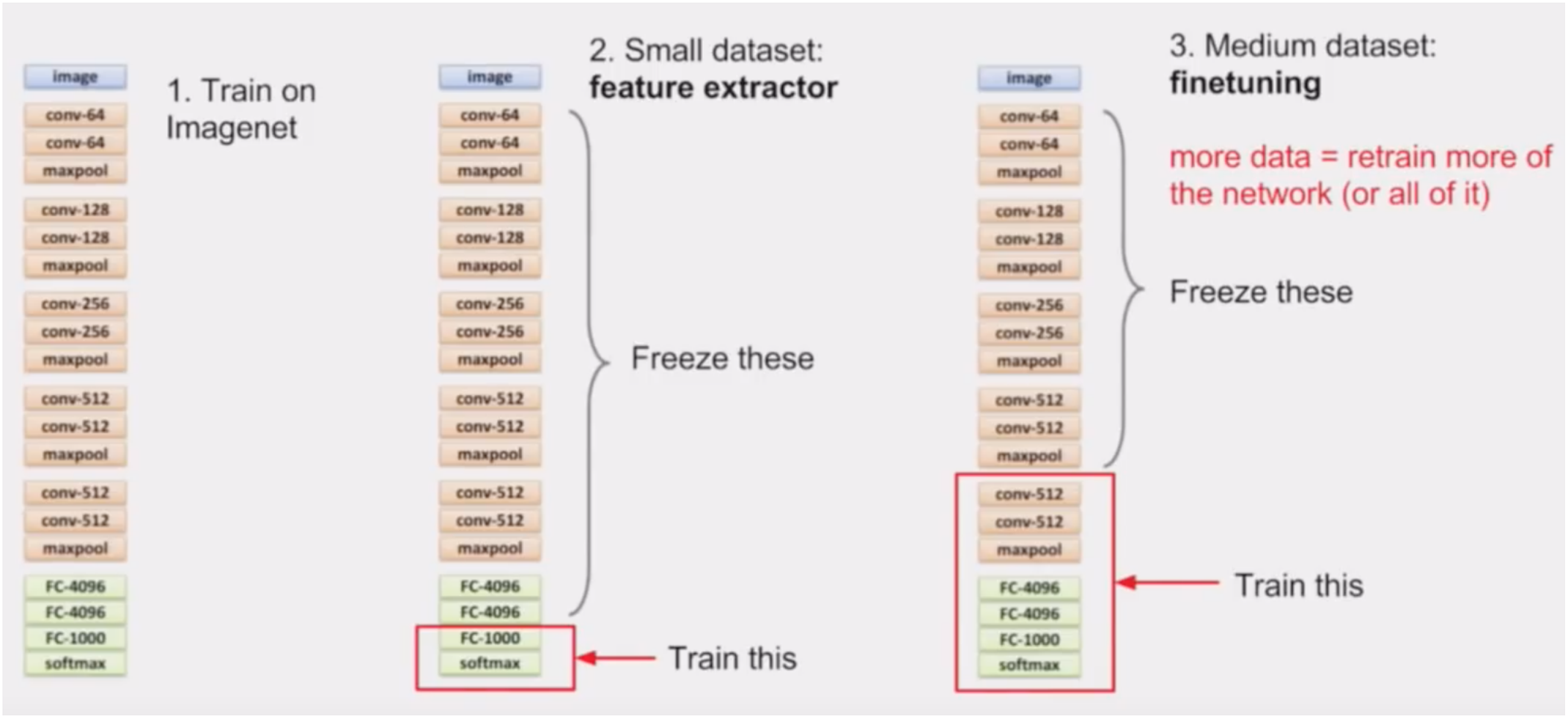
- More concretely: re-use the feature extraction part of pre-trained network on which years of engineering have been spent
- Traditional machine learning: generalise to unseen data based on patterns learned from the training data
- Transfer learning: kickstart the generalisation by starting from patterns learned for a different task

But not so different, let's say different in the same "domain" of tasks.

Transfer learning - discussions

- Transfer learning is about exporting knowledge into new environments
  - transfer learning is therefore a form of generalisation “across tasks”
- Many real-life problems do not have massive amounts of labelled data to train millions of parameters in deep models
  - Creating labelled data is expensive, so optimally leveraging existing datasets is key.
  - NLP Penn treebank dataset took 7 years to label for Part-of-Speech tagging
- There is a limit to the concept of transfer learning
  - If the data on which we transfer is too different, then it fails
  - “You should not trust a toddler that drives around in a toy car to be able to ride a Ferrari”

Transfer learning - best practices



Transfer learning - Keras

- Many pre-trained models for image recognition available in Keras
  - https://keras.io/applications/

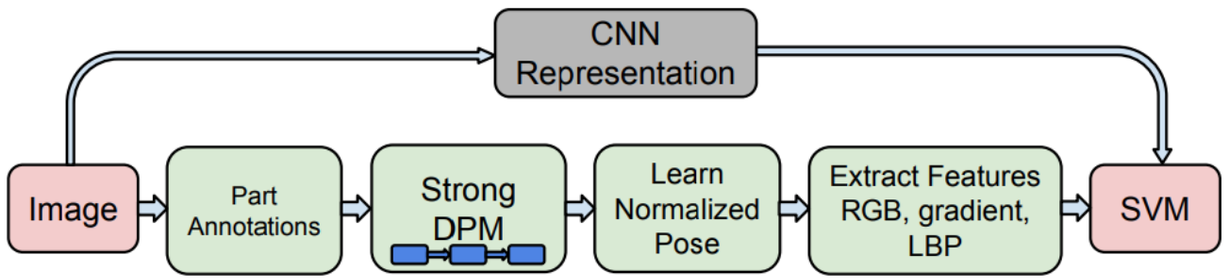
Trained and evaluated on ImageNet

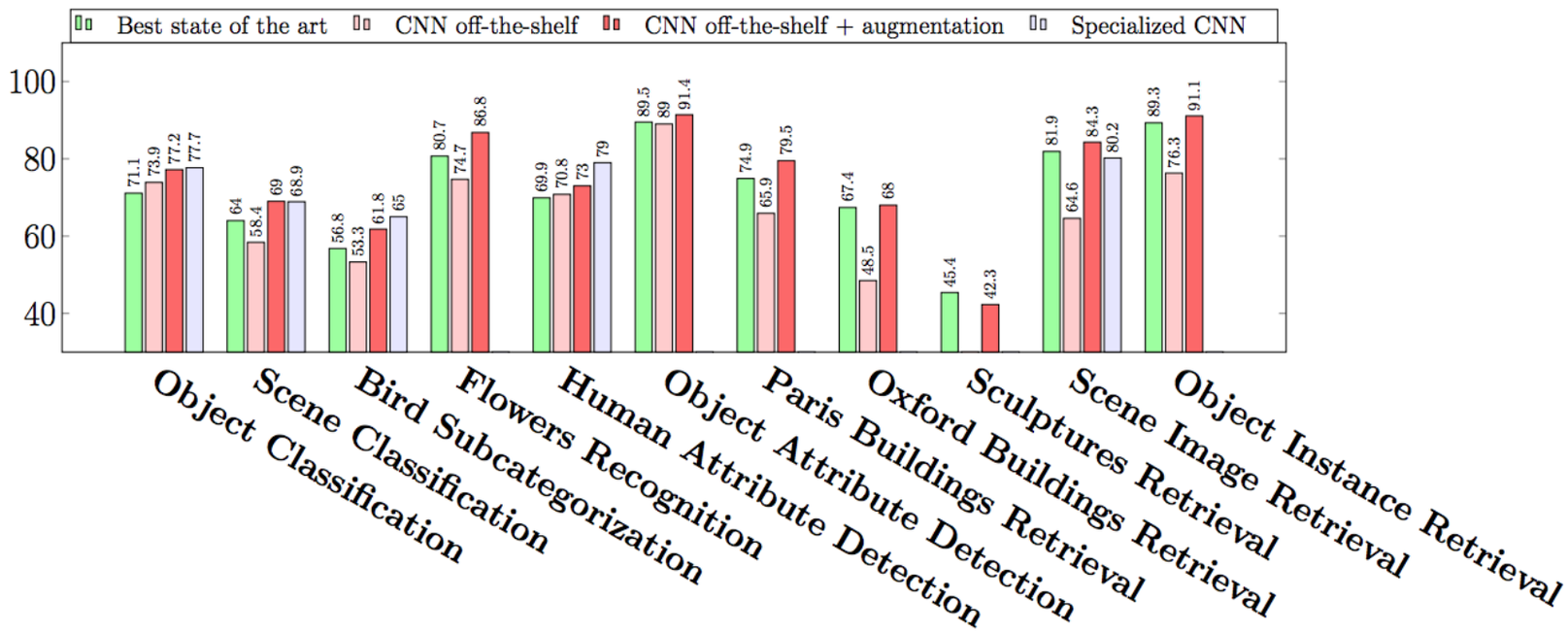
Model	Size	Top-1 Accuracy	Top-5 Accuracy	Parameters	Depth
Xception	88 MB	0.790	0.945	22,910,480	126
VGG16	528 MB	0.713	0.901	138,357,544	23
VGG19	549 MB	0.713	0.900	143,667,240	26
ResNet50	99 MB	0.749	0.921	25,636,712	168
InceptionV3	92 MB	0.779	0.937	23,851,784	159
InceptionResNetV2	215 MB	0.803	0.953	55,873,736	572
MobileNet	16 MB	0.704	0.895	4,253,864	88
MobileNetV2	14 MB	0.713	0.901	3,538,984	88
DenseNet121	33 MB	0.750	0.923	8,062,504	121
DenseNet169	57 MB	0.762	0.932	14,307,880	169
DenseNet201	80 MB	0.773	0.936	20,242,984	201
NASNetMobile	23 MB	0.744	0.919	5,326,716	-
NASNetLarge	343 MB	0.825	0.960	88,949,818	-

Transfer learning - beating state-of-the-art

- A. Razavian et al, "*CNN Features off-the-shelf: an Astounding Baseline for Recognition*", arXiv, 2014 https://arxiv.org/abs/1403.6382

- "Recent results indicate that the generic descriptors extracted from the convolutional neural networks are very powerful. This paper adds to the mounting evidence that this is indeed the case. We report on a series of experiments conducted for different recognition tasks using the publicly available code and model of the OverFeat network which was trained to perform object classification on ILSVRC13. We use features extracted from the OverFeat network as a generic image representation to tackle the diverse range of recognition tasks of object image classification, scene recognition, fine grained recognition, attribute detection and image retrieval applied to a diverse set of datasets. We selected these tasks and datasets as they gradually move further away from the original task and data the OverFeat network was trained to solve. Astonishingly, we report consistent superior results compared to the highly tuned state-of-the-art systems in all the visual classification tasks on various datasets."





Transfer learning in Keras - MobileNetV2

Import the MobileNetV2 pre-trained model from Keras

Then load the network architecture specifying the types of weights (here 'imaginet' weights).

Finally specify the input shape which should be equivalent to the one used for training this network.

```
import keras
from keras.datasets import cifar10
import numpy as np
from keras.applications.mobilenet_v2 import MobileNetV2
from keras.applications.mobilenet_v2 import preprocess_input
import scipy
from scipy import misc
import skimage
from skimage import transform
import os

# load the data
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
y_train = np.squeeze(y_train) # get (50000,) instead of (50000,1)
print('data loaded')

# load MobileNetV2 model + remove final classification layers
model = MobileNetV2(weights='imagenet', include_top=False,
                    input_shape=(224, 224,3))
print('model loaded')
```

https://arxiv.org/abs/1801.04381

Input	Operator	t	c	n	s
$224^2 \times 3$	conv2d	-	32	1	2
$112^2 \times 32$	bottleneck	1	16	1	1
$112^2 \times 16$	bottleneck	6	24	2	2
$56^2 \times 24$	bottleneck	6	32	3	2
$28^2 \times 32$	bottleneck	6	64	4	2
$14^2 \times 64$	bottleneck	6	96	3	1
$14^2 \times 96$	bottleneck	6	160	3	2
$7^2 \times 160$	bottleneck	6	320	1	1
$7^2 \times 320$	conv2d 1x1	-	1280	1	1
$7^2 \times 1280$	avgpool 7x7	-	-	1	-
$1 \times 1 \times 1280$	conv2d 1x1	-	k	-	

Remove the top layers - the one for the classification part. Keep the layers for the feature extraction. That could involve several layers in some cases.

Feature shape will be (50000, 7, 7, 1280) encoded as float32, so about 12.5GB !!!

Transfer learning in Keras - MobileNetV2

Due to the size to the time needed to get the features, we have here a strategy to store it on the disk. If the file exists, we simply reload it.

If file does not exist, we recompute the features:

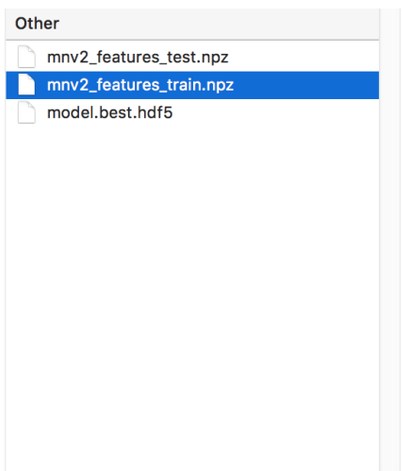
(1) Each CIFAR10 image is converted from 32x32x3 into an adequate input size of the network, i.e. 224x224x3

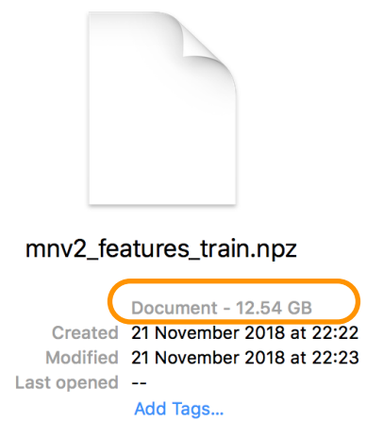
(2) We then need to pre-process the file, i.e. normalisations. This operation is dependent to the MobileNetV2 network and how images were preprocessed, so we call the preprocess_input() method provided with the network.

We can finally predict, i.e. compute the features. Attention this process takes time. Feature shape is (50000, 7, 7, 1280).

So in the end, one 32x32x3 image (encoded on 1 byte) using 3KB is transformed into a feature volume of 7x7x1280 (encoded on 4 bytes) using 251KB

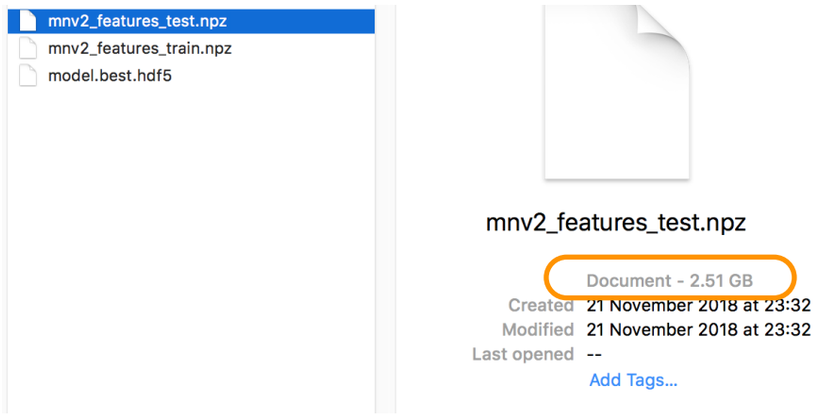
```
path = '/Users/henneber/Downloads/data/computed-features-cifar10/'

# obtain train features, read from file if already gotten
if os.path.exists(path + 'mnv2_features_train.npz'):
    print('features detected (train)')
    features = np.load(path + 'mnv2_features_train.npz')['features']
else:
    print('features file not detected (train)')
    print('calculating now ...')
    # first convert to correct size (upsize) and then to float32
    big_x_train = np.array([scipy.misc.imresize(x_train[i], (224, 224, 3))
                            for i in range(0, len(x_train))]).astype('float32')
    # pre-processing is specific to each network
    mnv2_input_train = preprocess_input(big_x_train)
    print('train data preprocessed')
    # predict and save features
    features = model.predict(mnv2_input_train, verbose=1)
    features = np.squeeze(features)
    # save features in an uncompressed archive
    np.savez(path + 'mnv2_features_train', features=features)
    print('train features saved (train)')

print('features ready (train)')
print(features.shape)
```





Transfer learning in Keras - MobileNetV2

Do the same thing for the test set.




```python
# obtain features (test)
if os.path.exists(path + 'mnv2_features_test.npz'):
    print('features detected (test)')
    features_test = np.load('mnv2_features_test.npz')['features_test']
else:
    print('features file not detected (test)')
    print('calculating now ...')
    # first convert to correct size (upsize) and then to float32
    big_x_test = np.array([scipy.misc.imresize(x_test[i], (224, 224, 3))
                           for i in range(0, len(x_test))]).astype('float32')
    # pre-processing is specific to each network
    mnv2_input_test = preprocess_input(big_x_test)
    # predict and save features (test)
    features_test = model.predict(mnv2_input_test, verbose=1)
    features_test = np.squeeze(features_test)
    np.savez(path + 'mnv2_features_test', features_test=features_test)
print('features saved (test)')
```

Transfer learning in Keras - MobileNetV2

You can now train a "simple" classifier on the features.

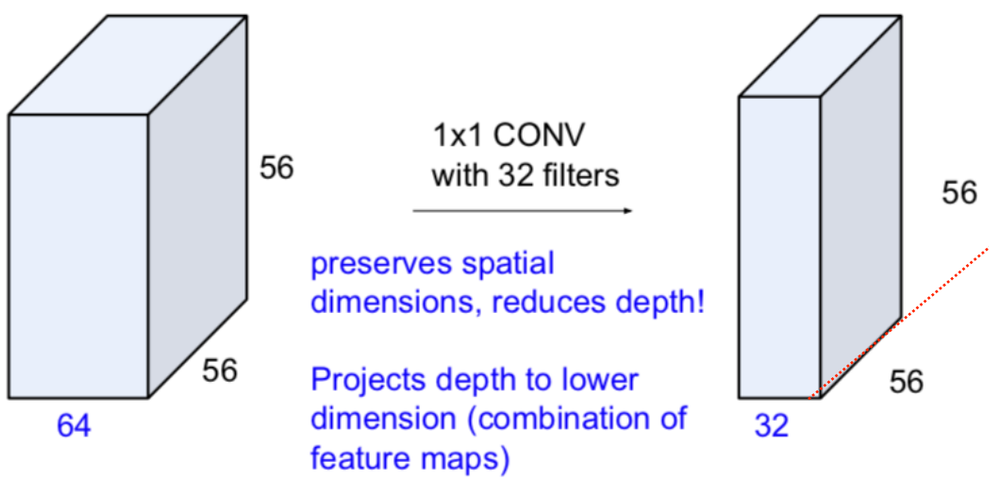
For very small dataset, you may want to use SVM that has good behaviour in such situations.

The feature size is a large stack of small (here 7x7) activation maps (here 1280), i.e. 62'720 dims. You probably need to reduce the dimension first, for example using **bottleneck layer.**

```
mlp.add(Conv2D(filters=128, kernel_size=1, strides=1,
               input_shape=(7, 7, 1280)))
```

To reduce the number of operation, 1x1 conv are used. So-called **bottleneck layers**.

Reminder: 1x1 convolutions



For example, using a 100 neuron MLP on CIFAR10: 83.2% accuracy

```
# evaluate test accuracy
X_test = features_test.reshape(10000, D)
score = mlp.evaluate(features_test, y_test, verbose=1)
accuracy = 100*score[1]

# print test accuracy
print('Test accuracy: %.4f%%' % accuracy)
```

```
10000/10000 [==============================] - 2s 246us/step
Test accuracy: 83.2000%
```

Extra note on transfer learning - Bayes law

- **Bayes rule:** Elect as winning category the one having the largest *a posteriori* probability. Doing so we are guaranteed to maximise the accuracy.

likelihood
"probability of observing x given class j"

a priori probability
"probability of class j"

$$P(C_k|\mathbf{x}) = \frac{p(\mathbf{x}|C_k)P(C_k)}{p(\mathbf{x})}$$

a posteriori probability
"probability of class j given observation x"

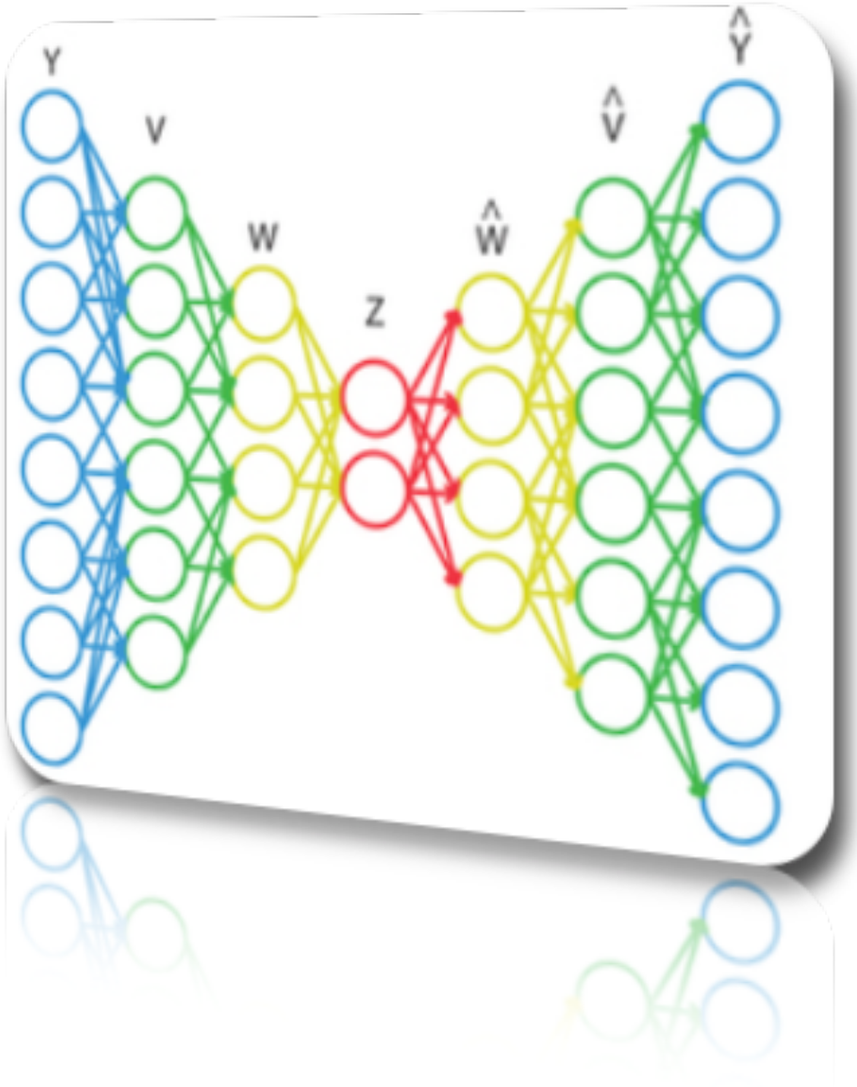
evidence = probability of x
"…unconditional to any class…"

- Usually, pre-trained networks are trained on "balanced" data sets, i.e. all the P(Ck) are equal to 1/K.
- If your "deployment" environment has different priors than the train/test data, you may need to re-scale your a posteriori probabilities using the new priors.

Auto-encoders

Definition

Loss



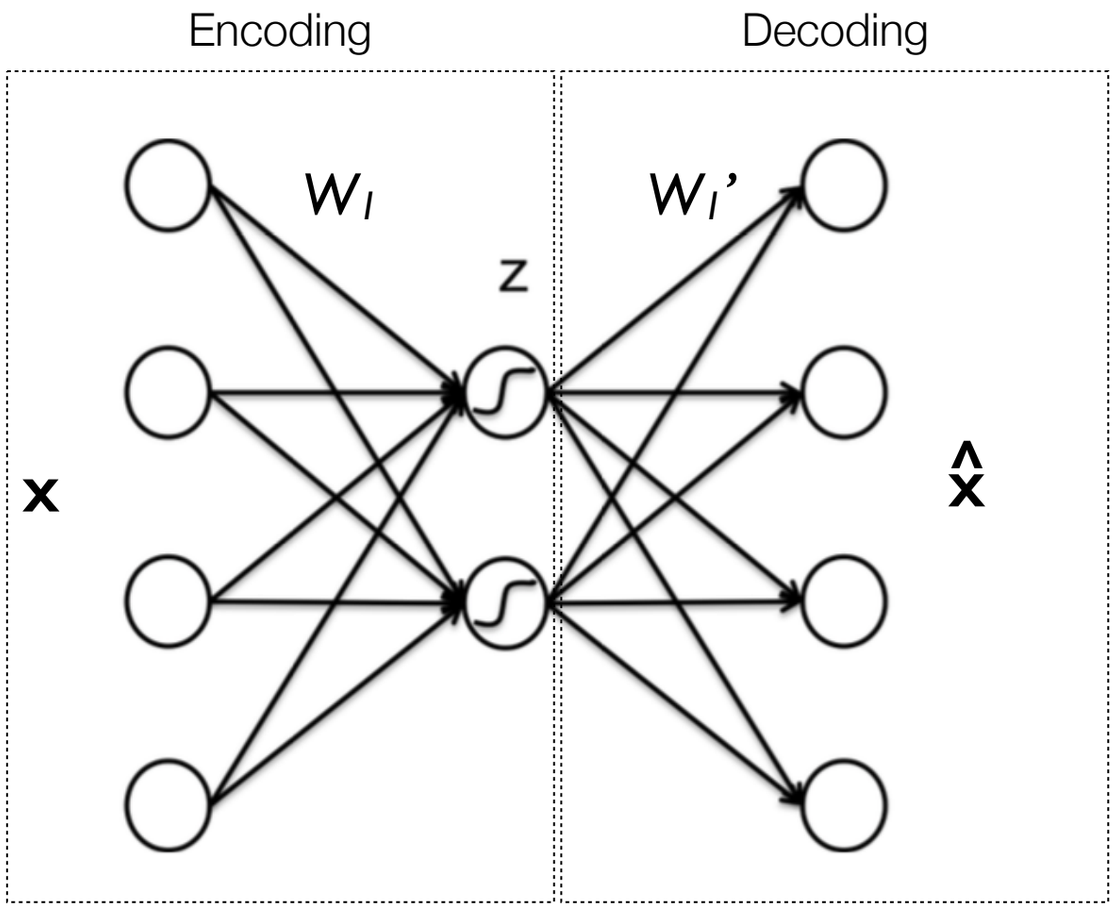
Source: Max Bernier, Konrad P. Kording, "Deep networks for motor control functions", frontiers in Computational Neuroscience, March 2015, Vol. 9

Definition

An **autoencoder** is a neural network trained to reproduce its input and able to discover “structures” and “efficient codings” of the input space.

- The simplest form of an autoencoder is a feedforward, non-recurrent neural network similar to the MLP.
- In such network, the output layer has the same number of nodes as the input layer.
- The mapping function $h_\theta(\mathbf{x})$ is trained to reconstruct its own inputs instead of predicting a target value.

$$\widehat{\mathbf{x}} = h_\theta(\mathbf{x})$$



$$\hat{\mathbf{x}} = h_\theta(\mathbf{x})$$

- In order to learn something else than the unity function, the network is often composed in such a way that the number of nodes in the hidden layer is smaller than the number of nodes in the input and output layers, forming a so-called "**diabolo**" network.
- The principle is to "cut" the network into two pieces after training

Loss function

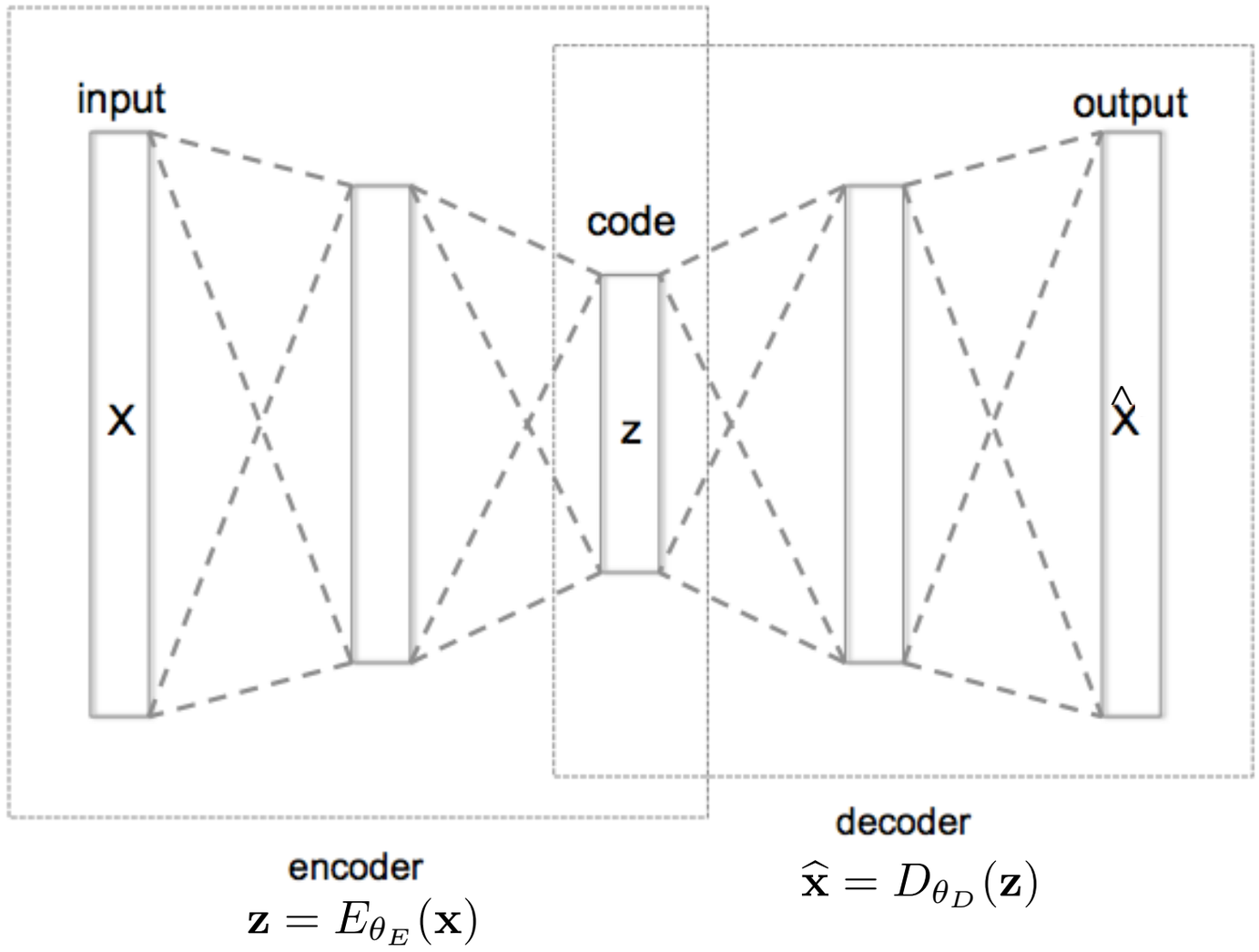
- Mean Square Error between input and reconstructed input

$$J(\theta) = J(\theta_E, \theta_D) = \frac{1}{2N} \sum_{n=1}^{N} (h_\theta(\mathbf{x}_n) - \mathbf{x}_n)^2$$

$$= \frac{1}{2N} \sum_{n=1}^{N} (\widehat{\mathbf{x}}_n - \mathbf{x}_n)^2$$

- Training as usual with one of the form of gradient descent
  - **Batch gradient descent**: use all N examples in each iteration
  - **Stochastic gradient descent**: use 1 example in each iteration
  - **Mini-batch gradient descent**: use b examples in each iteration

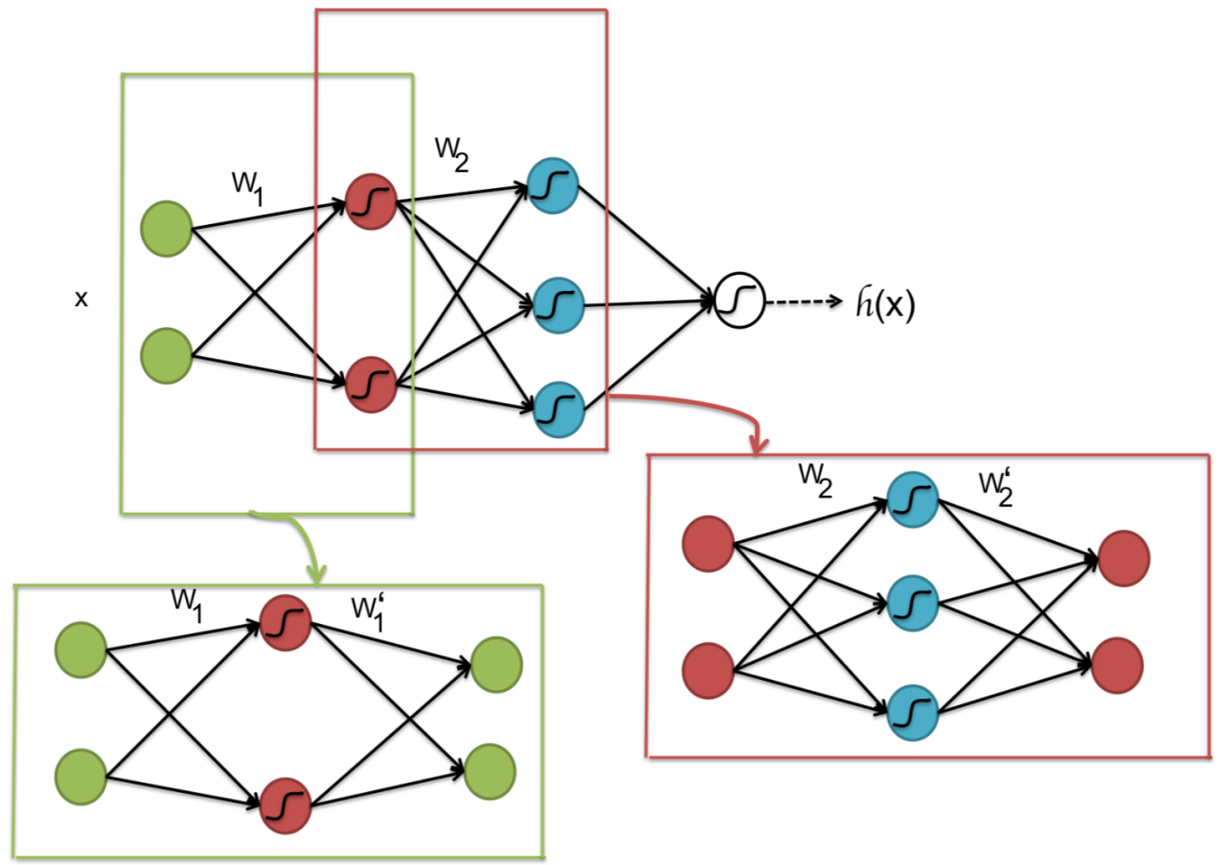
Stacked autoencoders - convolutional autoencoders



- The principle remains the same when
  - stacking dense layers
  - stacking convolution layers

Remark - layer by layer training

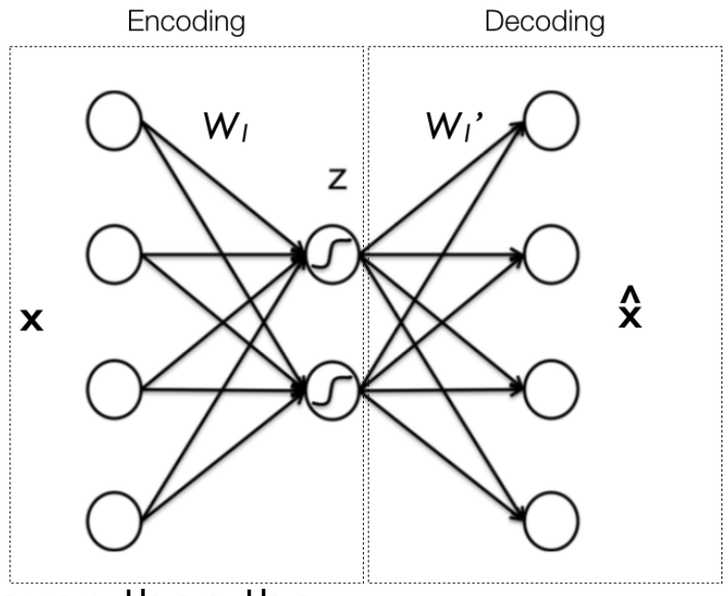
- Before the invention of robust regularisation and optimisation techniques, layer by layer training was used with stacked autoencoders:
  - To train the red neurons, we will train an autoencoder that has parameters W1 and W1′.
  - After this, we will use W1 to compute the outputs of the red neurons for all of our data
  - The parameters of the decoding process W1′ are then discarded.
  - The subsequent autoencoder uses the values for the red neurons as inputs, and trains an autoencoder to predict those values by adding a decoding layer with parameters W2′.
  - The parameters of the decoding process W2' are then discarded.
  - An so forth…
- It was a way "to go deeper"



From Quoc V. Le, Google Brain, "A Tutorial on Deep Learning. Part 2."

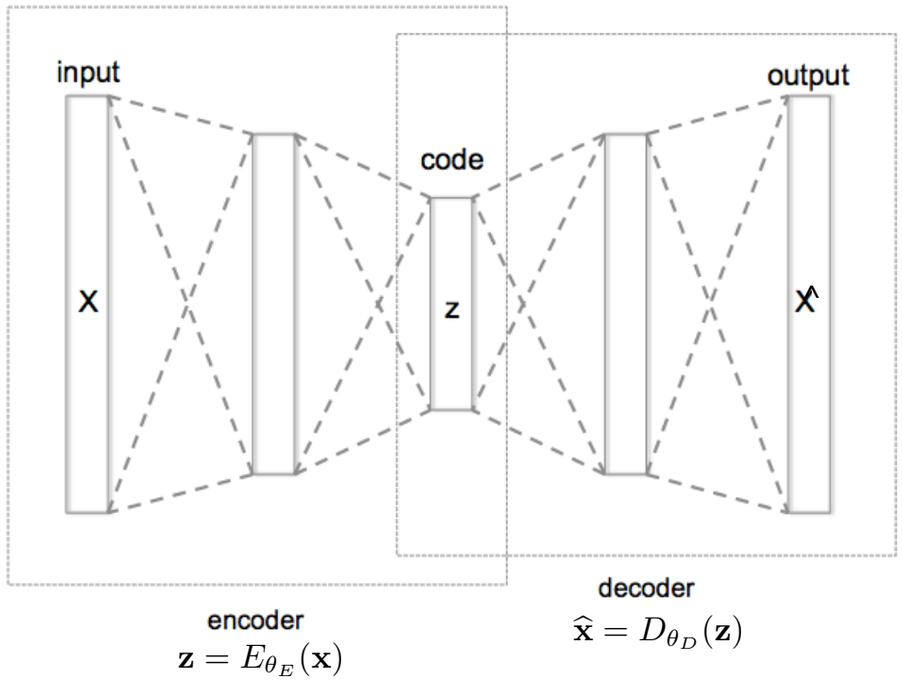
Usage 1: data compression / dimensionality reduction

- Principles:
  - Train an autoencoder on all your input data
  - Transmit the W_1' to your party
  - Encode: transmit the z values to your party instead of x
  - Decode: compute the reconstructed values $\hat{x}$
- Remarks:
  - if linear activations are used, or only a single sigmoid hidden layer, then the optimal solution to an autoencoder is related to principal component analysis (PCA)
  - chose dimension between your minimum and maximum bandwidth capacity
    - don't forget that you need to transfer the weights of the decoding part of your network
    - with acceptable loss for the decoding
      - the loss can be pre-computed



Usage 2: use the encoder to obtain features

- Principles:
  - Train a (conv)-autoencoder on your training data (including unlabelled data)
  - Use the encoding part to compute z
  - Use z as features to train a classifier (using the labelled part of your data)

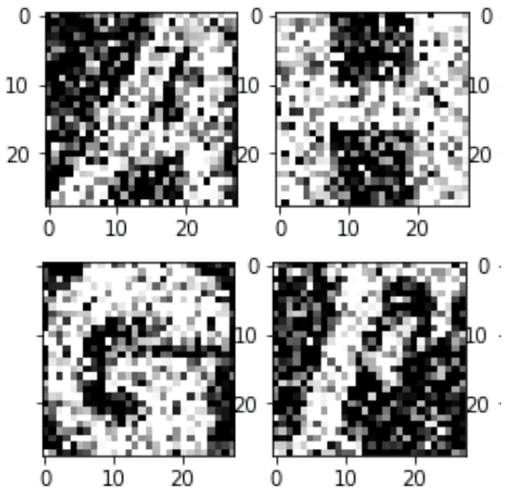


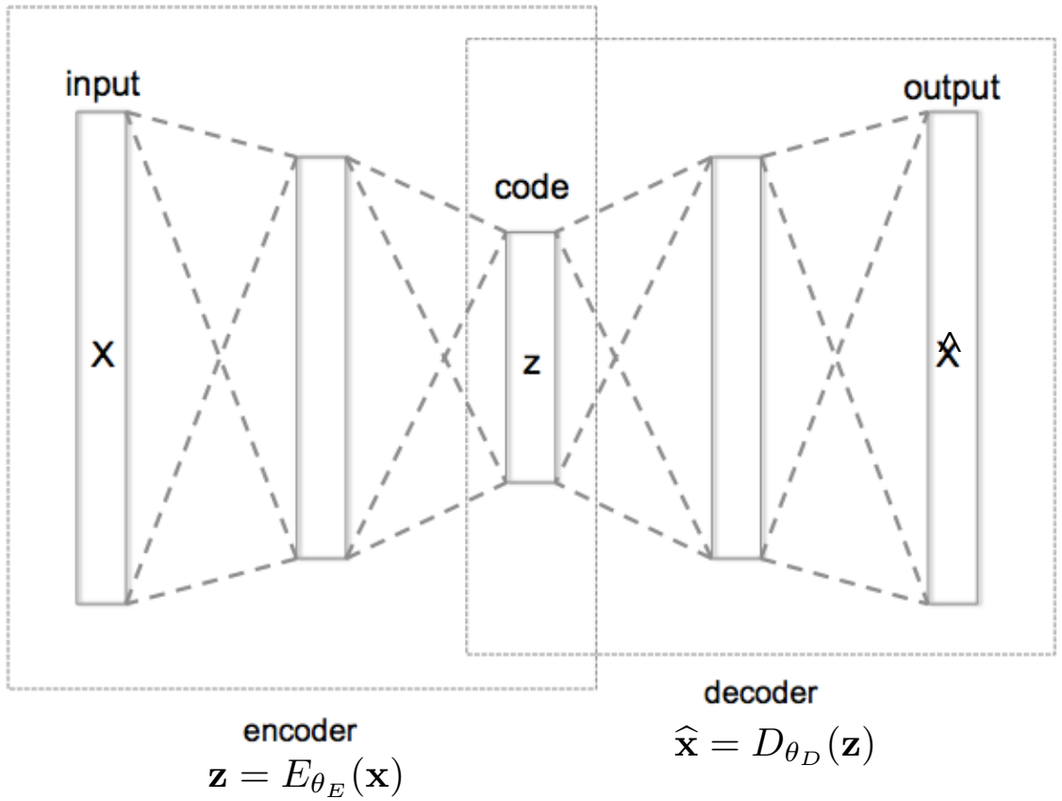
Transfer learning against autoencoder

- What do you expect about the quality of features obtained with a transfer learning approach vs an autoencoder approach?
- What are the advantage / disadvantage?

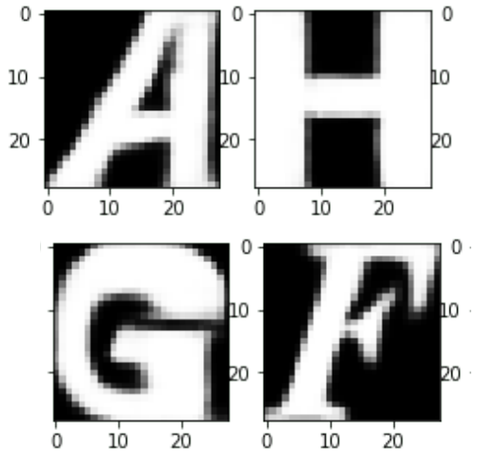
Usage 3: de-noising auto-encoders

Noisy image



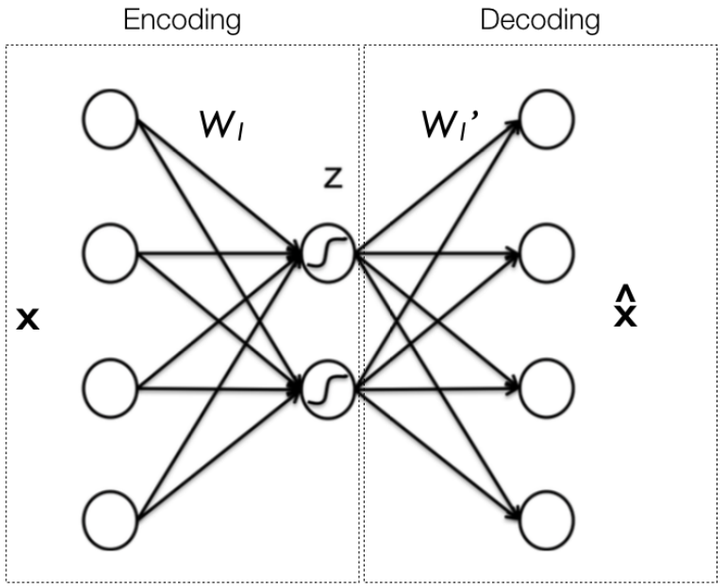


De-noised image

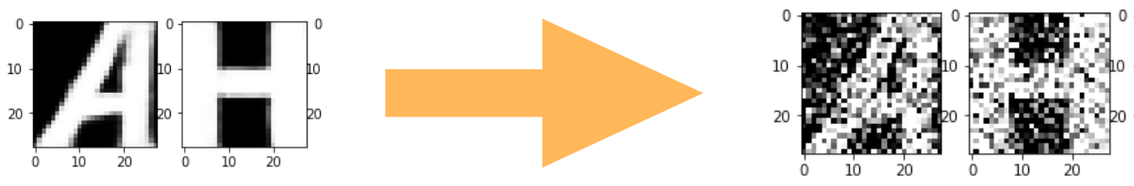


- Principles:
  - Training: add noise to your training data x and attempt to predict clean data
    - This will learn the network to “de-noise” the input data
  - Testing: input a noisy image and get a cleaner version
- If used to extract features
  - features show “robustness” to noise
  - generating artificial noise is also a form of data augmentation

Example in Keras - shallow dense



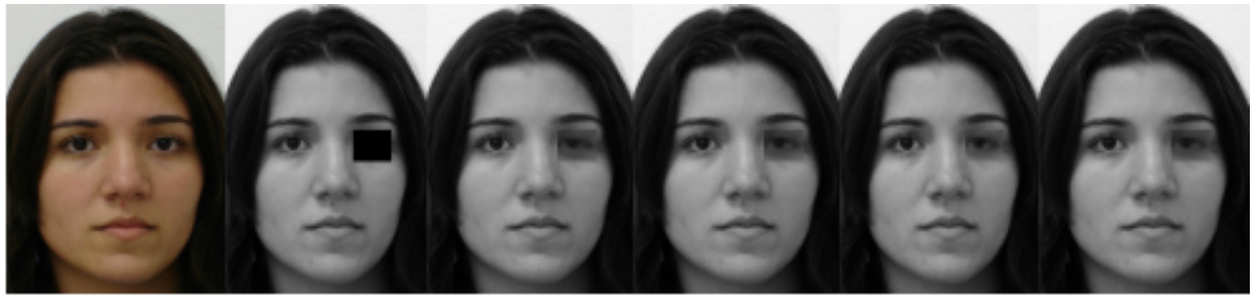

```
# size of our encoded representation
encoding_dim = 32

input_img = Input(shape=(784,))
encoded = Dense(encoding_dim, activation='relu')(input_img)
decoded = Dense(784, activation='sigmoid')(encoded)

autoencoder = Model(inputs=input_img, outputs=decoded)
encoder = Model(inputs=input_img, outputs=encoded)
# create a placeholder for an encoded (32-dimensional) input
encoded_input = Input(shape=(encoding_dim,))
# retrieve the last layer of the autoencoder model
decoder_layer = autoencoder.layers[-1]
# create the decoder model
decoder = Model(inputs=encoded_input,
                outputs=decoder_layer(encoded_input))
```



```
noise_factor = 0.4
X_train_noisy = X_train + noise_factor * np.random.normal(loc=0.0, scale=1.0, size=X_train.shape)
X_test_noisy = X_test + noise_factor * np.random.normal(loc=0.0, scale=1.0, size=X_test.shape)
X_train_noisy = np.clip(X_train_noisy, 0., 1.)
X_test_noisy = np.clip(X_test_noisy, 0., 1.)
```

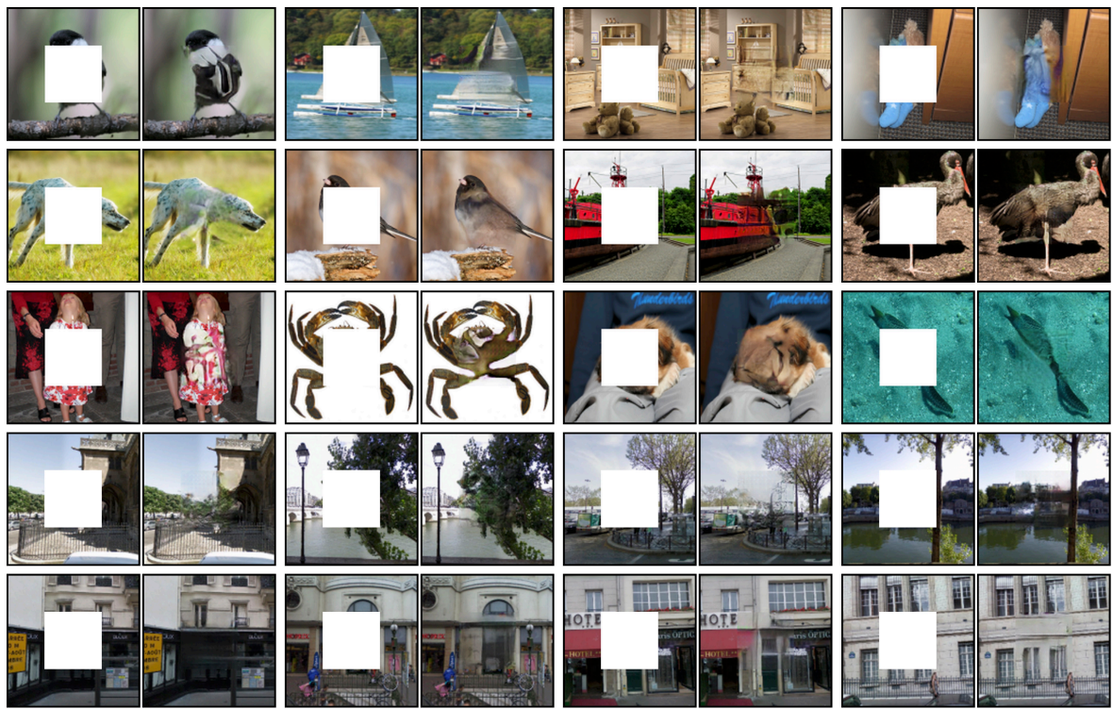
Usage 4: image in-painting

- The objective of image in-painting is to restore lost information in images by “painting” the lost region back in.



Source: O Shcherbakov and V Batishcheva, “Image inpainting based on stacked autoencoders”, 2014 J. Phys.: Conf. Ser. 536 012020 http://iopscience.iop.org/article/10.1088/1742-6596/536/1/012020/pdf

- Active research field, e.g. context encoders



Source: D. Pathak et al, “Context Encoders: Feature Learning by Inpainting”, 2016 CVPR http://people.eecs.berkeley.edu/~pathak/context_encoder/

Usage 5: initialise deep network for supervised learning

- Principles:
  - Pre-training step: train a stacked autoencoder using unsupervised data
  - Fine-tuning step 1: train the last layer using supervised data,
  - Fine-tuning step 2: use back-propagation to fine-tune the entire network using supervised data
- Researchers have shown that this pretraining idea improves deep neural networks.
- From 2006 to 2011, this approach gained much traction because the brain is likely doing unsupervised learning as well.
- Unsupervised learning is also more appealing because it makes use of inexpensive un-labeled data.
- Since 2012, this research direction however has gone through a relatively quiet period, because unsupervised learning is less relevant when a lot of labeled data are available.

From Quoc V. Le, Google Brain, “A Tutorial on Deep Learning. Part 2.”

Wrap-up

- Keras **functional API**
  - Create non-sequential architectures, multiple inputs, multiple features extraction etc
- **Transfer learning** is about exporting knowledge into new environments
  - Re-use features extraction layers of pre-trained networks
  - Allow to use deep learning on small train datasets
- **Autoencoders** are neural networks trained to reproduce its input
  - Discover a good internal representation of the input from which we can make some sense
  - Diabolo topology: shallow dense, stacked dense autoencoders or CNN autoencoders
  - 5 usages: data compression, feature extraction, de-noising, image in-painting, pre-training of deep network