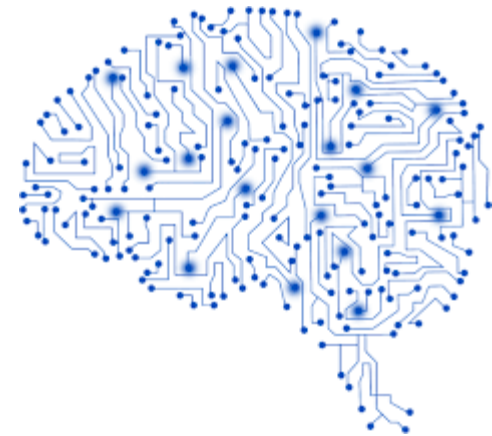


# Hands on deep learning

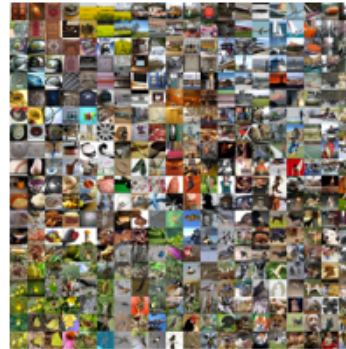


Alexandre Boucaud - [@alxbcd](#)

# What does "deep" means ?

IMAGENET

1 200 000 images  
1 000 classes



8 layers

**Error Rate:**  
**16.4%**

AlexNet (2012)

19 layers

**Error Rate:**  
**7.3%**

VGG (2014)

22 layers

**Error Rate:**  
**6.7%**

GoogleNet (2014)

152 layers

**Error Rate:**  
**3.57%**

Residual Net (2015)

more on these common net architectures [here](#)

# Why this recent trend ?

- specialized hardware

e.g. GPU, TPU, Intel Xeon Phi

# Why this recent trend ?

- specialized **hardware**

e.g. GPU, TPU, Intel Xeon Phi

- **data** availability

*big data era*

# Why this recent trend ?

- specialized **hardware**

e.g. GPU, TPU, Intel Xeon Phi

- **data** availability

*big data era*

- **algorithm** research

e.g. adversarial or reinforcement learning

# Why this recent trend ?

- specialized **hardware**

e.g. GPU, TPU, Intel Xeon Phi

- **data** availability

*big data era*

- **algorithm** research

e.g. adversarial or reinforcement learning

- **open source** tools

huge ecosystem right now

# Graphics Processing Unit (GPU)

- < 2000 : "graphics cards" (video edition + game rendering)



# Graphics Processing Unit (GPU)

- < 2000 : "graphics cards" (video edition + game rendering)
- 1999 : Nvidia coins the term "GPU" for the first time



# Graphics Processing Unit (GPU)

- **< 2000** : "graphics cards" (video edition + game rendering)
- **1999** : Nvidia coins the term "GPU" for the first time
- **2005** : programs start to be faster on GPU than on CPU

# Graphics Processing Unit (GPU)

- < 2000 : "graphics cards" (video edition + game rendering)
- 1999 : Nvidia coins the term "GPU" for the first time
- 2005 : programs start to be faster on GPU than on CPU
- 2016 : GPUs are part of our lives (phones, computers, cars, etc..)



credit: Nvidia Tesla V100

# Computational power

GPU architectures are **excellent** for the kind of computations required by the training of NN

$$\mathbf{D} = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix} + \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}$$

FP16 or FP32                      FP16                      FP16 or FP32

year	hardware	computation (TFLOPS)	price (K\$)
2000	IBM ASCI White	12	100 000 K
2005	IBM Blue Gene/L	135	40 000 K
2018	Nvidia Tesla V100	> 100	10 K

# Deep learning software ecosystem



theano



Microsoft  
CNTK

PYTORCH



Caffe2

dmlc  
***mxnet***

**gensim**

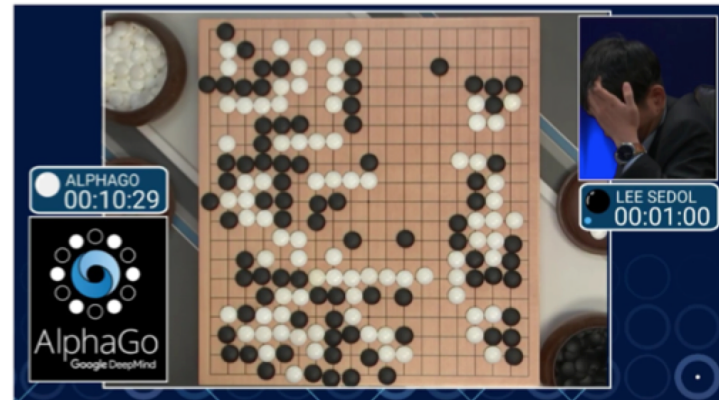
**spaCy**

# Deep learning today

- translation
- image captioning
- speech synthesis
- style transfer
- cryptocurrency mining
- self-driving cars
- games
- etc.



Style transfer - Gatvs (2015)



AlphaGo - DeepMind (2017)

# Deep learning today



Alexa - Amazon (2017)

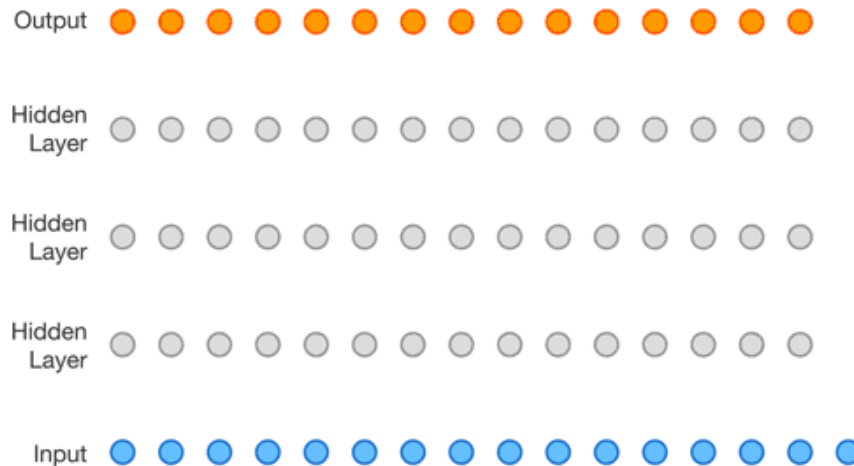


Skin cancer diagnostic – Stanford (2017)



StackGAN v2 – Zhang (2017)

# DL today: speech synthesis



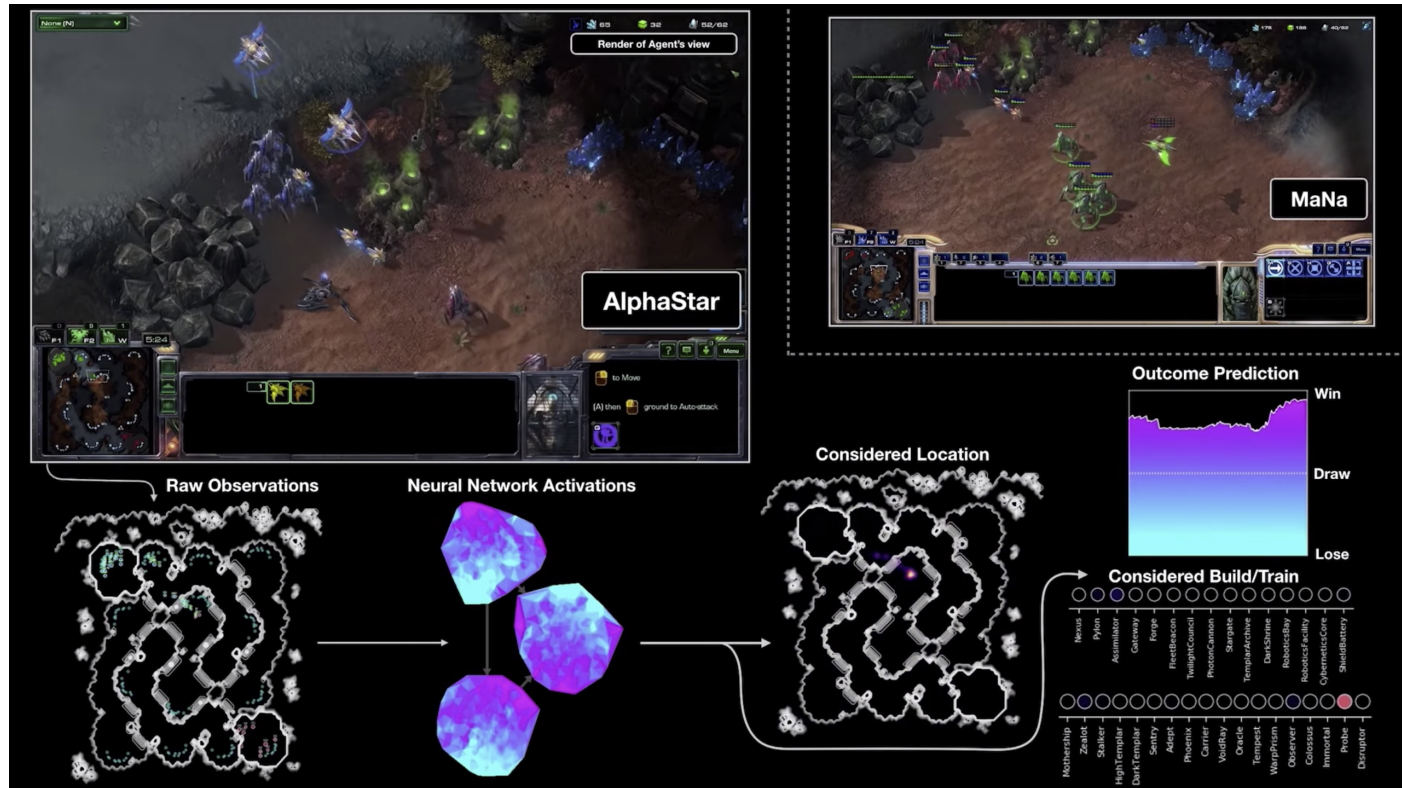
WaveNet - TTS with sound generation - DeepMind (2017)

# DL today: image colorisation



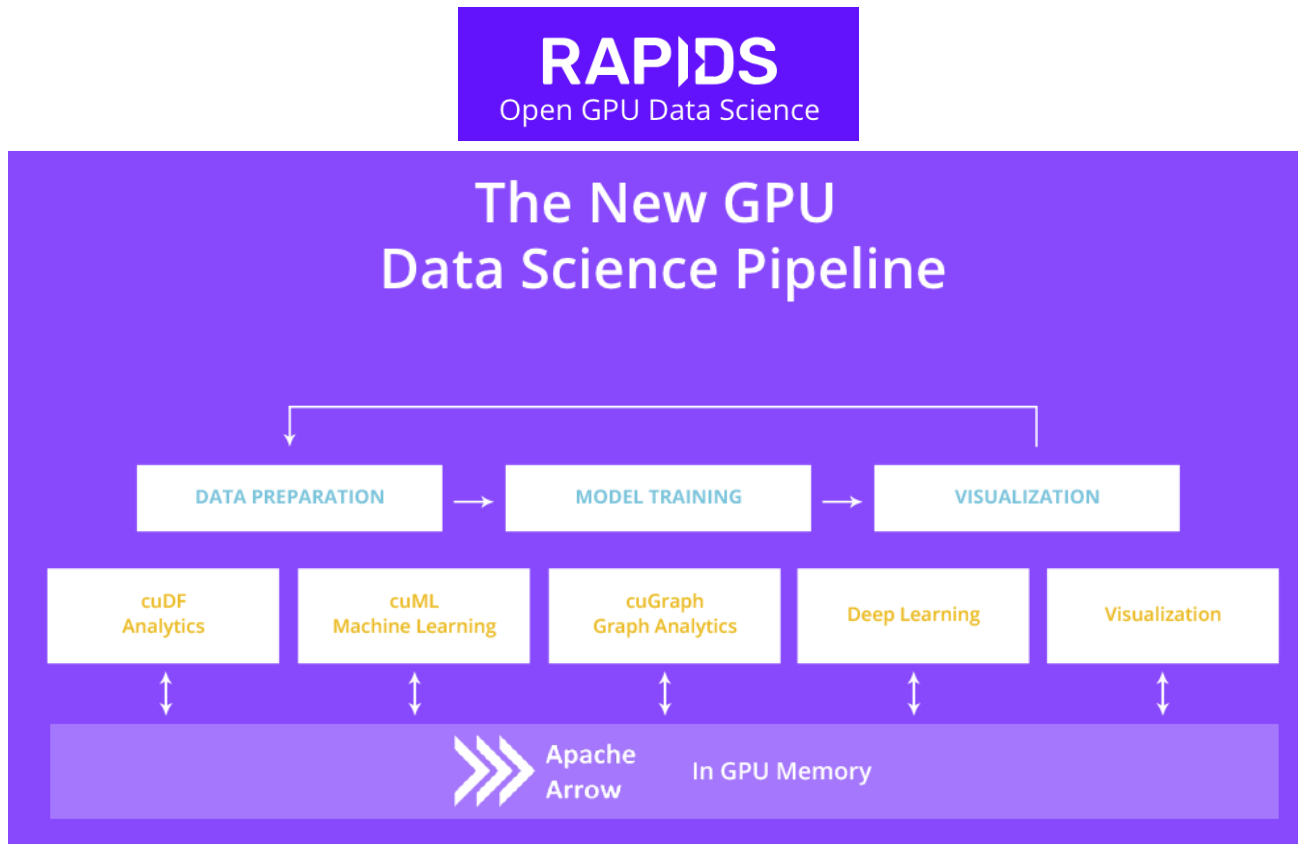
Real-time image colorization (2017)

# DL today: strategic game bots



## AlphaStar - Starcraft II AI - DeepMind (2019)

# DL today: data science world



[rapids.ai](https://rapids.ai) - Nvidia (2019)

# Outline

Neural nets - (dernier cours)

hidden layers - activation - backpropagation - optimization

# Outline

Neural nets - (dernier cours)

hidden layers - activation - backpropagation - optimization

Convolutional Neural Networks (CNN)

kernels - strides - pooling - loss - training

# Outline

Neural nets - (dernier cours)

hidden layers - activation - backpropagation - optimization

Convolutional Neural Networks (CNN)

kernels - strides - pooling - loss - training

In practice

step-by-step - monitoring your training

# Outline

## Neural nets - (dernier cours)

hidden layers - activation - backpropagation - optimization

## Convolutional Neural Networks (CNN)

kernels - strides - pooling - loss - training

## In practice

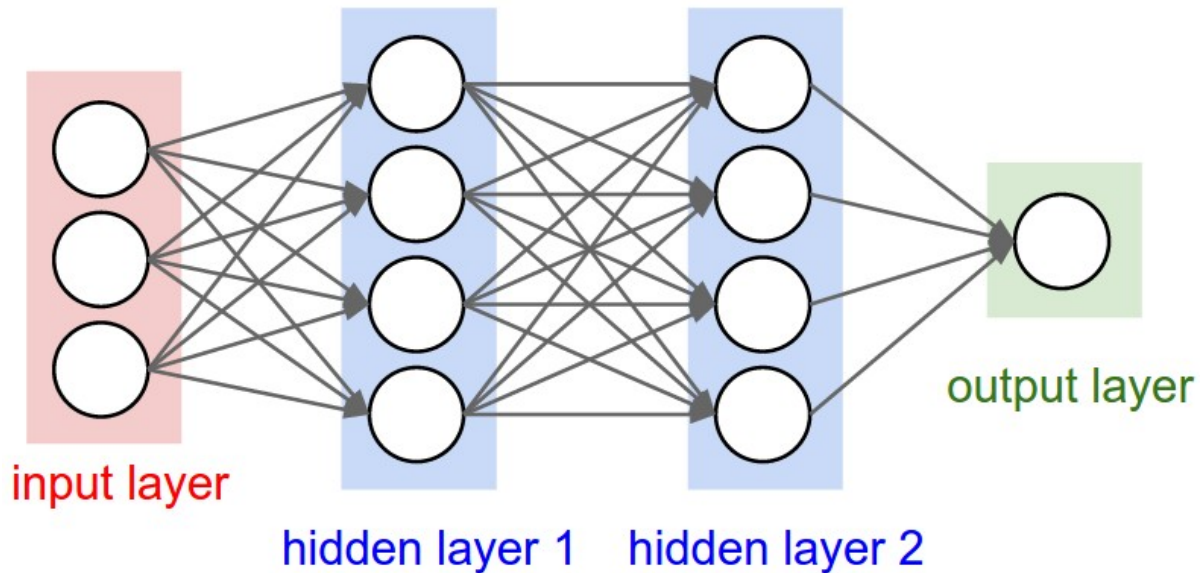
step-by-step - monitoring your training

## Common optimizations

data augmentation - dropout - batch normalisation

# Multi-layer perceptron

The **classical** neural network, with an input vector  $X_i$  where  $i$  is the sample number.



# Typical architecture

- **input neurons**: one per input feature
- **output neurons**: one per prediction dimension
- **hidden layers**: depends on the problem (~ 1 to 5)
- **neurons in hidden layers**: depends on the problem (~10 to 100)
- **loss function**: Mean Squared Error (MSE)
- **hidden activation**: ReLU
- **output activation**:
  - None
  - softplus (positive outputs)
  - sigmoid/tanh (bounded outputs)

# QUESTION:

How would you feed **images** to a MLP ?








# Convolutional Neural Networks

# Zoo of neural networks #1

A mostly complete chart of

## Neural Networks

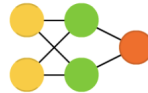
©2016 Fjodor van Veen - asimovinstitute.org

-  Backfed Input Cell
-  Input Cell
-  Noisy Input Cell
-  Hidden Cell
-  Probabilistic Hidden Cell
-  Spiking Hidden Cell
-  Output Cell
-  Match Input Output Cell
-  Recurrent Cell
-  Memory Cell
-  Different Memory Cell
-  Kernel
-  Convolution or Pool

Perceptron (P)



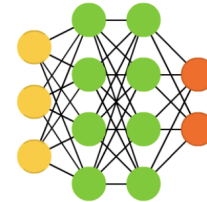
Feed Forward (FF)



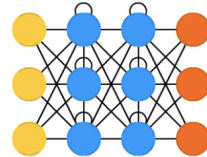
Radial Basis Network (RBF)



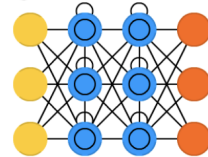
Deep Feed Forward (DFF)



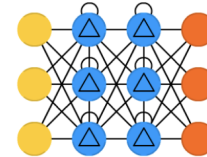
Recurrent Neural Network (RNN)



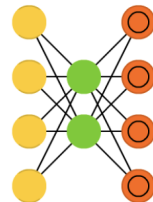
Long / Short Term Memory (LSTM)



Gated Recurrent Unit (GRU)



Auto Encoder (AE)



Variational AE (VAE)



Denoising AE (DAE)



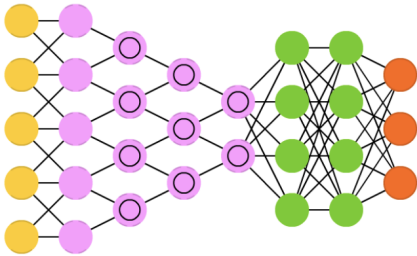
Sparse AE (SAE)



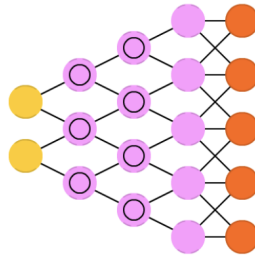
Neural network zoo - Fjodor van Veen (2016)

# Zoo of neural networks #2

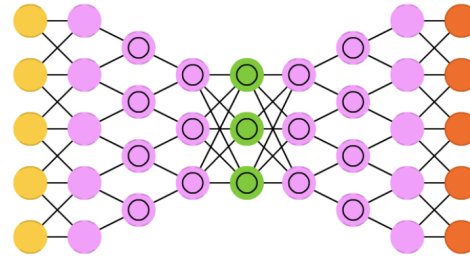
## Deep Convolutional Network (DCN)



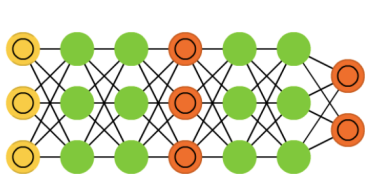
## Deconvolutional Network (DN)



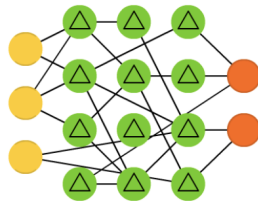
## Deep Convolutional Inverse Graphics Network (DCIGN)



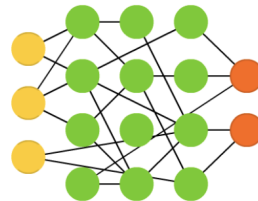
## Generative Adversarial Network (GAN)



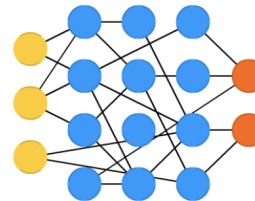
## Liquid State Machine (LSM)



## Extreme Learning Machine (ELM)

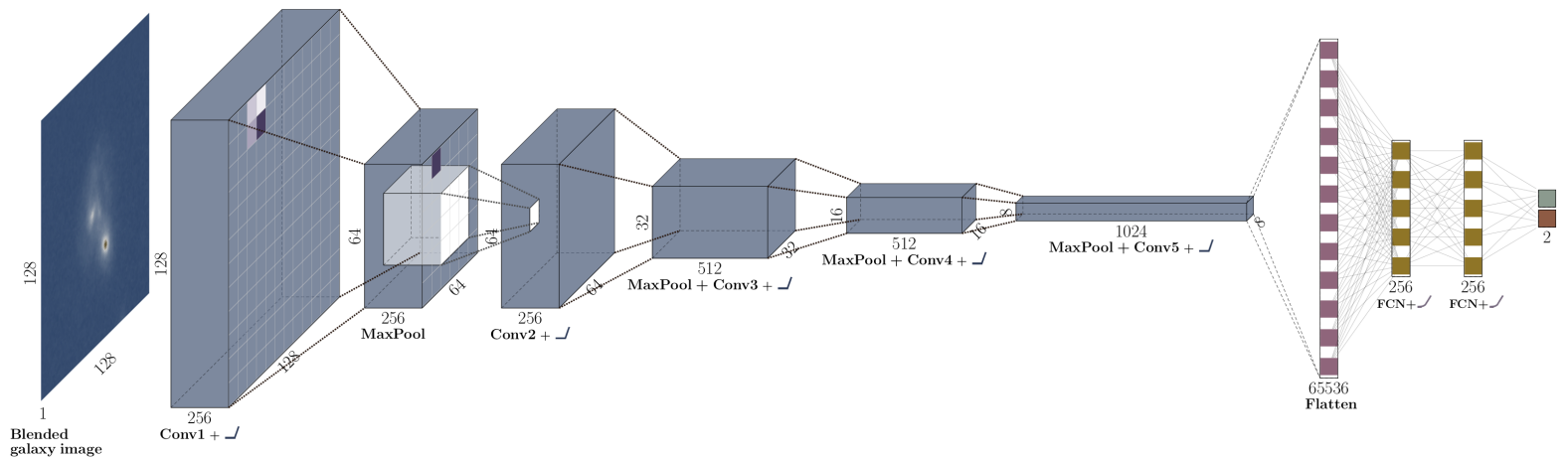


## Echo State Network (ESN)

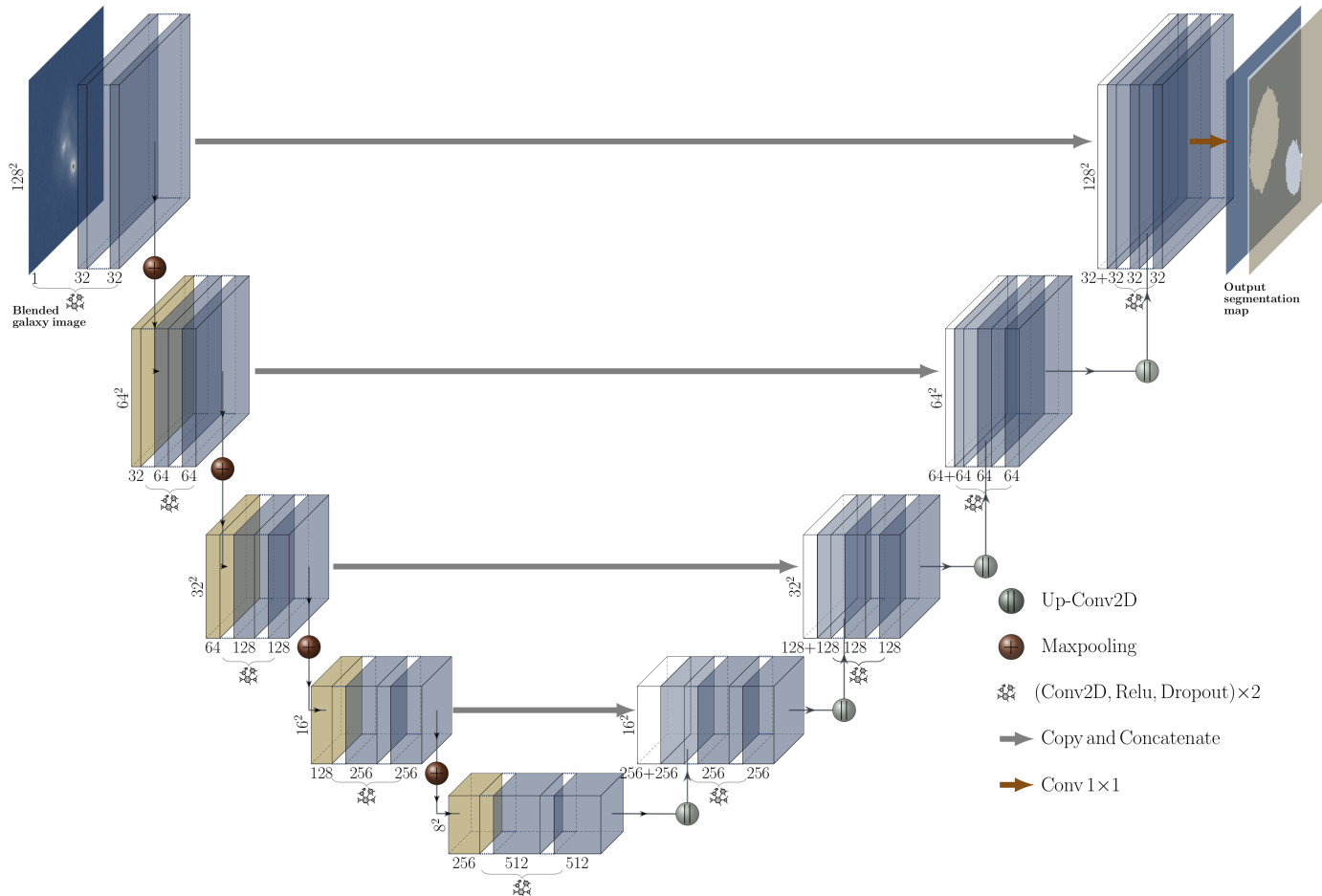


Neural network zoo - Fjodor van Veen (2016)

# Classical convolutional net (CNN)

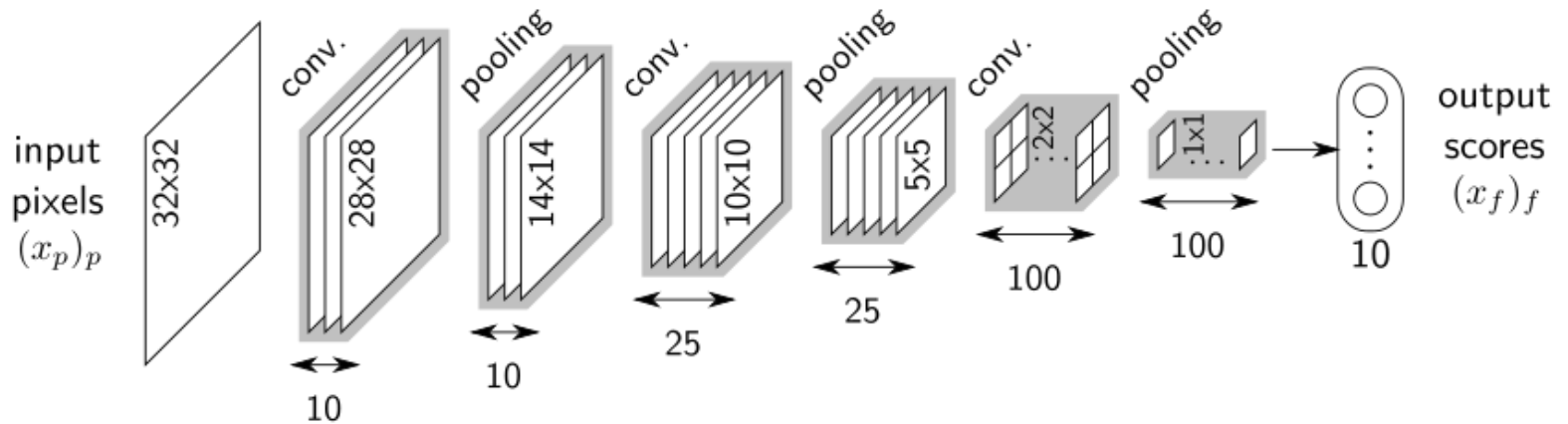


# Fully convolutional net (FCNN)



# Convolutional Neural Networks

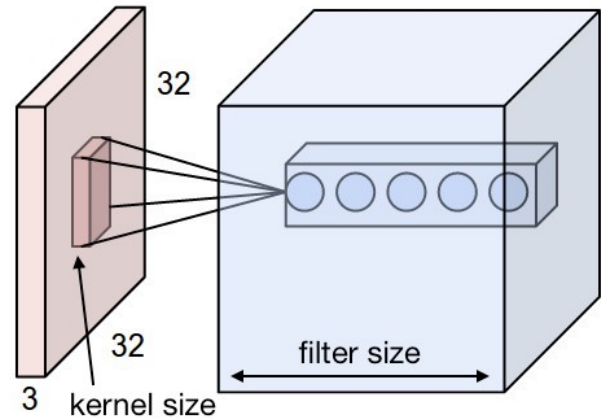
- elegant way of passing **tensors** to a network
- perform convolutions with **3D kernels**
- training optimizes kernels, not neuron weights



# Convolutional layers

```
from keras.models import Sequential
from keras.layers import Conv2D

model = Sequential()
# First conv needs input_shape
# Shape order depends on backend
model.add(
    Conv2D(15,      # filter size
           (3, 3),  # kernel size
           strides=1, # default
           padding='valid', # default
           input_shape=(32, 32, 3)))
# Next layers don't
model.add(Conv2D(16, (3, 3) strides=2))
model.add(Conv2D(32, (3, 3)))
```



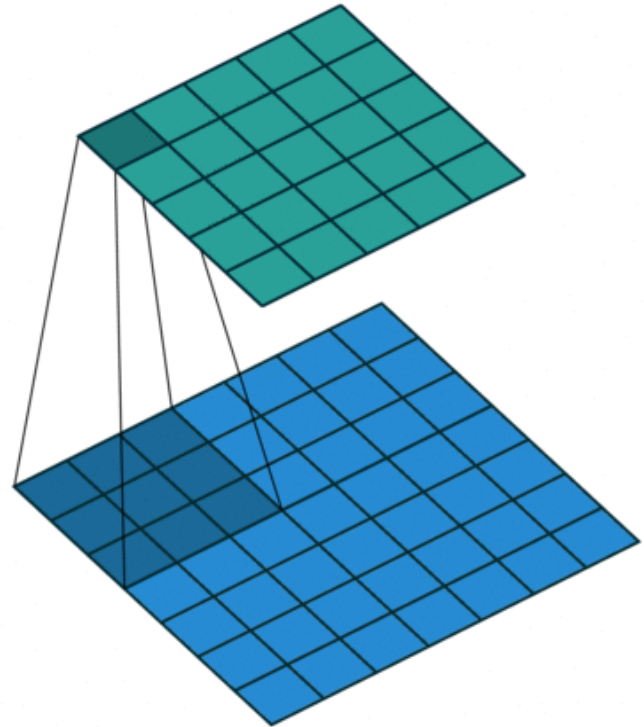
- **kernel properties:** size and number of filters
- **convolution properties:** strides and padding
- output shape depends on **all** these properties

# No strides, no padding

```
from keras.models import Sequential
from keras.layers import Conv2D

model = Sequential()
model.add(
    Conv2D(1, (3, 3),
           strides=1,      # default
           padding='valid', # default
           input_shape=(7, 7, 1)))
model.summary()
```

Layer (type)	Output Shape
=====	
conv2d (Conv2D)	(None, 5, 5, 1)
=====	
Total params: 10	
Trainable params: 10	
Non-trainable params: 0	

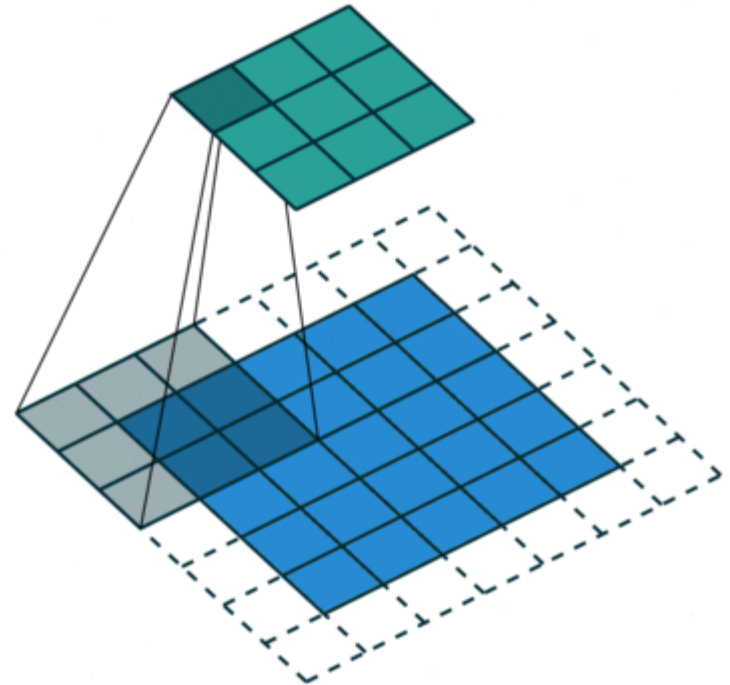


# Strides (2,2) + padding

```
from keras.models import Sequential
from keras.layers import Conv2D

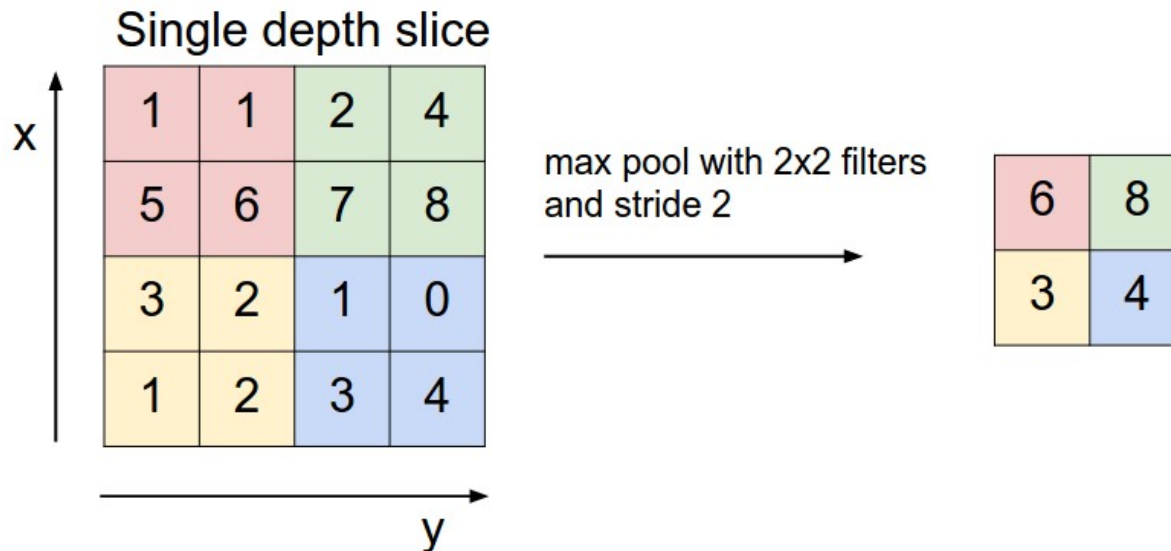
model = Sequential()
model.add(Conv2D(1, (3, 3),
                 strides=2,
                 padding='same',
                 input_shape=(5, 5, 1)))
model.summary()
```

Layer (type)	Output Shape
=====	
conv2d (Conv2D)	(None, 3, 3, 1)
=====	
Total params: 10	
Trainable params: 10	
Non-trainable params: 0	



# Pooling layers

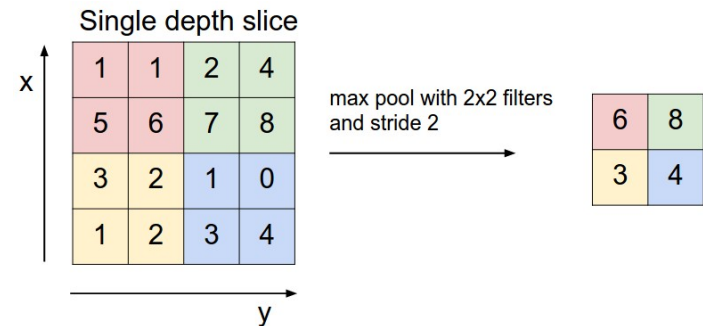
- reduces the spatial size of the representation (downsampling)  
=> less parameters & less computation
- common method: **MaxPooling** or **AvgPooling**
- common strides: (2, 2)



# Pooling layers

```
from keras.models import Sequential
from keras.layers import Conv2D
from keras.layers import MaxPool2D

model = Sequential()
model.add(Conv2D(1, (3, 3),
                 strides=1,
                 padding='same',
                 input_shape=(8, 8, 1)))
model.add(MaxPool2D((2, 2)))
model.summary()
```

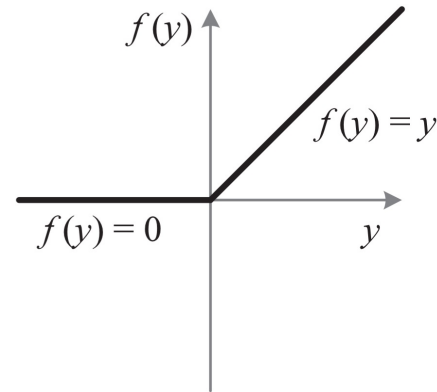


Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 8, 8, 1)	10
max_pooling2d_1 (MaxP	(None, 4, 4, 1)	0
Total params: 10		
Trainable params: 10		
Non-trainable params: 0		

# Activation

```
from keras.models import Sequential
from keras.layers import Conv2D

model = Sequential()
model.add(Conv2D(1, (3, 3),
                 activation='relu',
                 input_shape=(5, 5, 1)))
```

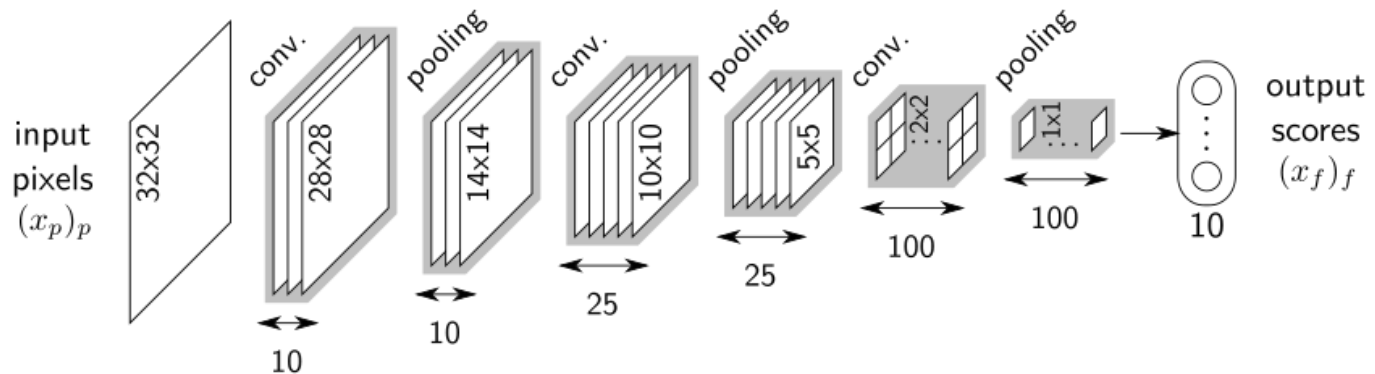


- safe choice\*: use **ReLU** or **variants** (PReLU, **SELU**) for the convolutional layers
- select the activation of the **last layer** according to your problem  
e.g. sigmoid for binary classification

\*not been proven (yet) but adopted empirically

# EXERCICE

on your own time, **write down the model** for the following architecture



how many **free parameters** does this architecture have ?

# SOLUTION

```
from keras.models import Sequential
from keras.layers import Conv2D
from keras.layers import MaxPool2D
from keras.layers import Flatten
from keras.layers import Dense

model = Sequential()

model.add(Conv2D(10, (5, 5),
                 input_shape=(32, 32, 1)))
model.add(MaxPool2D((2, 2)))
model.add(Conv2D(25, (5, 5)))
model.add(MaxPool2D((2, 2)))
model.add(Conv2D(100, (4, 4)))
model.add(MaxPool2D((2, 2)))
model.add(Flatten())
model.add(Dense(10))

model.summary()
```

Layer	Output Shape	Param #
Conv2D	(None, 28, 28, 10)	260
MaxPool2D	(None, 14, 14, 10)	0
Conv2D	(None, 10, 10, 25)	6275
MaxPool2D	(None, 5, 5, 25)	0
Conv2D	(None, 2, 2, 100)	40100
MaxPool2D	(None, 1, 1, 100)	0
Flatten	(None, 100)	0
Dense	(None, 10)	1010
Total params: 47,645		
Trainable params: 47,645		
Non-trainable params: 0		

# SOLUTION (bis)

```
from keras.models import Model
from keras.layers import Input
from keras.layers import Conv2D
from keras.layers import MaxPool2D
from keras.layers import Flatten
from keras.layers import Dense

input = Input((32, 32, 1))
x = Conv2D(10, (5, 5))(input)
x = MaxPool2D((2, 2))(x)
x = Conv2D(25, (5, 5))(x)
x = MaxPool2D((2, 2))(x)
x = Conv2D(100, (4, 4))(x)
x = MaxPool2D((2, 2))(x)
x = Flatten()(x)
output = Dense(10)(x)

model = Model(input, output)
model.summary()
```

Layer	Output Shape	Param #
=====		
Conv2D	(None, 28, 28, 10)	260
MaxPool2D	(None, 14, 14, 10)	0
Conv2D	(None, 10, 10, 25)	6275
MaxPool2D	(None, 5, 5, 25)	0
Conv2D	(None, 2, 2, 100)	40100
MaxPool2D	(None, 1, 1, 100)	0
Flatten	(None, 100)	0
Dense	(None, 10)	1010
=====		
Total params: 47,645		
Trainable params: 47,645		
Non-trainable params: 0		
=====		

# Loss and optimizer

Once your architecture (`model`) is ready, a **loss function** and an **optimizer must** be specified

```
model.compile(optimizer='adam', loss='mse')
```

or with better access to optimization parameters

```
from keras.optimizers import Adam
from keras.losses import MSE

model.compile(optimizer=Adam(lr=0.01, decay=0.1),
              loss=MSE)
```

Choose both according to the target output.

# Training

It's time to **train** your model on the data (X\_train, y\_train).

```
model.fit(X_train, y_train,  
          batch_size=32,  
          epochs=50,  
          validation_split=0.3) # % of data being used for val_loss evaluation
```

- **batch\_size:** # of images used before updating the model  
32 is a very good compromise between precision and speed\*
- **epochs:** # of times the model is trained with the full dataset

After each epoch, the model will compute the loss on the validation set to produce the **val\_loss**.

The closer the values of **loss** and **val\_loss**, the better the training.

\*see [Masters et al. \(2018\)](#)

# Callbacks

**Callbacks** are methods that act on the model during training, e.g.

```
from keras.callbacks import ModelCheckpoint
from keras.callbacks import EarlyStopping

# Save the weights of the model based on lowest val_loss value
chkpt = ModelCheckpoint('weights.h5', save_best_only=True)
# Stop the model before 50 epochs if stalling for 5 epochs
early = EarlyStopping(patience=5)

model.fit(X_train, y_train,
          epochs=50,
          callbacks=[chkpt, early])
```

# Callbacks

**Callbacks** are methods that act on the model during training, e.g.

```
from keras.callbacks import ModelCheckpoint
from keras.callbacks import EarlyStopping

# Save the weights of the model based on lowest val_loss value
chkpt = ModelCheckpoint('weights.h5', save_best_only=True)
# Stop the model before 50 epochs if stalling for 5 epochs
early = EarlyStopping(patience=5)

model.fit(X_train, y_train,
          epochs=50,
          callbacks=[chkpt, early])
```

- ModelCheckpoint saves the weights, which can be reloaded

```
model.load_weights('weights.h5') # instead of model.fit()
```

- EarlyStopping saves the planet.

In practice

# The right architecture

There is currently **no magic recipe** to find a network architecture that will solve your particular problem.

- \\_ (ゝツ) \\_ / -

But here are some advice to guide you in the right direction and/or get you out of trouble.

# A community effort

The Machine Learning community has long been  
a fervent advocate of

**open source**

which has fostered the spread of very recent  
developements even from big companies like Google.

Both **code and papers** are generally available  
and can be found **within a few clicks**.

# Start with existing (working) models

- look for a relevant architecture for your problem (arXiv, blogs, websites)

 Research at Google

Search

[Home](#) [Publications](#) [People](#) [Teams](#) [Outreach](#) [Blog](#) [Work at Google](#)

---

## SSD: Single Shot MultiBox Detector

### Venue

*Proceedings of the European Conference on Computer Vision (ECCV) (2016) (to appear)*

### Publication Year

2016

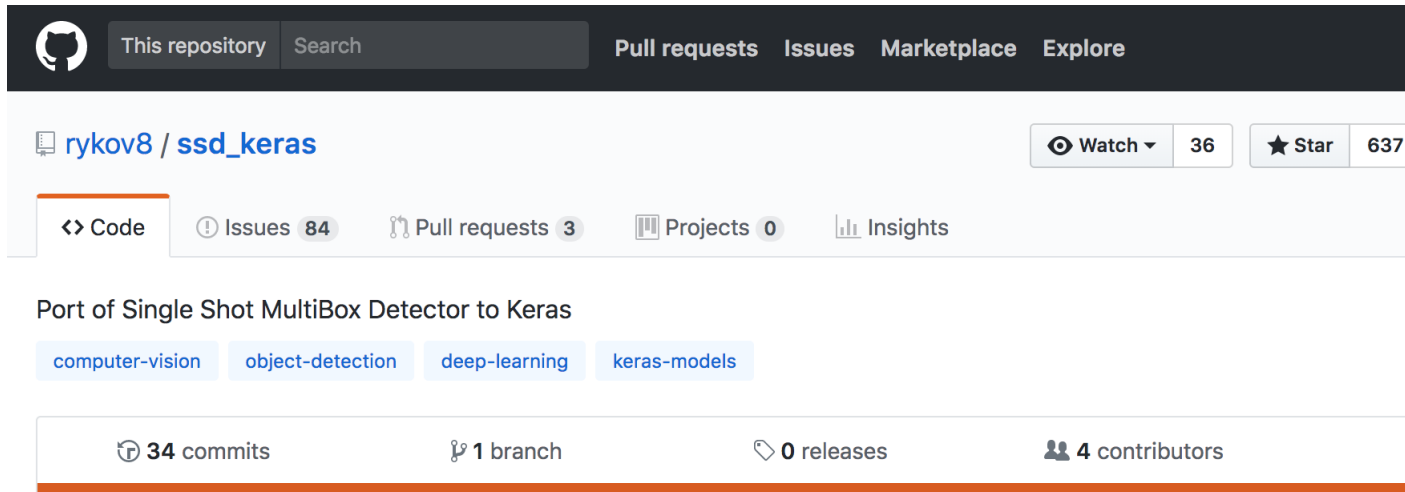
### Abstract

PDF

We present a method for detecting objects in images using a single deep neural network. Our approach, named SSD, discretizes the output space of bounding boxes into a set of bounding box priors over different aspect ratios and scales per feature map location. At prediction time, the network generates confidences that each prior corresponds to objects of interest and produces adjustments to the prior to better match the object shape. Additionally, the network combines


# Start with existing (working) models

- look for a relevant architecture for your problem (arXiv, blogs, websites)
- find an implementation on [GitHub](#) (often the case if algorithm is efficient)



The screenshot shows the GitHub interface for the repository `rykov8 / ssd_keras`. At the top, there's a dark navigation bar with the GitHub logo, a search bar, and links for Pull requests, Issues, Marketplace, and Explore. Below this, the repository name is displayed with a file icon. To the right, there are buttons for Watch (36) and Star (637). A secondary bar shows tabs for Code (selected), Issues (84), Pull requests (3), Projects (0), and Insights. The repository description is "Port of Single Shot MultiBox Detector to Keras". Below the description are tags: computer-vision, object-detection, deep-learning, and keras-models. At the bottom, a summary bar shows 34 commits, 1 branch, 0 releases, and 4 contributors.





This repository  Pull requests Issues Marketplace Explore

 [rykov8](#) / [ssd\\_keras](#) Watch 36 Star 637

[Code](#) [Issues 84](#) [Pull requests 3](#) [Projects 0](#) [Insights](#)

Port of Single Shot MultiBox Detector to Keras

[computer-vision](#) [object-detection](#) [deep-learning](#) [keras-models](#)

 34 commits  1 branch  0 releases  4 contributors

# Start with existing (working) models

- look for a relevant architecture for your problem  
(arXiv, blogs, websites)
- find an implementation on [GitHub](#)  
(often the case if algorithm is efficient)
- play with the examples and adjust to your inputs/outputs

# Start with existing (working) models

- look for a relevant architecture for your problem  
(arXiv, blogs, websites)
- find an implementation on [GitHub](#)  
(often the case if algorithm is efficient)
- play with the examples and adjust to your inputs/outputs
- use [pretrained nets](#) for the pre-processing of your data

# Start with existing (working) models

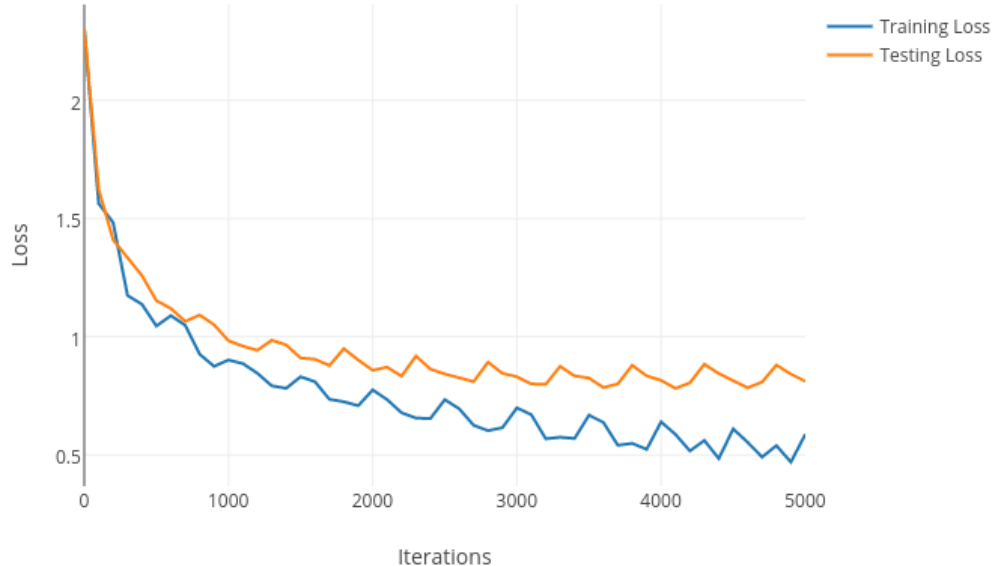
- look for a relevant architecture for your problem  
(arXiv, blogs, websites)
- find an implementation on [GitHub](#)  
(often the case if algorithm is efficient)
- play with the examples and adjust to your inputs/outputs
- use [pretrained nets](#) for the pre-processing of your data
- start tuning the model parameters..

# Plot the training loss

```
import matplotlib.pyplot as plt

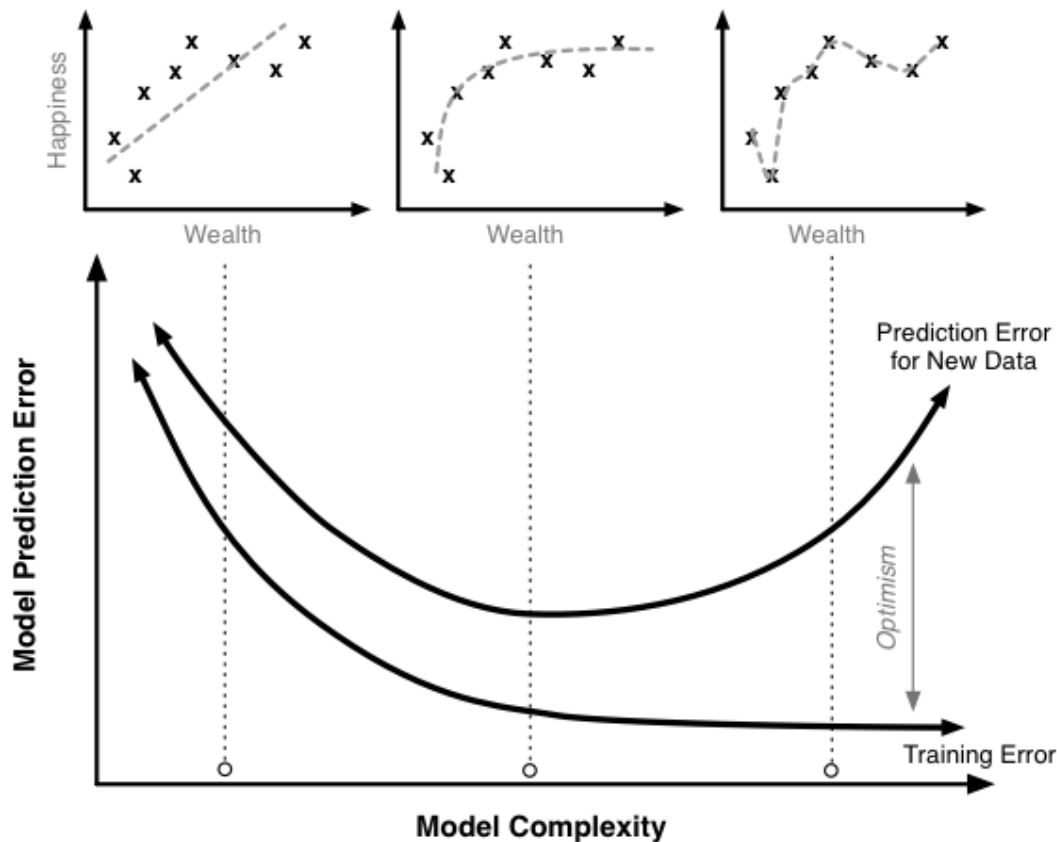
history = model.fit(X_train, y_train, validation_split=0.3)

# Visualizing the training
plt.plot(history.history['loss'], label='training')
plt.plot(history.history['val_loss'], label='validation')
plt.xlabel('epochs'); plt.ylabel('loss'); plt.legend()
```



# Plot the training loss

And look for the training **sweet spot** (before **overfitting**).



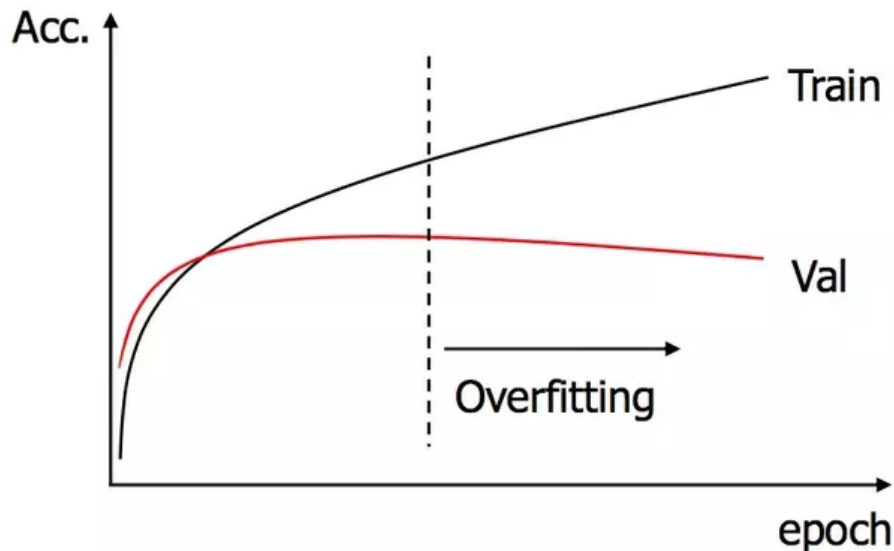
# Plot other metrics

```
import matplotlib.pyplot as plt

model.compile(..., metrics=['acc']) # computes other metrics, here accuracy

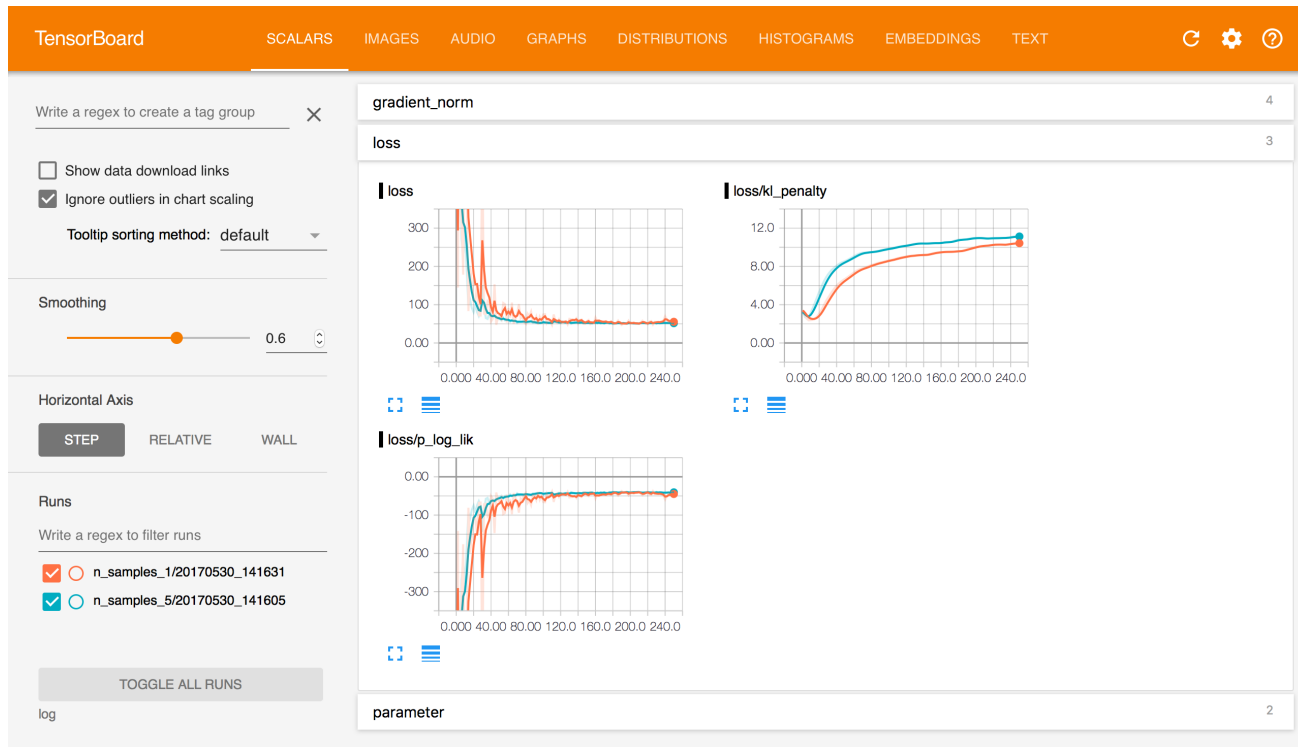
history = model.fit(X_train, y_train, validation_split=0.3)

# Visualizing the training
plt.plot(history.history['acc'], label='training')
plt.plot(history.history['val_acc'], label='validation')
plt.xlabel('epochs'); plt.ylabel('accuracy'); plt.legend()
```



# Tensorboard

There is a [Keras callback](#) to use Tensorboard



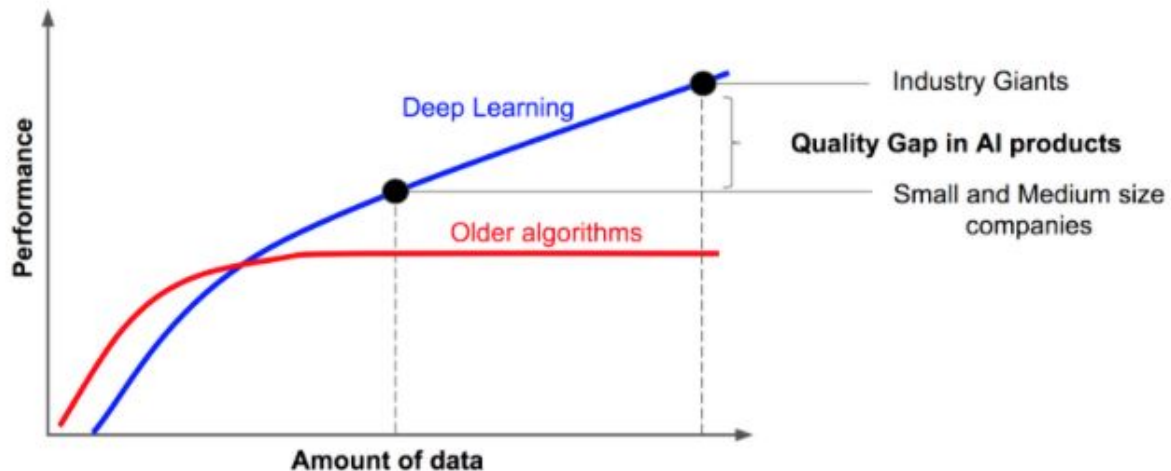
credit: [Edward Tensorboard tuto](#)

# Common optimizations

"avoiding overfitting"

# Data is key

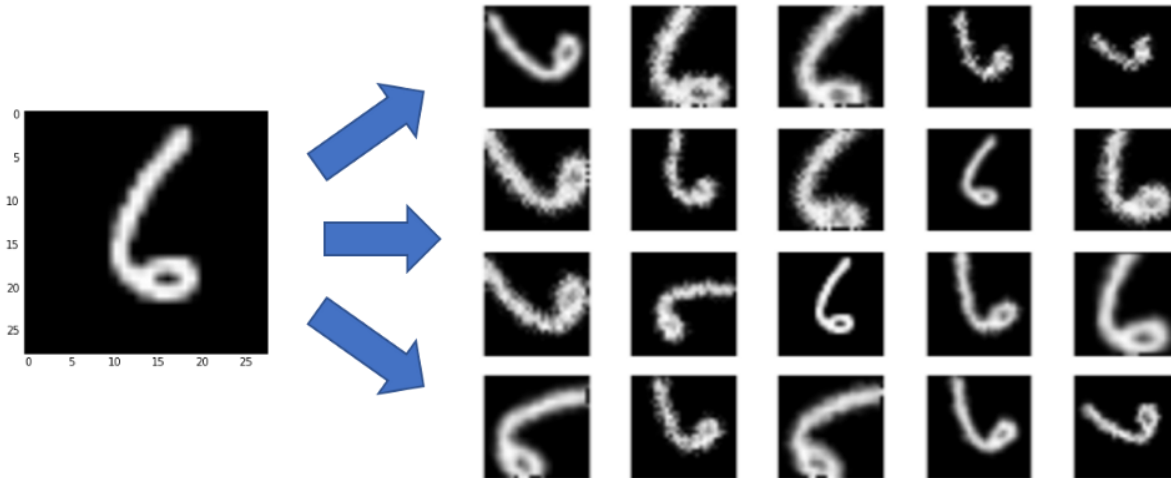
Deep neural nets need **a lot of data** to achieve good performance.



# Data is key

Deep neural nets need **a lot of data** to achieve good performance.

Use **data augmentation**.



# Data is key

Deep neural nets need **a lot of data** to achieve good performance.

Use **data augmentation**.

Choose a training set **representative** of your data.

# Data is key

Deep neural nets need **a lot of data** to achieve good performance.

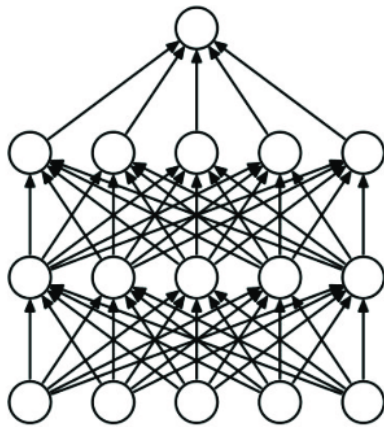
Use **data augmentation**.

Choose a training set **representative** of your data.

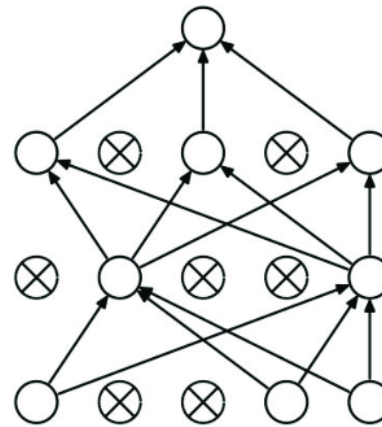
If you cannot get enough labeled data, use simulations or turn to **transfer learning** techniques.

# Dropout

A % of random neurons are **switched off** during training  
it mimics different architectures being trained at each step



(a) Standard Neural Network



(b) Neural Net with Dropout

Srivastava et al. (2014)

# Dropout

```
...  
from keras.layers import Dropout  
  
dropout_rate = 0.25  
  
model = Sequential()  
model.add(Conv2D(2, (3, 3), input_shape=(9, 9, 1)))  
model.add(Dropout(dropout_rate))  
model.add(Conv2D(4, (3, 3)))  
model.add(Dropout(dropout_rate))  
...
```

- regularization technique extremely effective
- prevents overfitting

**Note:** dropout is **not used during evaluation**, which accounts for a small gap between **loss** and **val\_loss** during training.

Srivastava et al. (2014)

# Batch normalization

```
...  
from keras.layers import BatchNormalization  
from keras.layers import Activation  
  
model = Sequential()  
model.add(Conv2D(..., activation=None))  
model.add(BatchNormalization())  
model.add(Activation('relu'))
```

- technique that adds robustness against bad initialization
- forces activations layers to take on a unit gaussian distribution at the beginning of the training
- must be used before non-linearities

Ioffe & Szegedy (2015)

# to go further..

Here are some leads (random order) to explore if your model do not converge:

- data normalization
- weight initialization
- choice of learning rate
- gradient clipping
- various regularisation techniques

# Thank you

Contact info:

[aboucaud.github.io](https://aboucaud.github.io)

@aboucaud on GitHub, GitLab

[@alxbcd](#) on Twitter

This presentation is licensed under a  
[Creative Commons Attribution-ShareAlike 4.0 International License](#)

