Applied Data Science

Mod.8 – Neural Networks – day 1

CAMBRIDGE SPARK

@cambridgespark

Weekend outline

Today :

- Neural Nets basics (Multilayer Perceptron or MLPs)
- Convolutional Neural Networks (CNNs)

Tomorrow :

- CNNs for time series
- RNNs and LSTMs for classification and regression

Before starting…

A lot of notions to cover and some will require you to come back to it if you want to fully grasp the notions:

- **essentials**: how an MLP works, the issues, where NNs can work and where they don't, how to apply pre-trained neural nets.
- **intermediate**: convolution, pooling, batch-training, batch-normalisation, activation functions, input/output layers
- **advanced**: weight regularisation, step-size scheme, dropout, vanishing gradient

Before starting…

A lot of notions to cover and some will require you to come back to it if you want to fully grasp the notions:

- **essentials**: how an MLP works, the issues, where NNs can work and where they don't, how to apply pre-trained neural nets.
- **intermediate**: convolution, pooling, batch-training, batch-normalisation, activation functions, input/output layers
- **advanced**: weight regularisation, step-size scheme, dropout, vanishing gradient

It's not so much the maths but rather NNs rely on *almost all* of classical ML $\rightarrow$ some concepts require time to process. We *strongly* recommend you work in pairs.

Neural Nets: the basics

- Introduction to NN
  - *NN for classification*
  - *The perceptron*
- Essential Maths
  - *Activation functions and loss functions*
  - *Gradient descent*
- Training and using a NN

Neural Net = set of interconnected neurons

- flexible class of models for supervised learning

Neural Net = set of interconnected neurons

- flexible class of models for supervised learning
- neuron $=$ small *processing unit* transforming an input (stimulus) into a signal

Neural Net = set of interconnected neurons

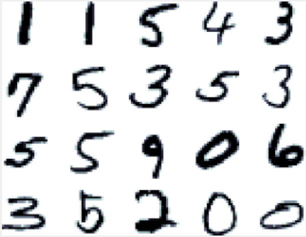
- flexible class of models for supervised learning
- neuron $=$ small *processing unit* transforming an input (stimulus) into a signal
- **aim**: imitate a complex function

Standard role for NN: classification

- training set $= \{(\text{object}, \text{class})_i\}_{i=1:N}$
- **goal**: label new objects
- **function to imitate**: classification function

Standard role for NN: classification

- training set $= \{(\text{object}, \text{class})_i\}_{i=1:N}$
- **goal**: label new objects
- **function to imitate**: classification function
- **Example**: classifying handwritten digits



The MNIST dataset

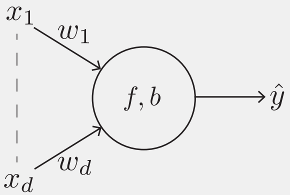
- objects : images of handwritten digits (28×28 px)
- labels : corresponding digit: $\{0, \dots, 9\}$.
- dataset : 70k labeled images,
  (60k for training, 10k for testing)

The MNIST dataset

- objects : images of handwritten digits (28$\times$28 px)
- labels : corresponding digit: $\{0, \dots, 9\}$.
- dataset : 70k labeled images,
  (60k for training, 10k for testing)

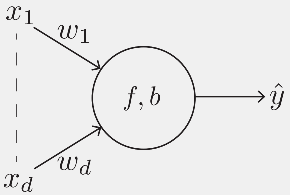
Note: this dataset has been shown to be *too-simple* but it helps to gain intuition. You will practice on more complicated datasets later.

Perceptron = 1-neuron NN



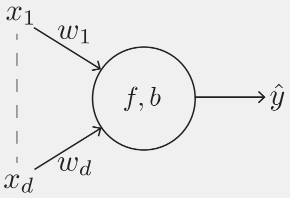
- input x with d components
- weight w with d components

Perceptron = 1-neuron NN



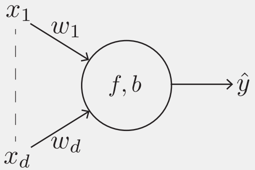
- input x with d components
- weight w with d components
- activation function f with bias b

Perceptron = 1-neuron NN



- input x with d components
- weight w with d components
- activation function f with bias b
- output $\hat{y}$ (class)

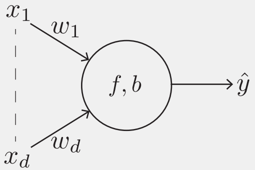
Inside a simple neuron…



- compute the weighted sum $s = \langle x, w \rangle$ (*stimulus*)

$$\langle x, w \rangle = \sum_{i=1}^{d} w_i x_i$$

Inside a simple neuron...



- compute the weighted sum $s = \langle x, w \rangle$ (*stimulus*)

$$\langle x, w \rangle = \sum_{i=1}^{d} w_i x_i$$

- compare stimulus s to bias b (*activation*):

$s > b$: return $\hat{y} = 1$ (*neuron fires*)

$s \leq b$: return $\hat{y} = 0$ (*neuron does not fire*)

Coding a perceptron…

Head to your notebook and…

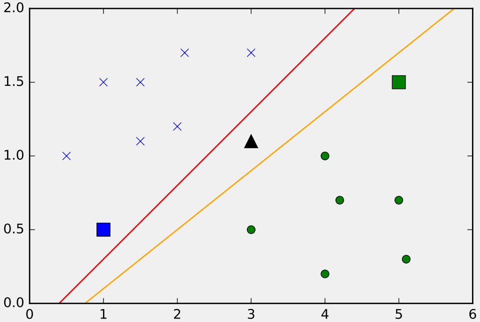
- see how a basic perceptron function can be coded
- try to use it to classify points
- what do you need to set? are there multiple solutions?

Perceptron: checkpoint 1

```python
def outPerceptron(x, w, b):
    innerProd = np.dot(x, w)
    output    = 0
    if innerProd > b:
        output = 1
    return output

def multiOutPerceptron2(X, w, b):
    return (np.dot(X, w) > b).astype(float)
```

Perceptron: checkpoint 2



Coding a perceptron…

Key points…

- parameters to set: weights w and bias b
- finding good parameters = training process

Coding a perceptron…

Key points…

- parameters to set: weights w and bias b
- finding good parameters = training process
- the neuron you coded answers a binary question:

"Does the input have a specific feature or not?"

Needed to train a NN: a loss and an optimiser

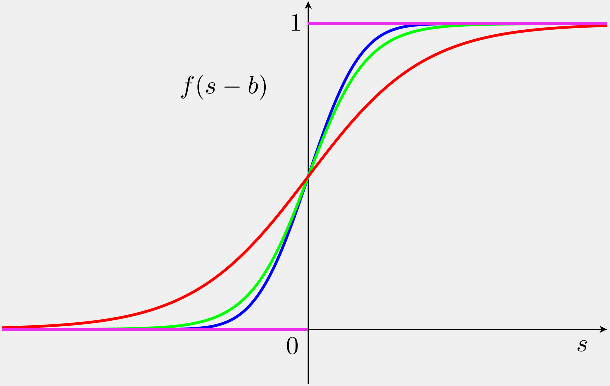
- a loss function is a way to measure the performances corresponding to a set of parameters $\theta = (w, b)$
- an optimiser is a procedure to go from a θ^0 to an *improved* θ^1 with lower loss.

Needed to train a NN: a loss and an optimiser

- a loss function is a way to measure the performances corresponding to a set of parameters $\theta = (w, b)$
- an optimiser is a procedure to go from a θ^0 to an *improved* θ^1 with lower loss.

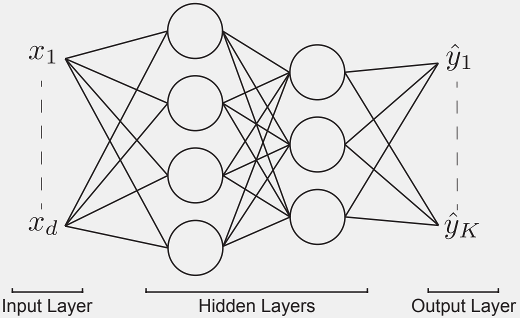
Recall: this is done in the *training phase*, performance and loss are measured on the training set .

Activation function : a teaser



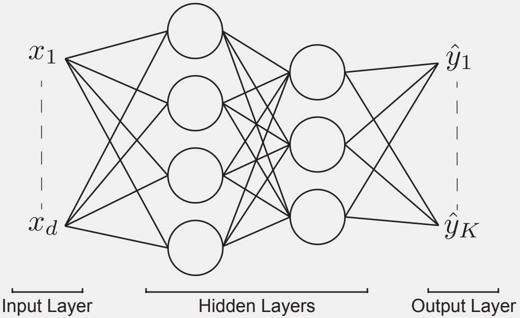
(more later…)

Neural network = set of interconnected neurons



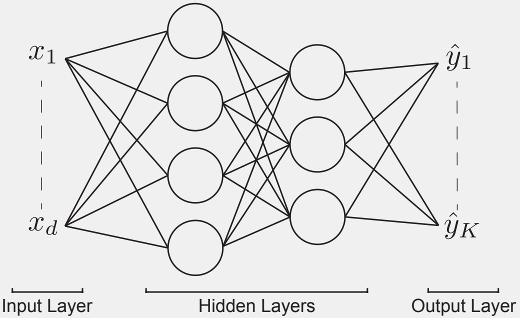
- an input layer with d nodes

Neural network = set of interconnected neurons



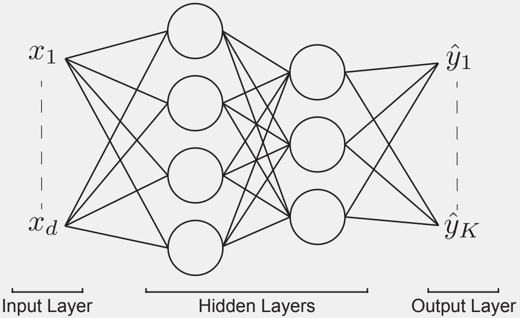
- an input layer with d nodes
- hidden layers each with a fixed number of neurons

Neural network = set of interconnected neurons



- an input layer with d nodes
- hidden layers each with a fixed number of neurons
- an interconnection pattern between layers

Neural network = set of interconnected neurons



- an input layer with d nodes
- hidden layers each with a fixed number of neurons
- an interconnection pattern between layers
- an output layer with K nodes

The architecture : a hard choice

- Depth : look at *complex interactions* of features
- Interconnection pattern : exploit problem-specific structure
- Ideal model :

The architecture : a hard choice

- Depth : look at *complex interactions* of features
- Interconnection pattern : exploit problem-specific structure
- Ideal model :
    - not too expensive to train $\rightarrow$ *simple architecture*
    - not overfitting $\rightarrow$ *simple architecture*
    - capture complexity of data $\rightarrow$ *complex architecture*

The architecture : a hard choice

- for generic problems, no one knows
- for specific problems, some architectures are known to perform well, often inspired from our brain

The architecture : a hard choice

- for generic problems, no one knows
- for specific problems, some architectures are known to perform well, often inspired from our brain
  - e.g.: convolutional NN for image classification

(more about all this later…)

Playing with the architecture

One nice way to learn about the impact of architectural choices when dealing with Neural Networks is the *tensorflow playground*

`http://playground.tensorflow.org`

Training = setting the parameters

- As for the perceptron, get good parameters
  but for all neurons this time

Training = setting the parameters

- As for the perceptron, get good parameters but for all neurons this time
- Typical architectures have thousands of neurons $\rightarrow$ many parameters to train (often millions)

Training = setting the parameters

- As for the perceptron, get good parameters
  but for all neurons this time
- Typical architectures have thousands of neurons $\rightarrow$ many parameters to train (often millions)
- Loss functions defined over (typically) huge training sets

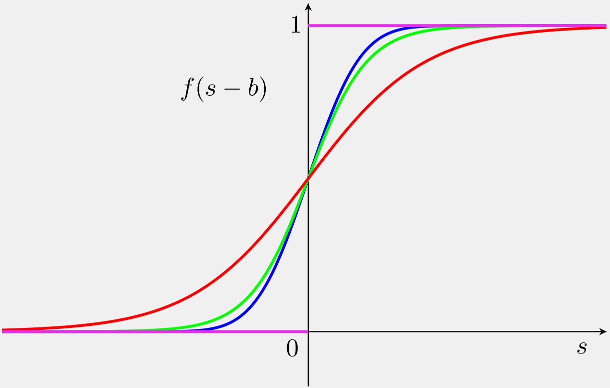
Training = setting the parameters

- As for the perceptron, get good parameters but for all neurons this time
- Typical architectures have thousands of neurons $\rightarrow$ many parameters to train (often millions)
- Loss functions defined over (typically) huge training sets
- The training of a NN is computationally hard but can be done using a simple class of optimisers .

ESSENTIAL MATHS

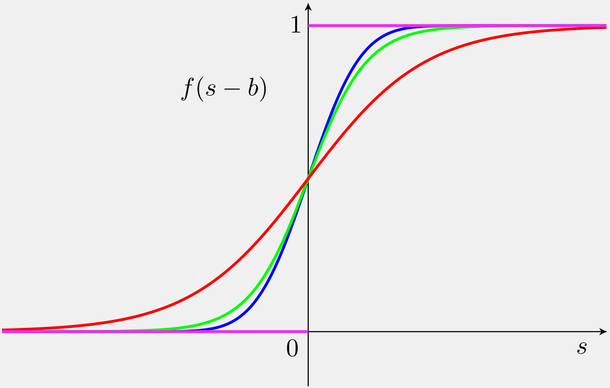
Sigmoid and hinge activation functions

- Sigmoid activation functions : smooth approximations of the step function



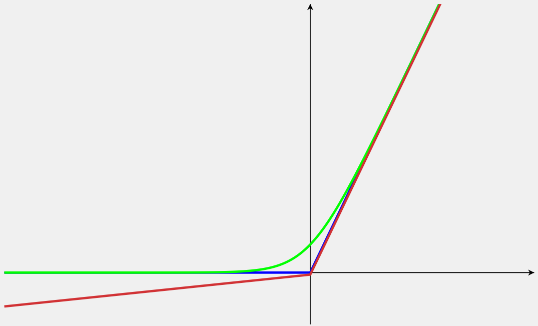
Sigmoid and hinge activation functions

- Sigmoid activation functions : smooth approximations of the step function
  - *tanh, logistic,...*



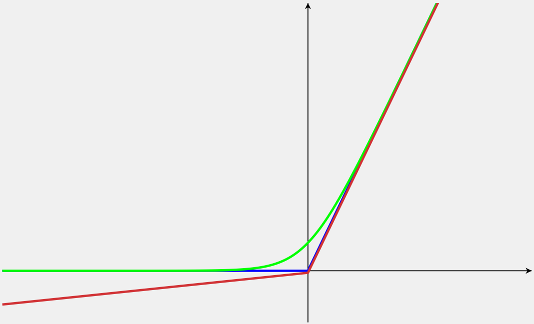
Sigmoid and hinge activation functions

- Hinge activation functions : idea of neuron firing more frequently when stimulated more



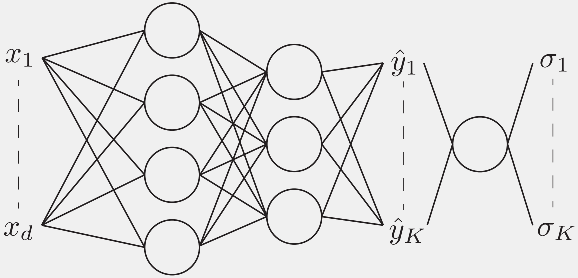
Sigmoid and hinge activation functions

- Hinge activation functions : idea of neuron firing more frequently when stimulated more
  - *Rectified Linear Units (ReLU), Leaky ReLU, softplus, …*



The softmax for the last layer

- No parameters, converts the K outputs into K scores
  - positive, sum to 1
  - can be interpreted as probability of input being in that category



Loss functions : way to rank parameters

- The misclassification loss function: counts the number of instances inaccurately classified (problem: not smooth)

Loss functions : way to rank parameters

- The misclassification loss function: counts the number of instances inaccurately classified (problem: not smooth)
- The quadratic loss function: sum the squares of errors, (same loss used in standard regression):

$$L(\theta) = \sum_{i=1}^{N} (\sigma_i(\theta) - y_i)^2$$

Loss functions : way to rank parameters

- The misclassification loss function: counts the number of instances inaccurately classified (problem: not smooth)
- The quadratic loss function: sum the squares of errors, (same loss used in standard regression):

$$L(\theta) = \sum_{i=1}^{N} (\sigma_i(\theta) - y_i)^2$$

- The cross entropy loss function:

$$L(\theta) = -\sum_{i=1}^{N} y_i \log \sigma_i(\theta) + (1 - y_i) \log(1 - \sigma_i(\theta))$$

Optimising the loss $\rightarrow$ gradient descent

$$\theta^{k+1} \quad = \quad \theta^k - \delta^k \nabla L(\theta^k)$$

Recall:

- δ^k is the step-size at step k (standard schemes such as ADAM are popular for training NNs)
- $\nabla L(\theta^k)$ is the gradient of the loss at the previous step
- provably leads to minimiser only if the loss function is convex (not the case for NNs…)

Gradient of the loss or backprop

Objective function is a sum over datapoints (as usual):

$$L(\theta) \quad = \quad \sum_{i=1}^{N} L_i(\theta)$$

The gradient is therefore $\nabla L(\theta) = \sum_{i=1}^{N} \nabla L_i(\theta)$.

- how do we compute $\nabla L_i(\theta)$?
- that's potentially a big sum (usually N is in the millions) $\rightarrow$ can we simplify this?

Computing $\nabla L_i(\theta)$

- the model is *layered* $\rightarrow$ the loss function is a composition
- gradient of composition requires "the chain rule"
- very efficient implementations exist $\rightarrow$ *central element in libraries such as TensorFlow or Torch.*

Dealing with a big sum

Instead of

$$\nabla L(\theta) \quad = \quad \sum_{i=1}^{N} \nabla_i(\theta)$$

use an approximation :

$$\hat{\nabla} L(\theta) \quad = \quad \frac{N}{|S|} \sum_{i \in S} \nabla L_i(\theta)$$

where S is a random subset of $\{1, \ldots, N\}$, a batch .

This is the principle behind stochastic gradient descent (SGD).

Adam and Xavier

- The training of Neural Nets is done with SGD
- State-of-the-art step-size schemes take this into account for example:
  - Xavier/Glorot Initialisation
  - Adam stepping scheme

Adam and Xavier

- The training of Neural Nets is done with SGD
- State-of-the-art step-size schemes take this into account for example:
  - Xavier/Glorot Initialisation
  - Adam stepping scheme

Brief explanation coming...

Adam stepping scheme ($\star\star$)

- keep track of recent gradient estimates in order to approximate the mean and the variance of the gradient at current step

$$\theta^{k+1} = \theta^k - \frac{\delta m^k}{\sqrt{\hat{\nu}^k} + \epsilon}$$

m^k and ν^k are estimates of the first and second moment of the gradients.

Adam stepping scheme ($\star\star$)

- keep track of recent gradient estimates in order to approximate the mean and the variance of the gradient at current step

$$\theta^{k+1} = \theta^k - \frac{\delta m^k}{\sqrt{\hat{\nu}^k} + \epsilon}$$

m^k and ν^k are estimates of the first and second moment of the gradients.

there are quite a few methods out there, no guarantees but empirical performances. ADAM is known to work quite well for Deep Learning.

Xavier initialisation ($\star\star$)

- generate initial weights from random normal with mean 0, variance $1/\tau$.
- initially, the neural network should not increase or decrease the variance of an instance that goes through it.
- variance of input = variance of output for a neuron

Xavier initialisation ($\star\star$)

- generate initial weights from random normal
  with mean 0, variance $1/\tau$.

- initially, the neural network should not increase or decrease the variance of an instance that goes through it.

- variance of input = variance of output for a neuron

- Forward : requires $\tau = n_{\text{in}}$

- Backward : requires $\tau = n_{\text{out}}$

Xavier initialisation ($\star\star$)

- generate initial weights from random normal with mean 0, variance $1/\tau$.
- initially, the neural network should not increase or decrease the variance of an instance that goes through it.
- variance of input = variance of output for a neuron
- Forward : requires $\tau = n_{\text{in}}$
- Backward : requires $\tau = n_{\text{out}}$

Compromise by Xavier Glorot & Joshua Bengio:

$$\tau = \frac{n_{\text{in}} + n_{\text{out}}}{2}$$

>>> Attacking MNIST!

Head to your notebook and...

- use `Keras` to write a 500x300 Neural Network
- train it and test it on MNIST

MNIST - checkpoint 1

```
model = Sequential()
model.add(Dense(500,input_shape=(784,)))
model.add(Activation('relu'))
model.add(Dense(300))
model.add(Activation('relu'))
model.add(Dense(10))
model.add(Activation('softmax'))
```

MNIST - checkpoint 2

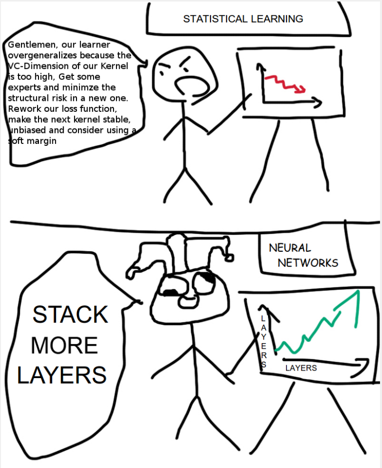
```
model.compile(loss='categorical_crossentropy',
              optimizer='adam', metrics=["accuracy"])

model.fit(images_train, labels_train,
          batch_size=100,
          epochs=10,
          verbose=2,
          validation_data = (images_test,labels_test))
```

Convolutional Neural Networks

- Introduction to deep learning
- The convolution operator
- Convolutional Neural Networks (CNNs)
- Regularisation
- Inspecting VGG-16

Introduction to **deep learning**



Going deep with neural networks

- Simple fully-connected neural networks (as described already) typically fail on high-dimensional datasets (e.g. images).
  - Treating each pixel as an independent input…

Going deep with neural networks

- Simple fully-connected neural networks (as described already) typically fail on high-dimensional datasets (e.g. images).
  - Treating each pixel as an independent input…
  - …results in $h \times w \times d$ new parameters per neuron in the first hidden layer…

Going deep with neural networks

- Simple fully-connected neural networks (as described already) typically fail on high-dimensional datasets (e.g. images).
  - Treating each pixel as an independent input…
  - …results in $h \times w \times d$ new parameters per neuron in the first hidden layer…
  - …quickly deteriorating as images become larger—requiring exponentially more data to properly fit those parameters!

Going deep with neural networks

- Simple fully-connected neural networks (as described already) typically fail on high-dimensional datasets (e.g. images).
  - Treating each pixel as an independent input…
  - …results in $h \times w \times d$ new parameters per neuron in the first hidden layer…
  - …quickly deteriorating as images become larger—requiring exponentially more data to properly fit those parameters!
- **Key idea:** downsample the image until it is small enough to be tackled by such a network!
  - Would ideally want to extract some useful features first…

Going deep with neural networks

- Simple fully-connected neural networks (as described already) typically fail on high-dimensional datasets (e.g. images).
  - Treating each pixel as an independent input...
  - ...results in $h \times w \times d$ new parameters per neuron in the first hidden layer...
  - ...quickly deteriorating as images become larger—requiring exponentially more data to properly fit those parameters!
- **Key idea:** downsample the image until it is small enough to be tackled by such a network!
  - Would ideally want to extract some useful features first...
- $\Longrightarrow$ exploit spatial structure !

The convolution operator

Enter the convolution operator

- Define a small (e.g. 3×3) matrix (the kernel, $\mathbf{K}$).
- Overlay it in all possible ways over the input image, $\mathbf{I}$.
- Record sums of elementwise products in a new image.

$$(\mathbf{I} * \mathbf{K})_{xy} = \sum_{i=1}^{h} \sum_{j=1}^{w} \mathbf{K}_{ij} \cdot \mathbf{I}_{x+i-1, y+j-1}$$

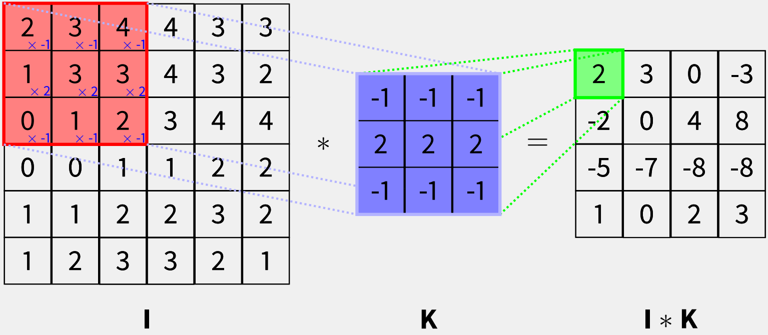
Enter the convolution operator

- Define a small (e.g. 3×3) matrix (the kernel, $\mathbf{K}$).
- Overlay it in all possible ways over the input image, $\mathbf{I}$.
- Record sums of elementwise products in a new image.

$$(\mathbf{I} * \mathbf{K})_{xy} = \sum_{i=1}^{h} \sum_{j=1}^{w} \mathbf{K}_{ij} \cdot \mathbf{I}_{x+i-1,y+j-1}$$

- This operator exploits structure—neighbouring pixels influence one another stronger than ones on opposite corners!

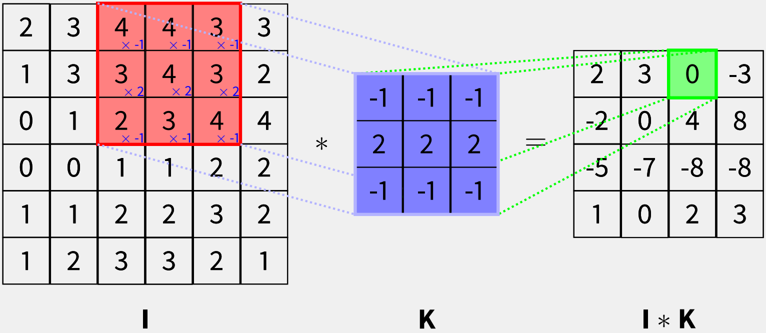
Convolution example



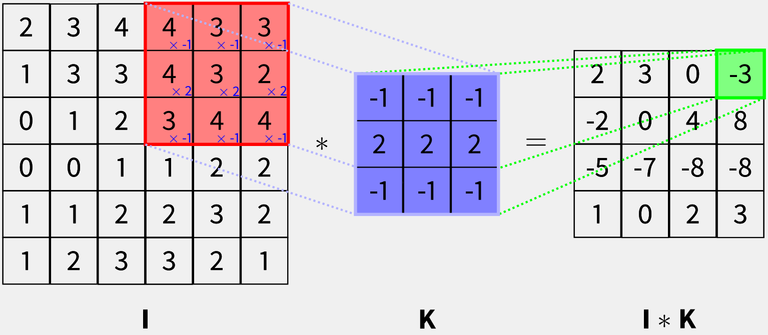
Convolution example

$$\underset{\mathbf{I}}{\begin{bmatrix} 2 & 3 & 4 & 4 & 3 & 3 \\ 1 & 3 & 3 & 4 & 3 & 2 \\ 0 & 1 & 2 & 3 & 4 & 4 \\ 0 & 0 & 1 & 1 & 2 & 2 \\ 1 & 1 & 2 & 2 & 3 & 2 \\ 1 & 2 & 3 & 3 & 2 & 1 \end{bmatrix}} * \underset{\mathbf{K}}{\begin{bmatrix} -1 & -1 & -1 \\ 2 & 2 & 2 \\ -1 & -1 & -1 \end{bmatrix}} = \underset{\mathbf{I} * \mathbf{K}}{\begin{bmatrix} 2 & 3 & 0 & -3 \\ -2 & 0 & 4 & 8 \\ -5 & -7 & -8 & -8 \\ 1 & 0 & 2 & 3 \end{bmatrix}}$$

Convolution example



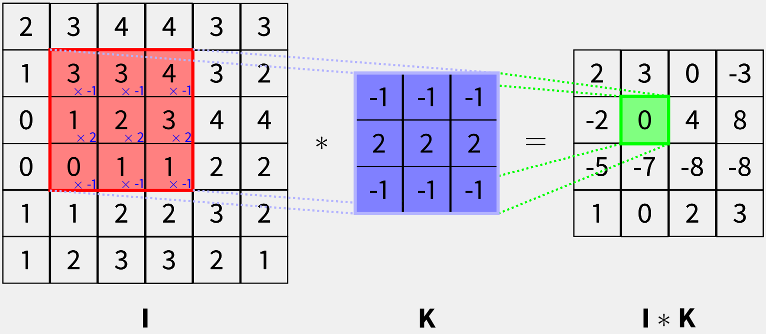
Convolution example



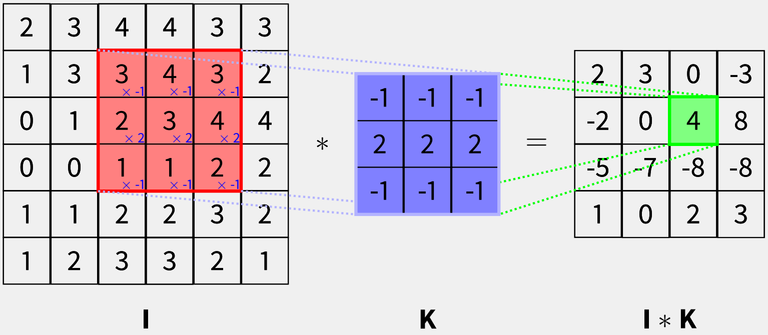
Convolution example

$$\underset{\mathbf{I}}{\begin{bmatrix} 2 & 3 & 4 & 4 & 3 & 3 \\ 1_{\times -1} & 3_{\times -1} & 3_{\times -1} & 4 & 3 & 2 \\ 0_{\times 2} & 1_{\times 2} & 2_{\times 2} & 3 & 4 & 4 \\ 0_{\times -1} & 0_{\times -1} & 1_{\times -1} & 1 & 2 & 2 \\ 1 & 1 & 2 & 2 & 3 & 2 \\ 1 & 2 & 3 & 3 & 2 & 1 \end{bmatrix}} * \underset{\mathbf{K}}{\begin{bmatrix} -1 & -1 & -1 \\ 2 & 2 & 2 \\ -1 & -1 & -1 \end{bmatrix}} = \underset{\mathbf{I} * \mathbf{K}}{\begin{bmatrix} 2 & 3 & 0 & -3 \\ \boxed{-2} & 0 & 4 & 8 \\ -5 & -7 & -8 & -8 \\ 1 & 0 & 2 & 3 \end{bmatrix}}$$

Convolution example



Convolution example



Convolution example

$$
\underbrace{\begin{bmatrix} 2 & 3 & 4 & 4 & 3 & 3 \\ 1 & 3 & 3 & 4 & 3 & 2 \\ 0 & 1 & 2 & 3 & 4 & 4 \\ 0 & 0 & 1 & 1 & 2 & 2 \\ 1 & 1 & 2 & 2 & 3 & 2 \\ 1 & 2 & 3 & 3 & 2 & 1 \end{bmatrix}}_{\mathbf{I}} * \underbrace{\begin{bmatrix} -1 & -1 & -1 \\ 2 & 2 & 2 \\ -1 & -1 & -1 \end{bmatrix}}_{\mathbf{K}} = \underbrace{\begin{bmatrix} 2 & 3 & 0 & -3 \\ -2 & 0 & 4 & 8 \\ -5 & -7 & -8 & -8 \\ 1 & 0 & 2 & \boxed{3} \end{bmatrix}}_{\mathbf{I} * \mathbf{K}}
$$

Convolution example (with padding)

0	0	0	0	0	0	0	0
0	2	3	4	4	3	3	0
0	1	3	3	4	3	2	0
0	0	1	2	3	4	4	0
0 × -1	0 × -1	0 × -1	1	1	2	2	0
0 × 2	1 × 2	1 × 2	2	2	3	2	0
0 × -1	1 × -1	2 × -1	3	3	2	1	0
0	0	0	0	0	0	0	0

I (padded)

$*$

-1	-1	-1
2	2	2
-1	-1	-1

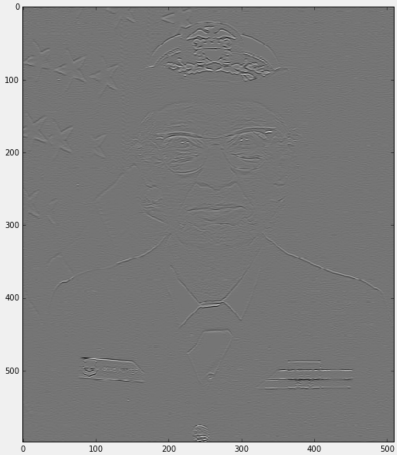
K

$=$

6	11	12	12	11	7
2	2	3	0	-3	-4
-2	-2	0	4	8	7
-3	-5	-7	-8	-8	-5
1	1	0	2	3	3
4	8	11	9	5	1

$\mathbf{I} * \mathbf{K}$

Applying convolutions

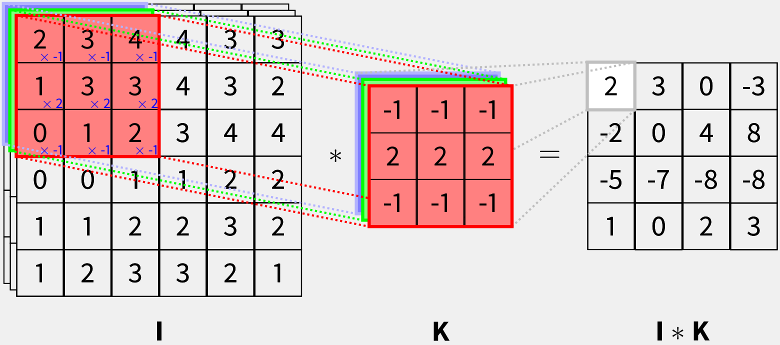


Applying convolutions

- Just by observing the convolved image, can you tell what kind of pattern the kernel detects?
- How would you design a kernel that detects vertical edges?
- What would the following kernel detect ?

```
kernel = np.array([[ 1,  1,  1],
                   [ 0,  0,  0],
                   [-1, -1, -1]])
```

Convolution with colour



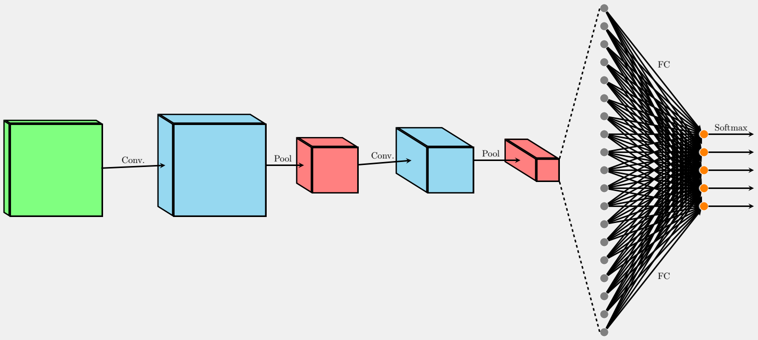
Convolution with colour

Convolution with colour

- Can you design a filter to detect the edge of Grace Hopper's left shoulder?

- [Hint: make sure the weights in your kernel add up to zero!]

Convolutional neural networks (CNNs)



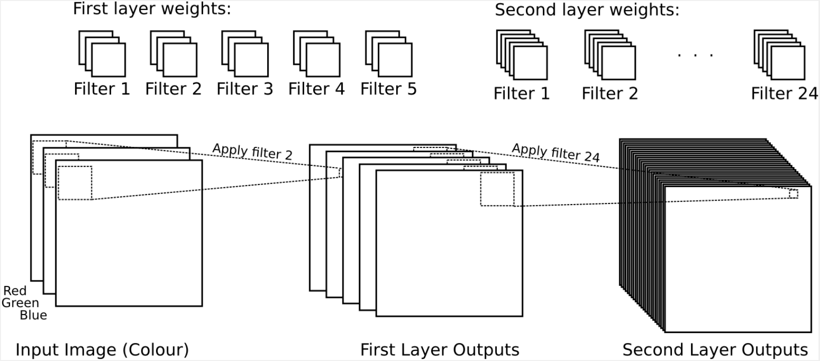
Convolutional layer

- A convolutional layer is specified by several kernels, to be applied to the output image of the previous layer.

- Convolving with one of the kernels (and potentially applying an activation function to every pixel) provides a single channel of the output image.

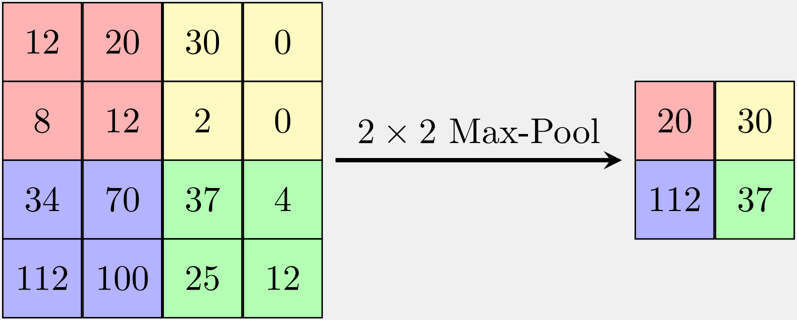
- Start with random kernels—let the network learn optimal ones by itself!
  - **N.B.** all we're doing is multiplying inputs by weights and adding them together $\implies$ we can learn in the same fashion as before!

Stacking convolutional layers



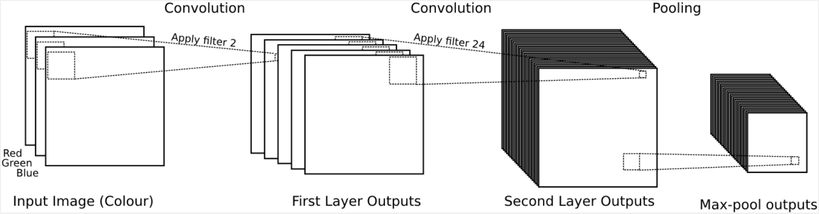

```
model.add(Conv2D(32, (3, 3),
      activation='relu',
       padding='same'))
```

Pooling layers

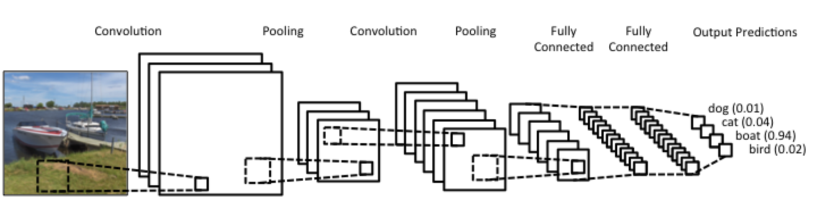



```
model.add(MaxPooling2D())
```

Stacking pooling layers

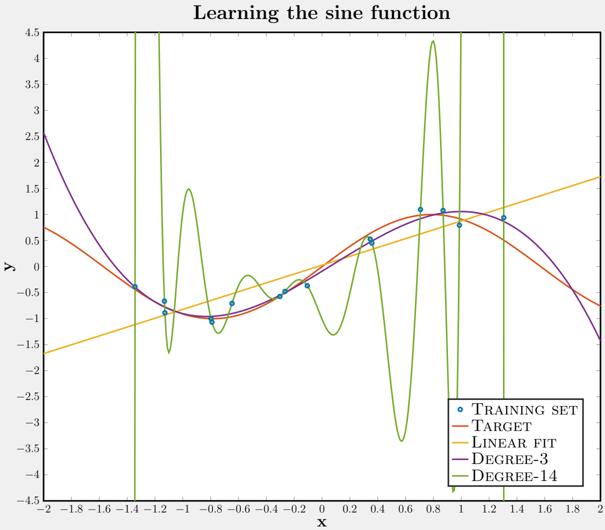


Putting it all together




```
model.add(Flatten())
model.add(Dense(256))
model.add(Activation('relu'))
```

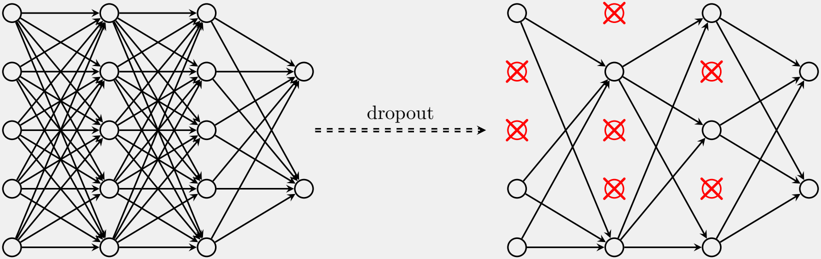
Regularisation



Why now?

- With previously covered networks and problems, overfitting tends not to become an issue.
- However, with CNNs and most image recognition problems, this becomes an extremely major issue!
- We will cover two "black magic" methods that are extremely good in practice…

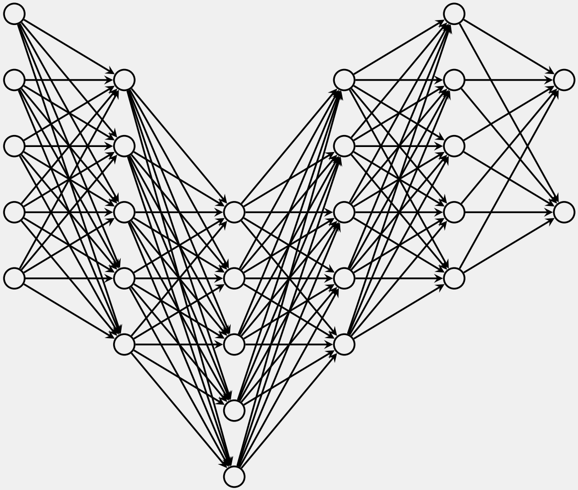
Dropout



- Randomly "kill" each neuron in a layer with probability p during training only...?!

```
model.add(Dropout(0.5))
```

Batch normalisation



```
model.add(BatchNormalization())
```

Batch normalisation

- Solution: renormalise outputs of the current layer across the current batch, $\mathcal{B} = \{x_1, \ldots, x_m\}$ (but allow the network to "revert" if necessary)!

$$\mu_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m} x_i \quad \sigma_{\mathcal{B}}^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2$$

$$\hat{x}_i = \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \varepsilon}} \qquad y_i = \gamma \hat{x}_i + \beta$$

where γ and β are trainable!

- Now ubiquitously used across deeper CNNs
  - Published in February 2015, $\sim$1600 citations by now!

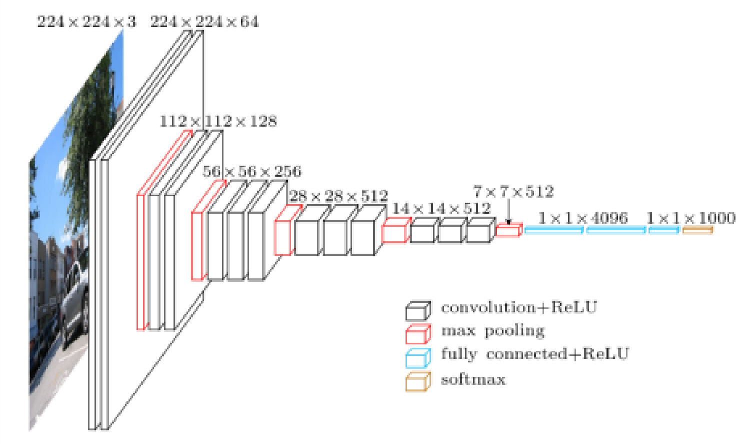
One last trick: data augmentation

```
datagen = ImageDataGenerator(
    width_shift_range=0.1,
    height_shift_range=0.1)
```

~~`model.fit(...)`~~

```
model.fit_generator(datagen.flow(
    X_train, y_train,
    batch_size=32),
  steps_per_epoch=len(X_train),
  epochs=100,
  validation_data=(X_test,
    y_test))
```

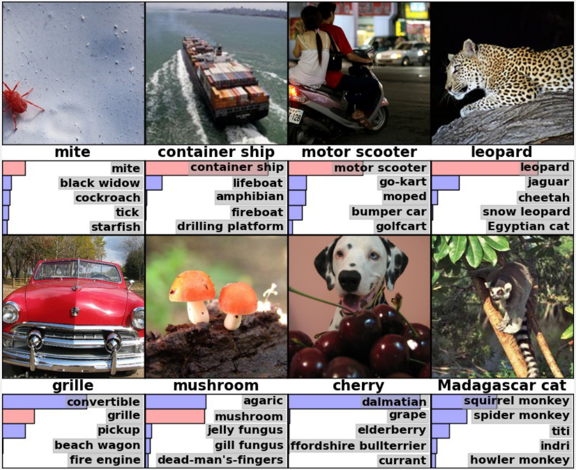
Inspecting VGG-16



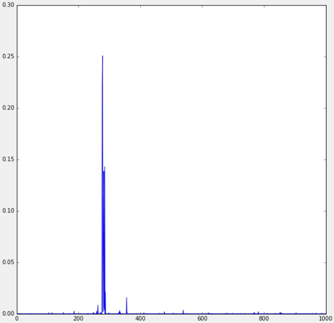
ImageNet

- A 1000-class image classification problem, with classes that are both very diverse (animals, transportation, people…) and very specific (100 breeds of dogs!).
- A state-of-the-art predictor needs to be very good at extracting features from virtually any image!
- Early success story of deep learning (2012); human performance ($\sim$ 94%) surpassed by a 150-layer neural network in 2015.
- Pre-trained models are readily available in deep learning libraries (such as Keras, which I will be using for all the demos).

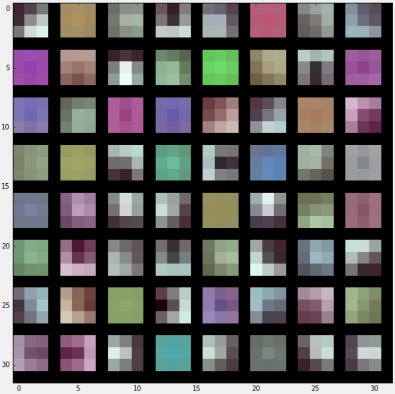
ImageNet classification

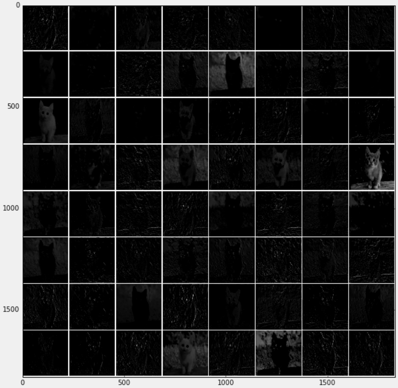


Passing data through the network



Looking inside: the first layer





Looking inside: deeper layers