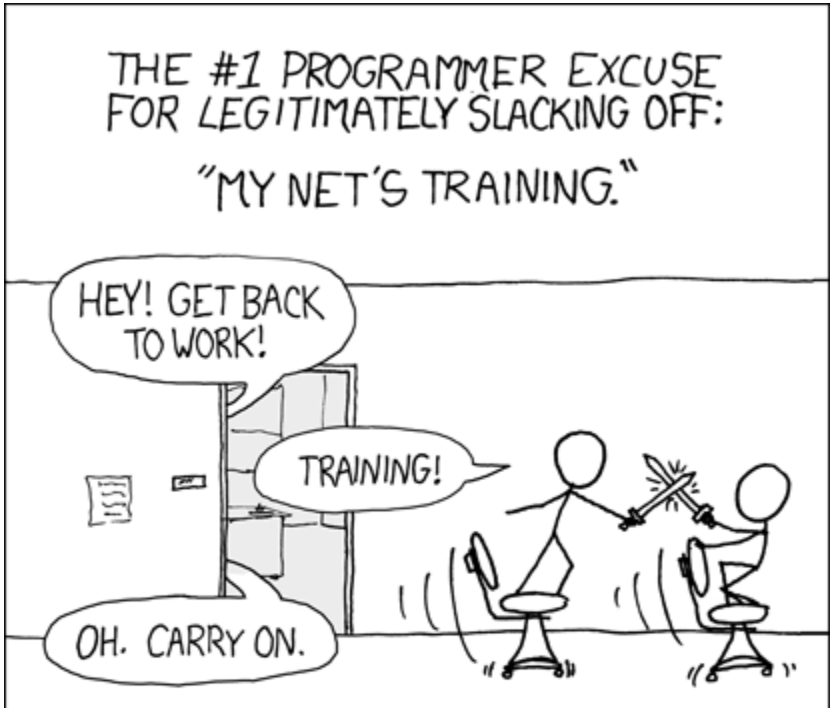
CMM560 Topic 7 - Convolutional Neural Networks



Aims of the Session

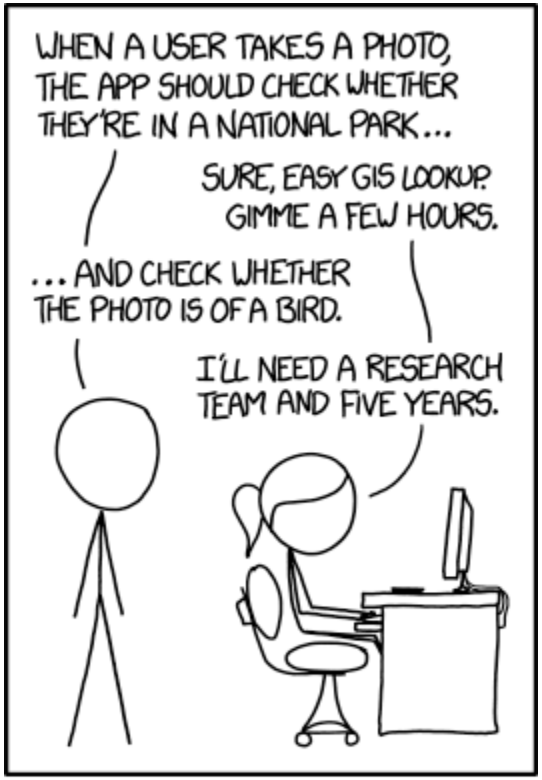
- Learn the particularities of Convolutional Neural Networks (CNNs)

Aims of the Session

- Learn the particularities of Convolutional Neural Networks (CNNs)
- Apply CNNs to image repositories in easy ways

Resources for the Lecture

- **Deep Learning with Python**. François Chollet. November 2017, ISBN 978161729443. Manning.
    - Very recommendable book, it was written by the author of `Keras`



IN CS, IT CAN BE HARD TO EXPLAIN THE DIFFERENCE BETWEEN THE EASY AND THE VIRTUALLY IMPOSSIBLE.

- For today's lecture (and for the lab as well) we will use the *Hello World!* of image datasets... **MNIST**

- For today's lecture (and for the lab as well) we will use the *Hello World!* of image datasets... **MNIST**
- This dataset contains 70'000 images (60k for training and 10k for testing) of handwritten numbers

- For today's lecture (and for the lab as well) we will use the *Hello World!* of image datasets... **MNIST**
- This dataset contains 70'000 images (60k for training and 10k for testing) of handwritten numbers
- The task is to recognise digits from 0 to 9 in 28×28 images

- For today's lecture (and for the lab as well) we will use the *Hello World!* of image datasets... **MNIST**

- This dataset contains 70'000 images (60k for training and 10k for testing) of handwritten numbers

- The task is to recognise digits from 0 to 9 in 28×28 images

- This dataset can be obtained ether by importing it through `Tensorflow` or `Keras`

In [ ]:

```
# Installing Tensorflow and Keras if not installed already
!pip install tensorflow==2.11.0
!pip install keras==2.11.0
```

In [1]:

```
# Import Keras with Tensorflow backend and download the dataset
import os
os.environ['KERAS_BACKEND'] = 'tensorflow'

from keras.datasets import mnist
(X_train, Y_train), (X_test, Y_test) = mnist.load_data()
```

```
In [1]: # Import Keras with Tensorflow backend and download the dataset
        import os
        os.environ['KERAS_BACKEND'] = 'tensorflow'

        from keras.datasets import mnist
        (X_train, Y_train), (X_test, Y_test) = mnist.load_data()

In [2]: print(X_train.shape,X_test.shape)

        (60000, 28, 28) (10000, 28, 28)
```

- Notice that contrary to my recommendation, this dataset is stored as a 3D matrix

- Notice that contrary to my recommendation, this dataset is stored as a 3D matrix
- This means that the dataset has 60k train/10k test rows, each one with a 28×28 image!

- Notice that contrary to my recommendation, this dataset is stored as a 3D matrix
- This means that the dataset has 60k train/10k test rows, each one with a 28×28 image!
- This is for us to visualise the samples better (afterwards you will see that images need to be flattened to be used)

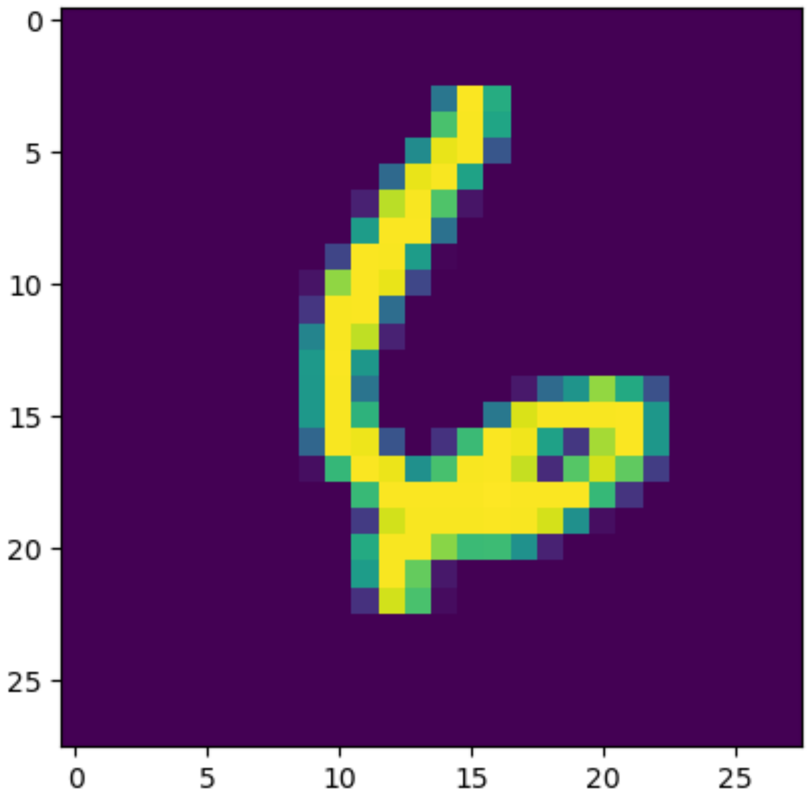
In [3]:

```python
import matplotlib.pyplot as plt

sample = 59000

print(X_train[sample].shape)
print('The number is: '+str(Y_train[sample]))
plt.imshow(X_train[sample])
```

```
(28, 28)
The number is: 6
```

Out[3]:

```
<matplotlib.image.AxesImage at 0x17c11cec2e0>
```


0
5
10
15
20
25
0
5
10
15
20
25

The convolutional operation

The convolutional operation

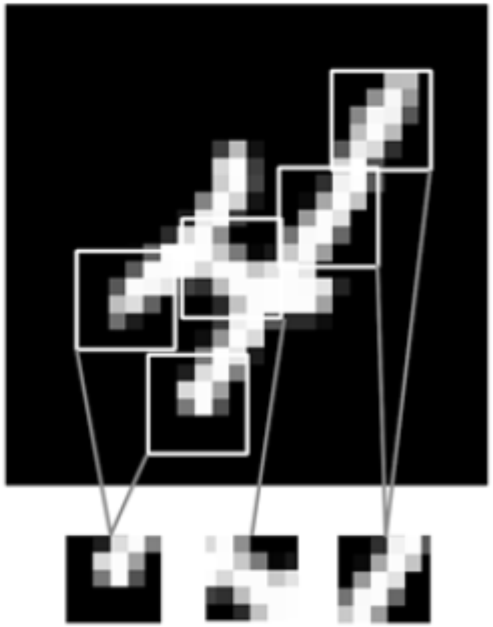
- Two types of layers:

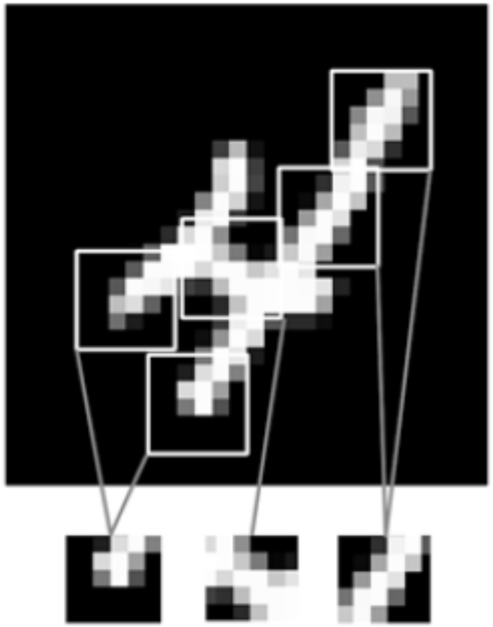
The convolutional operation

- Two types of layers:
- `Dense` layers learn global patterns

The convolutional operation

- Two types of layers:
- `Dense` layers learn global patterns
- `Conv` learn local patterns



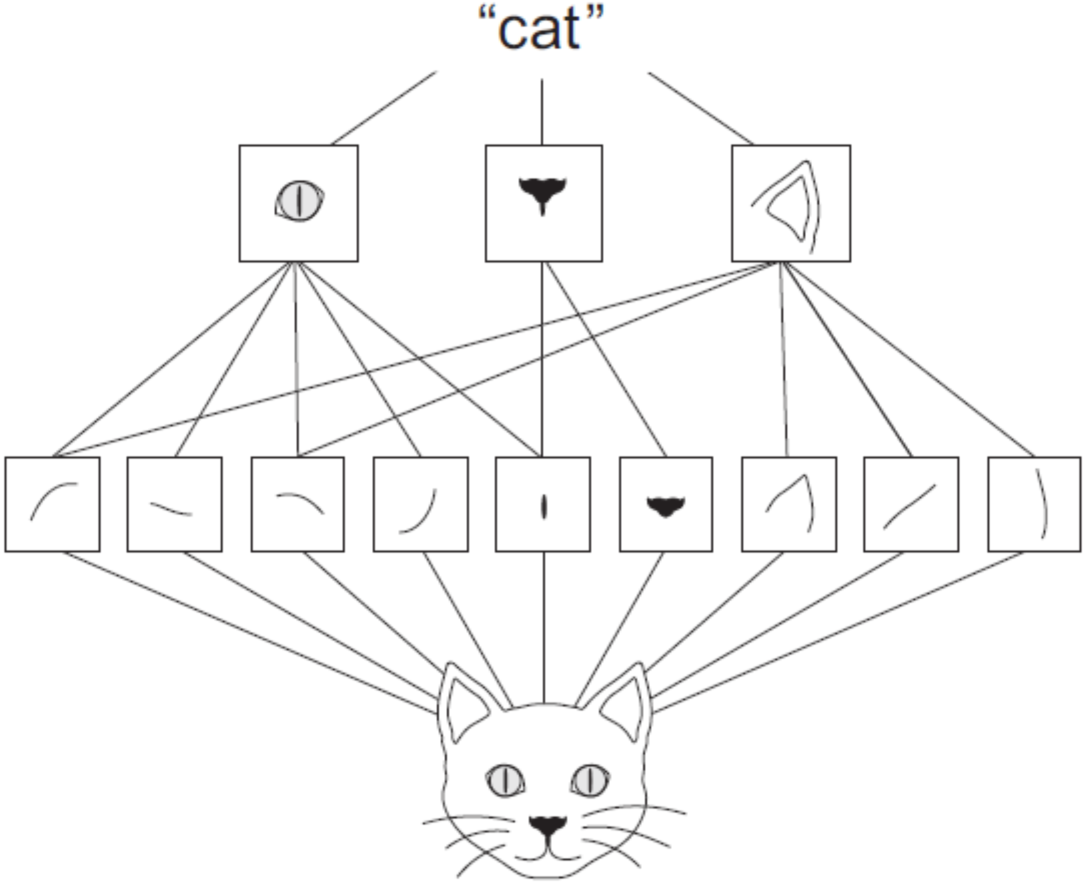


- CNNs not only classify, but also extract their own features!

- The patterns/features that CNNs learn are **translation invariant**
    - Once it learns it, it can recognise it anywhere in the image

- The patterns/features that CNNs learn are **translation invariant**
    - Once it learns it, it can recognise it anywhere in the image

- They can learn **spatial hierarchies** of patterns
    - Each layer learns different type of features
        - First layer learns edges, second learns larger patterns, and so on


"cat"

How do CNNs learn?

How do CNNs learn?

- The images of the **MNIST** dataset are all 28×28 pixels

How do CNNs learn?

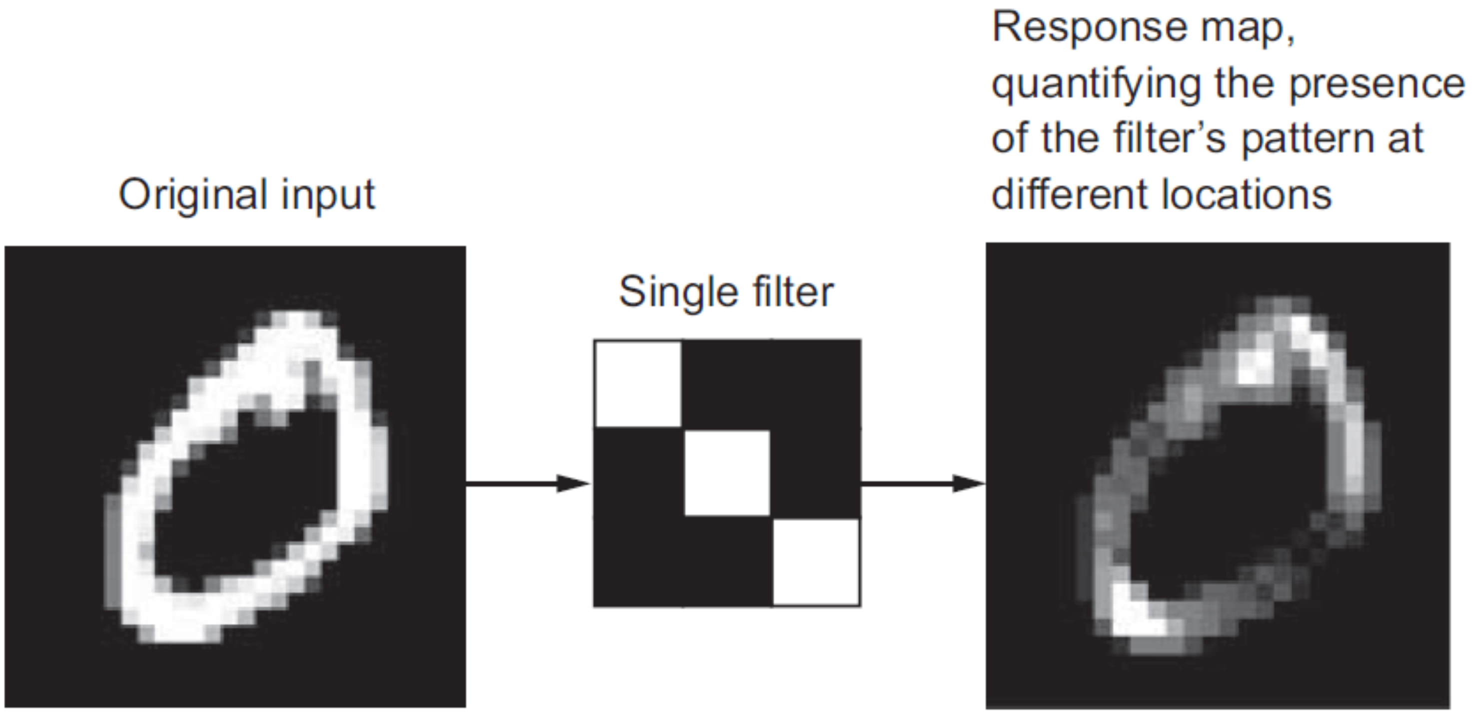
- The images of the **MNIST** dataset are all 28×28 pixels
- The first **convolution** layer takes a $(28, 28, 1)$ image and outputs a **feature map** of size $(26, 26, 32)$

How do CNNs learn?

- The images of the **MNIST** dataset are all 28×28 pixels
- The first **convolution** layer takes a $(28, 28, 1)$ image and outputs a **feature map** of size $(26, 26, 32)$
- This is because it computes 32 **filters** over the input!

How do CNNs learn?

- The images of the **MNIST** dataset are all 28×28 pixels
- The first **convolution** layer takes a $(28, 28, 1)$ image and outputs a **feature map** of size $(26, 26, 32)$
- This is because it computes 32 **filters** over the input!
- That means that after the first layer, the network transforms the training images into 32 output channels, each containing a 26×26 filter, which is a **response map**


Response map, quantifying the presence of the filter's pattern at different locations
Original input
Single filter

- A response map is the response of a filter at different locations of the input

- A response map is the response of a filter at different locations of the input
- This is how CNNs extract features; by applying filters over the images and finding responses to them!

Why is the response map 26×26?

Why is the response map 26×26?

Why 32 filters?

Basic Parameters of CNNs

Basic Parameters of CNNs

- **Size of the patches extracted:** Typically 3×3 or 5×5. In the example above, you can see 3×3

Basic Parameters of CNNs

- **Size of the patches extracted:** Typically 3×3 or 5×5. In the example above, you can see 3×3
- **Depth of the output feature map:** Number of filters. This can change, i.e. can start with a *depth* of 32 and finish with 64

In `Keras`, you can import a `Conv2D` layer, to which you can pass these values

In `Keras` , you can import a `Conv2D` layer, to which you can pass these values

- Conv2D(output_depth,(window_height, window_width))

How do Convolution Layers Work?

How do Convolution Layers Work?

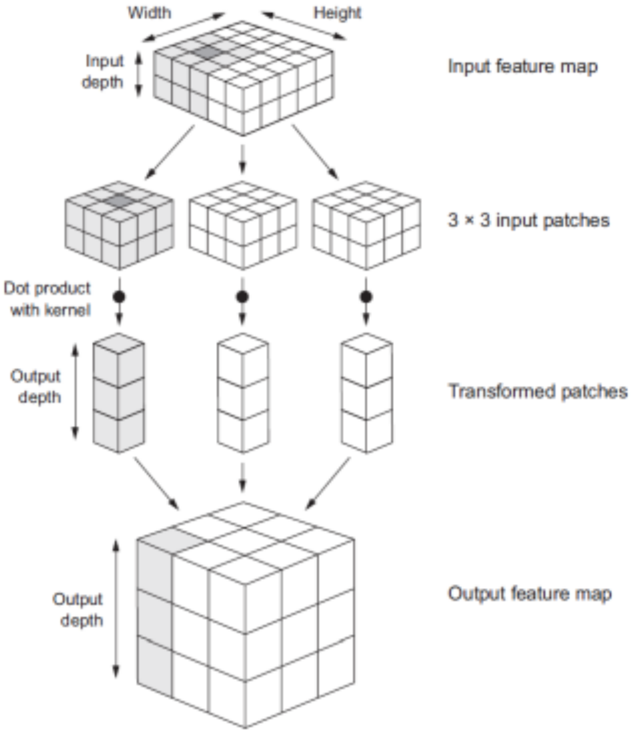
- A convolution layer works by sliding the patches over the 3D input map, stopping at every location and extracting the 3D patch

How do Convolution Layers Work?

- A convolution layer works by sliding the patches over the 3D input map, stopping at every location and extracting the 3D patch
- Each of these patches is flattened into a 1D vector of size (`output_depth` ,)

How do Convolution Layers Work?

- A convolution layer works by sliding the patches over the 3D input map, stopping at every location and extracting the 3D patch
- Each of these patches is flattened into a 1D vector of size (`output_depth` ,)
- The process can be best illustrated using the following figure:


Width
Height
Input depth
Input feature map
3 × 3 input patches
Dot product with kernel
Output depth
Transformed patches
Output depth
Output feature map

Border Effect and Strides

Border Effect and Strides

- Notice that we started with a $(28, 28, 1)$ image and we ended with a $(26, 26, 32)$ **feature map**!

Border Effect and Strides

- Notice that we started with a $(28, 28, 1)$ image and we ended with a $(26, 26, 32)$ **feature map**!
- Two main reasons for not having the same width & height:
  - The **border effect** (which can be countered by applying `padding`)
  - The use of **strides**

The Border Effect & Padding

The Border Effect & Padding

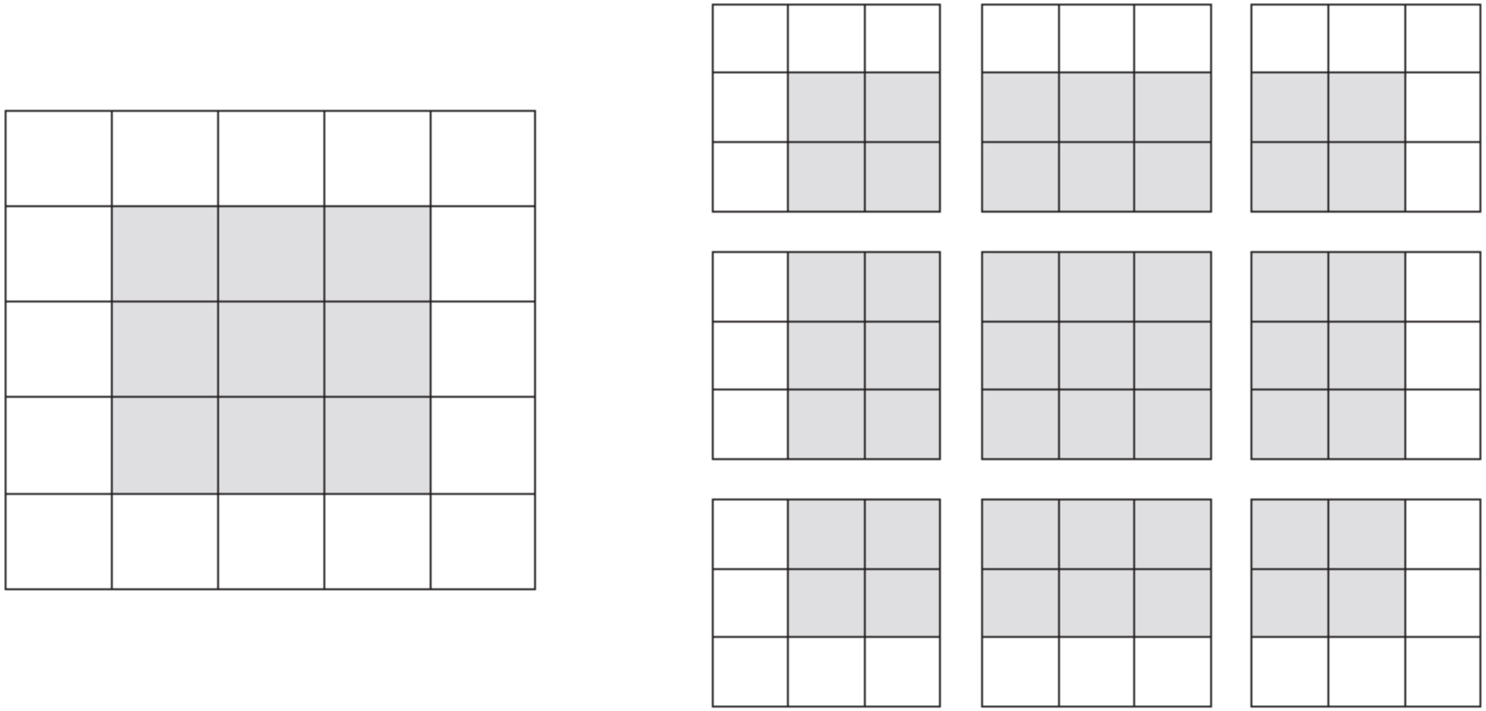
- Remember that a CNN uses a convolution layer that applies a filter for each position of the image, similar to sliding a window throughout the image

The Border Effect & Padding

- Remember that a CNN uses a convolution layer that applies a filter for each position of the image, similar to sliding a window throughout the image
- By nature, this sliding window cannot be centered exactly throughout the entire image!

The Border Effect & Padding

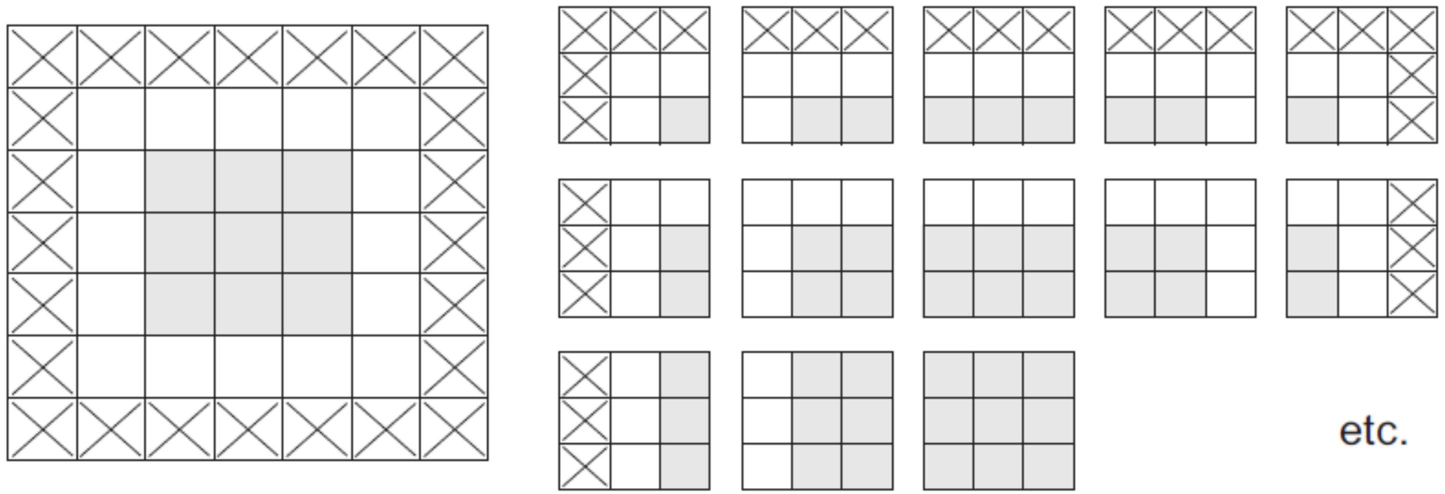
- Remember that a CNN uses a convolution layer that applies a filter for each position of the image, similar to sliding a window throughout the image
- By nature, this sliding window cannot be centered exactly throughout the entire image!
- For instance, in a 5×5 feature map, you could only center a 3×3 window in 9 positions as shown in the image below:



- If you want to get an output with the same size as the input, you need to apply `padding`

- If you want to get an output with the same size as the input, you need to apply `padding`
- In short, this is like adding a *margin* to the image so that the filter can be centered in more positions!

- If you want to get an output with the same size as the input, you need to apply `padding`
- In short, this is like adding a *margin* to the image so that the filter can be centered in more positions!
- In the next figure you can see how the 3×3 filter can be located in 25 positions now, thus delivering a 5×5 output


etc.

- In the `Conv2D` function in `Keras`, padding is enabled by setting the parameter `padding = 'valid'`

Striding

Striding

- When images are considerably large, you don't want the sliding window to stop at every position!

Striding

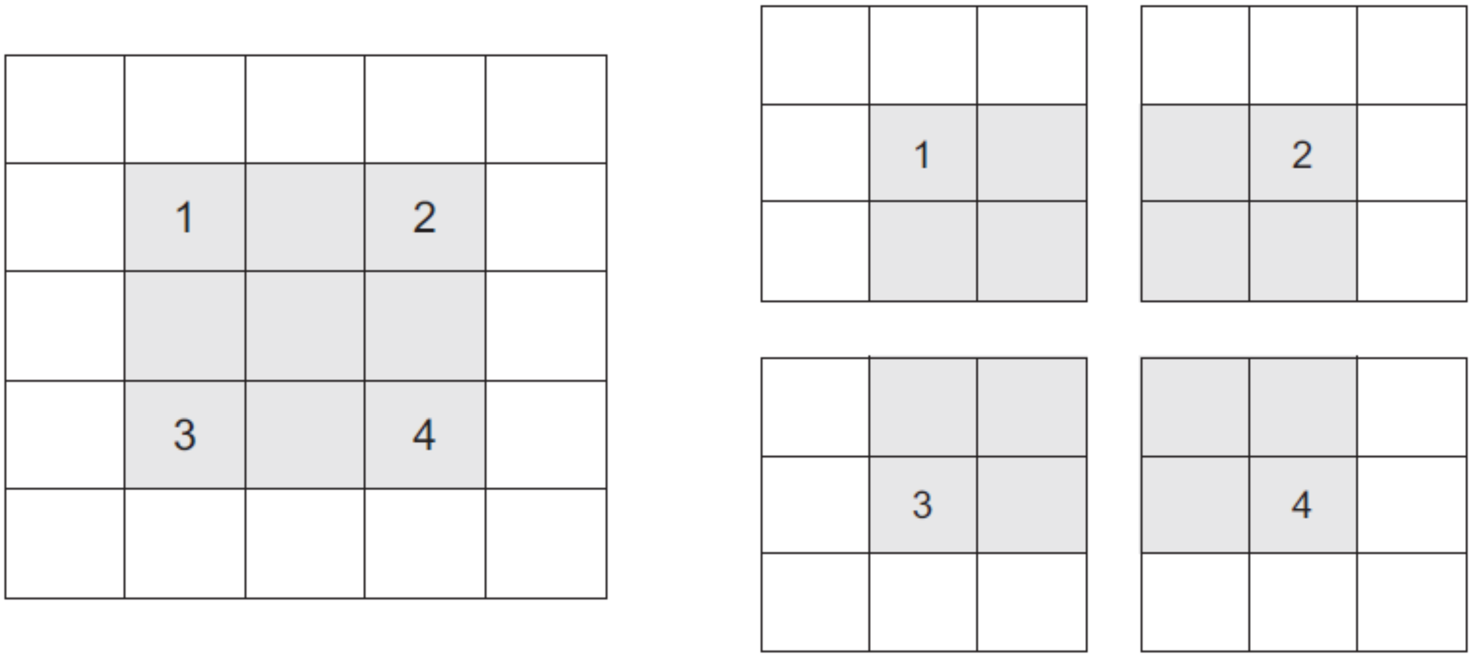
- When images are considerably large, you don't want the sliding window to stop at every position!
- You may introduce a parameter called `stride` which allows your convolution window to skip positions

Striding

- When images are considerably large, you don't want the sliding window to stop at every position!
- You may introduce a parameter called `stride` which allows your convolution window to skip positions
- A `stride=1` stops in every position, but for instance `stride=2` will make the filter to move with a step of 2, this skipping half of the positions!

Striding

- When images are considerably large, you don't want the sliding window to stop at every position!
- You may introduce a parameter called `stride` which allows your convolution window to skip positions
- A `stride=1` stops in every position, but for instance `stride=2` will make the filter to move with a step of 2, this skipping half of the positions!
- Recall the example presented before. Without considering padding, the 3×3 filter will only stop at four positions of the image


1
2
3
4
1
2
3
4

- This is rarely used in practice (people don't want to lose important features in between the image)

- This is rarely used in practice (people don't want to lose important features in between the image)
- It is more recommended to use `Max Pooling`

Max Pooling

Max Pooling

- You can add to your model a `Max Pooling` layer which halves your feature map

Max Pooling

- You can add to your model a `Max Pooling` layer which halves your feature map
- Max pooling extracts windows from the input much like a convolution

Max Pooling

- You can add to your model a `Max Pooling` layer which halves your feature map
- Max pooling extracts windows from the input much like a convolution
- Why should we use it? Imagine a CNN with no max pooling:

```
model_no_max_pool = models.Sequential()
model_no_max_pool.add(layers.Conv2D(32, (3, 3), activation='relu',
                      input_shape=(28, 28, 1)))
model_no_max_pool.add(layers.Conv2D(64, (3, 3), activation='relu'))
model_no_max_pool.add(layers.Conv2D(64, (3, 3), activation='relu'))
```

We can print the summary of the model with the following code:

We can print the summary of the model with the following code:

```
>>> model_no_max_pool.summary()

Layer (type)                 Output Shape              Param #
=================================================================
conv2d_4 (Conv2D)            (None, 26, 26, 32)        320
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 24, 24, 64)        18496
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 22, 22, 64)        36928
=================================================================
Total params: 55,744
Trainable params: 55,744
Non-trainable params: 0
```

- Notice that the number of parameters grow drastically after each layer
    - The final feature map (the one with $22 \times 22 \times 64 = 36'928$ parameters) has to be `flattened` and then a `Dense` layer has to be applied, resulting in 15 million parameters!

- Notice that the number of parameters grow drastically after each layer
    - The final feature map (the one with $22 \times 22 \times 64 = 36'928$ parameters) has to be `flattened` and then a `Dense` layer has to be applied, resulting in 15 million parameters!

- However, the model isn't learning a **hierarchy** of features! This means that as layers progress, the CNN would get smaller and smaller images and thus will be unable to learn the features

- Notice that the number of parameters grow drastically after each layer
    - The final feature map (the one with $22 \times 22 \times 64 = 36'928$ parameters) has to be `flattened` and then a `Dense` layer has to be applied, resulting in 15 million parameters!

- However, the model isn't learning a **hierarchy** of features! This means that as layers progress, the CNN would get smaller and smaller images and thus will be unable to learn the features

- You need the last layer to contain information about the whole image

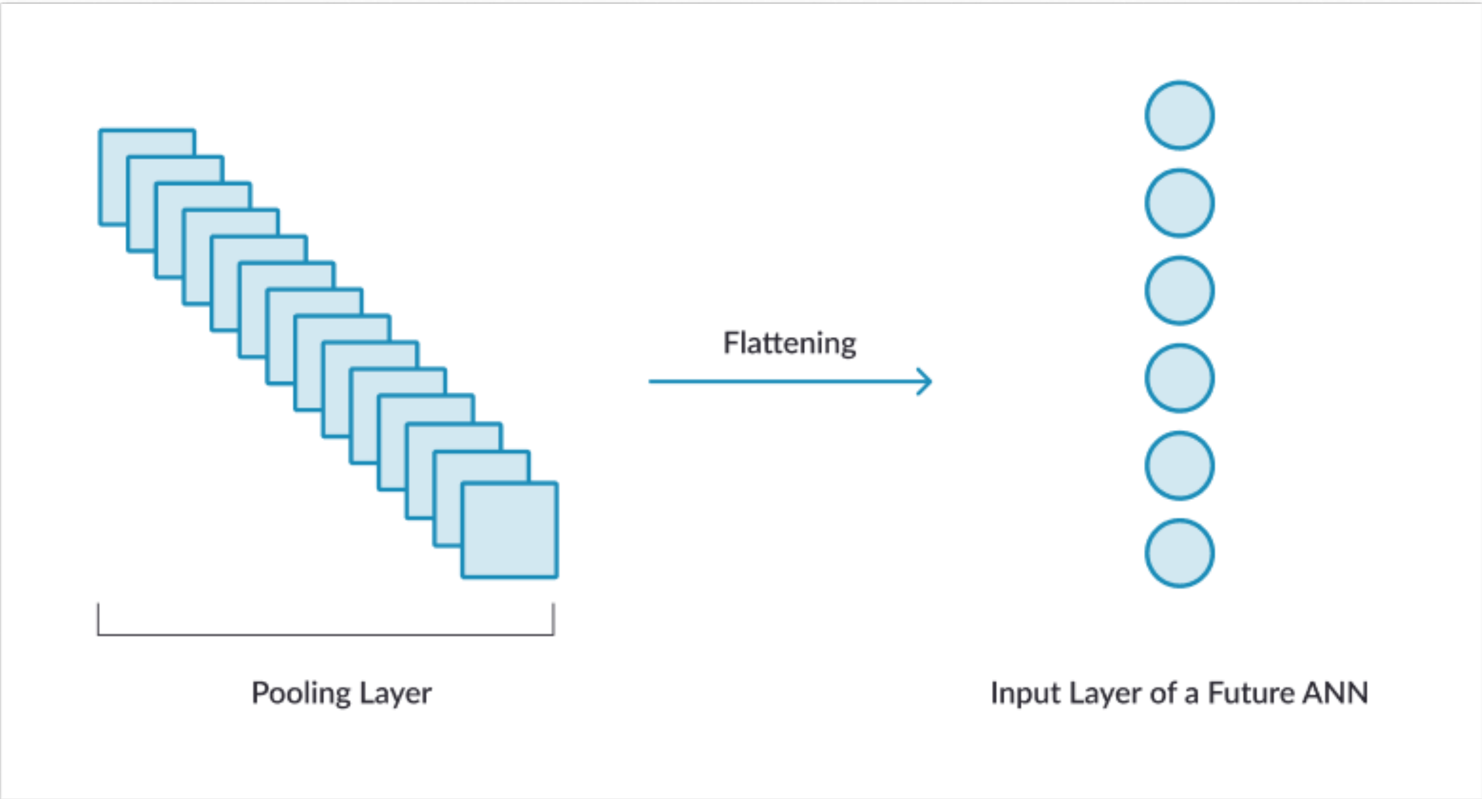
The Flatten Layer

The Flatten Layer

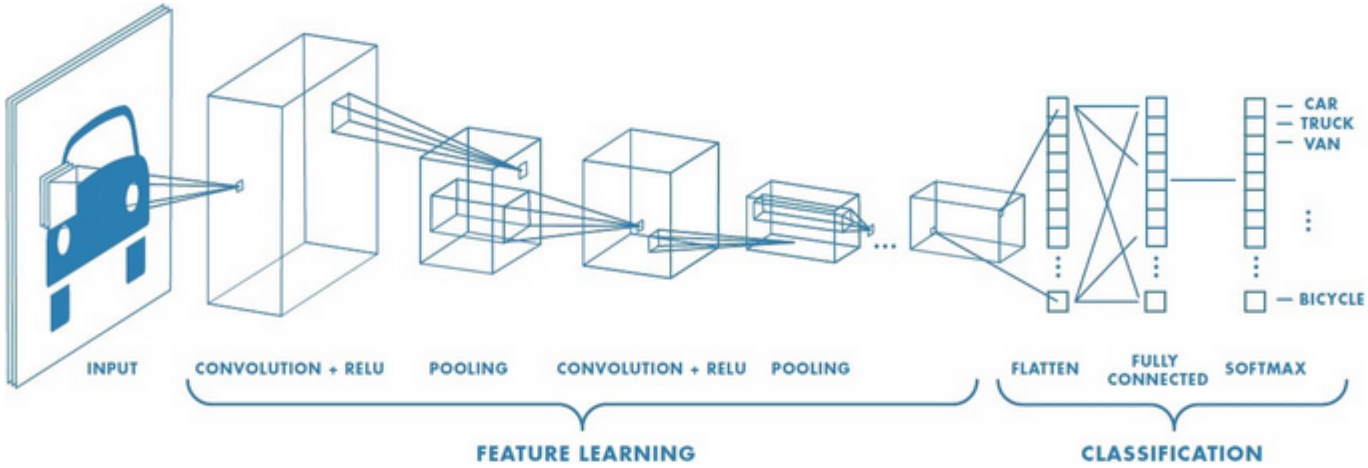
- Not to be confused with flattening an image!

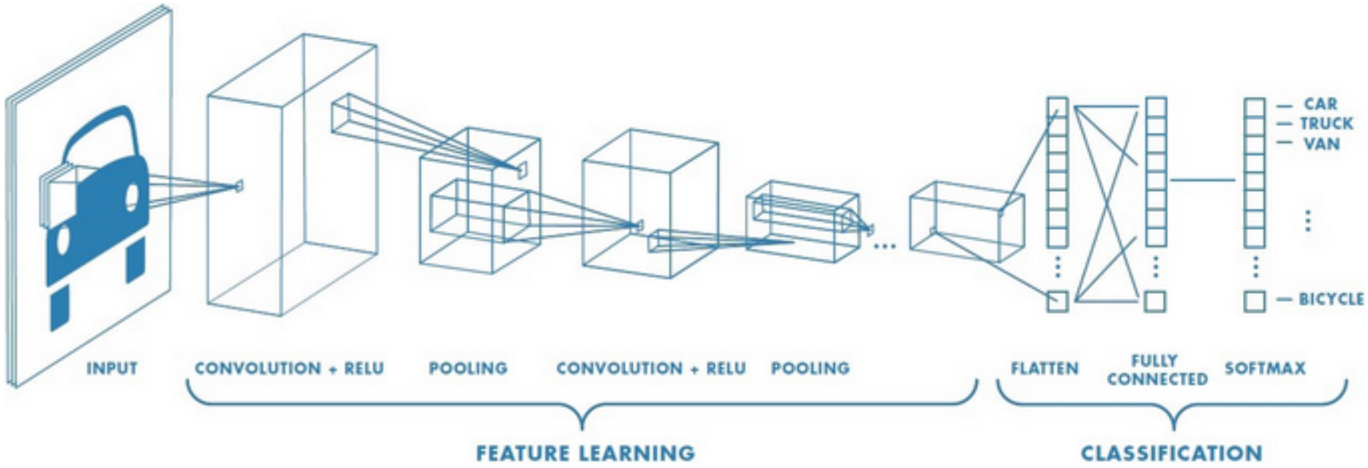
The Flatten Layer

- Not to be confused with flattening an image!
- After a convolution layer and before a `Dense` (i.e. fully connected layer), there is a `flatten` layer that transforms the matrix feature maps into vectors for the `Dense` layer to operate


Flattening
Pooling Layer
Input Layer of a Future ANN

Final example of a CNN


CAR
TRUCK
VAN
BICYCLE
INPUT
CONVOLUTION + RELU
POOLING
CONVOLUTION + RELU
POOLING
FLATTEN
FULLY CONNECTED
SOFTMAX
FEATURE LEARNING
CLASSIFICATION



I recommend you to also read this source if you have questions regarding any of the steps

Other useful concepts

- The following are not exclusive to CNNs, and are widely used in all NNs to further improve performance

Dropout

Dropout

- Reduces overfitting

Dropout

- Reduces overfitting
- It is rare that you use a fully connected layer as this may extract features that are too related to each other!

Dropout

- Reduces overfitting
- It is rare that you use a fully connected layer as this may extract features that are too related to each other!
- You implement a dropout percent per layer. This randomly disconnects layers from the previous layer into the current one

Epochs

Epochs

- Machine learning has by default an iterative nature (recall Gradient Descent)
    - The more you iterate things, the "better"!

Epochs

- Machine learning has by default an iterative nature (recall Gradient Descent)
    - The more you iterate things, the "better"!
- An epoch occurs when the entire dataset is passed forward and backward through the network once.

Epochs

- Machine learning has by default an iterative nature (recall Gradient Descent)
    - The more you iterate things, the "better"!
- An epoch occurs when the entire dataset is passed forward and backward through the network once.
- By passing the dataset multiple times, you can further reduce the loss and increase the training/validation accuracy

The more epochs, the better?

Batch Size

Batch Size

- Imagine training a network with the MNIST dataset

Batch Size

- Imagine training a network with the MNIST dataset
- You would need to pass 60k images in each epoch ! This would take a while!

Batch Size

- Imagine training a network with the MNIST dataset
- You would need to pass 60k images in each epoch ! This would take a while!
- You can set a `batch_size` to pass your data in chunks

Batch Size

- Imagine training a network with the MNIST dataset
- You would need to pass 60k images in each epoch ! This would take a while!
- You can set a `batch_size` to pass your data in chunks
- This may have an effect on your training results if the sequence of batches is not properly set
    - i.e. if you only pass batches from the negative class first, and then the positive one, your classifier may get biased towards the first class before being able to learn from the second.

The ADAM Optimiser

The ADAM Optimiser

- A faster way to optimise gradient descent compared to the more classical approaches

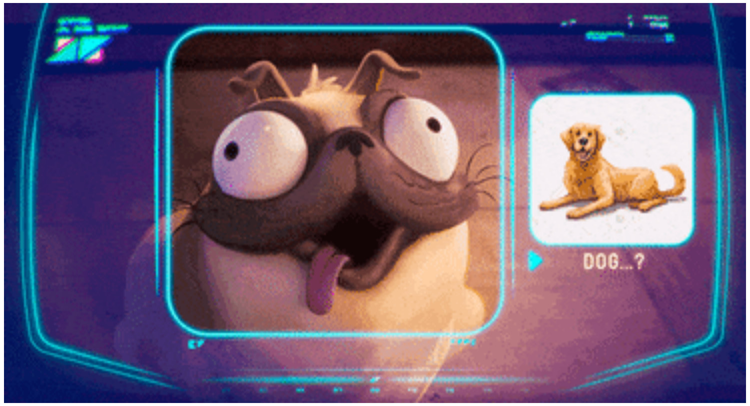
The ADAM Optimiser

- A faster way to optimise gradient descent compared to the more classical approaches
- It may obtain worse results, but compensates with speed!

The ADAM Optimiser

- A faster way to optimise gradient descent compared to the more classical approaches
- It may obtain worse results, but compensates with speed!
- Combination of RMSprop and Stochastic Gradient Descent with momentum. More info here


Daddy
Wecodeinpython
Daddy
Daddy
Daddy
whyatt


DOG...?

In [4]:

```
import warnings;
warnings.simplefilter('ignore')
from IPython.display import HTML
HTML('<iframe width="560" height="315" src="https://www.youtube.com/embed/vIc
```

Out[4]: