# AI Academy - Software Eng. Part B: DL + MLOps Course

Daniel Hen | July 2021

# A word about myself

- Data Scientist for **3** years @ **Fyber**
- Previously Software Developer
- Dealing mostly with tabular data, but also love Computer Vision and developed a crazy app with it, which you all can visit here
- Think that overall - continuously learning is the key for success!
- Linkedin | GitHub | Medium

# Small notes and logistics

- Breaks: 1 hour for lunch time, small ones here and there, when we need

  **Important** - it is really not easy to learn all day long

- Lectures will be held in a way that you will be exposed to both **theory** and **practical exercises**
- At the end of each topic, day, we will leave some time for Q&A, **you can ask everything, and if I don’t know it - I will get you an answer :-)**
- Most of the course content will be available via Jupyter Notebooks + this presentation

# Project

- You will do a project which you can eventually share it with your GitHub, colleagues and professional network!
- It will contain materials + tech we have learned during the course
    - Weights & Biases
    - Flask API
    - Docker
    - Deep Learning model training, evaluation, saving in ONNX

# Schedule

Overall, we have 5 days (**July 13th, 18th, 20th, 25th, 27th**)

- Day 1 (July 13th) - Course Intro, Intro to DL
- Day 2 (July 18th) - Intro to DL (CNN, Regularization, Activation Functions)
- Day 3 (July 20th) - Intro to DL (leftovers), Project Presentation
- Day 4 (July 25th) - DL leftovers, W&B, ONNX & Docker, Flask API
- Day 5 (July 27th) - Leftovers from Day 4, Project work time, Q&A time

# Intro to Deep Learning

# Let's discuss Deep Learning

- Artificial Neural Network
  - Neuron
  - Input, Hidden, Output layers
  - Backpropagation
  - Gradient Descent
  - Loss function
- FC Neural Network(s) - (in Keras)
- Convolutional Neural Network(s) (in Keras)
- Overfitting, Underfitting and other phenomena in DL(?)

# Motivation #1

***So, what exactly is deep learning ?***

***And, why is it generally better than other methods on image, speech and certain other types of data?***

*'Deep Learning' means using a neural network with several layers of nodes between input and output*

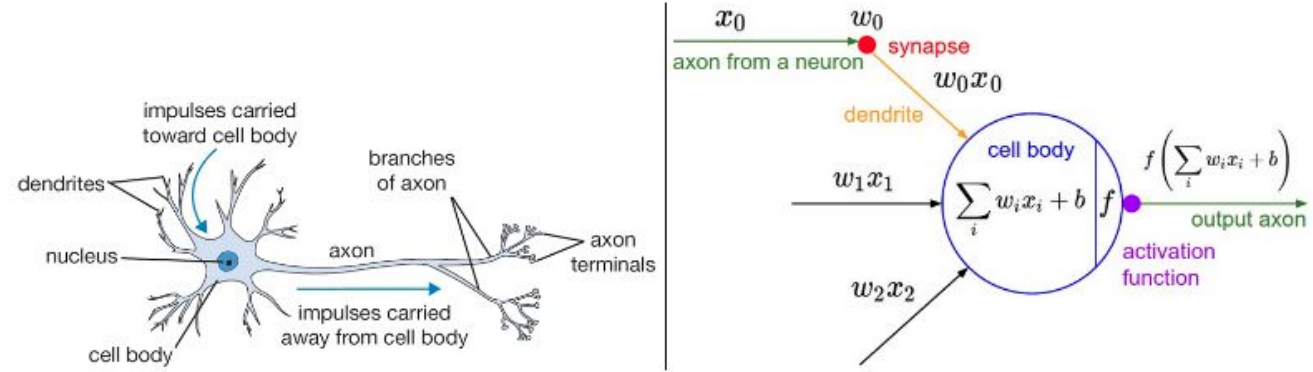
*The series of layers between input & output do feature identification and processing in a series of stages, just as our **brains** seem to*

# Motivation #2

- DL has become one of the hottest fields in the current day and became very popular!
- Eventually, it is very easy to implement DL today, as there are many frameworks - PyTorch, Tensorflow (with Keras), Simply NumPy, etc.
- Just implementing this without knowing the concepts is like playing with a black box - you know it's working, you don't know how!
- Whether it's for your own usage, industry / academia, you will be tweaking, changing, building DL models, so you should definitely be aware of what's going on, as much as possible, including the math.
- In this course, we'll dive into Neural Networks by providing step by step explanation, so you'll be able to grasp it.

# What is a Neuron?

(hint: gets input, does magic, produces output)




A cartoon drawing of a biological neuron (left) and its mathematical model (right).

Source: CS231n

As can be seen, a neuron is a function which gets inputs from several “axon’s & synapse’s”, aggregates it (SUM in this case), and produces another output, which is being used by the neuron after (as an input). It is working serially

# So, eventually - Neuron = Aggregation of data!

The base notation stands behind the fact that a neuron is a function, which:

1. gets some data
2. Does a manipulation on top of it
3. Outputs data that will represent in some way the input

The basic notation works has some “high-school math” ideas. That is:

***Z = w0 + x1 * w1 + x2 * w2 + … + xn * wn***

# Let's see an example of that "function"

| A | B | AND |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

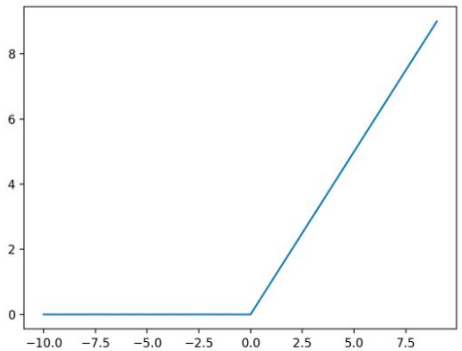
Given inputs A, B, we want to perform some aggregation so we will "collect" all of the signals from them. A slightly simple example can be:

```
def and_neuron(a, b):
    w0 = -0.5
    w1 = 0.6
    w2 = 0.4
    z = w0 + w1 * a + w2 * b // -0.5 + 0.6*a + 0.4*b
    thresh = lambda x: 1 if x>= 0.5 else 0 // what if we use different things?
    r = thresh(z)
    print(r)
```

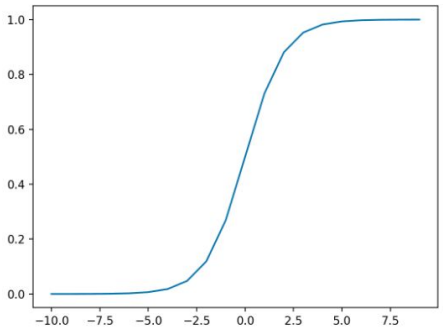
# Activation Functions

Activation functions are a critical part of the design of a neural network.
The choice of activation function in the **hidden layer** will control how well the network model learns the training dataset.
The choice of activation function in the **output layer** will define the type of predictions the model can make.
As such, a careful choice of activation function must be made for each deep learning neural network project.

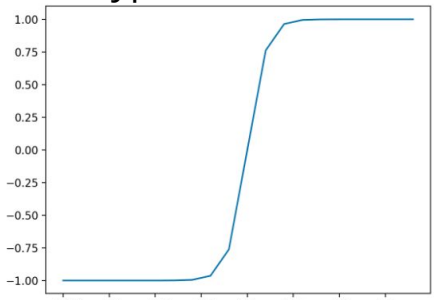
Relu




Sigmoid




Hyperbolic Tan



# Class Ex. 1

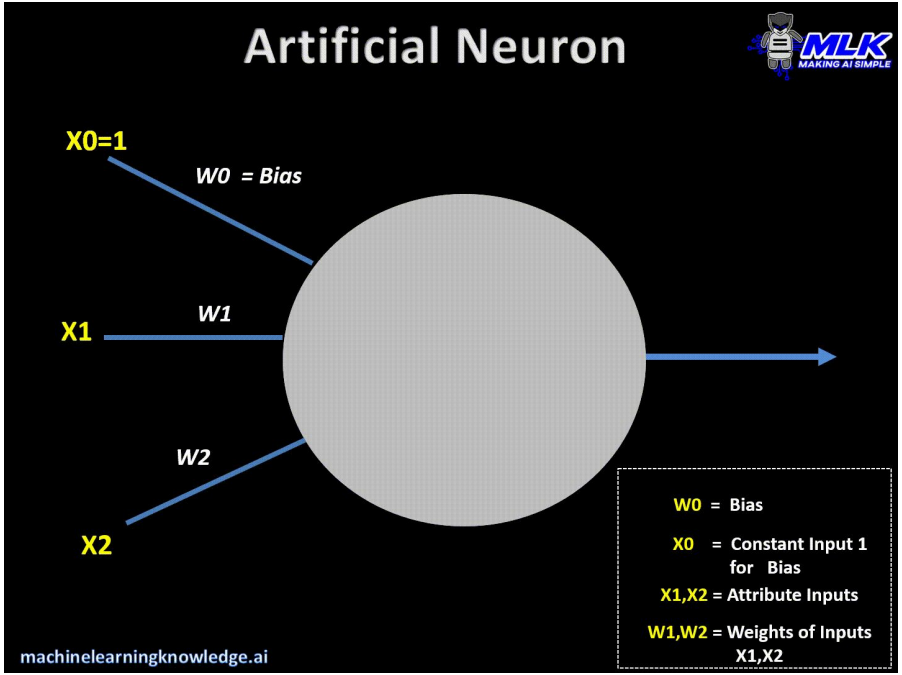
Let’s practice coding in NumPy!

The goal is as follows:
Code a neuron which will implement the above, and will try out the relevant Activation Functions we’ve been going through

Notebook Name: ***Daniel_NumPy_NN_Ex1_Neuron.ipynb***

Good Luck!

# Wrap up Forward Pass - Full Example



# Q&A Time

# Different layers, Same network

**Elements of a Neural Network**

Input Layer

This layer accepts input features.
It provides information from the **outer world** to the network, no computation is performed at this layer, nodes here just pass on the information(features) to the hidden layer.

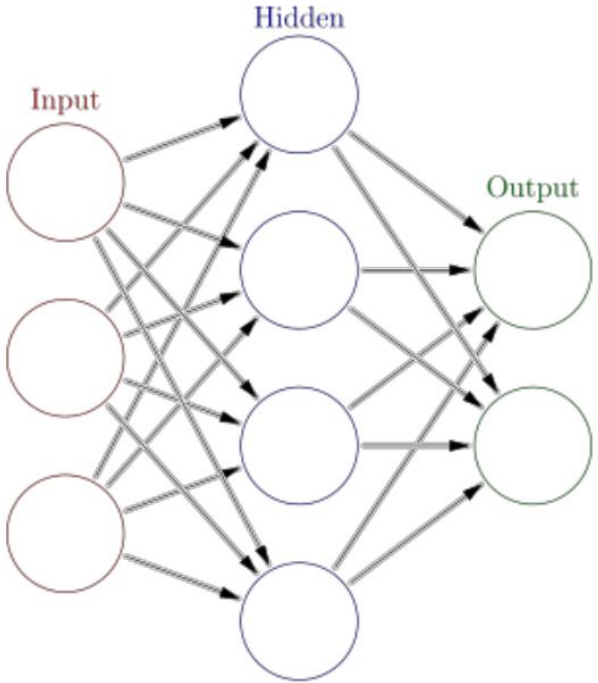
Hidden Layer

Nodes of this layer are not exposed to the outer world, they are the part of the abstraction provided by any neural network.
Hidden layer performs all sort of computation on the features entered through the input layer and transfer the result to the output layer.

Output Layer

This layer bring up the information learned by the network to the **outer world**.

# Different layers, Same network



# Backpropagation - How the network is learning

**Backpropagation** is a way of computing gradients of expressions through recursive application of **chain rule**.

Understanding of this process and its subtleties is critical for you to understand, and effectively develop, design and debug neural networks.

**<u>This is the way a Neural Network is actually learning!</u>**

**Problem statement**. The core problem studied is as follows: We are given some function f(x) where x is a vector of inputs and we are interested in computing the gradient of f at x (i.e. $\nabla f(x)$).

Why gradients?

# Backpropagation

An example:

Given f(x1, x2) = x1*x2, we want to model the data so our model will be able to learn

X1 = a, x2 = b, how to update x1 and x2 to make f(x, y) more accurate (w.r.t our loss function)?

Follow Gradient Directions!

f(x1, x2) = x1*x2 => derivF / derivX1 = X2,

derivF / derivX2 = X1

X1 = a + 0.01*b

X2 = b + 0.01*a

f(x1, x2) = a*b => (a + 0.01*b)(b + 0.01*a) // a = 4, b = -3

= 4 * (-3) => 3.97 * -2.96 => -11.7512

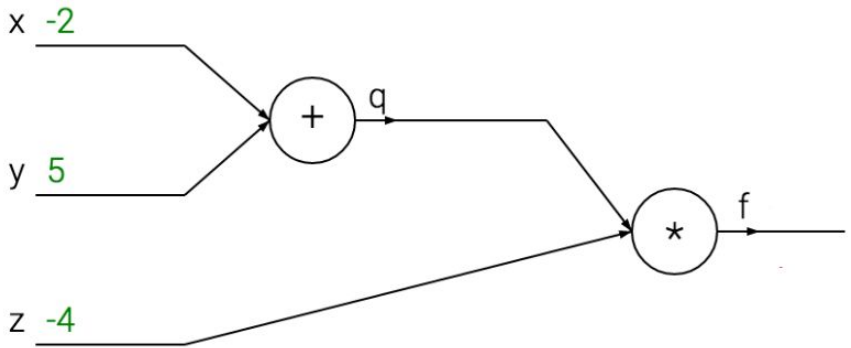
# Backpropagation

A more complicated example:

f(x, y, z) = (x + y) * z

q(x, y) = (x + y) => f(x, y, z) = q(x, y) * z

Using the chain rule: $$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial x}$$



# Gradient Descent

Gradient descent is an optimization algorithm that's used when training a machine learning model.

It's based on a convex function and tweaks its parameters iteratively to minimize a given function to its local minimum.

Gradient descent is a way to minimize an objective function (our loss function)

Minimizing our loss function => increasing our model's accuracy => learning!

**A gradient** simply measures the change in all weights with regard to the change in error. You can also think of a gradient as the **slope of a function**.

**In mathematical terms, a gradient is a partial derivative with respect to its inputs.**

# Gradient Descent

To minimize the function, we can instead follow the negative of the gradient, and thus go in the direction of steepest descent. This is gradient descent. Formally, if we start at a point x0, and move a positive distance alpha in the direction of the negative gradient, then our new and improved x_1 will look like this:

$$x_1 = x_0 - \alpha \nabla f(x_0)$$

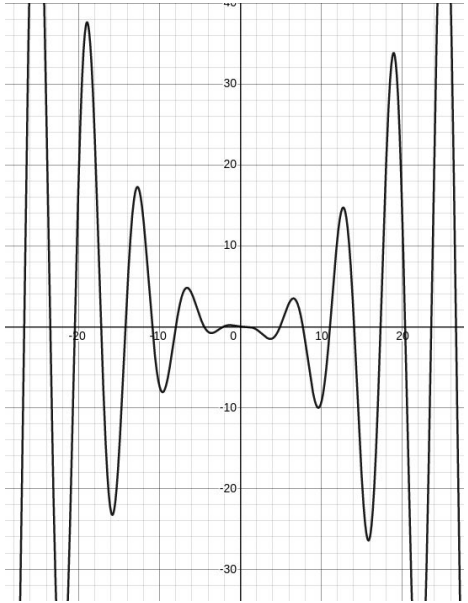
More generally:

$$x_{n+1} = x_n - \alpha \nabla f(x_n)$$

# Gradient Descent

Starting from an initial guess x0, we keep improving little by little until we find a local minimum. This process may take thousands of iterations, so we typically implement gradient descent with a computer. Given the below function:

$$f(x) = \frac{x^2 \cos(x) - x}{10}$$

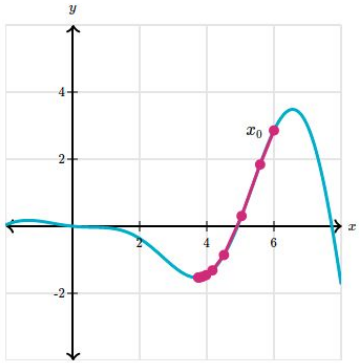


# Gradient Descent

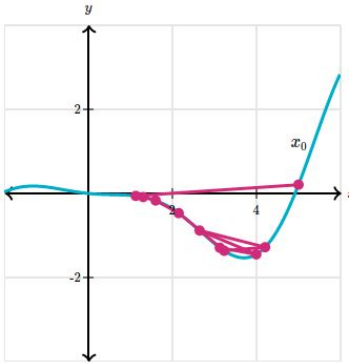
$$f(x) = \frac{x^2 \cos(x) - x}{10}$$

f'(x) = $$\frac{-x^2 \sin(x) + 2x \cos(x) - 1}{10}$$

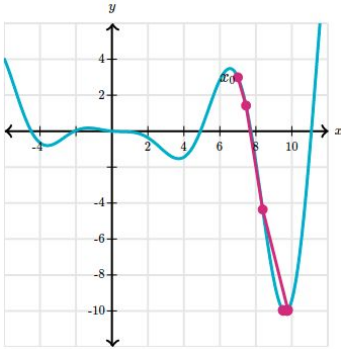
X0 = 6, alpha = 0.2




X0 = 6, alpha = 1.5

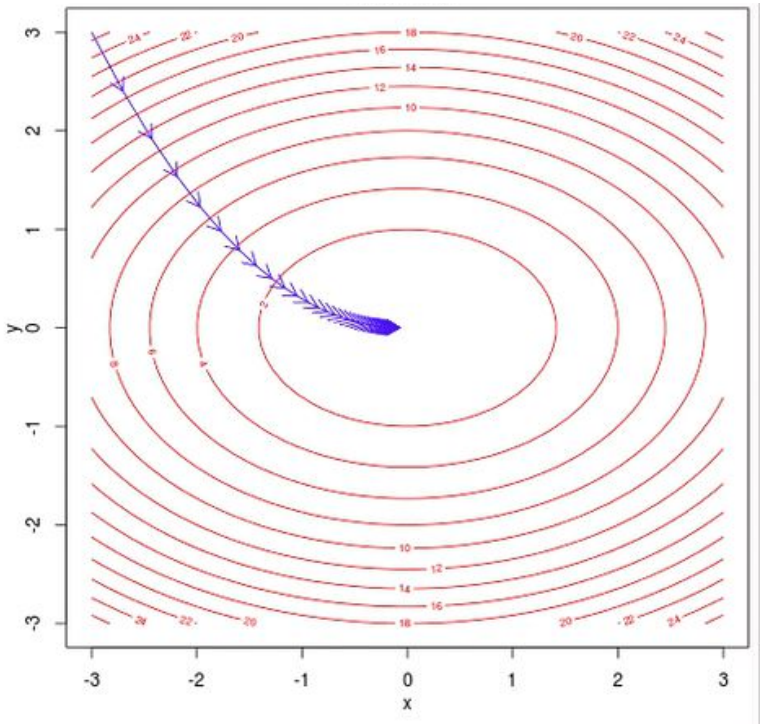



X0 = 7, alpha = 0.2




Derivative from here

# Gradient Descent



# Loss Functions

Q: Why do we need a loss function?

A: So that we will be able to measure the model against the reality, plus we will be able to present some results to anyone interested! (e.g. train error was X, test error was 0.5X...)

As part of the optimization algorithm, the error for the current state of the model must be estimated repeatedly.

This requires the choice of an error function, conventionally called a **loss function**, that can be used to estimate the loss of the model so that the weights can be updated to reduce the loss on the next evaluation.

# Loss Functions - the main ones

| ***Regression*** | ***Classification*** |
| --- | --- |
| MSE | Binary Cross Entropy |
| MAE | Hinge Loss |

# Loss Functions - the main ones

**Regression**

MSE

$$\frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

MAE

$$\frac{\sum_{i=1}^{n}|y_i - x_i|}{n}$$

| Prediction (Y^) | Label (Y) | **MAE** |
|---|---|---|
| 1 | 2 | (\|1-2\| + \|2-4\| + \|3-3\|)/3 = **1** |
| 2 | 4 | **MSE** |
| 3 | 3 | ((1-2)^2 + (2-4)^2 + (3-3)^2)/3 = **1.67** |

# Loss Functions - the main ones

**Classification**

Binary Cross Entropy (Log-loss)

$$-\frac{1}{N}\sum_{i=1}^{N} y_i \cdot log(p(y_i)) + (1 - y_i) \cdot log(1 - p(y_i))$$

Hinge Loss

$$\max\left(0, 1 - yf(x)\right)$$

| P(Y^) | Y | **BCE** |
|---|---|---|
| 0.95 | 1 | -⅓ * [(1 * log(0.95)) + (1 * log(0.35)) + (0 * log(0.03))]<br>= -⅓ * [-0.02 -0.45] = 0.15 |
| 0.35 | 1 | **Hinge Loss** |
| 0.03 | 0 | max(0, 1-0*0.03) = 1 +<br>max(0, 1-1*0.35) = 0.65 +<br>max(0, 1-1*0.95) = 0.05 = **1.7** |

# First Neural Network - Ex. 2

Ex. 2 is called - **Daniel_Ex2_First_Neural_Network.ipynb**

# Summing it all together into a Fully Connected Neural Network

**Now that we have the main components, we can compose a fully connected neural network**

1. A Neuron is needed in order to perform computation on inputs
2. A Layer is a set of neurons
3. (Forward Pass) Input layer gets inputs from the data we have
4. (Forward Pass) Hidden layer(s) gets inputs from the input layer and passes it onto the output layer
5. (Forward Pass) Output layer gets inputs from the hidden layer(s) and passes it onto the loss function
6. Loss function computes loss based on y^ and y (prediction and label)
7. Backpropagation algorithm, using Gradient Descent, comes in place in order to update the weights and to learn
8. Iteratively 1-7 again and again until convergence...

**And there you go - the pseudo code of a Neural Network!**

# Q&A Time

# Fully Connected Neural Networks

As you saw in the Ex.2, if we want a larger network, we will need to think about each function's derivative, derivative it accordingly, and populate the input forward and backward, again and again, playing with a lot of parameters, and inputs.

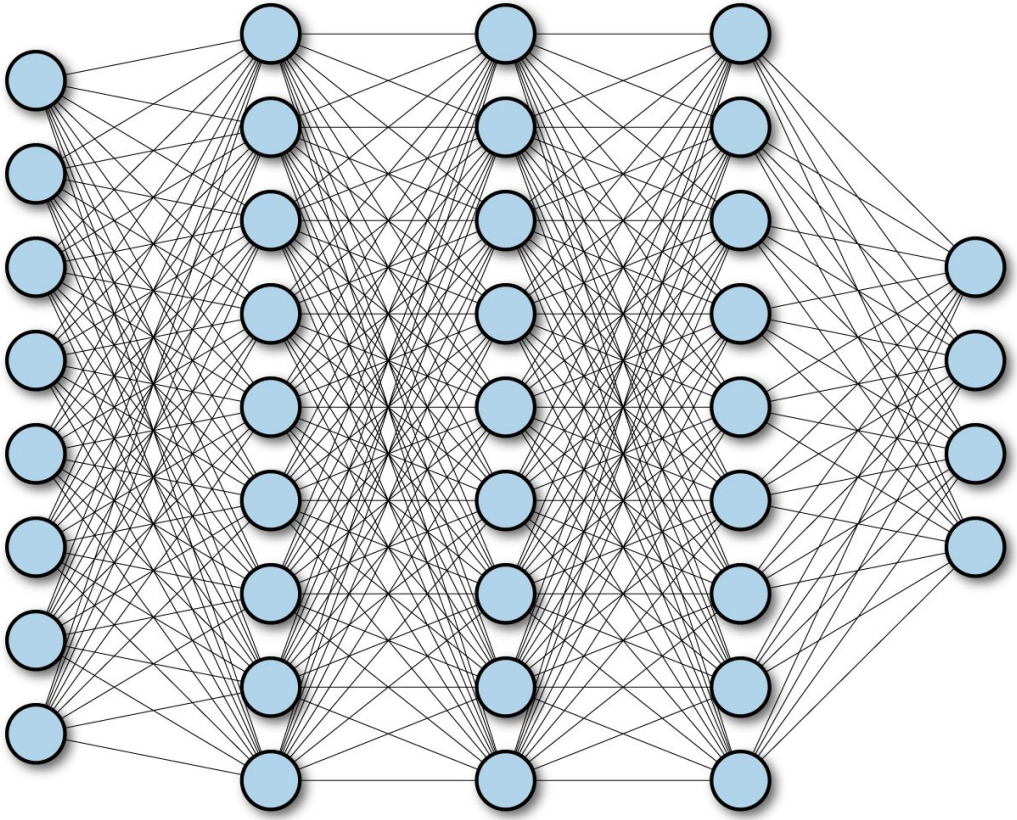
This is not scalable, of course.

Luckily, we have python libraries who does this for us behind the scenes.

Now you are going to meet Keras, and eventually you'll get familiar with this library, which will make it pretty easy to build up a NN

Fun time! Let's practice Fully connected NNs without coding (Let's take ~15-20 min for this)

# Fully Connected Neural Networks



# Fully Connected Neural Networks

A fully connected Neural Network is a network composed from many neurons, and many layers!

Meaning:

- There is not only 1-d input (or 2-d...)
- There is not only 1 hidden layer
- There is 1 output layer which can be composed of >1-d

*"One of the main advantages of **deep learning** lies in being able to solve complex problems that require discovering hidden patterns in the data and/or a deep understanding of intricate relationships between a large number of interdependent variables."* (source)

# Fully Connected Neural Networks - Ex. 3

Keras API - get familiar with this powerful API for building Neural Networks

Ex. 3 is called “**DL_in_Keras_Ex3.ipynb**”

# Convolutional Neural Networks

A Convolutional Neural Network, also known as CNN or ConvNet, is a class of neural networks that specializes in processing data that has a **grid-like** topology, such as an **image**.

A digital image is a binary representation of visual data.

It contains a series of pixels arranged in a grid-like fashion that contains pixel values to denote how bright and what color each pixel should be.

# Convolutional Neural Networks

**Motivation**: why do we need CNN?

As we previously saw, FC NN usually receive an input (a vector), and transform it through a series of hidden layers.

Each hidden layer is made up of a “bunch” of neurons, and each neuron is **fully** connected to **all** neurons in the previous layer.

Regular Neural Networks don’t scale well to full images, and why is that?

Let’s think of CIFAR-10 (images of 32x32x3) - A single fully-connected neuron in a first hidden layer of a regular Neural Network would need to be 32*32*3 = 3072 weights.

This seems cool up till here, but what would be the case if the image was 200*200*3 ? => 120,000 weights! This scales largely pretty easily, and can lead to bad phenomena such as overfitting, and just way more computation than actually needed

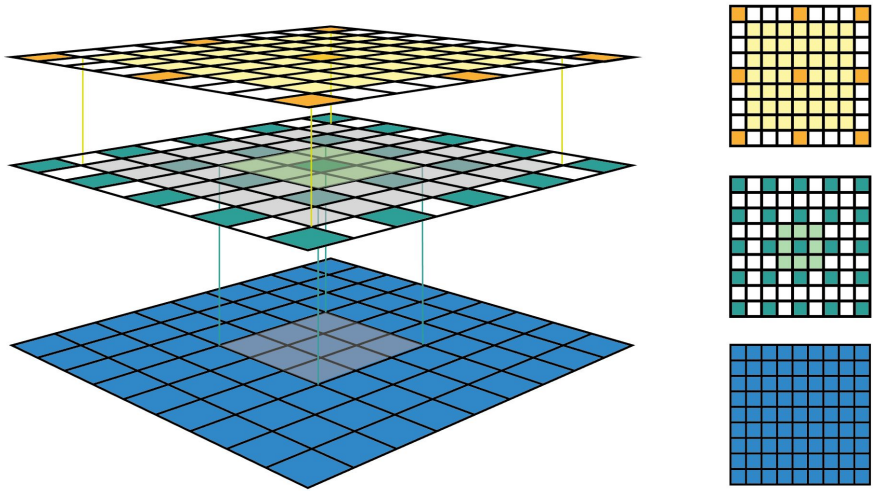
# Convolutional Neural Networks

| ***Fully Connected Neural Network*** | ***Convolutional Neural Network*** |
|---|---|
| Uses Fully Connected Layers (as we saw) | Uses Partially Connected Layers |
| Can be used only for small images | Can be used for any image |
| Considers entire image for learning the image “patterns” | Considers only the part of an image, aka **Receptive Field** |
| # of parameters will be very high. E.g. for 28x28 images, if a layer has 32 neurons, the # of params will be 28x28x32 = 25,088. Consequently, computations per each of these values | # of parameters will be very less compared to **FCNN**. For 28x28 image, if the layer has 32 filters (we’ll learn soon), each filter size = 5 (we’ll learn soon), num of parameters will be (5*5*1 + 1) * 32 = 832. Computations accordingly |
| **No translation invariant**. Once a regular DNN has learned a pattern in one location, it won’t learn to recognize it in other location. In short, we will need to re-train the model | **Translation invariant**. Once a CNN has learned a pattern in one location, it can recognize it in other location. In short, weights can be reused even if the image is rotated, shifted, etc. |

# Convolutional Neural Networks

**Receptive Field**

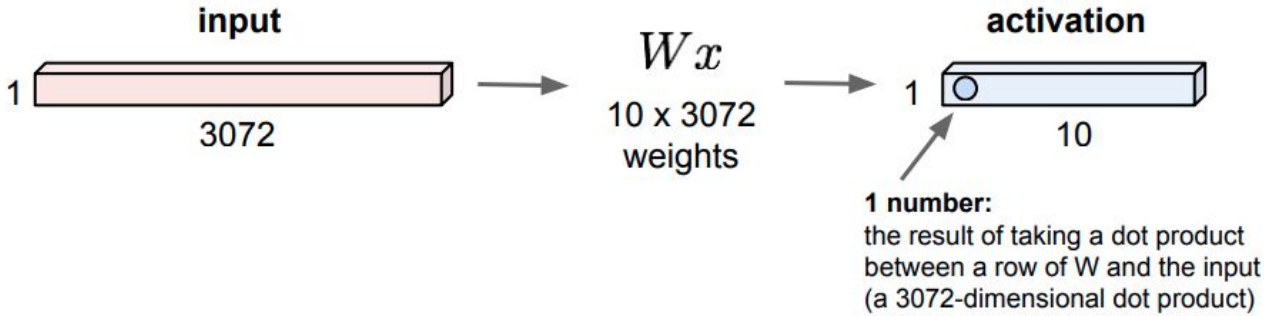
*(The receptive field is defined as the region in the input space that a particular CNN's feature is looking at (i.e. be affected by) . A receptive field of a feature can be described by its center location and its size)*



# Convolutional Neural Networks

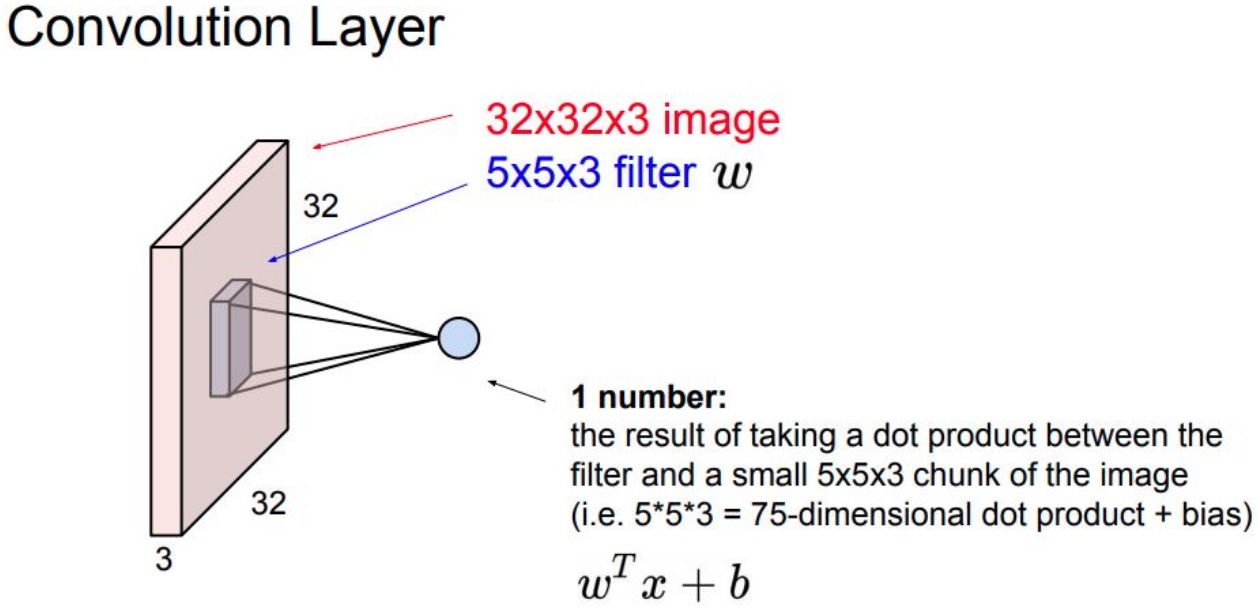
## Fully Connected Layer

32x32x3 image -> stretch to 3072 x 1



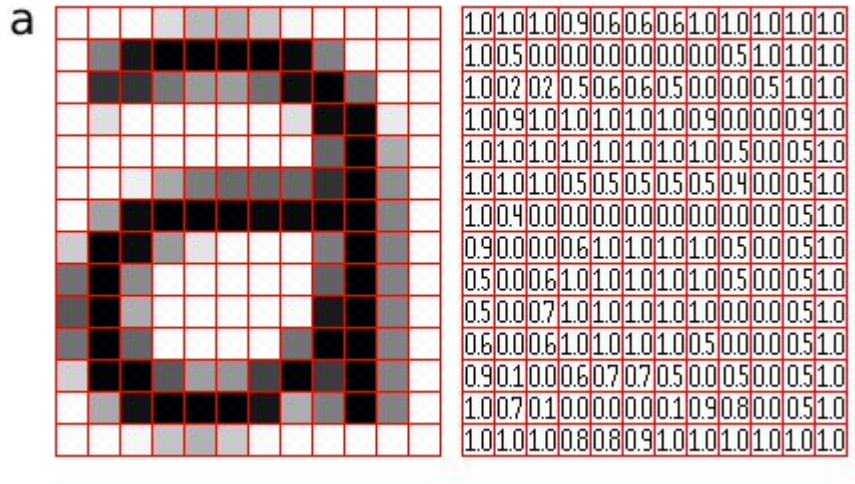

source

# Convolutional Neural Networks



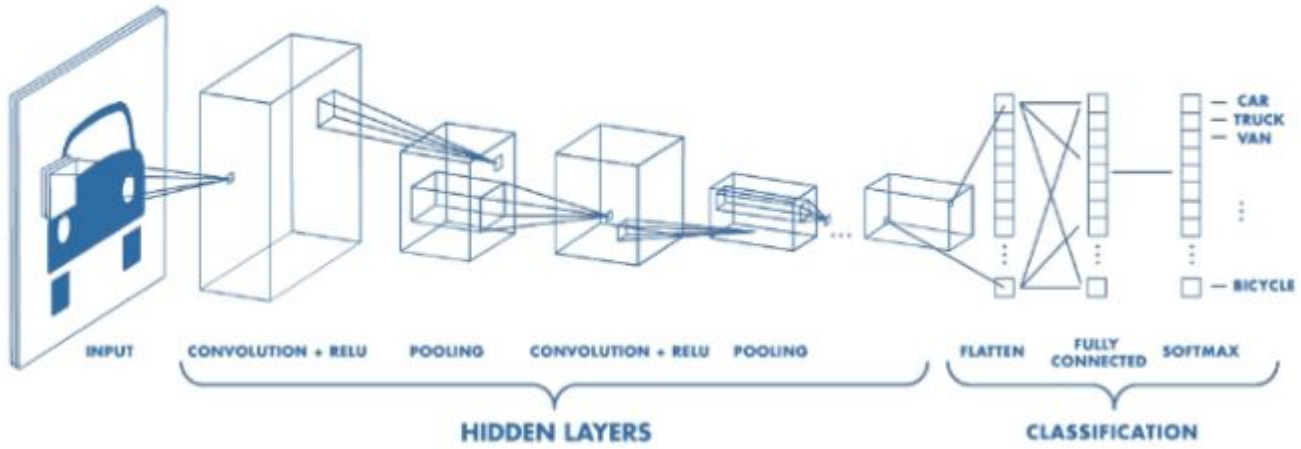

source

# Convolutional Neural Networks



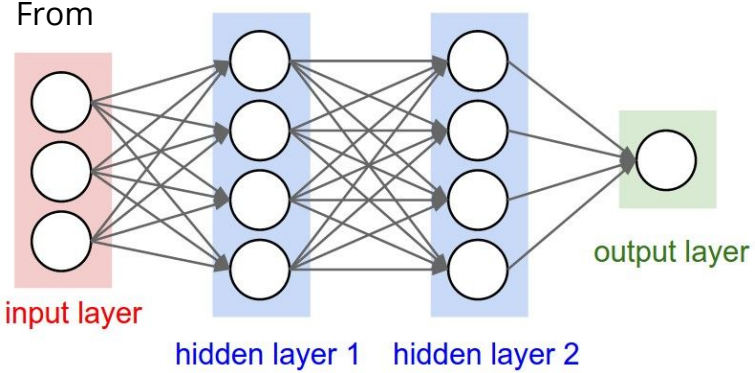

source

# Convolutional Neural Networks

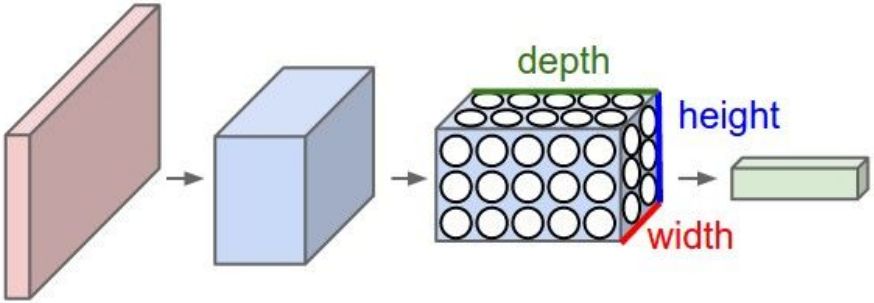


# Moving from FC NNs to CNN

From




To




source

# Moving from FC NNs to CNN

Similarities

- They are both made up of neurons that have learnable weights and biases.
- Each neuron receives some inputs, performs a dot product and optionally follows it with a non-linearity.
- A loss function still exists and being “learned” and updated via the training process

Differences

- ConvNet architectures make the explicit assumption that the inputs are **images**, which allows us to encode certain properties into the architecture.
- These then make the forward function more efficient to implement and **vastly reduce** the amount of parameters in the network

# Moving from FC NNs to CNN

3D volumes of neurons

CNNs take advantage of the fact that the input consists of images and they constrain the architecture in a more sensible way

In particular, unlike a regular Neural Network, the layers of a ConvNet have neurons arranged in 3 dimensions: **width, height, depth**.

For example, the input images in CIFAR-10 are an input volume of activations, and the volume has dimensions 32x32x3 (width, height, depth respectively, and 3 stands for RGB, color, image).

# CNN Layers

A ConvNet (CNN) is a sequence of layers, and every layer of this ConvNet transforms one volume of activations to another through a differentiable function.

As opposed to Neural Networks, here there are several types of layers.

We use three main types of layers to build ConvNet architectures:

- Convolutional Layer
- Pooling Layer
- Fully-Connected Layer (exactly as seen in regular Neural Networks)

Now we'll dive into each of them in order to understand:

- What's going in
- What's being done
- What's going out

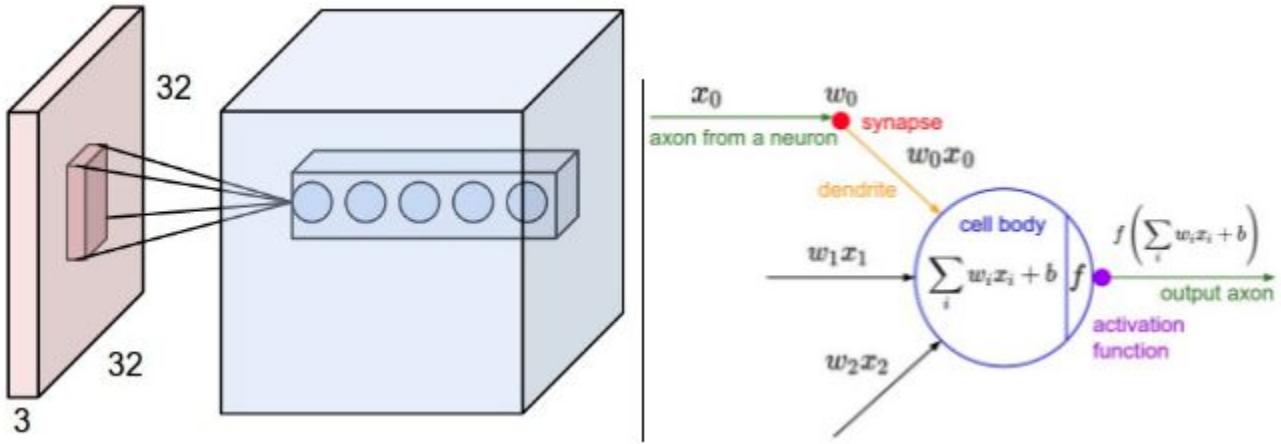
# Convolutional Layer

The Convolutional layer is the core building block of a CNN, and basically it does most of the computational heavy lifting.

Main Attributes:

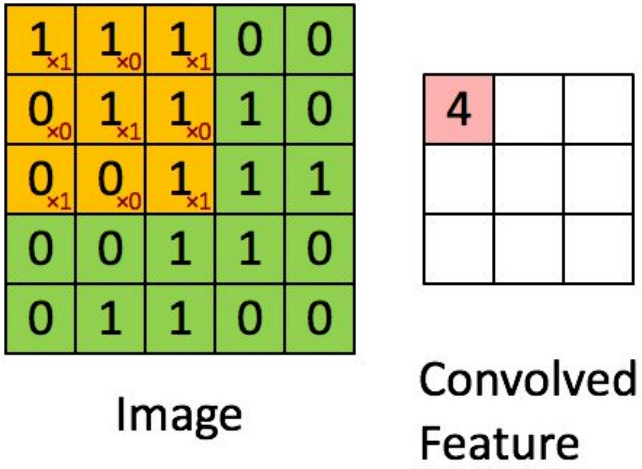
- The CONV layer parameters consist of a set of **learnable** filters
- Every filter is small in terms of w, h, but extends through the full depth of the input volume. For example, a typical filter on a first layer of a ConvNet might have size 5x5x3 (i.e. 5 pixels width and height, and 3 because images have depth 3, the color channels)
- During the forward pass, we “convolve” each filter across the width and height of the input volume and compute **dot products** between the entries of the filter and the input at any position
- Intuitively, the network will learn filters that activate when they see some type of visual feature such as an edge of some orientation or a blotch of some color on the first layer, or eventually entire honeycomb or wheel-like patterns on higher layers of the network

# Convolutional Layer




source

# Convolutional Layer




source

# Convolutional Layer

**Class Exercise** - Let's do a math, "by-hand" example.
These are the things that really helps to understand the bits behind this fascinating algo!

***Given an image h:***

| | | | | | |
|---|---|---|---|---|---|
| 10 | 8 | 10 | 5 | 10 | 0 |
| 7 | 12 | 4 | 8 | 10 | 0 |
| 4 | 10 | 5 | 8 | 10 | 10 |
| 6 | 6 | 6 | 1 | 2 | 10 |
| 8 | 5 | 5 | 5 | 2 | 8 |
| 9 | 0 | 0 | 0 | 2 | 5 |

*

***Given a filter f:***

| | | |
|---|---|---|
| 1 | 2 | 1 |
| 0 | 0 | 0 |
| -1 | -2 | -1 |

=

***We can produce Matrix G***

| | | | |
|---|---|---|---|
| | | | |
| | | | |
| | | | |
| | | | |

# Convolutional Layer

**Class Exercise** - Let's do a math, "by-hand" example.
These are the things that really helps to understand the bits behind this fascinating algo!

***Given an image h:***

| | | | | | |
|---|---|---|---|---|---|
| **10** | **8** | **10** | 5 | 10 | 0 |
| **7** | **12** | **4** | 8 | 10 | 0 |
| **4** | **10** | **5** | 8 | 10 | 10 |
| 6 | 6 | 6 | 1 | 2 | 10 |
| 8 | 5 | 5 | 5 | 2 | 8 |
| 9 | 0 | 0 | 0 | 2 | 5 |

*

***Given a filter f:***

| | | |
|---|---|---|
| **1** | **2** | **1** |
| **0** | **0** | **0** |
| **-1** | **-2** | **-1** |

=

***We can produce Matrix G***

| | | | |
|---|---|---|---|
| **7** | ... | ... | ... |
| ... | ... | ... | ... |
| ... | ... | ... | ... |
| ... | ... | ... | ... |

# Convolutional Layer

Now it's your turn!

***Given an image h:***

| | | | | | |
|---|---|---|---|---|---|
| **10** | **8** | **10** | 5 | 10 | 0 |
| **7** | **12** | **4** | 8 | 10 | 0 |
| **4** | **10** | **5** | 8 | 10 | 10 |
| 6 | 6 | 6 | 1 | 2 | 10 |
| 8 | 5 | 5 | 5 | 2 | 8 |
| 9 | 0 | 0 | 0 | 2 | 5 |

*

***Given a filter f:***

| | | |
|---|---|---|
| **1** | **2** | **1** |
| **0** | **0** | **0** |
| **-1** | **-2** | **-1** |

=

***We can produce Matrix G***

| | | | |
|---|---|---|---|
| **7** | | | |
| | | | |
| | | | |
| | | | |

# Convolutional Layer

How should we know the size of the new matrix that we generate?
Luckily, someone already developed formulas for this as well :)

N_out = [(N_in + 2p - f)/s + 1]
D_out = num_filters

*Where,*

- *N_out = the size of the output **w/h** (following convolve)*
- *N_in = the size of the input **w/h***
- *P = Padding (you'll see in a bit)*
- *S = Stride (you'll see in a bit)*
- *F = filter size*
- *Num_filters = number of filters applied on that input (e.g. 64)*

# Convolutional Layer Parameters

Stride & Padding - what are they all about?

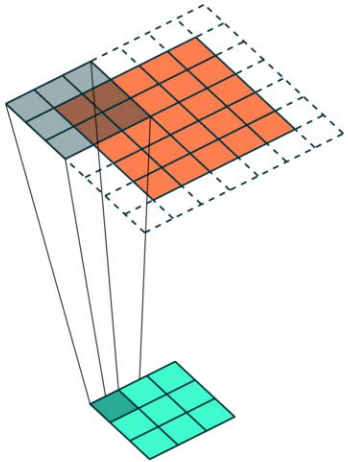
**Stride** - “our jumps” in which we slide the filter
When the stride is **one** then we move the filters **one** pixel at a time.
When the stride is **2** then the filters jump **2** pixels at a time as we slide them around.
Larger stride will produce smaller output volumes spatially.

**Padding** - sometimes it will be convenient to pad the input volume with zeros around the border. The size of this zero-padding is a hyperparameter, which eventually decides on the output size (w, h).
Most APIs (Keras, PyTorch, ...) support zero padding vs. 1..X padding

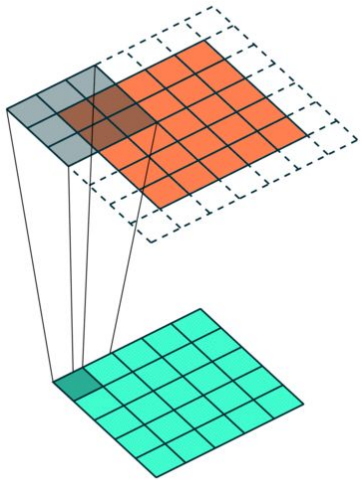
# Convolutional Layer Parameters

Let's see some examples - from here, and here

*Stride (s) = 2, Padding (p) = 1*



*Stride (s) = 1, Padding (p) = 1*



*Stride (s) = 1, Padding (p) = 1*

| | | | | | | |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 60 | 113 | 56 | 139 | 85 | 0 |
| 0 | 73 | 121 | 54 | 84 | 128 | 0 |
| 0 | 131 | 99 | 70 | 129 | 127 | 0 |
| 0 | 80 | 57 | 115 | 69 | 134 | 0 |
| 0 | 104 | 126 | 123 | 95 | 130 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Kernel

| | | |
|---|---|---|
| 0 | -1 | 0 |
| -1 | 5 | -1 |
| 0 | -1 | 0 |

| | | | | |
|---|---|---|---|---|
| 114 | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |

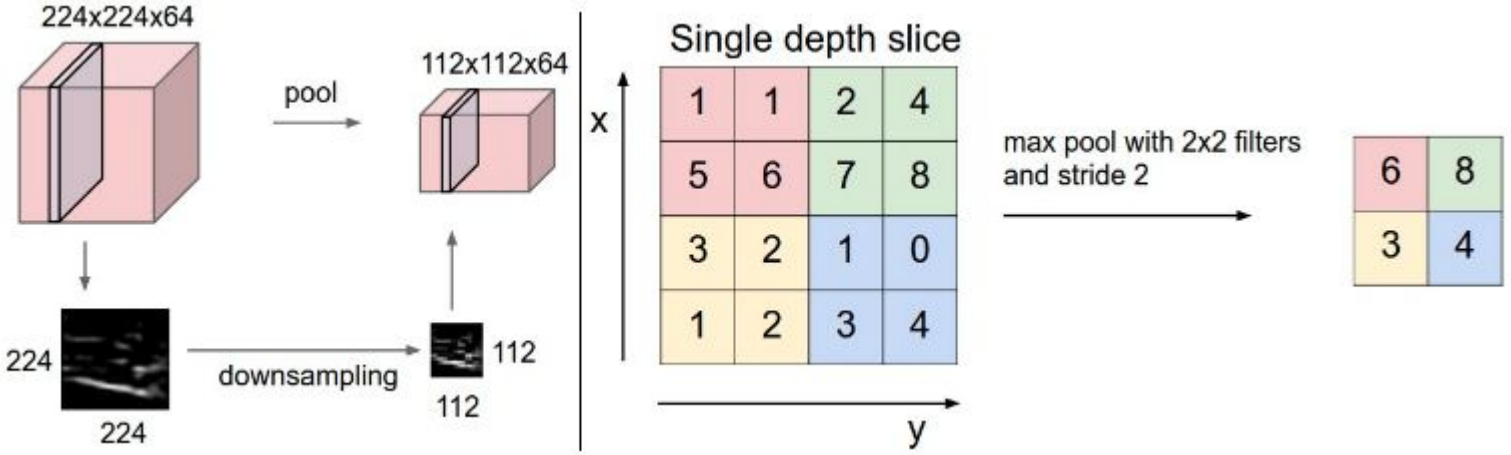
# Pooling Layer

Another crucial layer in CNNs is the pooling layer.

- It is common to periodically insert a Pooling layer in-between successive Conv layers in a ConvNet architecture
- **Its function is to progressively** reduce the spatial size of the representation to reduce the amount of parameters and computation in the network, and hence to also control overfitting
- The Pooling Layer operates independently on every depth slice of the input and resizes it spatially (by w, h), using the MAX / AVG operation. The most common form is a pooling layer with filters of size 2x2 applied with a stride of 2 downsamples every depth slice in the input by 2 along both width and height, discarding 75% of the activations
- Every MAX/AVG operation would in this case be taking a max over 4 numbers (little 2x2 region in some depth slice)
- The **depth** dimension remains unchanged

# Max Pooling example

Pooling (Max in this case) works as follows:




source

# Max Pooling example

Luckily, we can also calculate output sizes following any pooling operation!

- Given: W1, H1, D1 (e.g. 28*28*3)
- Stride = S, filter_size = F (e.g. 2)
- Output will be -
  - W2 = (W1 - F) / S + 1
  - H2 = (H1 - F) / S + 1
  - D2 = D1 // meaning -> Pooling does not “touch” the input depth

source

# Let's code your first CNN!

Ex. 4 is called "**DL_CNN_in_Keras_Ex4.ipynb**"

(Reminder: Use kernel: *conda_tensorflow2_p36*)

In this ex., you are going to implement your first CNN in Keras!

You will see the different layers of CNN, in Keras, using the Sequential API.

For that (and for the next exercise), let's go over a quick overview of CNN in Keras

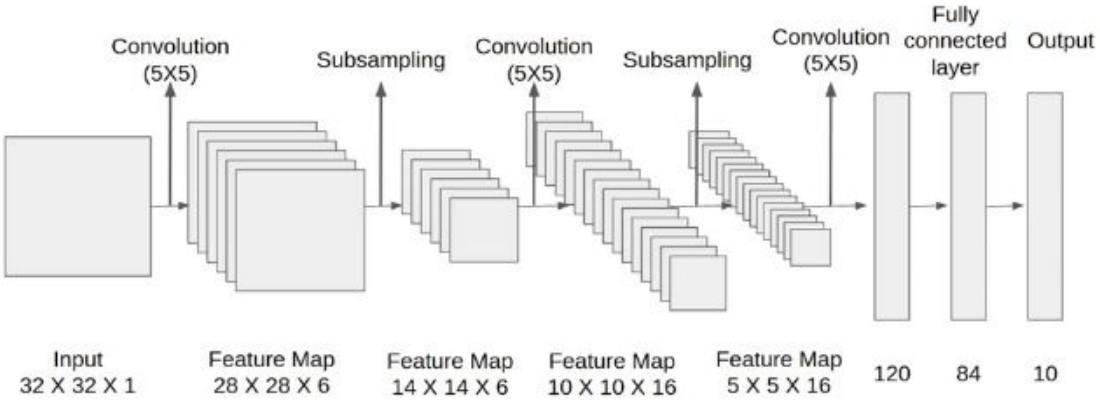
# CNN - Main Architectures

There are many “known” architectures, that were developed by great researchers along the years, and are in usage even today.

This architectures were tested on many datasets, and even on “baseline” datasets, that helps to evaluate how good a model is “overall”.

Now, we’ll go through few of these and will learn how and why are the speciality of each.

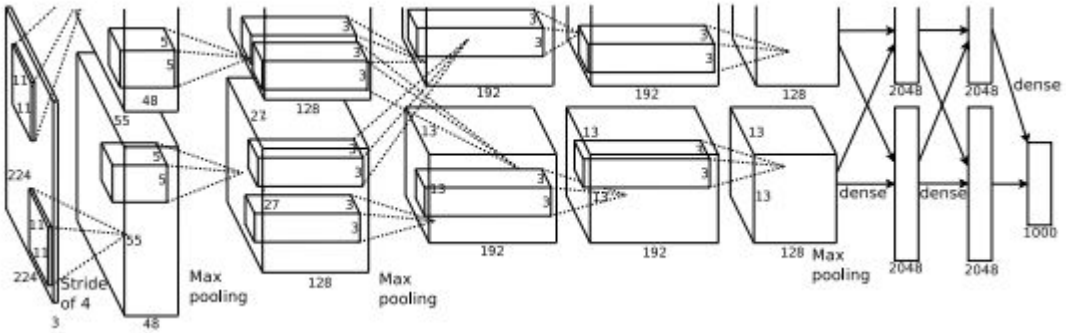
# CNN - LeNet-5




Convolutional Filters were 5*5, applied at stride=1 with pooling
layers were 2*2 applied at stride 2
architecture is:
**[*CONV-POOL-CONV-POOL-FC-FC*]**
Subsampling == Pooling

[LeCun et al., 1998]

# CNN - AlexNet



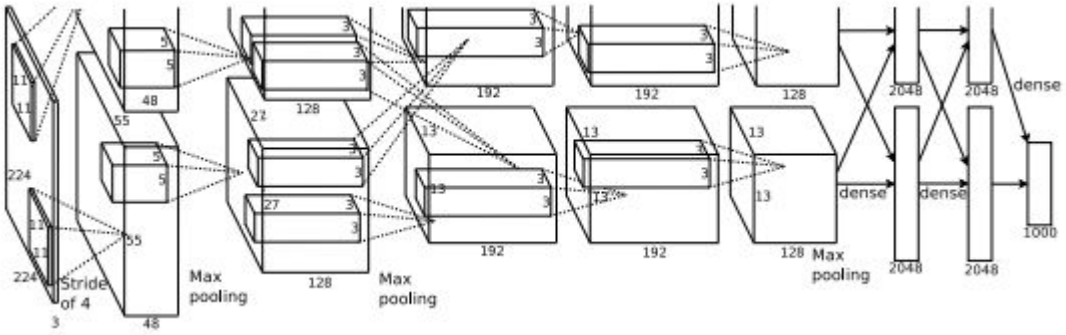

[227x227x3] **INPUT**
[55x55x96] **CONV1**: 96 11x11 filters at stride 4, pad 0
[27x27x96] **MAX POOL1**: 3x3 filters at stride 2
[27x27x96] **NORM1**: Normalization layer
[27x27x256] **CONV2**: 256 5x5 filters at stride 1, pad 2
[13x13x256] **MAX POOL2**: 3x3 filters at stride 2
[13x13x256] **NORM2**: Normalization layer
[13x13x384] **CONV3**: 384 3x3 filters at stride 1, pad 1
[13x13x384] **CONV4**: 384 3x3 filters at stride 1, pad 1
[13x13x256] **CONV5**: 256 3x3 filters at stride 1, pad 1
[6x6x256] **MAX POOL3**: 3x3 filters at stride 2
[4096] **FC6**: 4096 neurons
[4096] **FC7**: 4096 neurons
[1000] **FC8**: 1000 neurons (class scores)

[Krizhevsky et al., 2012]

Input: 227x227x3 images
First layer (CONV1): 96 of 11x11 filters applied at stride 4 =>
Q: what is the output volume size?

55*55*96, and why?
(227-11)/4+1 = 55 -> w = h
C (or depth) = 96 (num of filters)

# CNN - AlexNet

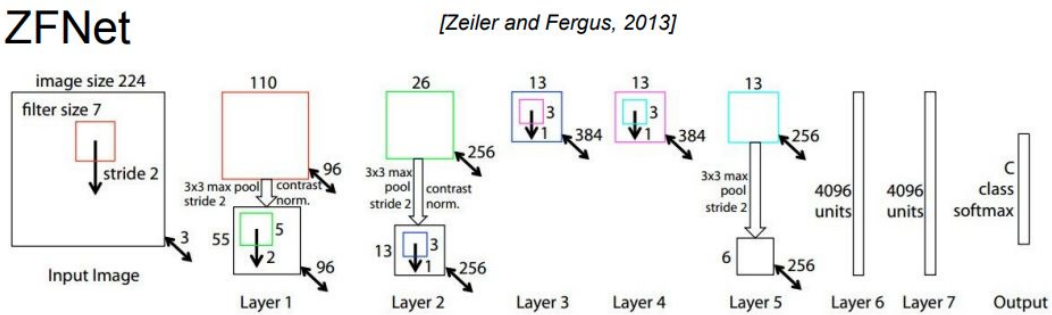



*What was special here?*

- *First use of ReLU!*
- *Used norm layers (not common today)*
- *Historical note: Trained on GTX 580 GPU with only 3 GB of memory. Network spread across 2 GPUs, half the neurons (feature maps) on each GPU*
- *Was a "baseline" for a more advanced networks (you'll see in a bit)*

[Krizhevsky et al., 2012]

# CNN - ZFNet

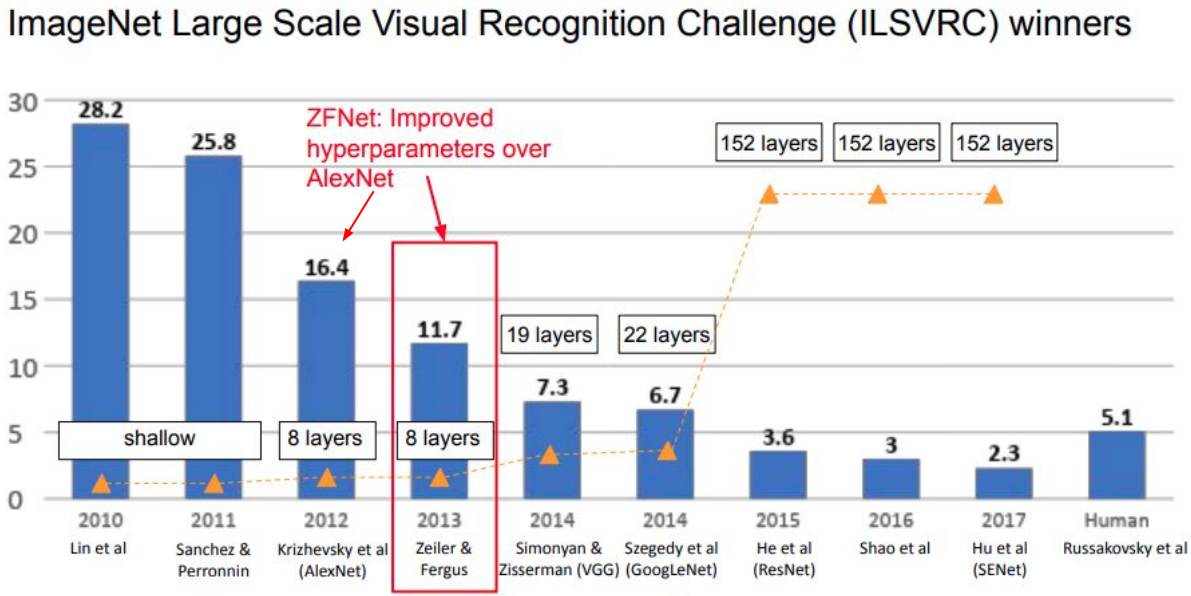



Actually, pretty similar to AlexNet, but…

1. **CONV1**: change from (11x11 stride 4) to (7x7 stride 2)
2. **CONV3,4,5**: instead of 384, 384, 256 filters use 512, 1024, 512

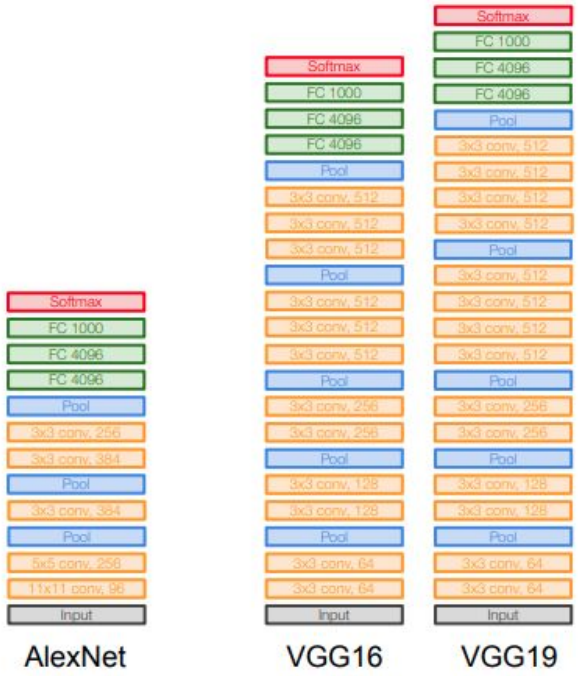
[Zeiler and Fergus, 2013]

# CNN - ZFNet




[Zeiler and Fergus, 2013]

# CNN - VGGNet (Very Deep…)

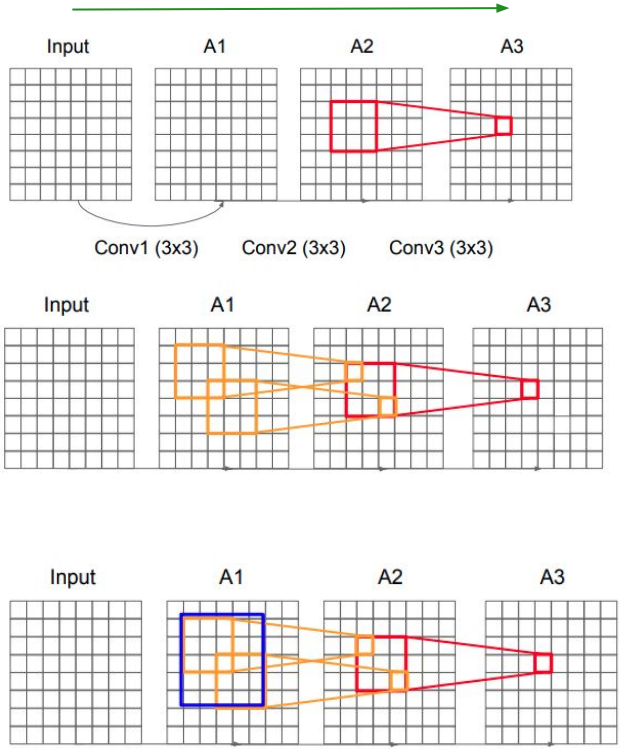



***The Key: Small filters, Deeper networks***
8 layers (AlexNet)
-> 16 - 19 layers (VGG16Net)
Only 3x3 CONV stride 1, pad 1
and 2x2 MAX POOL stride 2

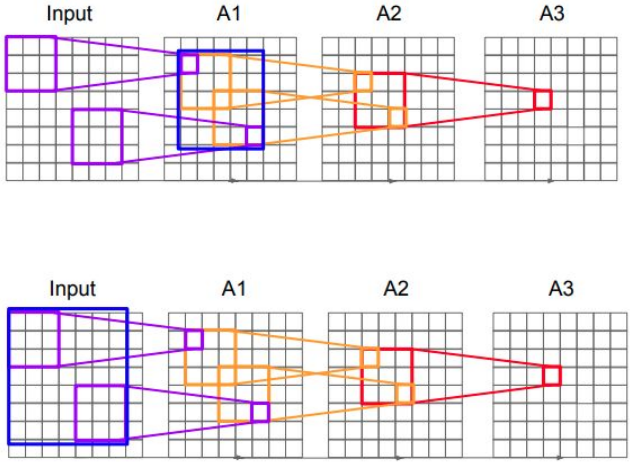
**Results on ImageNet**
From the “best” ZFNet score, 11.7% top 5 error in ILSVRC’13, to 7.3% top 5 error in ILSVRC’14 !

[Simonyan and Zisserman, 2014]

# CNN - VGGNet (Very Deep…)




***Why use 3 of 3*3 (stack) of filters?***
***Remember the "Receptive Field" we discussed***




*Stack of three 3x3 conv (stride 1) layers has same effective receptive field as one 7x7 conv layer, plus we use less parameters:*
***3 * (3^2*C^2) vs. (7^2C^2) for C channels per layer***

source

# CNN - Let's talk parameters

(not counting biases)

INPUT: [224x224x3] memory: 224*224*3=150K params: 0
CONV3-64: [224x224x64] memory: 224*224*64=3.2M params: (3*3*3)*64 = 1,728
CONV3-64: [224x224x64] memory: 224*224*64=3.2M params: (3*3*64)*64 = 36,864
POOL2: [112x112x64] memory: 112*112*64=800K params: 0
CONV3-128: [112x112x128] memory: 112*112*128=1.6M params: (3*3*64)*128 = 73,728
CONV3-128: [112x112x128] memory: 112*112*128=1.6M params: (3*3*128)*128 = 147,456
POOL2: [56x56x128] memory: 56*56*128=400K params: 0
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*128)*256 = 294,912
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*256)*256 = 589,824
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*256)*256 = 589,824
POOL2: [28x28x256] memory: 28*28*256=200K params: 0
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*256)*512 = 1,179,648
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*512)*512 = 2,359,296
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*512)*512 = 2,359,296
POOL2: [14x14x512] memory: 14*14*512=100K params: 0
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
POOL2: [7x7x512] memory: 7*7*512=25K params: 0
FC: [1x1x4096] memory: 4096 params: 7*7*512*4096 = 102,760,448
FC: [1x1x4096] memory: 4096 params: 4096*4096 = 16,777,216
FC: [1x1x1000] memory: 1000 params: 4096*1000 = 4,096,000

VGG16

source

# CNN - Let's talk parameters and memory

(not counting biases)

INPUT: [224x224x3] memory: 224*224*3=150K params: 0
CONV3-64: [224x224x64] memory: **224*224*64=3.2M** params: (3*3*3)*64 = 1,728
CONV3-64: [224x224x64] memory: **224*224*64=3.2M** params: (3*3*64)*64 = 36,864
POOL2: [112x112x64] memory: 112*112*64=800K params: 0
CONV3-128: [112x112x128] memory: 112*112*128=1.6M params: (3*3*64)*128 = 73,728
CONV3-128: [112x112x128] memory: 112*112*128=1.6M params: (3*3*128)*128 = 147,456
POOL2: [56x56x128] memory: 56*56*128=400K params: 0
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*128)*256 = 294,912
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*256)*256 = 589,824
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*256)*256 = 589,824
POOL2: [28x28x256] memory: 28*28*256=200K params: 0
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*256)*512 = 1,179,648
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*512)*512 = 2,359,296
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*512)*512 = 2,359,296
POOL2: [14x14x512] memory: 14*14*512=100K params: 0
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
POOL2: [7x7x512] memory: 7*7*512=25K params: 0
FC: [1x1x4096] memory: 4096 params: 7*7*512*4096 = **102,760,448**
FC: [1x1x4096] memory: 4096 params: 4096*4096 = 16,777,216
FC: [1x1x1000] memory: 1000 params: 4096*1000 = 4,096,000

TOTAL memory: 24M * 4 bytes ~= 96MB / image (only forward! ~*2 for bwd)
TOTAL params: 138M parameters

Note:

Most memory is in early CONV

Most params are in late FC

source

# CNN - Competition Time!

Now it's your time to shine!

You will be given a dataset (MNIST), a specific code for data preprocessing, and your goal is to reach at least **92%** accuracy! (**on the test set**).

**You should implement one of the models (LeNet, AlexNet, …) from scratch, OR you can define your own architecture!**

Ex. 5 is called "**DL_CNN_Competition_Ex5.ipynb**"

(Reminder: Use kernel: *conda_tensorflow2_p36*)

# CNN & NN - Competition Time!

Now we are going to have a nice competition, this time not between you and an accuracy.

This time, the competition is between models!

You will be given a dataset (Boston Housing prices), and you will be ask to predict a price using two models - NN and CNN.

You will need to build them, compile them, train them, and evaluate their performance.

At the end, you will need to decide who is better based on the evaluation.

The goal is to be able to see how NNs performs on non-classical problems (e.g. regression! Usually you saw it on classic ML, not this time)

Ex. 6 is called "**DL_Ex6_CNN_NN_Competition_on_Regression_Keras.ipynb**"

(Reminder: Use kernel: *conda_tensorflow2_p36*)

# CNN & NN - A possible solution

A possible solution and walkthrough notebook to Ex. 6 can be found here

# Q&A Time

# Regularization (in DL)

Why do we need regularization?

Think about this case:

Loss (y, x, W) = 1/n * L(f(x, W), y)

Such that: we found some W that gets us to ***Loss(y, x, W) = 0***

Is it the most unique W? No!

Why? 2W (W multiplied by 2) also gets us to Loss(y, x, W) = 0

$$L(W) = \frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i)$$

Data loss: Model predictions should match training data

source

# Regularization

Our goal is to “punish” our model, so eventually he won’t do “too good” on the data.

This “too good” phenomena is also known as overfitting.

In a word - overfitting is knowing pretty well the training data, but know nothing when it comes to new, unseen data.

$$L(W) = \underbrace{\frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i)}_{\text{Data loss}} + \underbrace{\lambda R(W)}_{\text{Regularization}}$$

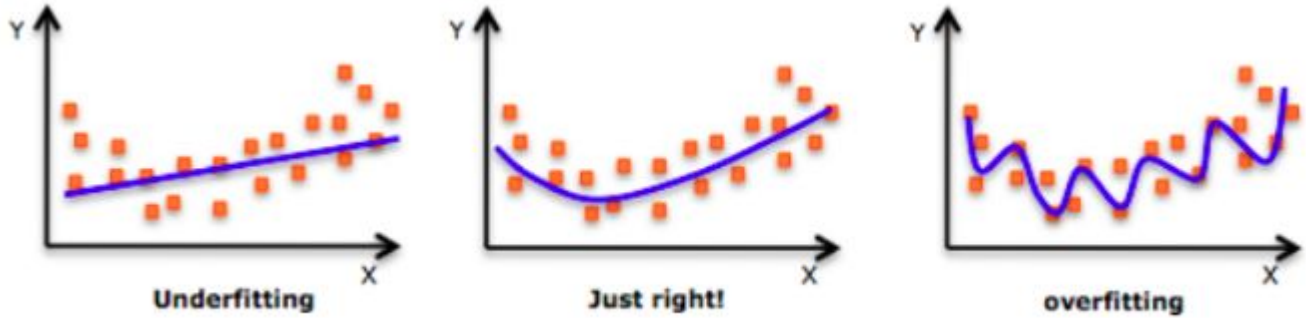
**Data loss**: Model predictions should match training data

**Regularization**: Prevent the model from doing *too* well on training data

Where λ is regularization power

source

# Regularization

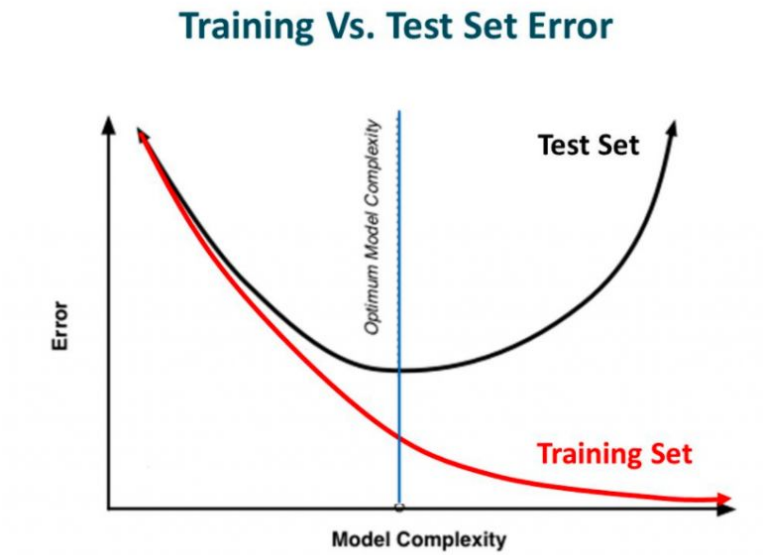



source

# Regularization

while going towards the right, the complexity of the model increases such that the training error reduces but the testing error doesn't.

A more complex model (more neurons, layers, depth, ...) usually will get you to a lower error on the training set, however that's not 100% the case on the test set




source

# Regularization - main types in DL

There are several types of regularization in DL. Let's go over the main ones -

- L1 Regularization
- L2 Regularization
- Dropout
- Data Augmentation
- Early Stopping

**Most common**

source

# Regularization - L1 Regularization

L1 Regularization - also known as “Lasso Regression”

Lasso Regression (Least Absolute Shrinkage and Selection Operator) adds

“absolute value of magnitude” of coefficient as penalty term to the loss function.

In mathematical notations, this means that:

$$\sum_{i=1}^{n}(Y_i - \sum_{j=1}^{p} X_{ij}\beta_j)^2 + \lambda \sum_{j=1}^{p} |\beta_j|$$

source

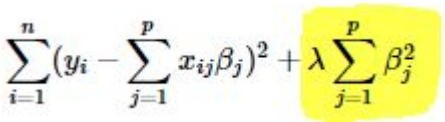
# Regularization - L2 Regularization

L2 Regularization - also known as “Ridge Regression”

Ridge regression adds “squared magnitude” of coefficient as penalty term to the

loss function. Here the highlighted part represents L2 regularization element.

In mathematical notations, this means that:



$$\sum_{i=1}^{n}(y_i - \sum_{j=1}^{p} x_{ij}\beta_j)^2 + \lambda \sum_{j=1}^{p} \beta_j^2$$

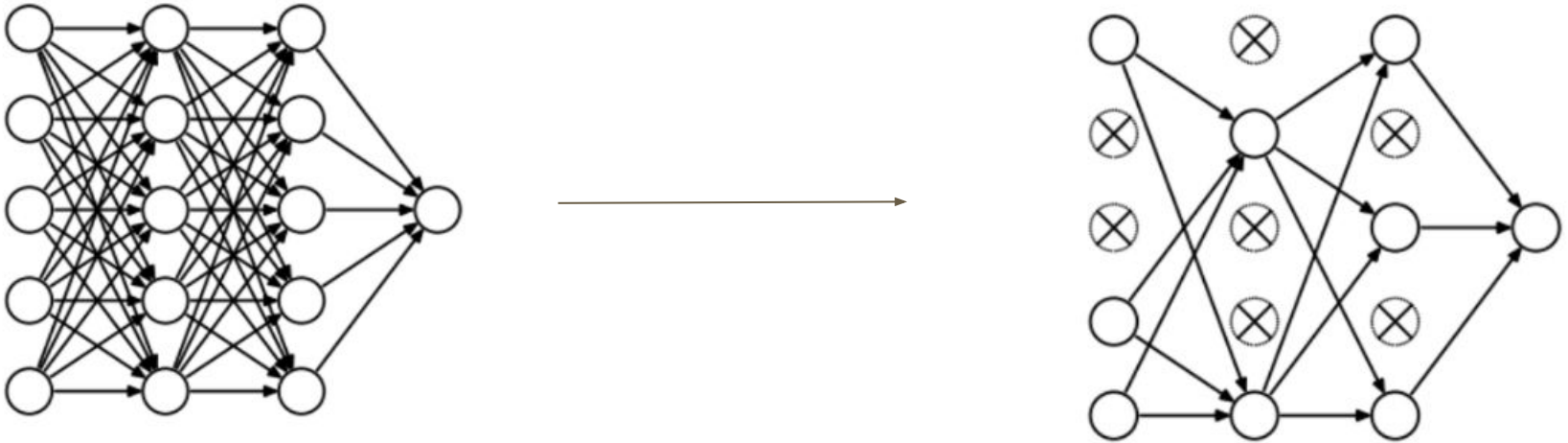
```
from keras import regularizers
model.add(Dense(64, input_dim=64,

kernel_regularizer=regularizers.l2(0.01)
```

source

# Regularization - Dropout

Dropout Technique - one of the most interesting techniques in DL

Dropout produces very good results and is widely used in Deep Learning models.

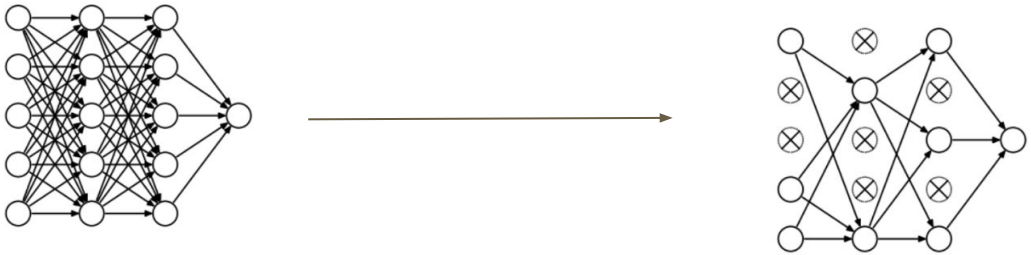


source

# Regularization - Dropout

Dropout Technique - one of the most interesting techniques in DL

Dropout produces very good results and is widely used in Deep Learning models.



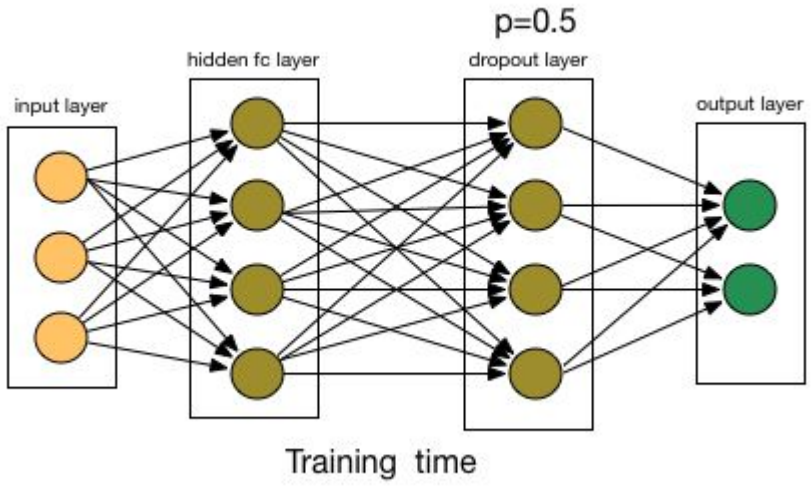
So what does dropout do?

At every iteration, it randomly selects few nodes, with probability P, and removes them along with all of their **incoming and outgoing** connections as shown above.

So each iteration has a different set of nodes and this results in a different set of outputs.

source

# Regularization - Dropout




Source: chatbotslife

# Regularization - Dropout

Dropout represents to us the idea of "**Ensemble models**".

Ensemble modeling is a process where multiple diverse models are created to predict an outcome, either by using many different modeling algorithms or using different training data sets

Ensemble models usually perform better than a single model as they capture more randomness.

Similarly, dropout also performs better than a normal neural network model.

This probability of dropout is a hyperparameter which we'll need to tweak.

```
from keras.layers.core import Dropout

model = Sequential([
 Dense(output_dim=hidden1_num_units, input_dim=input_num_units, activation='relu'),
 Dropout(0.25),

Dense(output_dim=output_num_units, input_dim=hidden5_num_units, activation='softmax')])
```

Source: chatbotslife

# Regularization - Data Augmentation

Another cool way to reduce overfitting is to use Data Augmentation.

Basically, it is the process of creating from nothing (in hebrew it sounds better...), and eventually is increasing the size of our dataset.

For example, if we are dealing with images data, we can pretty easily (and super easy with our APIs) to increase our dataset size by applying some transformations onto images.

For example:


Source

# Regularization - Data Augmentation

Why it works?

- More data gives more “insights” to the model. E.g.: the digit “2” can be also “upside-down”, shifted to left, etc.
- More diversified dataset, not only the same “1”
- increasing generalization ability of the models, now they can be much better
- helping resolve class imbalance issues in classification (mostly for imbalanced classification problems)

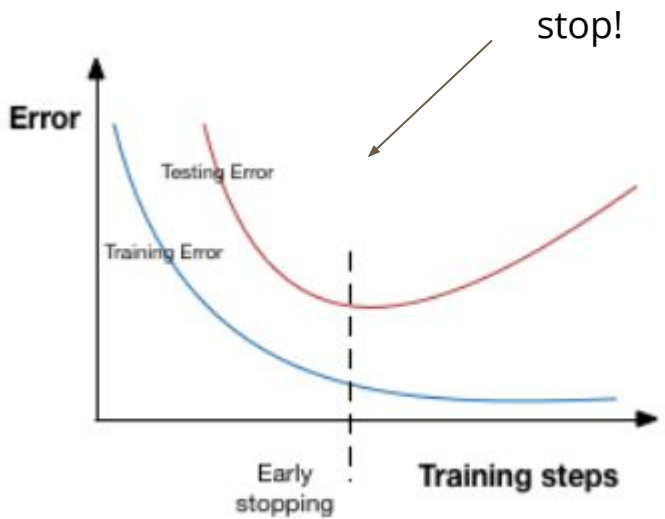
```
from keras.preprocessing.image import ImageDataGenerator
datagen = ImageDataGenerator(horizontal_flip=True)
datagen.fit(train)
```

Source

# Regularization - Early Stopping

Early stopping is a technique where we want to make sure we don't start to overfit.

Specifically, when the performance on a validation/test set gets worse from an iteration to an iteration, we will immediately stop the training process.




Source

# Regularization - Early Stopping

Early stopping is a technique where we want to make sure we don’t start to overfit.

Specifically, when the performance on a validation/test set gets worse from an iteration to an iteration, we will immediately stop the training process.

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

```
EarlyStopping(monitor='val_err', patience=5)
```

- **monitor** denotes the metric we should monitor in order to know if we are getting worse / not
- **Patience** is the number of epochs we will be “patient” about, until we’ll stop training, given there was no improvement

Source

# Regularization - Ex. time

Let’s practice Regularization in Keras!

In the following exercise, you will have a chance to implement several regularizations in Keras, and check models with and without them, in order to see who performs better (with? without?)

The Ex. name is “**Daniel_Ex7_Regularization.ipynb”**

**Good luck!**

# Regularization Dropout - Ex. time

Let’s practice Dropout!

As mentioned, one of the greatest techniques that helps in reducing overfitting in DL is Dropout.

In the following exercise, you will have a chance to try out different dropout rates (p), and will be able to check which had the greatest effect on the model, in terms of performance.

The Ex. name is “**Daniel_Ex8_Dropout.ipynb”**

(Reminder: Use kernel: *conda_tensorflow2_p36*)

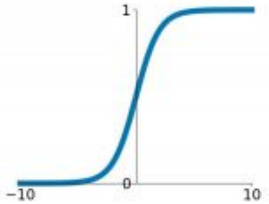
**Good luck!**

# Q&A Time

# Activation Functions

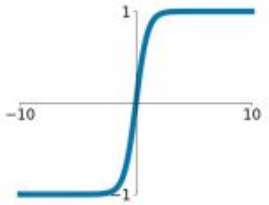
**Sigmoid**

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



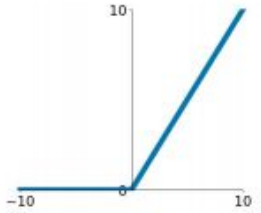

**tanh**

$$\tanh(x)$$



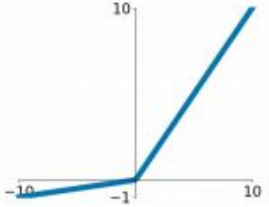

**ReLU**

$$\max(0, x)$$




**Leaky ReLU**

$$\max(0.1x, x)$$

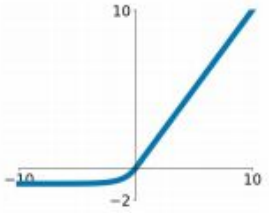



**Maxout**

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

**ELU**

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$




Source

# Activation Functions

***Why do we need Activation Functions?***

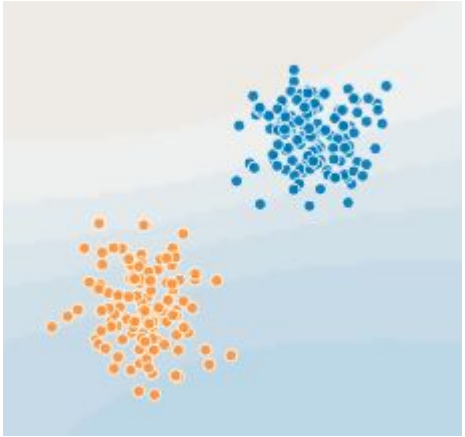
Using an activation function on a linear transformation, e.g. activation(wx+b), adds another step at each relevant layer during the forward + backward propagation.

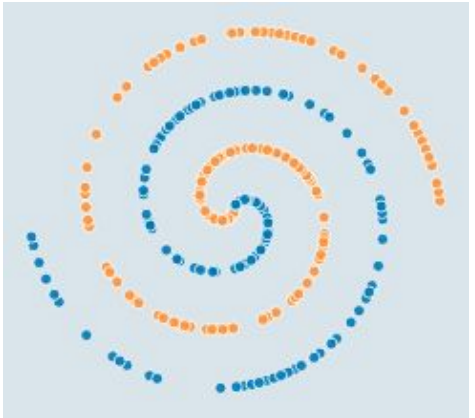
The question is - do we need it if it just adds complexity?

Think about a neural network without activation function. In this case, every neuron will generate a linear transformation on the **inputs** using the **weights & biases**.

Most of the times - our data is not linearly splitted (and for linear split we don't need NNs).

Linear transformations make the neural network simpler, but this network would be less powerful and will not be able to learn the complex patterns from the data.





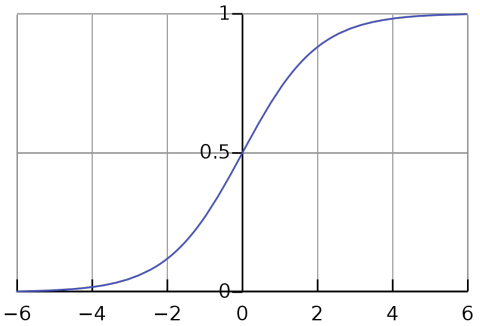
# Activation Functions

***Let's talk about the main activation functions***

**Sigmoid**

$$\sigma(x) = 1/(1 + e^{-x})$$

- Squashes input between 0 and 1
- Historically popular since they have nice interpretation as a saturating "firing rate" of a neuron
- Numeric examples:
  - sigmoid(6) = 1/(1 + e^-6) = 0.997
  - sigmoid(100) = 1/(1+e^-100) = 1
  - sigmoid(-100) = 1/(1+e^100) ~~ 0
- Problems:
  - As you see, outputs for very small "x"s can be 0
  - Output is not zero-centered
  - Exponent is an expensive operation

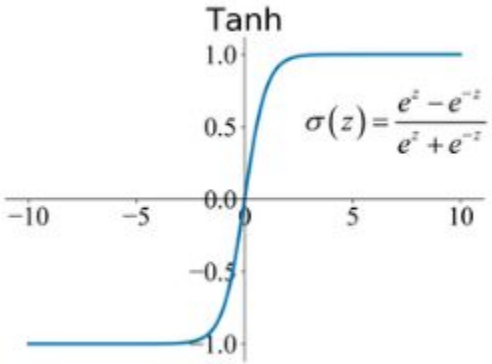


# Activation Functions

***Let's talk about the main activation functions***

**Tanh (Tangent Hyperbolic)**

- Squashes input between -1 and 1
- Zero centered
- Still kills gradients :(
- Numeric example:
  - tanh(0) = (e^6 - e^-6) / (e^6 + e^-6) = 0
  - tanh(100) = (e^100 - e^-100) / (e^100 + e^-100) = 1
  - tanh(-100) = (e^-100 - e^100) / (e^-100 + e^100) = -1

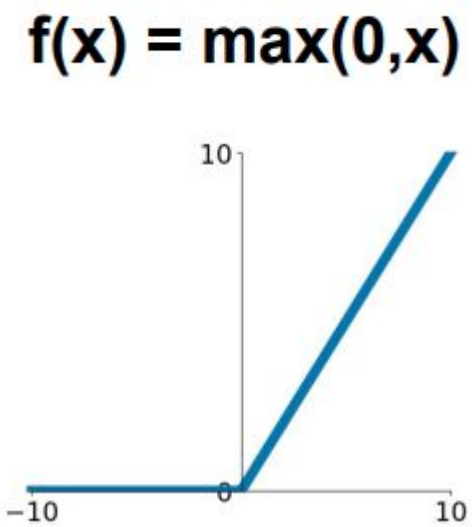


# Activation Functions

***Let's talk about the main activation functions***

**ReLU (Rectified Linear Unit)**

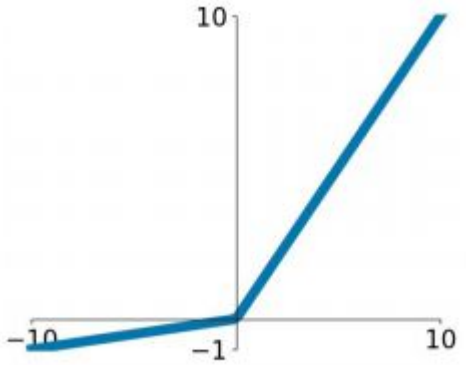
- Does not saturate (in the positive region)
- Very computationally efficient
- Not zero-centered output
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)
- Numeric example:
  - f(6) = 6
  - f(-4) = 0
  - f(0) = 0



# Activation Functions

***Let's talk about the main activation functions***

$$f(x) = \max(0.01x, x)$$




**Leaky ReLU (Leaky Rectified Linear Unit)**

- Does not saturate (in the positive region)
- Very computationally efficient
- Will not die!
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)
- Numeric example:
  - f(6) = 6
  - f(-4) = -0.04
  - f(0) = 0

(There is another version of Leaky ReLU, such as: Parametric Rectifier (PReLU)), which gets this 0.01 and turns it into a parameter (alpha).

Then, this parameter can be learned via Back Propagation

# Activation Functions

***Summary - let's see some code!***

Keras Tutorial

# Framework (Keras) Overview + Tutorial

Following some feedback I got, now it's time to have real overview on the code you are learning here.

We are going to invest the next few hour / two in understanding how Keras works, what can you do with it, and some very cool features it has. Let's go over it, you have the link if you need (for going over it within your own time)

**Keras Tutorial**

# Overfitting / Underfitting in DL

This relatively short lecture will focus on two main phenomena in DL - Overfitting and Underfitting, specifically in DL.

For that, let us define the two -

**Overfitting**:

When an algorithm fits the training data “too good”, but when a new data point arrives it knows nothing about it, and predicts poorly.

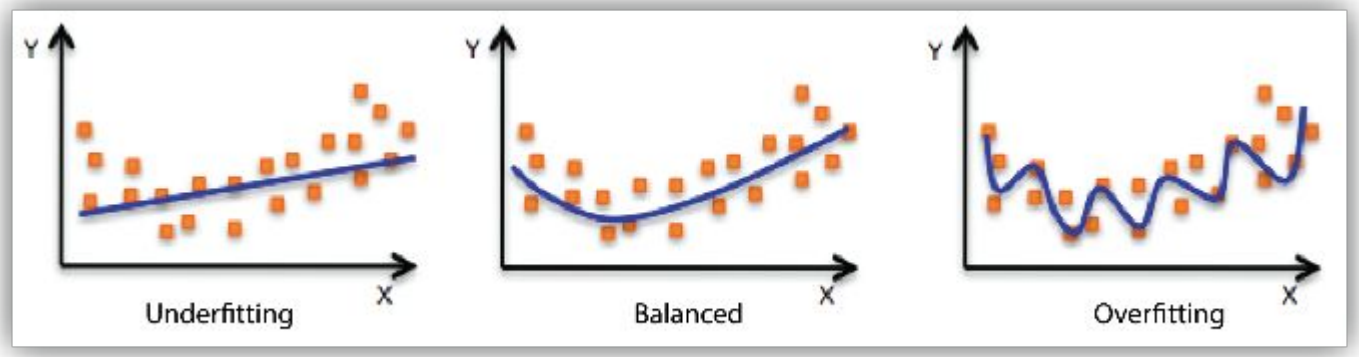
**Underfitting**:

When an algorithm is not fitting the training data well, it is called “underfitting” in terms of its performance.

Usually, this comes when a model is too simple to the data (e.g. a line for splitting 2 classes when the data is not linearly splittable)

In order to know if we are overfitting / underfitting, we MUST monitor our loss functions AND our model’s performance

# Overfitting / Underfitting in DL




source

# Overfitting / Underfitting in DL

We can take several steps when one of these phenomenon occur, and below you will be able to find few of them -

**Underfitting**:

- For example, we can add new features to our X, and more feature “products” (e.g. x1*x2)
- We can also change the types of feature processing used (e.g., increasing n-grams size in text related tasks)
- In case we are using regularization (which we discussed), we can maybe decrease the amount of regularization used, so we won’t “punish” our model too much, such that it won’t be able to learn
- Remove noise / outliers from the data (maybe they just “bother” the training process)
- Increase the number of epochs, so the model will “see” the same data at least few more times
    - **Clarification** - You can think of a for-loop over the number of **epochs** where each loop proceeds over the **training dataset**. Within this for-loop is another nested for-loop that iterates over each **batch** of samples, where one **batch** has the specified “**batch size**” number of samples

source

# Overfitting / Underfitting in DL

We can take several steps when one of these phenomenon occur, and below you will be able to find few of them -

**Overfitting**:

- Increase training data - we can for example generate or get more data (e.g. more days, more users, more images, ...)
- Feature Selection - choose the relevant features and not just “all of the data you have”. Not all of the features are relevant
- Reduce model complexity - maybe we don’t need 1000 layers of 1000 neurons each?
- Early stopping during the training phase - to make sure we don’t harm our eval loss + accuracy
- Use L1 (Lasso) and L2 (Ridge) Regularization techniques
- Use dropout for neural networks in order to diversify the models and to make sure they generalize

source

# Weight Initialization in DL

One of the core components in having good models in DL are weight initializations.

This process is a core component because it directly affects our training process.

This is because of the fact that eventually, and after all layers, neurons, convolutional, pooling, and any mathematical operations, we are willing to update our weights such that it will get us to the “best” mapping of X->Y (in supervised), and will minimize our loss.

There are many ways to initialize our weights (W, b), within this lecture we’ll discuss the main ones.

source

# Weight Initialization in DL

***Why do we want this?***

Except of getting our loss to minimum, we would like to prevent layer activation outputs from **exploding or vanishing** during the course of a forward pass through a deep neural network. If either occurs, loss gradients will either be too large or too small to flow backwards beneficially, and the network will take longer to converge, if it is even able to do so at all.

Meaning, in simple words - our gradients (which control our weight updates) will be too small or too large to be updated. In both ways, we might “undershoot” or “overshoot” our goal (which is convergence).

source

# Weight Initialization in DL

***Random Initialization***

The first initialization we'll see is random.

This means that we will initialize our weights without too much "domain knowledge" on what's going within the network.

In Keras, Random Initialization is pretty simple and requires few lines of code, which we'll see in a bit.

source

# Weight Initialization in DL

***Xavier Initialization***

This initialization is named after  Xavier Glorot, following his research with Yoshua Bengio (one of the greatest DL researches out there).

This initialization basically takes into consideration two parameters - the number of connections in the current layer, and the number of connections in the next layer.

Xavier initialization sets a layer's weights to values chosen from a random uniform distribution that's bounded between

$$\pm \frac{\sqrt{6}}{\sqrt{n_i + n_{i+1}}}$$

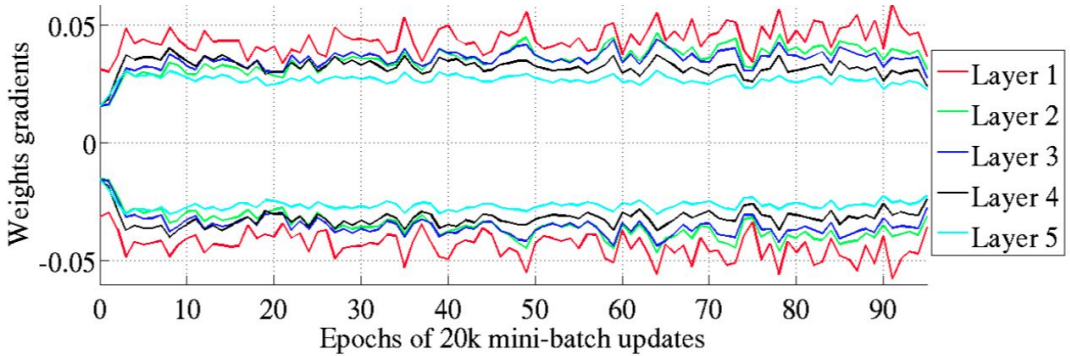
source

# Weight Initialization in DL

***Xavier Initialization***

Xavier Glorot and Bengio believed that Xavier weight initialization would maintain the variance of activations and back-propagated gradients all the way up or down the layers of a network.

In their experiments they observed that Xavier initialization enabled a 5-layer network to maintain near identical variances of its weight gradients across layers.




source

# Weight Initialization in DL

***Kaiming (He) Initialization***

One of the challenges that Xavier initialization had is the fact that there are some “non-normal” layers, such as non-linearity (e.g. Activation function such as ReLU).

This was the basic idea and motivation for Kaiming He et. al to propose their own method for weights initialization.

Their method is tailored for Deep NNs that does use these kinds of asymmetric, non-linear activations.

source

# Weight Initialization in DL

***Kaiming (He) Initialization***

In their 2015 paper, He et. al. demonstrated that Deep NNs (e.g. a 22-layer CNN) would converge much earlier if the following input weight initialization strategy will be as follows:

1. Create a tensor with the dimensions appropriate for a weight matrix at a given layer, and populate it with numbers randomly chosen from a standard normal distribution
2. Multiply each randomly chosen number by $\sqrt{2}/\sqrt{n}$ where n is the number of incoming connections coming into a given layer from the previous layer's output (also known as the "fan-in")
3. Bias tensors are initialized to zero

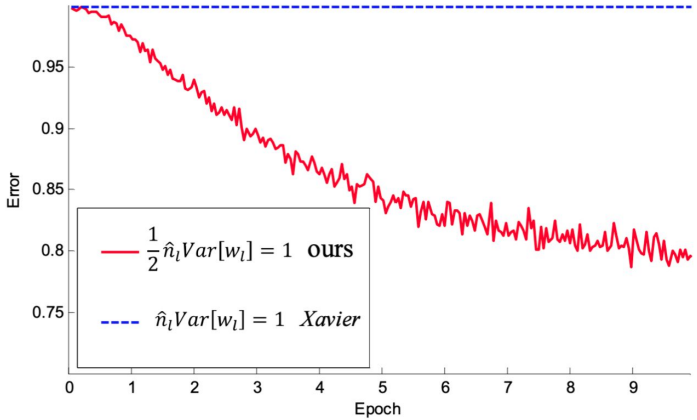
source

# Weight Initialization in DL

***Kaiming (He) Initialization***

Interesantly, when they trained even deeper networks that **used ReLUs**, He et. al. found that a 30-layer CNN using Xavier initialization didn't learn at all.

However, when the same network was initialized according to the three-step procedure we just mentioned, it enjoyed substantially greater convergence.




source

# Weight Initialization in DL

***Keras Tutorial on Layer weights initialization***

Basically, this two methods are widely used, though most of the DL networks does have some ReLU activations somewhere, so He. (Kaiming) would be the way to go.

Now, let's see how this is done using Keras

# Optimization Methods - Overview

Up till now, you have met with with one main Optimization Algorithm - Gradient Descent.

As a reminder, we are trying to minimize our loss function with respect to the models relevant parameters (usually W, B), in order to make sure we are able to minimize our errors, and thus getting better with our predictions.

However, there are several methods, and this lecture will now walk you through into the main ones which are used, alongside some concrete code examples.

# Optimization Methods

- Gradient Descent
  - Batch gradient descent (GD variant)
  - Stochastic gradient descent (GD variant)
  - Mini-batch gradient descent (GD variant)
- Challenges with the classic Gradient Descent (and its variants)
- Momentum
- Nesterov accelerated gradient
- Adagrad
- Adadelta
- RMSprop
- Adam

source

# Optimization Methods - Gradient Descent

**Gradient descent - reminder**

Gradient descent is a way to minimize an objective function $J(\theta)$ parameterized by a model's parameters $\theta \in R^d$,

by updating the parameters in the opposite direction of the gradient of the objective function $\nabla_\theta J(\theta)$ w.r.t. to the parameters. The learning rate $\eta$ (eta) determines the size of the steps we take to reach a (local) minimum.

**In simple words, we follow the direction of the slope of the surface created by the objective function downhill until we reach a valley**.

# Optimization Methods - Gradient Descent

**Gradient descent variants**

There are three variants of gradient descent, which differ in how much data we use to compute the gradient of the objective function. Depending on the amount of data, we make a trade-off between the accuracy of the parameter update and the time it takes to perform an update.

- Batch gradient descent
- Stochastic gradient descent
- Mini-batch gradient descent

# Optimization Methods - BGD

**Batch Gradient Descent**

The Vanilla version of gradient descent, aka batch gradient descent, computes the gradient of the cost function w.r.t. to the parameters θ for the **entire** training dataset:

$\theta = \theta - \eta \cdot \nabla_\theta J(\theta)$.

As we need to calculate the gradients for the whole dataset to perform just **one** update, batch gradient descent can be very slow and is intractable for datasets that don't fit in memory. Batch gradient descent also doesn't allow us to update our model **online**, i.e. with new examples on-the-fly.

# Optimization Methods - BGD

**Batch Gradient Descent**

In code, Batch Gradient Descent can look something like this:

```
for i in range(num_epochs):
  params_grad = evaluate_gradient(loss_function, data, params)
  params = params - learning_rate * params_grad
```

# Optimization Methods - SGD

**Stochastic Gradient Descent**

Stochastic gradient descent (SGD) in contrast performs a parameter update for **each** training example x(i) and label y(i):

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x(i); y(i)).$$

Meaning, every time we have a new data point (x, y), we make sure to update our parameters (w, b - theta in a generic form).

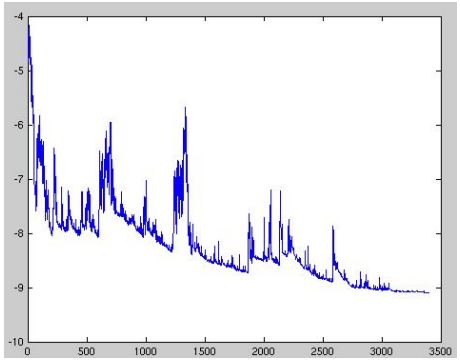
# Optimization Methods - SGD

**Stochastic Gradient Descent**

Batch gradient descent performs redundant computations for large datasets, as it recomputes gradients for similar examples before each parameter update.

SGD does away with this redundancy by performing one update at a time. It is therefore usually much faster and can also be used to learn online.

SGD performs frequent updates with a high variance that cause the objective function to fluctuate heavily, due to the "every step - update" nature of it -



# Optimization Methods - SGD

**Stochastic Gradient Descent**

While batch gradient descent converges to the minimum of the basin the parameters are placed in, SGD's fluctuation, on the one hand, enables it to jump to new and potentially better local minima.

On the other hand, this ultimately complicates convergence to the exact minimum, as SGD will keep overshooting.

Its code fragment simply adds a loop over the training examples and evaluates the gradient w.r.t. each example

In code, SGD can be seen as something like:

```
for i in range(nb_epochs):
  for example in data:
    params_grad = evaluate_gradient(loss_function, example, params)
    params = params - learning_rate * params_grad
```

# Optimization Methods - MBGD

**Mini-Batch Gradient Descent**

Mini-batch gradient descent finally takes the best of both worlds and performs an update for every mini-batch of

n training examples:

$$\theta=\theta-\eta \cdot \nabla \theta J(\theta;x(i:i+n);y(i:i+n)).$$

This way, it:

- Reduces the variance of the parameter updates, which can lead to more stable convergence
- Can make use of highly optimized matrix optimizations common to state-of-the-art deep learning libraries that make computing the gradient w.r.t. a mini-batch very efficient.

Common mini-batch sizes range between 50 and 256, but can vary for different applications.

Mini-batch gradient descent is typically the algorithm of choice when training a neural network and the term SGD usually is employed also when mini-batches are used.

Note: In modifications of SGD in the rest of this post, we leave out the parameters x(i:i+n);y(i:i+n) for simplicity.

# Optimization Methods - MBGD

**Mini-Batch Gradient Descent**

In code, MBGD can be seen as something like:

```
for i in range(nb_epochs):
  for batch in get_batches(data, batch_size=50): // a function that splits data into mini-batches
    params_grad = evaluate_gradient(loss_function, batch, params)
    params = params - learning_rate * params_grad
```

# Challenges with Vanilla Gradient Descent

**Challenges**

Vanilla mini-batch gradient descent, however, does not guarantee good convergence, but offers a few challenges that need to be addressed:

1. Choosing a proper learning rate can be difficult, this is a hyperparameter that needs to be tuned. A learning rate that is too **small** leads to painfully slow convergence, while a learning rate that is too **large** can hinder convergence and cause the loss function to fluctuate around the minimum or even to diverge

2. Learning rate “schedules” try to adjust the learning rate during training by reducing the learning rate according to a predefined schedule (every X epochs…) or when the change in objective between epochs falls below a threshold. These schedules and thresholds, however, have to be defined in advance and are thus unable to adapt to a dataset's characteristics

3. Additionally, the same learning rate applies to all parameter updates. If our data is sparse and our features have very different frequencies, we might not want to update all of them to the same extent, but perform a larger update for rarely occurring features

# GD Optimization Algorithms

**Momentum**

SGD has trouble navigating areas where the surface curves much more steeply in one dimension than in another, which are common around local optima.

In these scenarios, SGD oscillates across the slopes of the “ravine” while only making hesitant progress along the bottom towards the local optimum as in below -

Image 2: SGD without momentum

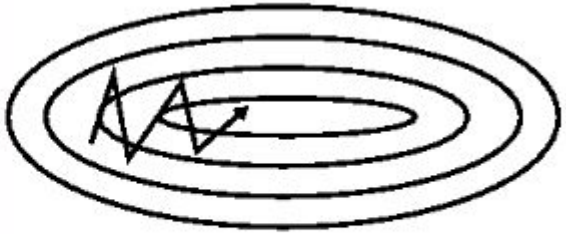


Image 3: SGD with momentum

source

# GD Optimization Algorithms

**Momentum**

In code, Momentum looks like something as:

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$
$$\theta = \theta - v_t$$

Where momentum term (γ) usually set to 0.9 or similar value.

Essentially, when using momentum, it's like we push a ball down a hill.

You can also think of γ as the "velocity" of a ball rolling downhill, building up speed (and momentum) according to the direction of the gradient/slope of the hill.

The ball accumulates momentum as it rolls downhill, becoming faster and faster on the way (until it reaches its terminal velocity if there is air resistance, i.e. γ<1).

The same thing happens to our parameter updates:

The momentum term increases for dimensions whose gradients point in the same directions and reduces updates for dimensions whose gradients change directions. As a result, we gain faster convergence and reduced oscillation.

source

# GD Optimization Algorithms

**Nesterov accelerated gradient (NAG)**

The downside of “Momentum”, However, is that a ball that rolls down a hill, blindly following the slope, is highly unsatisfactory.

We'd like to have a smarter ball, a ball that has a notion of where it is going so that it knows to slow down before the hill slopes up again.

Nesterov accelerated gradient (NAG) is a way to give our momentum term this kind of prescience.
We know that we will use our momentum term $\gamma v_{t-1}$ to move the parameters $\theta$.
Computing $\theta - \gamma v_{t-1}$ thus gives us an approximation of the next position of the parameters
(the gradient is missing for the full update), a rough idea where our parameters are going to be.

source

# GD Optimization Algorithms

**Nesterov accelerated gradient (NAG)**

We can now effectively look ahead by calculating the gradient not w.r.t. to our current parameters θ but w.r.t. the approximate future position of our parameters.

In code, NAG looks something like this:

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1})$$

$$\theta = \theta - v_t$$

source

# GD Optimization Algorithms

**RMSprop**

RMSprop was proposed by the father of back-propagation, Geoffrey Hinton. Gradients of very complex functions like neural networks have a tendency to either vanish or explode as the data propagates through the function (*refer to vanishing gradients problem).

RMSprop deals with the above issue by using a moving average of squared gradients to normalize the gradient. This normalization balances the step size (momentum), decreasing the step for large gradients to avoid exploding, and increasing the step for small gradients to avoid vanishing.

$$E[g^2]_t = 0.9E[g^2]_{t-1} + 0.1g_t^2$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

source

# GD Optimization Algorithms

**Adam**

Adaptive Moment Estimation (Adam) is another method that computes adaptive learning rates for each parameter.
In addition to storing an exponentially decaying average of past squared gradients vt like RMSprop, Adam also keeps an exponentially decaying average of past gradients mt, similar to momentum.
Whereas momentum can be seen as a ball running down a slope, **Adam** behaves like a heavy ball with friction, which thus prefers flat minima in the error surface.
We compute the decaying averages of past and past squared gradients mt and vt respectively as follows:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

source

# GD Optimization Algorithms

**Adam**

mt and vt are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients respectively, hence the name of the method.
As mt and vt are initialized as vectors of 0's, the authors of Adam observe that they are biased towards zero, especially during the initial time steps, and especially when the decay rates are small
(i.e. β1 and β2 are close to 1).
They counteract these biases by computing bias-corrected first and second moment estimates:

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t} \qquad \hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

They then use these to update the parameters just as we have seen in Adadelta and RMSprop, which yields the Adam update rule:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t.$$

source

# GD Optimization Algorithms

**Adam**

The authors propose default values of **0.9** for β1, 0.999 for β2, and 10^−8 for ϵ. They show empirically that Adam works well in practice and compares favorably to other adaptive learning-method algorithms.

As of today, Adam is one of the most used optimization algorithms out there.
So, the rule is - when in doubt, use Adam.

Also, most libraries (Keras, PyTorch) use the relevant optimizers we just discussed so it's pretty easily to use these, just by "picking" the relevant one for your trial and error.

A deeper intro to Adam including math calculations, for those interested, can be found here

source

# Project Instructions

Let's talk about the project - I want you to have something you can be proud of

1. Dataset: Fashion MNIST
2. Build & train a Deep Learning network (CNN / NN) using Keras API that will be able predict the dataset's X instance class

   (you already know how to do it)
3. Train at least 2 times the model, and log some parameters using W&B (will be learned)
4. Save the trained model in ONNX format (will be learned) - **optional**
5. Wrap the model in Flask API (will be learned)
   a. Predict API - will generate a prediction probability of the relevant class (e.g. 0.97 that this image is a Dress)
   b. Make sure that the Dataset is within the machine so you will be able to send a request with the path of some image
6. Once wrapped, wrap both the API + the model into a docker container (will be learned)
7. Project is considered done once you are able to query the model and get a prediction("Y") with a some "X", and you are able to do so within the wrapping docker container
   a. in our case, X is defined as an image
   b. In our case, Y is defined as a label

# Project Instructions

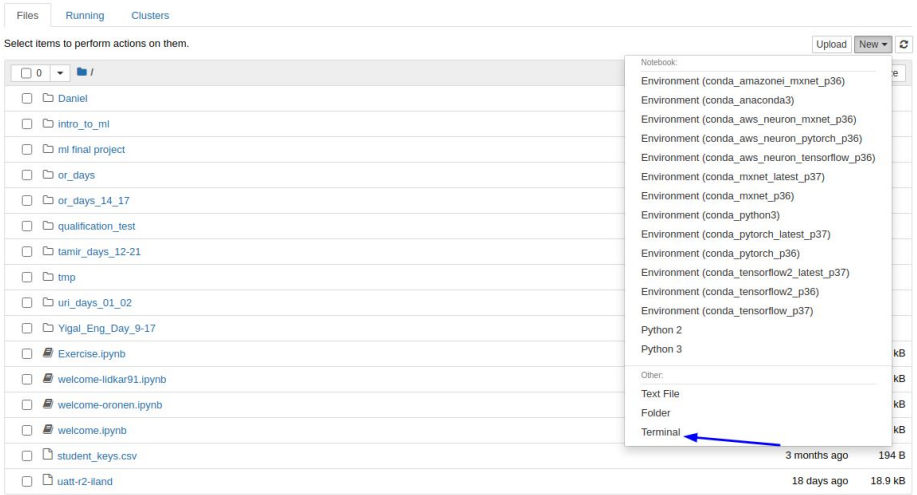
To sum up, the tech stack you are going to use are:

- Deep Learning model using **Keras API**
- **Flask API** for model hosting
- **Weights  & Biases** for experiment tracking
- **Docker** for containerizing the above
- (optional - up to you - can be relevant **Python** libraries for dataset preprocessing + **ONNX**)

A great tutorial which covers most of the stuff (ML, Flask, Docker) can be found here

# Project Instructions

The project will be done from the terminal of your machine:



# Project Instructions

You should make sure you have:

- Docker installed (just type “docker” in the terminal and see no error)

```
ubuntu@ip-172-31-18-63:~$ docker


Usage:  docker [OPTIONS] COMMAND

A self-sufficient runtime for containers
```

- Flask installed (you can check the below commands in the terminal if you want)


- **Note: In the project - make sure to use Port 8080 (As 8888 is already taken by jupyter!)**

**Project submission by Aug 5th, 2021, 23:59:00 PM, and should include the 3 below (in each student’s bucket create a folder “daniel-project”):**

- DockerFile
- DL Model code (train + test)
- Flask code

# Project Instructions

# Deep Learning - Summary Ex. (Ex 9)

***Summary for the DL chapter***

The dataset you will use is the “data.csv” (CTR dataset) which you all familiar (from the last chapter)

Pick the below features:

- User state (X)
- User isp (X)
- Device Maker (X)
- click - y (binary label 1 / 0)

# Deep Learning - Summary Ex. (Ex 9)

***Summary for the DL chapter***

The task:

- Build a NN (or CNN with Conv1D - you have Ex. 6 as a reference) and get the first 10K records (df.limi(10_000)
- Manipulate dataset (e.g. one hot encoding for the categorical features...)
- Split data into 80%-20% (train, test accordingly)
- Train the network on train data (with validation of 0.1)
- Predict the network on test data
- Have graphs of: monitor val accuracy, train accuracy, val loss, train loss
- Have a final classification report (Accuracy, F1, Recall)

(Reminder: Use kernel: *conda_tensorflow2_p36*)

**Good luck!**

# Weights and Biases

# W&B - Intro

Weights & Biases is the machine learning platform for developers to build better models faster.

Use W&B's lightweight, interoperable tools to quickly **track experiments, version** and **iterate on datasets, evaluate model performance, reproduce models, visualize results** and spot regressions, and **share findings** with colleagues.

Set up W&B in 5 minutes, then quickly iterate on your machine learning pipeline with the confidence that your datasets and models are tracked and versioned in a reliable system of record.

# What W&B contains

- ***Experiment Tracking***

  Using few lines of code, it is super easy to spin up a new experiment, track its relevant configurations, see how they increase / decrease accuracy, reiterate...

- ***Integrations***

  W&B supports integrations with many platforms e.g. PyTorch, Keras, AWS SageMaker, Databricks, ...

  In simple words, it means that we can use this tool in any modern platform today

- ***Hyperparameter Tuning***

  W&B offers “sweeps” which is a component that automates hyperparameter optimization and explore the space of possible models

# What W&B contains

- ***Data + Model Versioning***

  This is a very cool feature, in which one can version not only code but also data and model. It is known as “artifacts”

- ***Collaborative Reports***

  One can generate reports that contains relevant graphs and text about experiment, share it with stakeholders. Super relevant for **explainability, data analysis** purposes

- ***Self Hosting***

  The last component is called “self hosting”, and basically it is spinning up a server (locally / remote) that will host W&B, thus one can keep its data safe

# Let's dive into each of the components

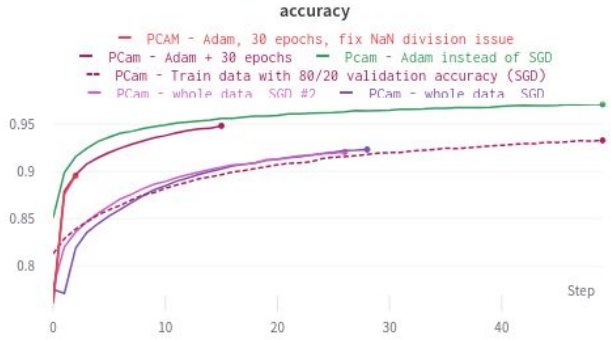
**Experiment Tracking**

What's an experiment? Essentially, it's training a model over some data and getting results.

However, the difference relies in the fact that we are trying several configurations, e.g.:

- Different learning rate (eta)
- Different optimizer (adam? sgd?)
- Different number of epochs (10? 20? 30?)
- Different data split
- ...

# Let’s dive into each of the components

# Let’s dive into each of the components

**Integrations**

As mentioned, integrations focuses on bringing this product into the main and modern ML frameworks (e.g. PyTorch), repositories (e.g. HuggingFace), services (e.g. AWS SageMaker).

It basically contains tons of information about how to customize the W&B integration into each of these, as each one of them has its own set of configurations, connection, etc.

# Let's dive into each of the components

**Integrations**

*ML / DL frameworks*

- Keras
- PyTorch
- PyTorch Lightning
- PyTorch Ignite
- TensorFlow
- TensorBoard
- Fastai
- Scikit

*ML / DL repositories*

- Hugging Face
- spaCy
- YOLOv5
- Simple Transformers
- Catalyst
- XGBoost & LightGBM

*ML / DL / other tools*

- SageMaker
- Kubeflow
- Docker
- Databricks
- Ray Tune
- OpenAI Gym

# Let’s dive into each of the components

**Hyperparameter Tuning (Optimization)**

In ML & DL, hyperparameter optimization or tuning is the problem of choosing a set of optimal hyperparameters for a learning algorithm.

A hyperparameter is a parameter whose value is used to control the learning process, for example:

- Learning rate: 0.01/ 0.02 / 0.1 / 0.001 / ...
- Early stopping (by number of epochs): 7 / 10 / 20 / ...

# Let’s dive into each of the components

**Hyperparameter Tuning (Optimization)**

There are several strategies to hyperparameter tuning, such as:

- Random Search Optimization
- Grid Search Optimization
- Bayesian Optimization

Let’s walk each one of them accordingly...

# Let's dive into each of the components

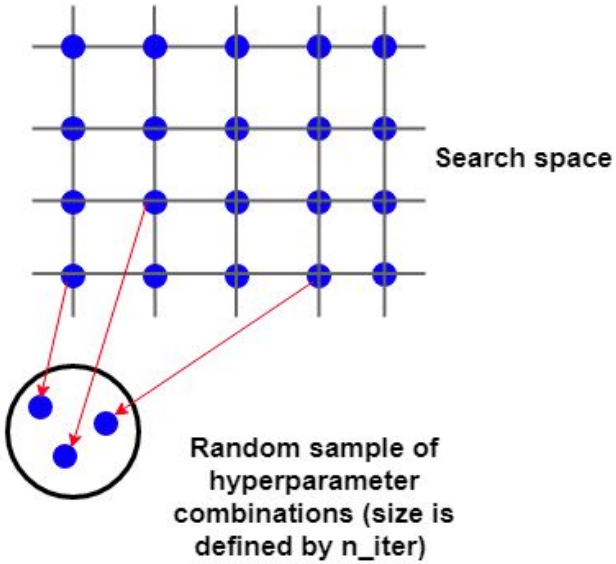
**Hyperparameter Tuning (Optimization) - Random Search**

Given a search space with some parameters,

Let us pick some random combination of them and

run the model accordingly, store results, and

then try other set of parameters.




source

# Let's dive into each of the components

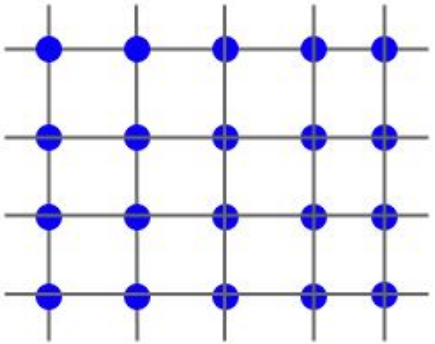
**Hyperparameter Tuning (Optimization) - Grid Search**

Given a search space with some parameters,

Let us pick **any** combination (e.g. x1, y1, x2, y2, ...)

of them and run the model accordingly, store results, and

then try other set of parameters.



source

# Let's dive into each of the components

**Hyperparameter Tuning (Optimization) - Bayesian Search**

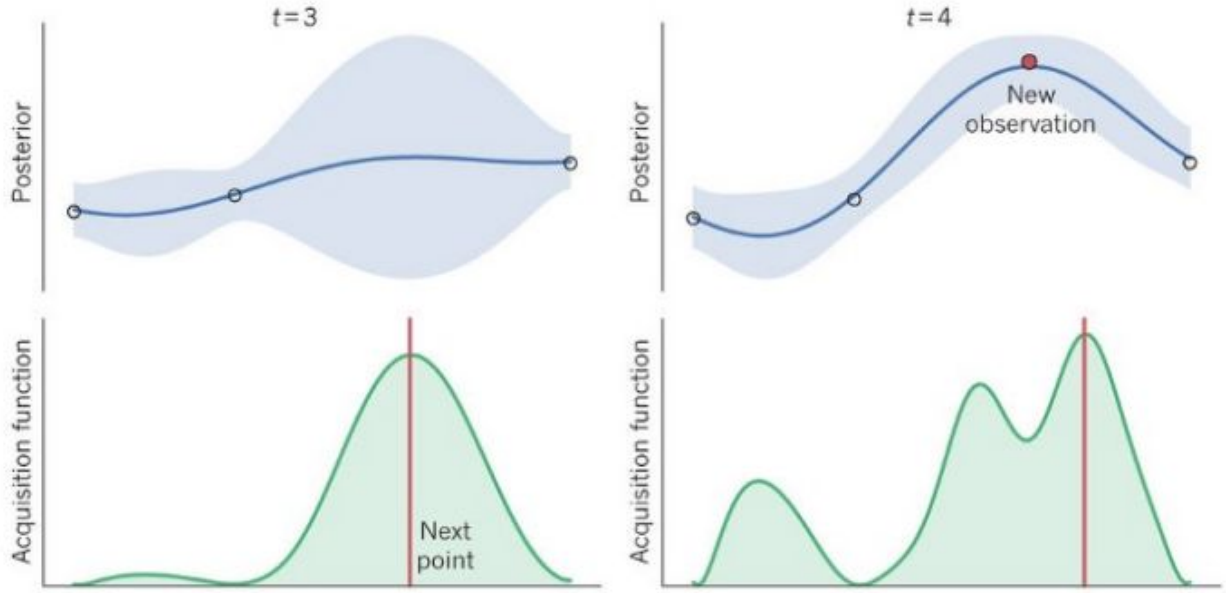
Bayesian Optimization is an approach that uses Bayes Theorem to direct the search in order to find the **minimum** or **maximum** of an objective function.

It is an approach that is most useful for objective functions that are complex, noisy, and/or expensive to evaluate

Out of the 3 methods presented, bayesian is most popular because it uses data to evaluate historical performance, and acts accordingly in the future (next iteration)

# Let's dive into each of the components

**Hyperparameter Tuning (Optimization) - Bayesian Search**




Choosing our next possible max point, with trade-off between **Exploration & Exploitation.**

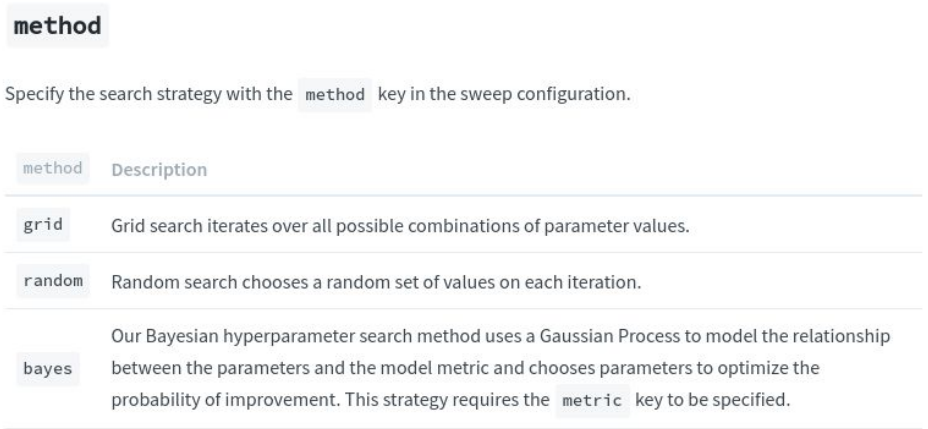
source

# Let's dive into each of the components

**Hyperparameter Tuning (Optimization)**

All of this theory is implemented as a *"sweep"* in W&B.

A "sweep" is an hyperparameter experiment. It contains several configurations in which one can have a full run of experiment, and then act accordingly!



`method`

Specify the search strategy with the `method` key in the sweep configuration.

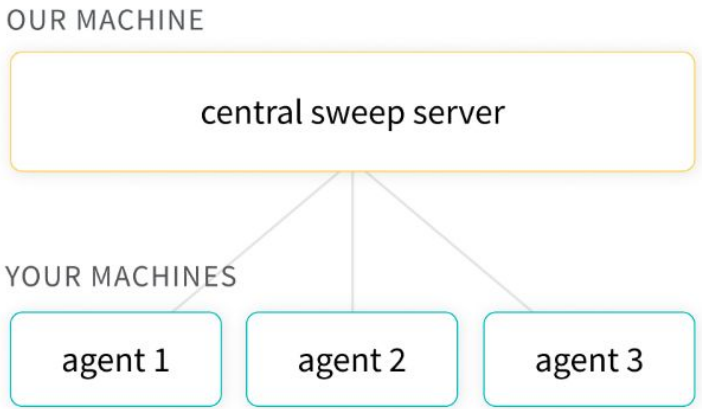
| `method` | Description |
|---|---|
| `grid` | Grid search iterates over all possible combinations of parameter values. |
| `random` | Random search chooses a random set of values on each iteration. |
| `bayes` | Our Bayesian hyperparameter search method uses a Gaussian Process to model the relationship between the parameters and the model metric and chooses parameters to optimize the probability of improvement. This strategy requires the `metric` key to be specified. |

source

# Let's dive into each of the components

**Hyperparameter Tuning (Optimization)**

It works by communicating with a W&B server, which based on our data (search space, model, parameters, ...), it knows which iteration should one try next (also based on the method), and sends it back to the agent (our servers)




source

# Let's dive into each of the components

**Data & Model Versioning (in W&B Artifacts)**

A crucial ability for **reproducibility** is having anything related to our coding versioned!

In ML projects, that is:

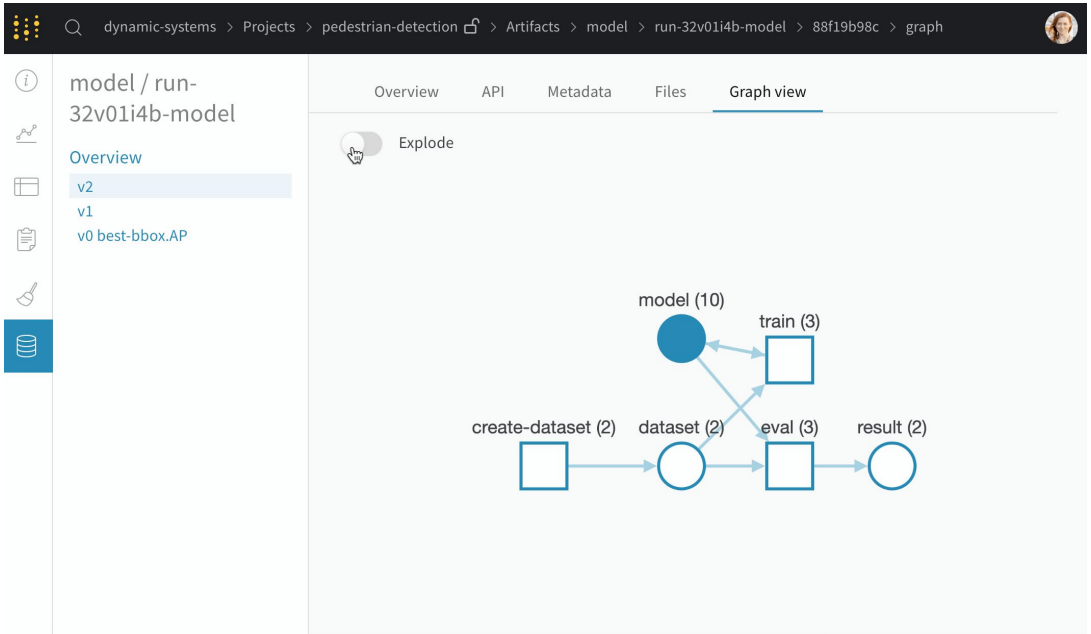
- Code + Project structure (for that we have GitHub)
- ML Model
- Data

You can think of an **artifact** as a versioned folder of data.

You can store entire datasets directly in artifacts, or use artifact references to point to data in other systems like AWS S3, GCP, or your own system.

# Let's dive into each of the components

**Data & Model Versioning**




source

# Let’s dive into each of the components

**Collaborative Reports**

Collaborative Reports let one organize and embed visualizations, describe findings, share updates with collaborators, and more.

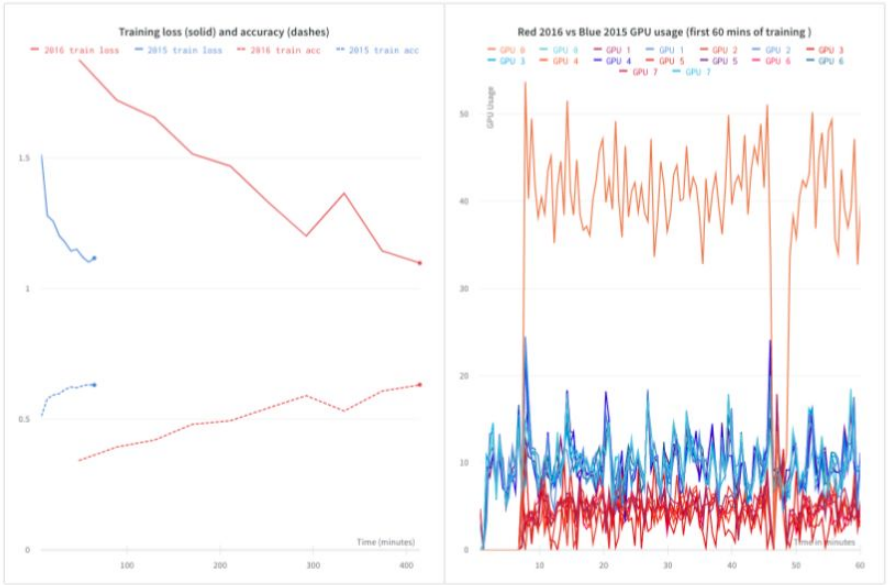
Main use-cases are:

- ***Notes***: Add a graph with a quick note to yourself
- ***Collaboration***: Share findings with your colleagues
- ***Work log***: Track what you've tried and plan next steps

# Let's dive into each of the components

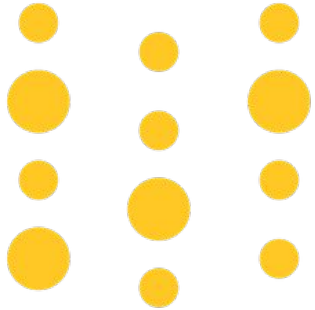
## Inception V3 parallelizes better than Inception-ResNet-V2

Plotting my GPU usage across the two models explains what's happening. The right plot below shows the GPU utilization percentage for each of 8 GPUs in red colors for the Red 2016 Inception-ResNet V2 model and in blue colors for Blue 2015 Inception V3. In the Red 2016 model, GPU 0 (top orange line) does way more work: 6-8X the work of the other 7 GPUs (dropping at the end of the first epoch, at around 46 minutes). In the Blue 2015 model, all 8 GPUs share work much more evenly. Using Inception V3 instead of Inception-ResNet-V2 for this task will let me iterate much faster.




source

# W&B - Exercise time!



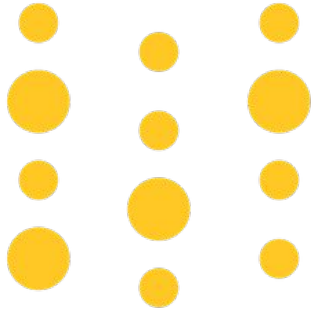
Now it's your time! Ex. name: **Intro_to_Keras_with_W&B.ipynb**

This exercise is a long one, but definitely worth it.

You will be:

- Training a CNN using Keras API
- Log some parameters using wandb
- Send data using *WandbCallback*
- Make predictions
- See these developments, data and results in w&b
- In case it is not working on our machines, you can use Colab via here

# W&B - Q&A Time



# ONNX

# What is ONNX ?

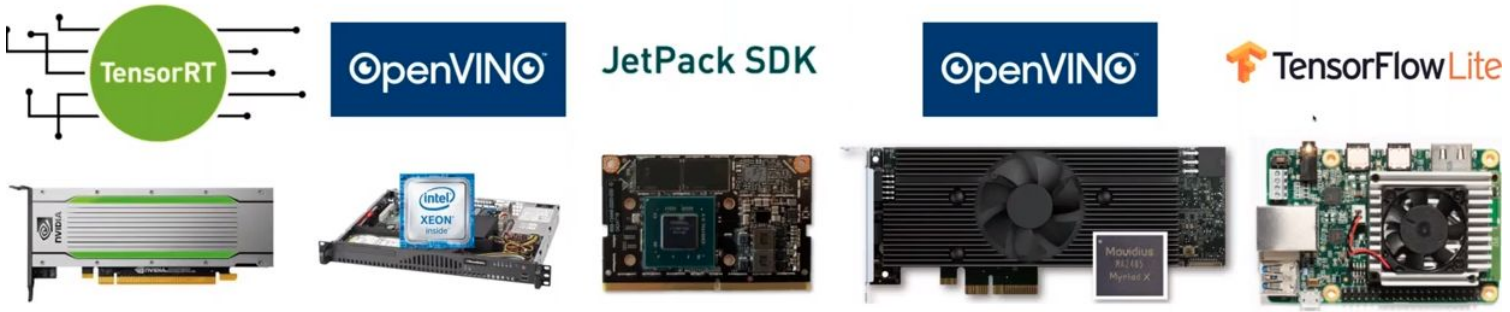
*"ONNX is an open format* ***built to represent machine learning models****. ONNX defines a common set of operators — the building blocks of machine learning and deep learning models — and a common* ***file format*** *to enable AI developers to use models with a variety of frameworks, tools, runtimes, and compilers*" (see onnx.ai).

# Why ONNX ?

There are many frameworks, each one of them has its own speciality

# Why ONNX ?

The problem: **No Interoperability** - it's not easy to move from Keras to PyTorch models, nor from developing on a machine to edge devices deployment!




source

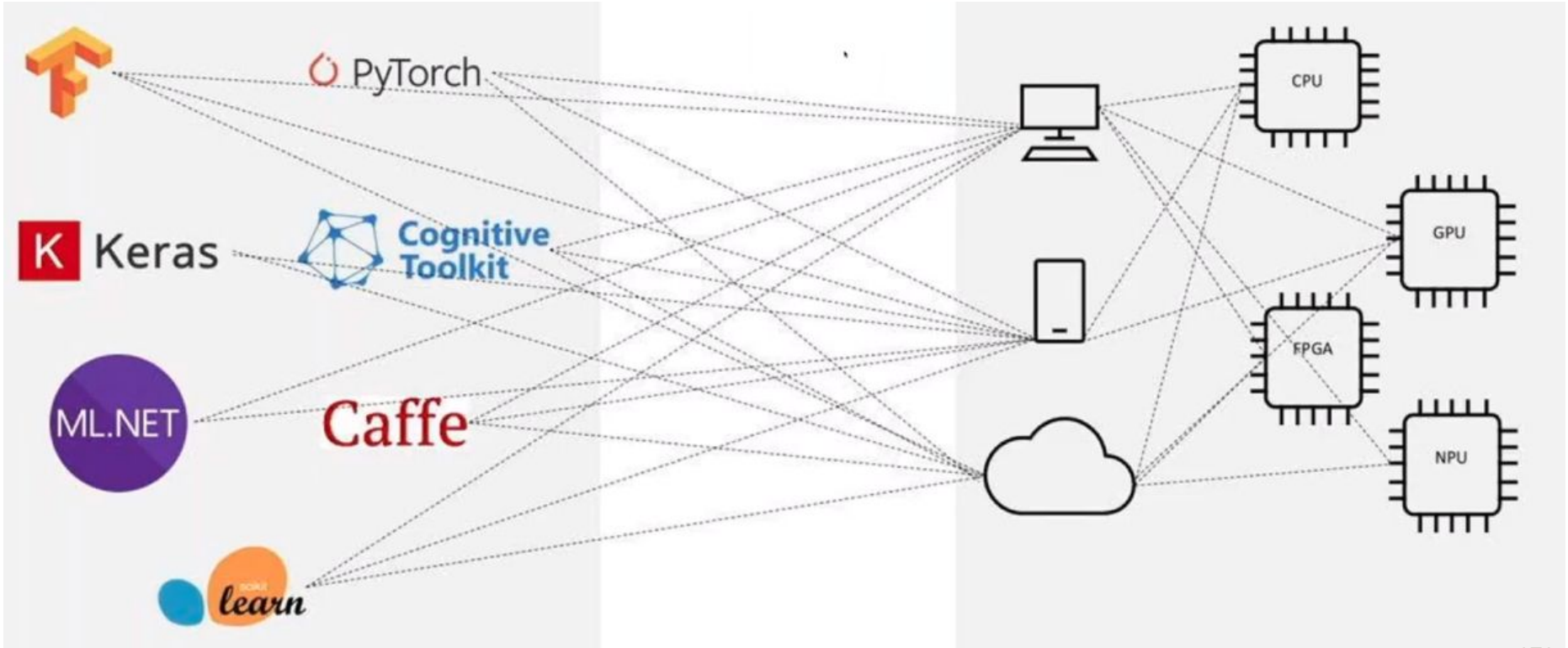
# Why ONNX ?

Notice the intense process:

1. Diverse toolkits, frameworks
2. training environments (local? cloud? dc?)
3. optimizing stack
4. production stack
5. data drift and lower accuracy? Redo #1 - #5

# Why ONNX ?

*"The ML forest"*



# Here Comes ONNX

**ONNX** - **O**pen **N**eural **N**etwork **E**xchange

The main goal of ONNX is to allow diversity in the ML space.

The specificity of ONNX even allows one to automatically compile the stored operations to lower level languages for embedding on various devices.

Effectively, an ONNX file will contain all you need to know to re-instantiate a full data processing pipeline when moving from one platform to the other (e.g. from PyTorch to Keras vice versa).

Allows to reduce a lot of moving parts when deploying and dealing with ML/DL models.

# ONNX advantages

The main goal of ONNX is to allow diversity in the ML space.

The specificity of ONNX even allows one to automatically compile the stored operations to lower level languages for embedding on various devices.

Effectively, an ONNX file will contain all you need to know to re-instantiate a full data processing pipeline when moving from one platform to the other (e.g. from PyTorch to Keras vice versa).

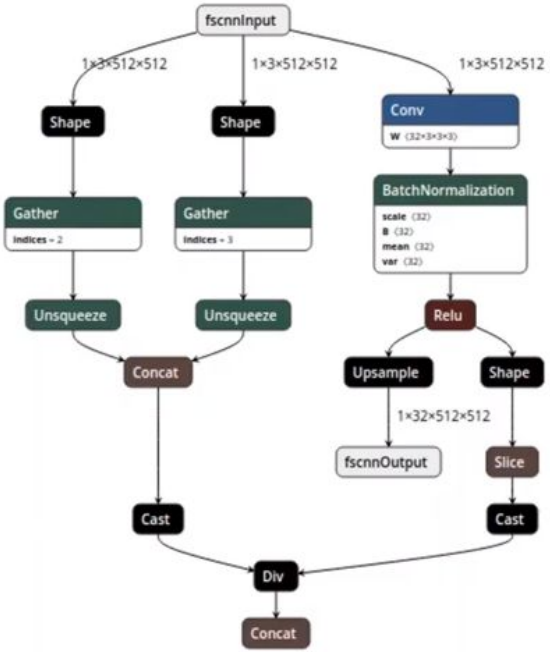
# ONNX community

# ONNX Design Principles

- Support traditional ML + DL
- Flexible enough to keep up with rapid changes
- Compact and cross-platform (supports PyTorch, Scikit-Learn, Keras, ...)
- Standardized list of well-defined **operators** informed by real world usage
- Code written in one format -> code being “translated” to ONNX format -> ONNX can read and process the code accordingly

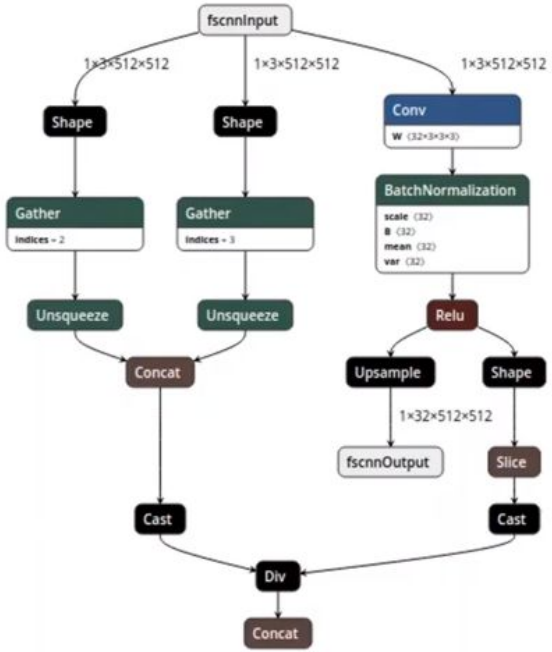
# ONNX File Format

- A model is a container for a graph that represents it (like a computation graph in NN)
- Each model has a version info, metadata, and some acyclic data flow graph
- Each graph is composed of inputs and outputs
- Each graph has its own data types



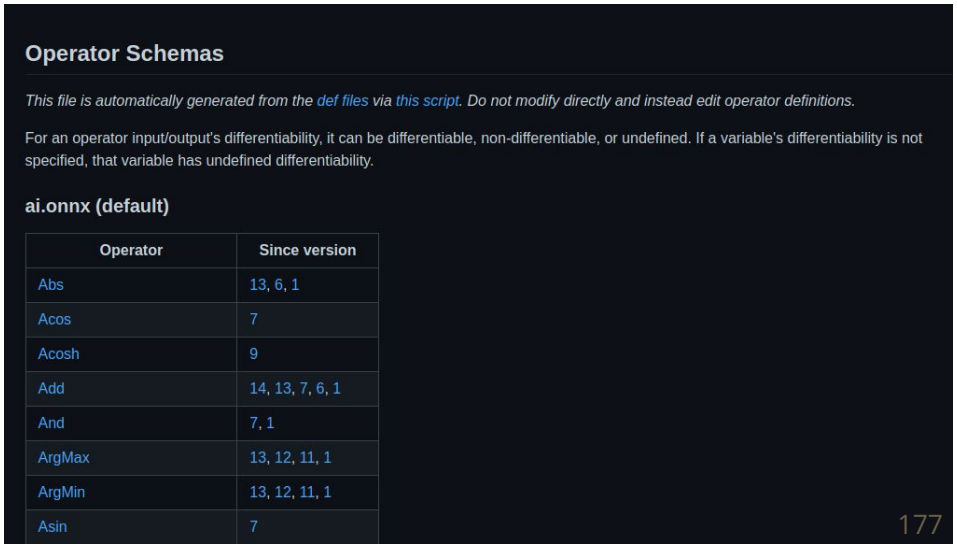
# ONNX Data Types

- **Tensor Type**
  - int8, int16, int32, int64
  - uint8, uint16, uint32, uint64
  - float16, float, double
  - Bool
  - String
  - ...



# ONNX Operators

- Maps a component in a model to ONNX operator
- Each component (e.g. Relu activation layer) is converted into an operator
- (Huge) List of supported operators can be found here



## Operator Schemas

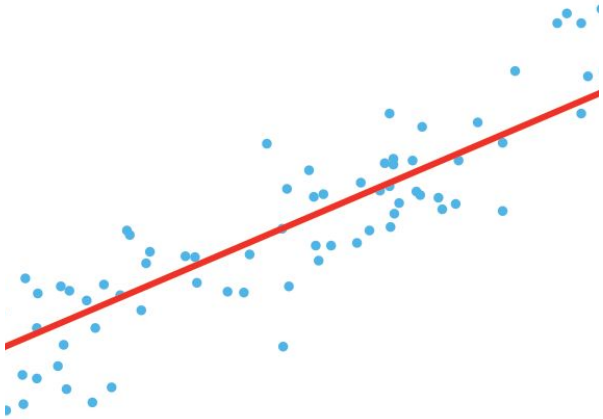
*This file is automatically generated from the def files via this script. Do not modify directly and instead edit operator definitions.*

For an operator input/output's differentiability, it can be differentiable, non-differentiable, or undefined. If a variable's differentiability is not specified, that variable has undefined differentiability.

### ai.onnx (default)

| Operator | Since version |
|---|---|
| Abs | 13, 6, 1 |
| Acos | 7 |
| Acosh | 9 |
| Add | 14, 13, 7, 6, 1 |
| And | 7, 1 |
| ArgMax | 13, 12, 11, 1 |
| ArgMin | 13, 12, 11, 1 |
| Asin | 7 |

# ONNX Use-Cases



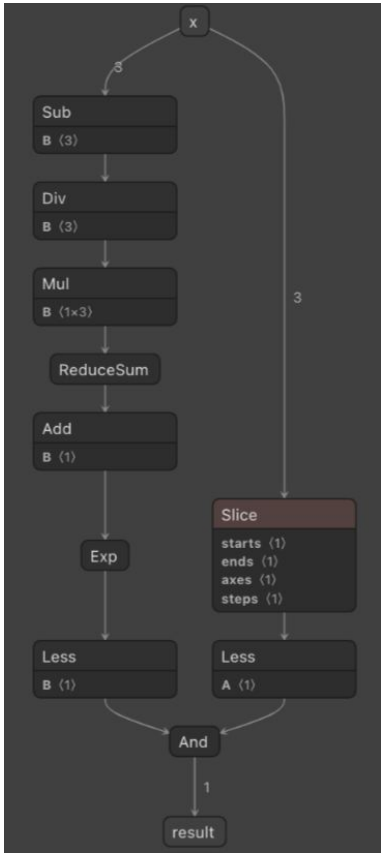
- Say you train a model to predict, for example,

  house prices (aka *Regression Problem*)
- You have several preprocessing steps to do:
  - Scaling inputs
  - Changing from categorical to numerical representation (e.g. ohe...)
  - Log transform our Y label (to make it same scale)
- Then, you can infer using a trained model
- Then, you will need to transform the price back to the original scale

# ONNX Use-Cases

- So, instead doing all of this in SKLearn, you can do it on ONNX, and then this pipeline can be relevant to any framework, ML tech, language!
- This can be done relatively easy by creating “nodes” within that graph, where each graph is responsible to a different step

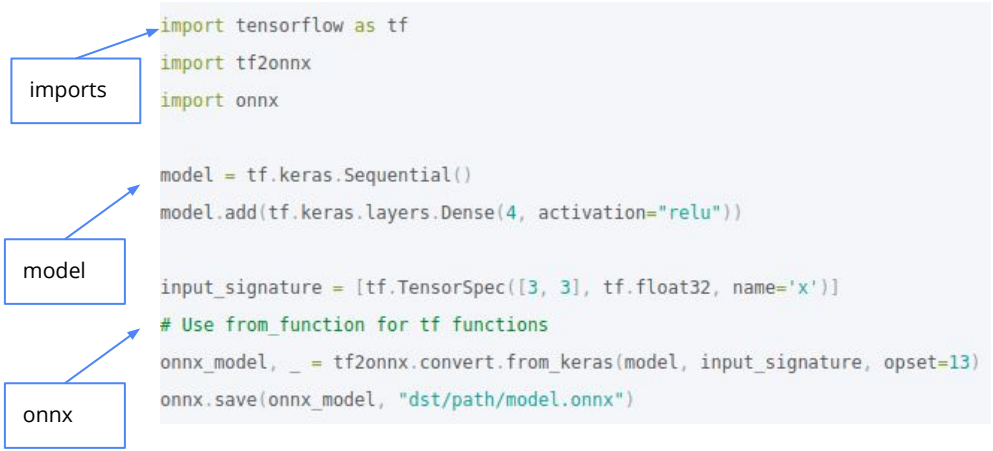
# ONNX Use-Cases



# ONNX Use-Cases

***Inference ML models - from platform to platform***

Consider the following block of code:




```
import tensorflow as tf
import tf2onnx
import onnx

model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(4, activation="relu"))

input_signature = [tf.TensorSpec([3, 3], tf.float32, name='x')]
# Use from_function for tf functions
onnx_model, _ = tf2onnx.convert.from_keras(model, input_signature, opset=13)
onnx.save(onnx_model, "dst/path/model.onnx")
```

# ONNX Use-Cases

***Inference ML models - from platform to platform***

imports

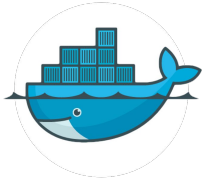
Onnx rt

assertion

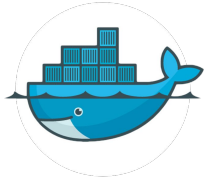
```python
import onnxruntime as ort
import numpy as np

# Change shapes and types to match model
input1 = np.zeros((1, 100, 100, 3), np.float32)

sess = ort.InferenceSession("dst/path/model.onnx")
# Set first argument of sess.run to None to use all model outputs in default order
# Input/output names are printed by the CLI and can be set with --rename-inputs and --rename-outputs
# If using the python API, names are determined from function arg names or TensorSpec names.
results_ort = sess.run(["output1", "output2"], {"input1": input1})

import tensorflow as tf
model = tf.saved_model.load("path/to/savedmodel")
results_tf = model(input1)

for ort_res, tf_res in zip(results_ort, results_tf):
    np.testing.assert_allclose(ort_res, tf_res, rtol=1e-5, atol=1e-5)

print("Results match")
```

# ONNX Q&A Time

# Docker



# What is Docker?



*"Docker helps developers bring their ideas to life by **conquering the complexity of app development**. We simplify and accelerate development workflows with an integrated dev pipeline and through the consolidation of application components"*

In short, Docker is a software container platform.

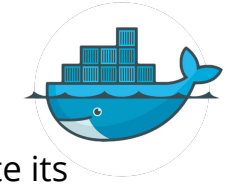
You can **create containerized applications, automate the deployment** and have fun!

**Packing, shipping, and running — made simpler, easier & definitely, faster!**

With Docker, applications can comfortably run **no matter where they are**.

Using Docker enables you to **package your entire application with all of its dependencies** with zero headache for compatibility issues or machine dependency.

# What is Docker?



Docker has a lot of components which anyone can easily use in order to automatically update its programs.

Now we'll go through the basic jargon of Docker, in order to make sure we're aligned -

- VMs
- Docker Containers
- Docker Files + Docker Hub
- Docker commands
- Docker Images

Then, we'll dive into a real ML example (which you can browse for your projects...)!

# VMs

Simplest term - VM is nothing but an emulation of a real computer that can execute programs of sorts.

A virtual machine (VM) is a program on a computer that works like it is a separate computer inside the main computer.

Alongside that, another component is called “Hypervisor”, which is basically what Virtual Machines run **on top of**.

The program that controls VMs is called a hypervisor and the computer that is running the virtual machine is called the host.

# VMs

- **VM** - a program on a computer that works like a separate computer, inside a real computer
- **Hypervisor** - VMs run on top of that component, and it controls them
- **Host** - the computer that runs VMs (aka the “real computer”)

# Docker Containers

The concept of Docker containers, similar to that of VM, also aims to achieve isolation in terms of applications and programs.

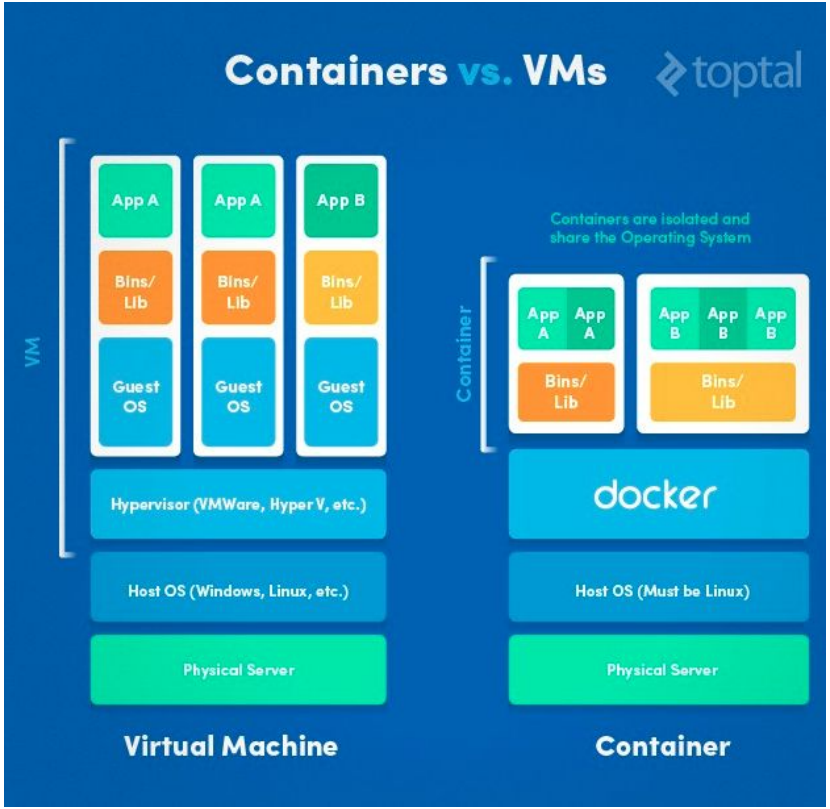
Containers, as the term suggests, **“contains”** every little thing that an application will need in order to be executed.

Think of it as a transportable box, filled with tools and libraries, configurations, settings, and all that is required for it to be executed.

# Then what's the diff between Docker Containers and VMs???

*"Virtual machines and containers differ in several ways, but the primary difference is that **containers provide a way to virtualize an OS so that multiple workloads can run on a single OS instance**. With VMs, the hardware is being virtualized to run multiple OS instances" (source)*

Containers don't require a full OS instance, allowing multiple containers to run smoothly on a single host machine.




source

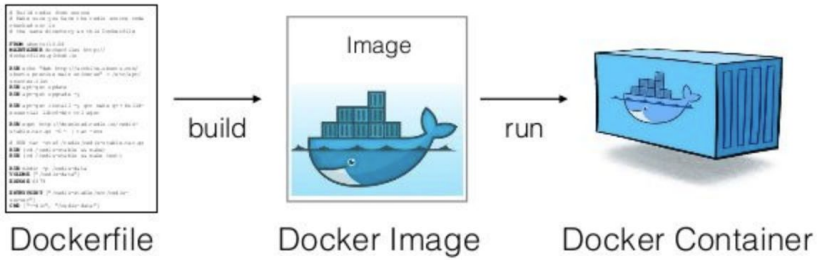
# How to create a Docker Container?

Now that we know what a docker container is, let's talk about the process for docker container creation.

In one line, **dockerfile** builds a **docker image**, from which you can run **docker container**.

A Dockerfile is a simple text file, with lines of instructions on how to build a docker image. So it's basically a blueprint for images, that's all.

Docker Images are blueprints for containers, similar to Class in Object Oriented Programming.




source

# Docker Files

Docker can build images automatically by reading the instructions from a **Dockerfile**.

A **Dockerfile** is simply a text document that contains all the commands a user could call on the command line to generate an image.

Using docker build users can create an automated build that executes several command-line instructions in succession, one after the other.

# Docker Files

Format: txt file
Content:

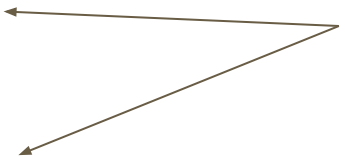
```
# Comment
INSTRUCTION arguments
```

Where instruction is a reserved word (usually 1-word) that tells **what** to do (e.g. RUN / ECHO / ...), and arguments will tell **how** to do the specified action.

Examples:

```
FROM ubuntu
CMD echo "Hello Dockerfile"
```

```
FROM ubuntu
CMD ["/usr/bin/wc","--help"]
```



**FROM** will take an existing image (like import code...)
**CMD** will run a console in the command line, in this case echo

# Docker Files

There are many commands in a dockerfile!

It helps to think about each of them as a function that gets inputs and performs them)

Let's go over the main ones:

**RUN - will run whatever instructions are given**

- RUN echo hello world -> will print to screen hello world
- RUN dir c:/ ->will run list files on c drive
- RUN ["/bin/bash", "-c", "echo hello"] -> get a set of arguments and will execute them one by one, by order, from left to right

**FROM - The FROM instruction initializes a new build stage and sets the Base Image for subsequent instructions.**

As such, a valid Dockerfile must start with a FROM instruction. The image can be any valid image – it is especially easy to start by pulling an image from the Public Repositories.

The public repositories is known as **Docker Hub** (we'll get there soon)!

*(All taken from here)*

# Docker Files

**ARG - allows to define arguments that will be used somewhere within the dockerfile, like variables inside a specific function**

- `ARG CODE_V=latest`
- `FROM my_relevant_image:${CODE_V}`

**CMD - Will perform relevant actions in the command line, as you would have write these manually**

- `CMD  /code/run-app.py --eta 0.1 --num_layers 10`
- `CMD  ["/code/run-app.py", "--eta 0.1", "--num_layers 10"]`

**The LABEL instruction adds metadata to an image. A LABEL is a key-value pair (think of tags)**

- `LABEL "com.example.vendor"="ACME Incorporated"`
- `LABEL com.example.label-with-value="foo"`
- `LABEL version="1.0"`
- `LABEL description="This text illustrates \`
  `that label-values can span multiple lines."`

# Docker Files

**ENV - Defines environment variables. This value will be in the env for all subsequent instructions in the build stage**

- `ENV MY_NAME="John Doe"`
- `ENV MY_DOG=Rex\ The\ Dog`
- `ENV MY_CAT=fluffy`

**ADD - adds relevant configurations / files to whatever you basically want**

- `ADD --chown=55:mygroup files* /somedir/`
- `ADD test.txt /absoluteDir/`

**COPY - as it is, copies whatever you basically want FROM TO**

- `COPY hom* /mydir/`
- `COPY hom?.txt /mydir/`

# Docker Files

**Order** matters in a docker file!

There are specific order rules that one should follow in order to make sure the dockerfile is indeed valid.

1. Place static instructions higher in the order. Instructions like, but not limited to, EXPOSE, VOLUME, CMD, ENTRYPOINT, and WORDIR whose value is not going to change once it is set
2. Place dynamic instruction lower in the order. Instructions like ENV (when using variable substitution), ARG
3. Place dependency RUN instructions before ADD or COPY instructions
4. Place ADD and COPY instructions after RUN instructions for installing dependencies but before dynamic instructions

# Docker Files - a full example

```
FROM ubuntu


RUN \
 apt-get update && \
 apt-get install -y wget sudo supervisor python-software-properies && \
 apt-get update && \
 add-apt-repository ppa:saltstack/salt && \
 apt-get update


EXPOSE 4505 4506


CMD ["/sbin/sshd"]
```

Note how we combined few “RUN” commands

# Docker Images

**Docker Images** are the output of the Dockerfiles.

This basically means that using the **dockerfile**, we can build a **docker image**, that will eventually can be run and thus generate us the **docker container**.

We can create Docker Images / we can use existing ones!

For using existing ones, there's the "github" of Docker - "*Docker Hub*".

Let's have a quick review of Docker Hub

# Docker Commands

- **docker run** - a command that takes a docker image, and creates a docker container from it
- **docker start –** starts one or more stopped containers
- **docker stop –** stops one or more running containers
- **docker build –** builds an image from a Dockerfile
- **docker pull –** pulls an image or a repository from a registry
- **docker push –** pushes an image or a repository to a registry
- **docker export –** exports a container's filesystem as a tar archive
- **docker search –** searches the Docker Hub for images

# Docker Flags

**Given this example:**

```
docker run -d \
-v "/${PWD}:/workspace" \
-p 8080:8080 \
--name "ml-workspace" \
--env AUTHENTICATE_VIA_JUPYTER="mytoken" \
--shm-size 2g \
--restart always \
dagshub/ml-workspace:latest
```

*the -d flag causes Docker to start the container in "detached" mode. A simple way to think of this is to think of **-d** as running the container in "the background," just like any other Unix process*

*-v stands for “volume”. It allows you to retain your work (files) after the container shuts down, and to access them from outside the container. It does this by mapping your current working folder (where you execute the docker run command), denoted as /${PWD}, to a /workspace folder inside the container's virtual file system. If you'd like to change that, you can change this argument appropriately*

*The -p argument exposes the 8080 port. In essence it means that after you run this on a computer, your container will be accessible via http://{computer-ip}:8080. If you're running this on your local system, that address will be http://localhost:8080*

*This generates a unique identifier for our container for future reference. As it might imply, this name should be unique per your system, so if you make multiple containers from the same image, you'll need to define different names for them. --name is also useful to add meaning to our container. If you don't define a name, a meaningless one will be generated for you automatically*

# Docker Flags

**Given this example:**

*docker run* ***-d*** \

***-v*** *"/${PWD}:/workspace"* \

***-p*** *8080:8080* \

***--name*** *"ml-workspace"* \

***--env*** *AUTHENTICATE_VIA_JUPYTER="mytoken"* \

***--shm****-size 2g* \

***--restart*** *always* \

*dagshub/ml-workspace:latest*

*The --env flag defines environment variables for your container. This can vary wildly between containers, and so it's hard to give a generic use case for it.*

*This flag is used to define the shared memory of your container (the more the better). Remember that this uses the same RAM as your regular system, so if you set it too high it might slow your computer down. A good size would be somewhere between 2g and 8g for most use cases*

*A restart policy controls whether the Docker daemon restarts a container after exit.*
*Using the always option which means Docker will try to keep the container running even if the system restarts. This is great in order to keep your project context intact.*

There are many flags and commands, you can see all here

# Docker - The ML Full Use-Case + Recipe

**Let's say we are developing a ML model, and we want to containerize it, so we will be able to eventually host it, meaning we want to "dockerize" our model**.

The steps needed are:

- Install Docker
- Create Dockerfile (as an empty txt file)
- Fill in Dockerfile with as many as relevant packages, installations that we need
  - Mostly we can get it with FROM ubuntu:18.04 (for example)
- Add requirements.txt file with the relevant python packages (e.g. Pandas, TF, Keras, ...)
  - We can use *pip freeze* to get list of packages + specific versions

# Docker - The ML Full Use-Case + Recipe

**Let's say we are developing a ML model, and we want to containerize it, so we will be able to eventually host it, meaning we want to "dockerize" our model**.

The steps needed are:

- Add to Dockerfile "RUN" command with relevant packages (e.g. apt-install python3-dev - a minimal version of python)
- Add to Dockerfile "RUN" command with relevant packages from requirements.txt (i.e. *RUN pip install -r requirements.txt*)
- Build your dockerfile in order to make it a Docker Image (use *"Docker build -f dockerfile"*, for example)
- "Docker run" your docker image
- Eventually, you should be able to do anything in that new containerized environment
    - Let's say we have a train script called train.py, you should be able to run *python train.py* inside the docker container!
    - Let's say we have a test script called test.py, you should be able to run *python test.py* inside the docker container!
    - Let's say you had a file "app.py" that contain some Flask API logic, for example (:-)), you should be able to host and activate it within the Docker container (Note: Porting issues)

# Flask (a short intro)

# What is Flask?

Flask is a web application framework written in Python.

Flask is very **Pythonic**.

It's easy to get started with Flask, because it doesn't have a huge learning curve.

On top of that it's very explicit, which increases readability.

Basically, a web app framework represents a collection of libraries and modules that enable web application developers to write applications without worrying about low-level details such as protocol, thread management, and so on. Flask manages all of it!

Let's go over Flask in a more detailed way!

# What is Flask?

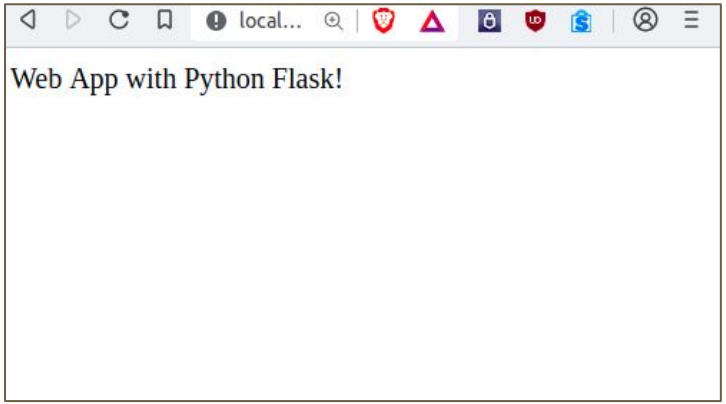
To get started with Flask, all you can run is few lines of basic code:

*(**script**)*

```
from flask import Flask

app = Flask(__name__)



@app.route('/')

def hello_world():

    return 'Web App with Python Flask!'

if __name__ == '__main__':

    app.run()
```

*(**terminal**)*

```
python server.py

 * Serving Flask app "hello"

 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```



# Q&A Time

# A full ML example

## (training/pre-trained, prediction, hosting and containerizing...)

# Use Case

Let's say we want to:

- Develop a model that will know to classify images correctly (with Keras)
- Create an API for the model (with Flask)
- Containerize it (using Docker) so we'll be able to run it on multiple environments

*Inspired by this*

# ImageNet

ImageNet is an Image Database, and it is a “pivot” for many many DL models.

Let’s go over it.

It contains 1000 classes (full list here), e.g.:

*{...*

*54: 'hognose snake, puff adder, sand viper',*

*55: 'green snake, grass snake',*

*56: 'king snake, kingsnake',*

*57: 'garter snake, grass snake',*

*58: 'water snake',*

*...}*

# Deep Learning Model - Meet the best technique for baselines!

Instead of developing from scratch, a lot of ML practitioners use what is known as “pre-trained” models.

Meaning, a model that was trained on some task (e.g. ImageNet), and now if we need it for another task (e.g. classify dogs vs. cats), we can definitely use it and thus “save” ourselves precious time.

Training time can be hours, days, … so sometimes it is worth this effort.

This technique is also called “transfer learning”, because we transfer our learning from one task to another, and it is very common in the industry to use that technique.

# Deep Learning Model - Meet the best technique for baselines!

Luckily, for our use-case, Keras supports pre-trained models which can be pretty easily used.

Also, feel free to check Keras pre-trained models (Google it)

```
import tensorflow as tf
from keras.models import load_model
from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array
from keras.applications.vgg19 import preprocess_input
from keras.applications.vgg19 import decode_predictions

model = tf.keras.models.load_model('./weights/my_model.h5', compile=False)
```

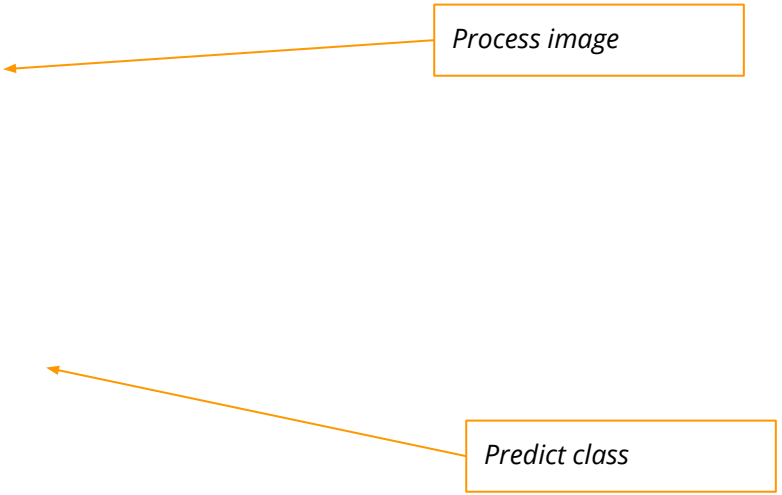
*Relevant imports*

*Loading a pretrained model (the weights were downloaded before)*

Following this step, we have a Keras model which can be used for training, evaluation, prediction, and everything we know

# Deep Learning Model - test code

```python
def process_image(image):
    # convert the image pixels to a numpy array
    image = img_to_array(image)
    # reshape data for the model
    image = image.reshape((1, image.shape[0], image.shape[1], image.shape[2]))
    # prepare the image for the VGG model
    image = preprocess_input(image)

    return image

def predict_class(image):
    # predict the probability across all output classes
    yhat = model.predict(image)
    # convert the probabilities to class labels
    label = decode_predictions(yhat)
    # retrieve the most likely result, e.g. highest probability
    label = label[0][0]
    # return the classification
    prediction = label[1]
    percentage = '%.2f%%' % (label[2]*100)

    return prediction, percentage
```




We can also write relevant functions that will be used for preprocessing, predicting, etc.

# Deep Learning Model

```
if __name__ == '__main__':
    ''' for test'''
    # load an image from file
    image = load_img('../image.jpg', target_size=(224, 224))
    image = process_image(image)
    prediction, percentage = predict_class(image)
    print(prediction, percentage)
```

*We are ready, let's test the model*

# Next - Flask API!

Now that we are happy with our model, it’s time to wrap it up with an API.

Flask helps us a lot with that, as we’ve seen!

As a reminder, Flask is a micro web framework written in Python, it provides functionalities for building web applications, managing HTTP requests, rendering templates and so on.

We can also use Flask-Uploads which allows your application to flexibly and efficiently handle file uploading and serving the uploaded files.

# Next - Flask API!

```python
from flask import Flask, render_template, request
from flask_uploads import UploadSet, configure_uploads, IMAGES

from keras.preprocessing.image import load_img
# the pretrained model
from model import process_image, predict_class

app = Flask(__name__)

photos = UploadSet('photos', IMAGES)

# path for saving uploaded images
app.config['UPLOADED_PHOTOS_DEST'] = './static/img'
configure_uploads(app, photos)

# professionals have standards :p
@app.route('/home', methods=['GET', 'POST'])
def home():
    welcome = "Hello, World !"
    return welcome
```

*Relevant imports*

*Instantiation (app, photos)*

*configuration*

*Routing (HTTP GET, POST)*

# Flask with HTML enrichment

Now, let's say we want to add a new route to our app. We can create, for example, an ***upload.html*** file that will be used for this purpose.

```
<html>
<head>
  <title>Upload</title>
</head>
<body>
<form method=POST enctype=multipart/form-data action="{{ url_for('upload') }}"
    <input type=file name=photo>
    <input type="submit">
</form>
</body>
</html>
```

# Flask with HTML enrichment

So we are ready with our HTML, we can update our **upload.py** route file (from before) -

```
@app.route('/upload', methods=['GET', 'POST'])
def upload():
    if request.method == 'POST' and 'photo' in request.files:
        filename = photos.save(request.files['photo'])

        image = load_img('./static/img/'+filename, target_size=(224, 224))
        image = process_image(image)
        prediction, percentage = predict_class(image)

        answer = "For {} : <br>The prediction is : {} <br>With probability = {}".format(filename, prediction, percentage)
        return answer
    return render_template('upload.html')


if __name__ == '__main__':
    #app.run(debug=True)
    app.run(host='0.0.0.0', debug=True)
```

Run python **upload.py** in CMD line, and then go to, for

example - *http://localhost:5000/upload*)

**Note: In the project - make sure to use Port 8080**

# Containerize our app with Docker

Model Development done, Flask API **done** - now we need to containerize it!

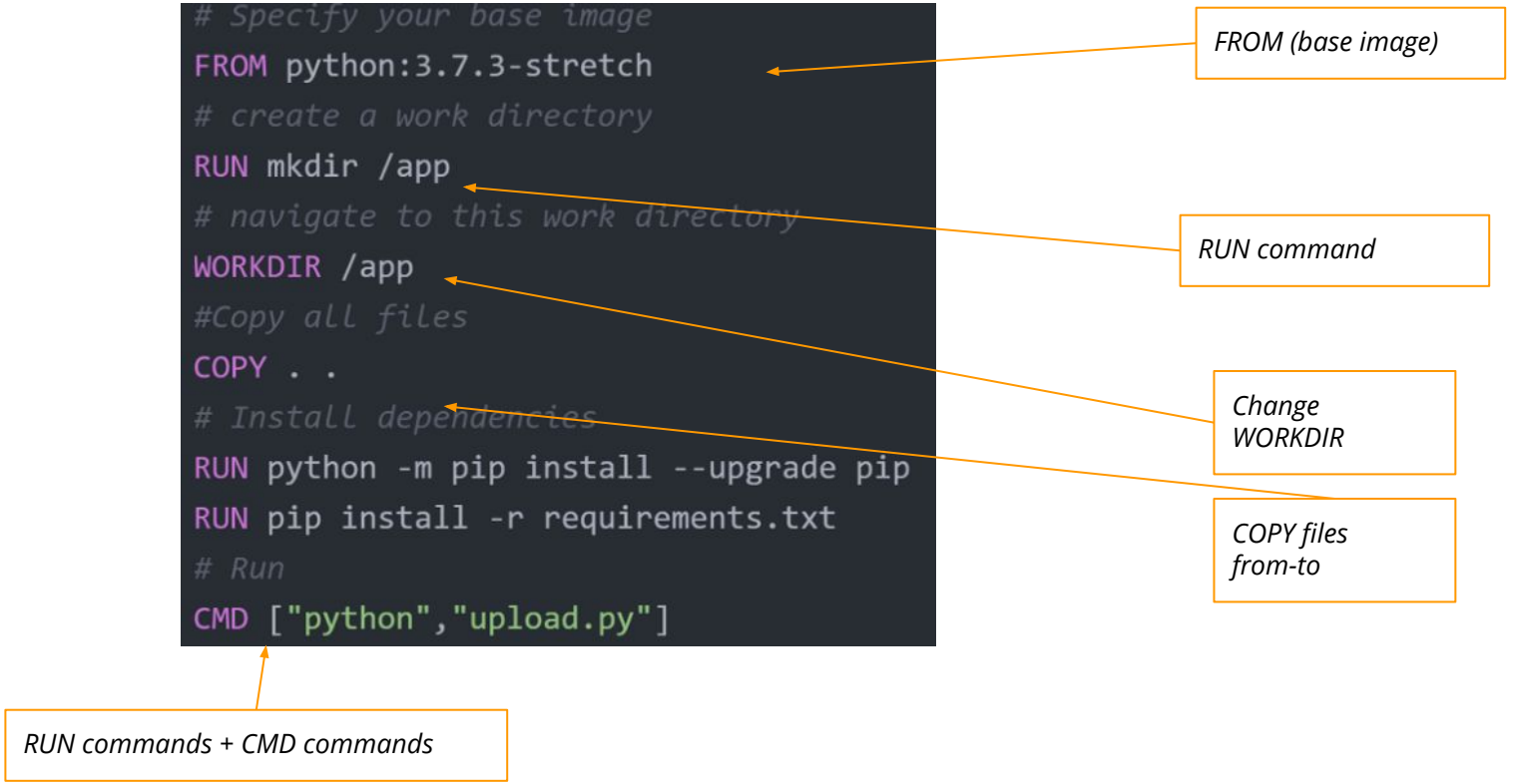
Reminder: In short, Docker allows us to create reproducible environments. So if you're moving your application to a Cloud resource - and generally you will - then you can easily and surely deploy it without worrying about the dependencies, versions or recipient system.

1. Make sure you have Docker installed
2. Create a file called requirements.txt in the main directory, and add the following:

```
Flask==1.1.2
Flask-Reuploaded==0.3.2
tensorflow==2.3.1
Keras==2.4.3
Keras-Preprocessing==1.1.2
...
(in case you need other, you
can use pip freeze)
```

# Containerize our app with Docker

3. Create a Dockerfile which contains the instructions for building your Docker image:



# Containerize our app with Docker

4. In a terminal, run the following command to build the Docker image:

```
docker build -f Dockerfile -t recog_container:api .
```

```
docker run -p 5000:8080 -d recog_container:api
```

5. Once this is running, you should be able to view your app running in your browser at

http://localhost:5000/upload

Using the upload API, one can upload images, get a prediction, and finalize the project :-)