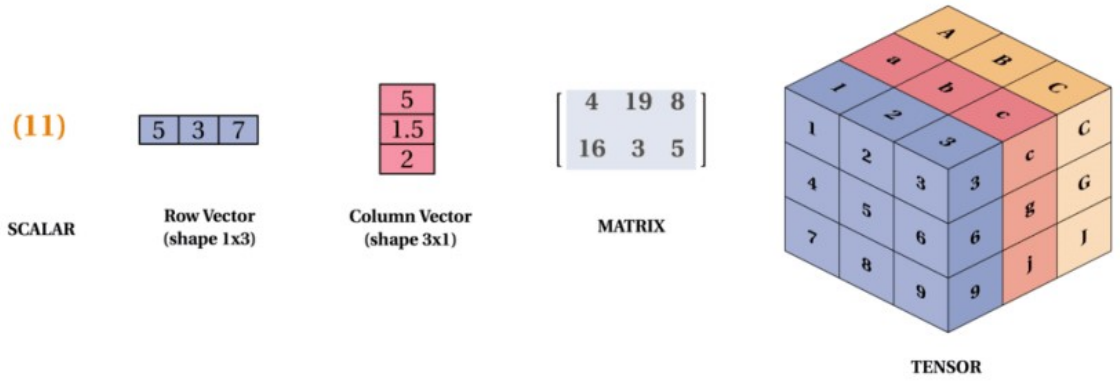
TensorFlow

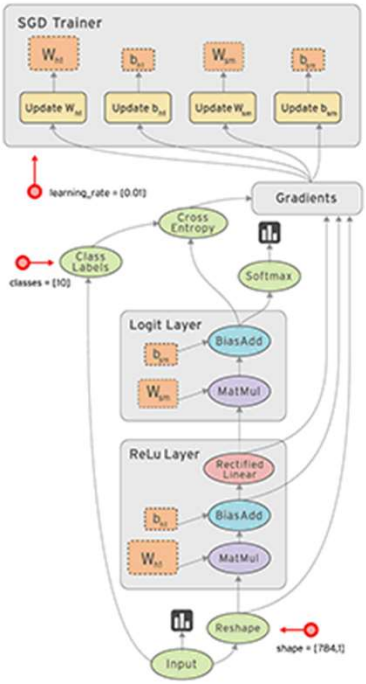
Tensor

- In mathematics, a tensor is an algebraic object that describes a multilinear relationship between sets of algebraic objects related to a vector space
- Tensors are the standard way in which data is represented in TensorFlow

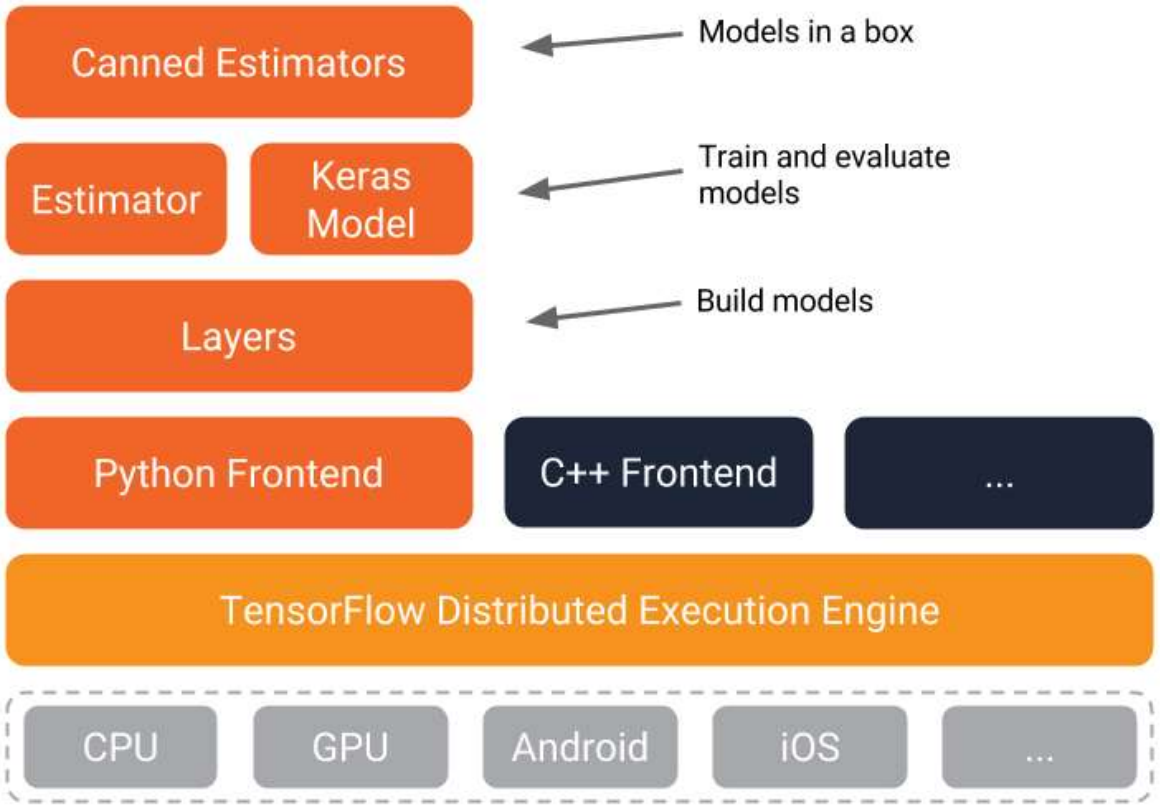


TensorFlow

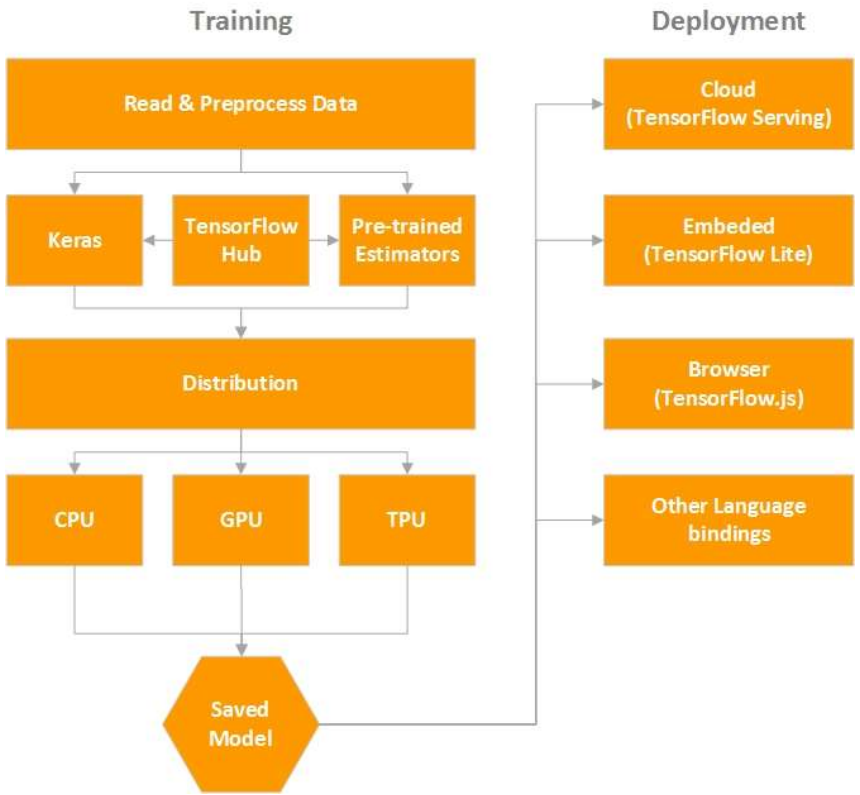
- TensorFlow is an end-to-end open source platform for machine learning. It has a comprehensive, flexible ecosystem of tools, libraries and community resources that lets researchers push the state-of-the-art in ML and developers easily build and deploy ML powered applications.
- TensorFlow is simply referring to the flow of the Tensors in the computational graph.



TensorFlow Architecture



TensorFlow Architecture



Installation and Basic Practice

- Consult**:** https://www.tensorflow.org/install
  - **pip install tensorflow** will do the job just fine
- You can configure TensorFlow for GPU and CPU
  - In late 2010, Stanford researchers found that GPU was also very good at matrix operations and algebra so that it makes them very fast for doing these kinds of calculations. Deep learning relies on a lot of matrix multiplication.
- Let's do some basic practice with TensorFlow
  - Initialization of Tensors
  - Mathematical Operations
  - Indexing
  - Reshaping
  - Placeholders
  - Graphs

PyTorch

Tensors in PyTorch

- Tensors are a specialized data structure that are very similar to arrays and matrices. In PyTorch, we use tensors to encode the inputs and outputs of a model, as well as the model's parameters.
- Tensors are similar to NumPy's ndarrays, except that tensors can run on GPUs or other hardware accelerators.

```
import torch
import numpy as np

data = [[1, 2],[3, 4]]
x_data = torch.tensor(data)
```

Datasets and Data Loaders

- Code for processing data samples can get messy and hard to maintain; we ideally want our dataset code to be decoupled from our model training code for better readability and modularity.
- PyTorch provides two data primitives: **torch.utils.data.DataLoader** and **torch.utils.data.Dataset** that allow you to use pre-loaded datasets as well as your own data.
- Dataset stores the samples and their corresponding labels, and DataLoader wraps an iterable around the Dataset to enable easy access to the samples.

Transformation

- Data does not always come in its final processed form that is required for training machine learning algorithms. We use transforms to perform some manipulation of the data and make it suitable for training.
- All TorchVision datasets have two parameters **-transform** to modify the features and **target_transform** to modify the labels - that accept callables containing the transformation logic.

Building the Neural Network

- Neural networks comprise of layers/modules that perform operations on data.
- The **torch.nn** namespace provides all the building blocks you need to build your own neural network.
- Every module in PyTorch subclasses the **nn.Module**.
- A neural network is a module itself that consists of other modules (layers). This nested structure allows for building and managing complex architectures easily.

Saving and Loading Models

- PyTorch models store the learned parameters in an internal state dictionary, called state_dict. These can be persisted via the torch.save method:

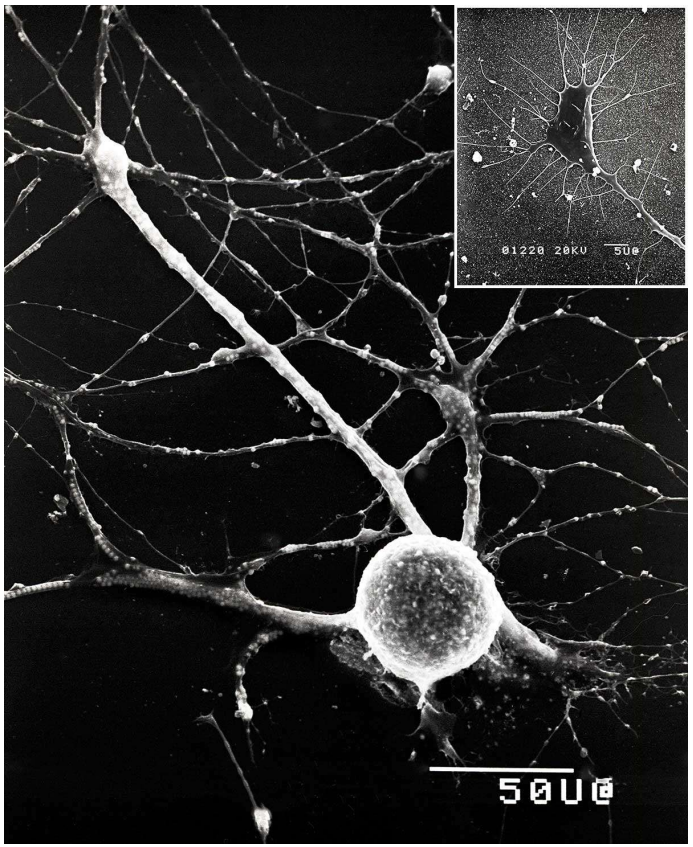
```
model = models.vgg16(pretrained=True)
torch.save(model.state_dict(), 'model_weights.pth')
```

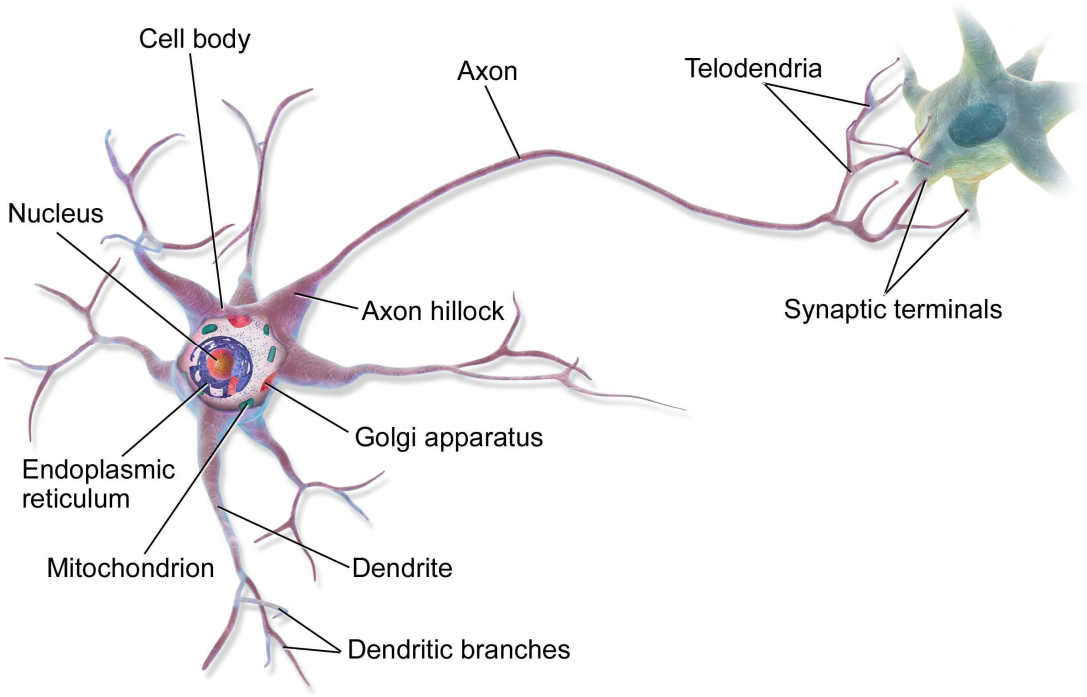
- To load model weights, you need to create an instance of the same model first, and then load the parameters using load_state_dict() method.

```
model = models.vgg16()
model.load_state_dict(torch.load('model_weights.pth’))
model.eval()
```

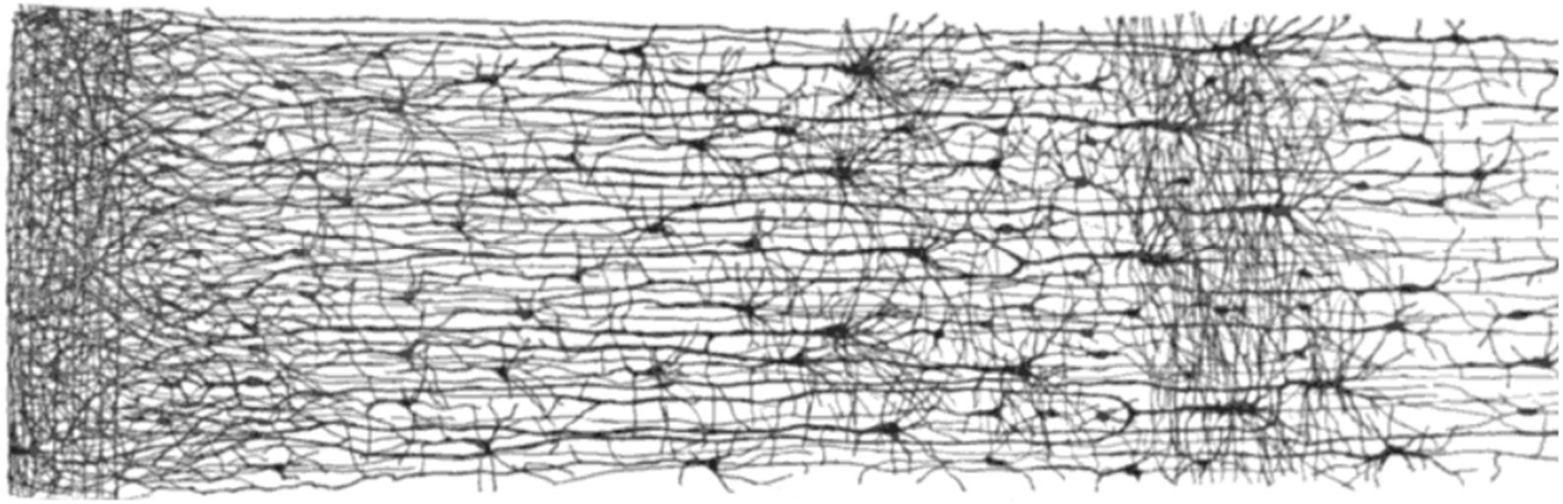
Neural Networks

Biological Neuron





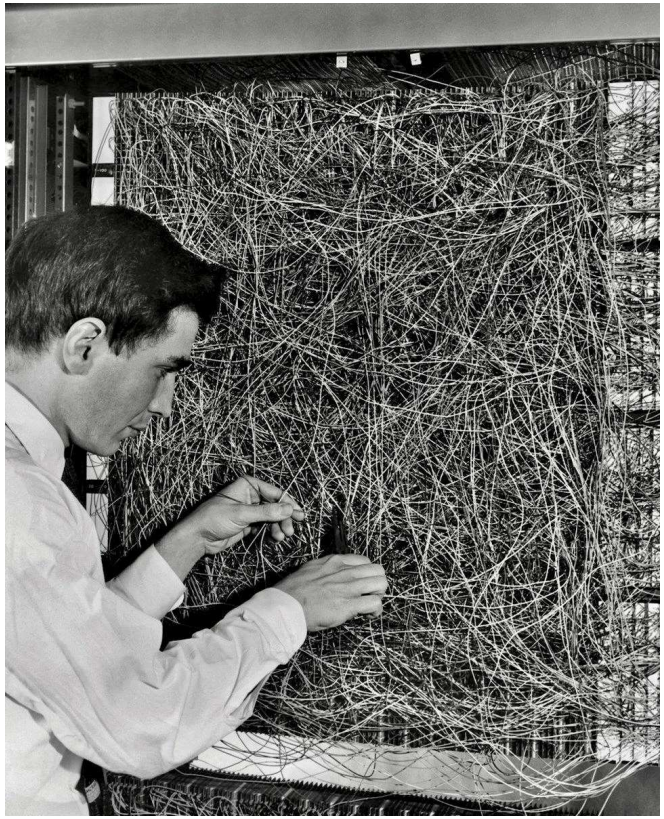
Multiple Layers of Neural Network



Multiple Layers in a biological neural network
in the human cortex

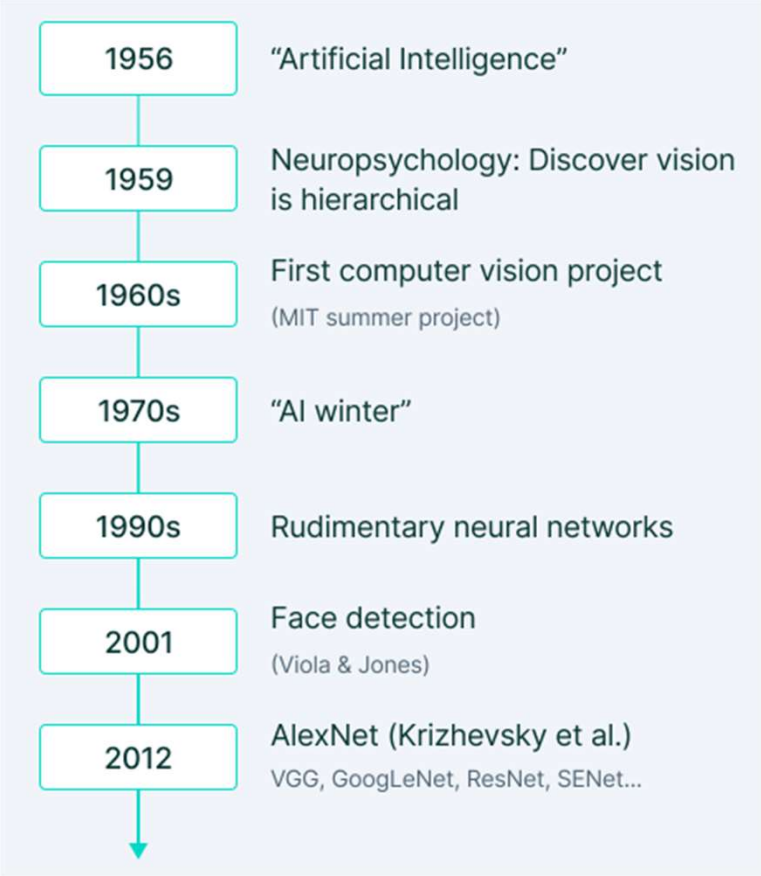
Perceptron

- A perceptron was a form of neuron/neural network introduced in 1958 by Frank Rosenblatt
- Rosenblatt's book **Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms**, summarized his work on perceptrons at the time
- Amazingly, even back then, he had forseen a huge potential:
  - "… perceptron may eventually be able to learn, make decisions and translate languages."

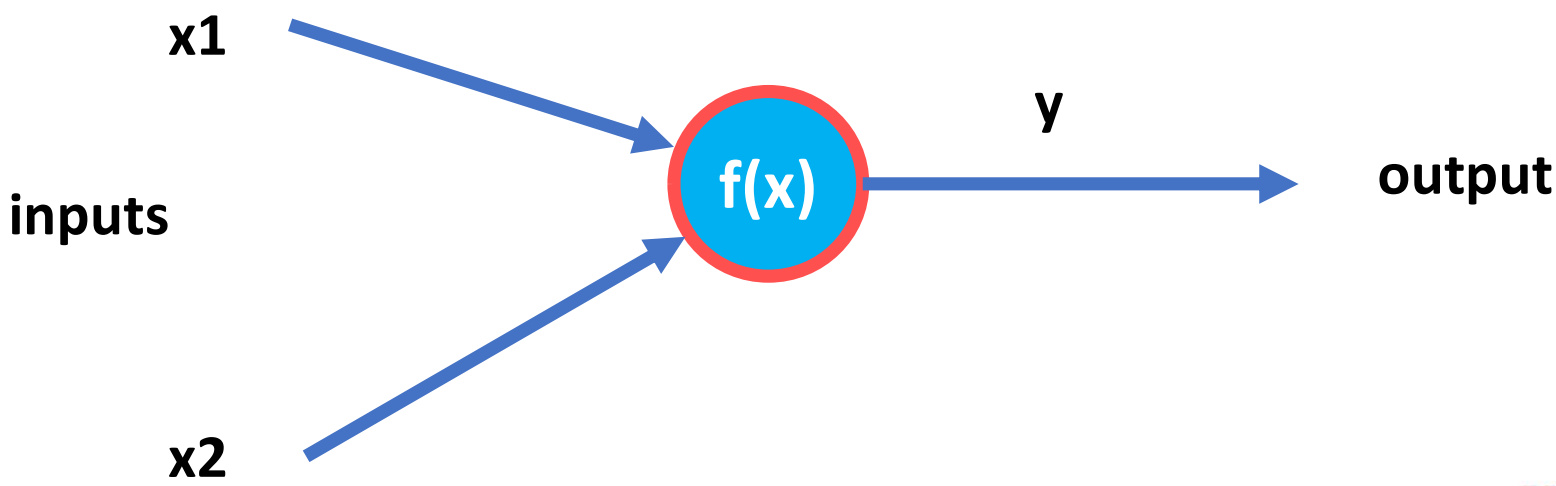


Perceptron

- In 1969, Marvin Minsky and Seymour Papert published their book Perceptrons
- It suggested that there were severe limitations to what perceptrons could do
- This marked the beginning of what is known as AI Winter, with little funding into AI and Neural Networks in the 1970s
- Fortunately for us, we now know the amazing power of neural networks, which all stem from the simple perceptron model.
- Let's now see how a simple biological neuron can be modelled into perceptron



Perceptron: Building a Model

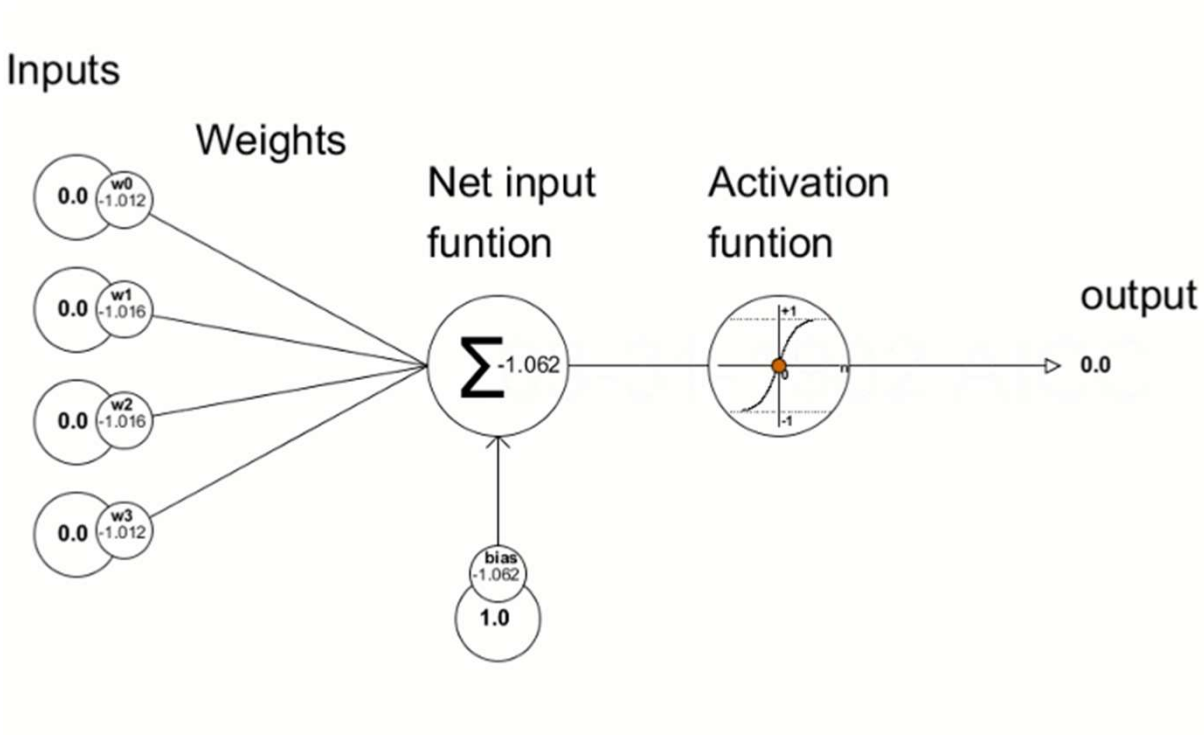


$$f(X) = \beta_0 + \sum_{k=1}^{K} \beta_k A_k$$

Bias

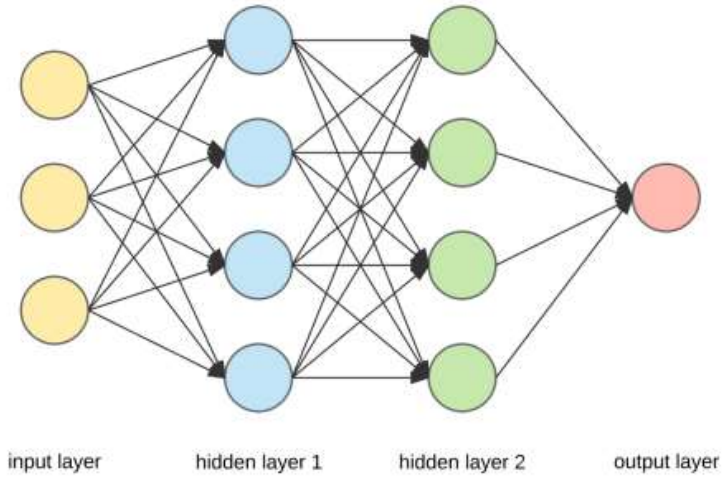
- The “bias” is a threshold that the neuron (perceptron) has placed on the inputs
- The input x*w should cross the bias to be effectively used by the neuron
- Suppose b = -10, x*w should cross 10 to be useful

Working


Inputs
Weights
Net input funtion
Activation funtion
output
0.0
w0 -1.012
0.0
w1 -1.016
0.0
w2 -1.016
0.0
w3 -1.012
Σ -1.062
bias -1.062
1.0
+1
-1
0.0

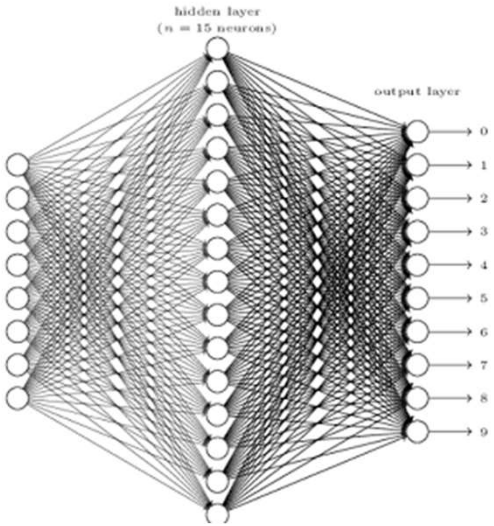
Neural Networks

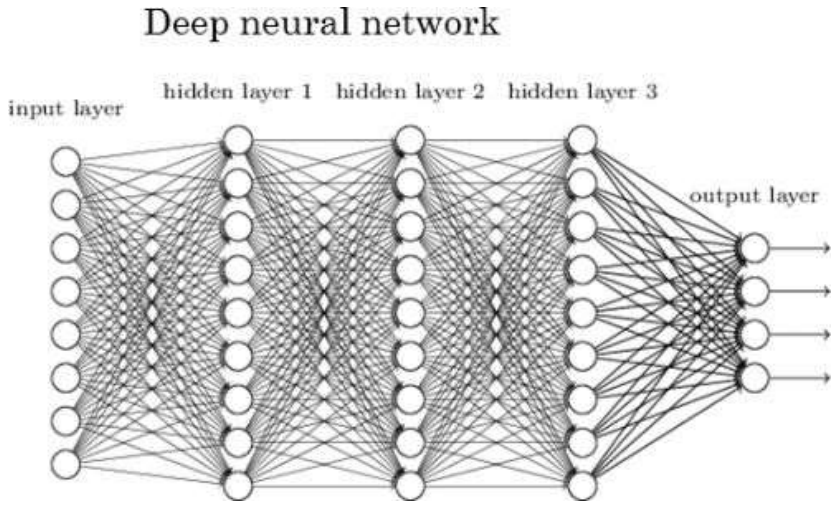
- A single perceptron will not be useful for learning anything significant. Therefore we build a network of perceptrons interconnected in layers.
- It is called as the **multi-layer perceptron model**



The hidden layers become more and more like black boxes

Deep Neural Networks





Neural Networks

- What is incredible about the neural network framework is that it can be used to approximate any continuous function
- Zhou Lu and later on Baris Hanin proved mathematically that Neural Networks can approximate any convex continuous function
- Reference: https://en.wikipedia.org/wiki/Universal_approximation_theorem

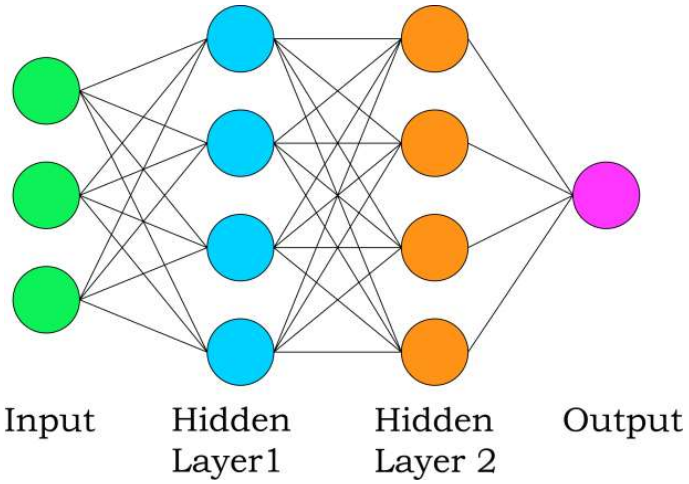
Neural Networks

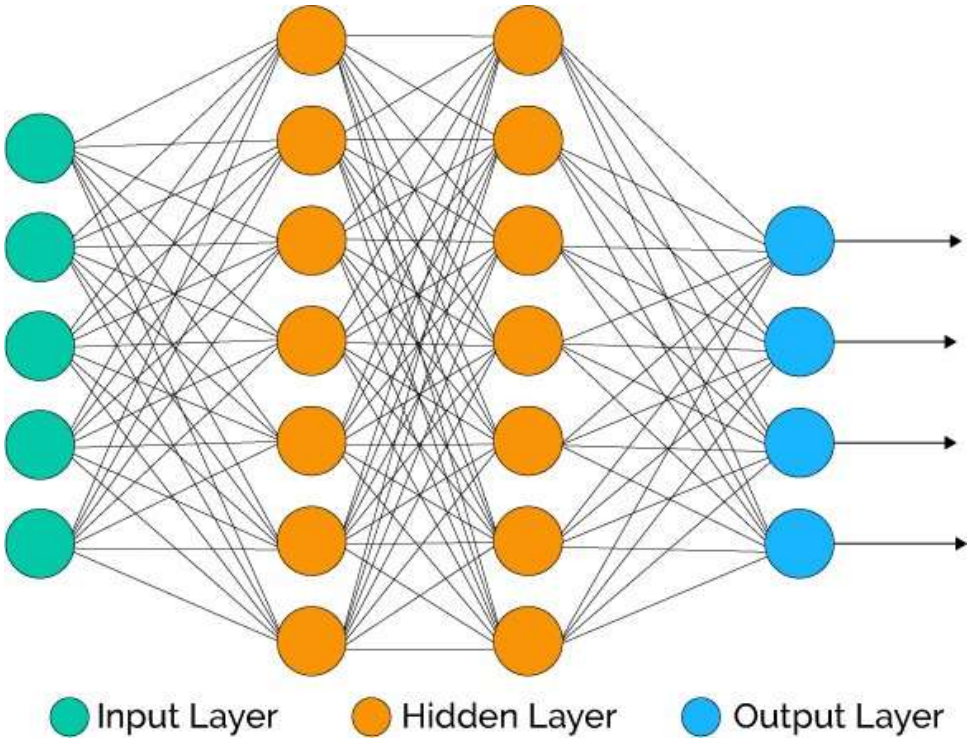
- Previously in out simple model we saw that the perceptron itself contained a very simple summation function f(x)
- For most cases however that won't be useful, we might want to set constraints to the outputs, especially classification tasks
- In classification tasks, it would be useful to have all the outputs fall between 0 and 1
- These values can then present probability assignments for each class
- We will have to use **activation functions** to set boundaries to output values from the neuron

Multi-class Classification

- Notice all activation functions make sense for a single output, either a continuous label or trying to predict a binary classification.
- But, what should be do if we have a multi-class situation?
- There are two main types of multi-class situation:
  - Non-exclusive classes: a data point can belong to multiple classes
    - Example: photo can have multiple tags such as family, beach, vacation, etc
  - Mutually exclusive classes: a data point belong to only one class
    - Example: photo can belong to grey scale or color, but not both at the same time
- One easy way to organize multiple classes is to simply have one output node per class

Multi-class Classification





Multi-class Classification

Multi-class Classification

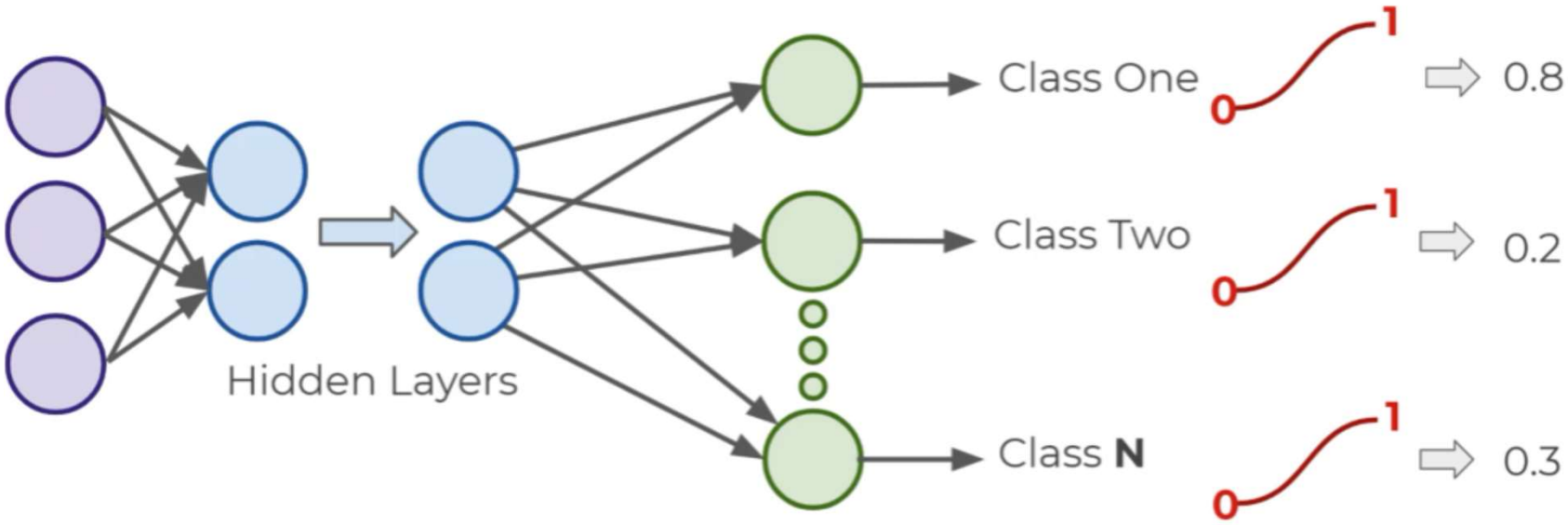
- Non-exclusive classes: data can belon to multiple classes

Data Point 1	A,B
Data Point 2	A
Data Point 3	C,B
...	...
Data Point N	B

	A	B	C
Data Point 1	1	1	0
Data Point 2	1	0	0
Data Point 3	0	1	1
...	...	...	...
Data Point N	0	1	0

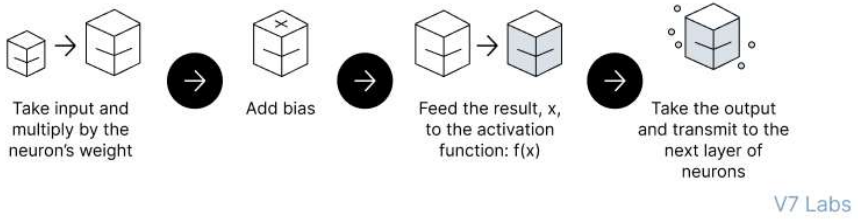
Multi-class Classification

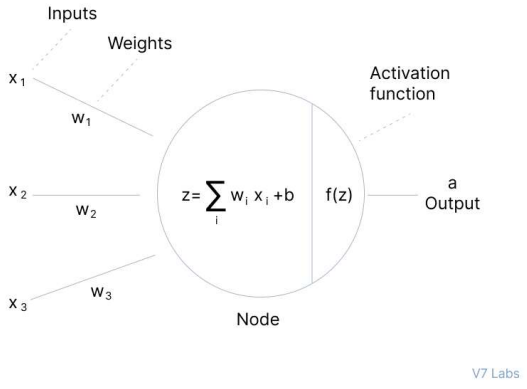
- Sigmoid Function can be used at the output for non-exclusive classes the output of which is a probability measure



Activation Function

- The activation function of a node defines the output of that node given an input or set of inputs.
- The primary role of the Activation Function is to transform the summed weighted input from the node into an output value to be fed to the next hidden layer or as output.

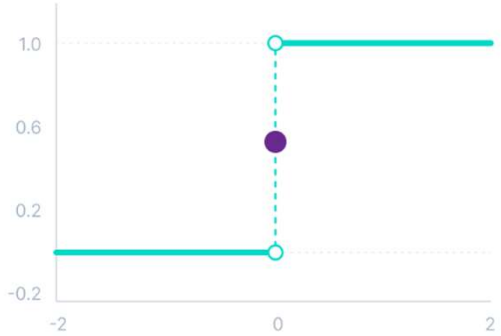




Binary Step Function

- Binary step function depends on a threshold value that decides whether a neuron should be activated or not.
- Here are some of the limitations of binary step function:
  - It cannot provide multi-value outputs—for example, it cannot be used for multi-class classification problems.
  - The gradient of the step function is zero, which causes a hindrance in the backpropagation process.

$$f(x) = \begin{cases} 0 & for\ x < 0 \\ 1 & for\ x \geqslant 0 \end{cases}$$

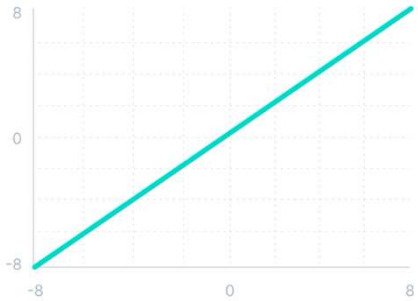


Linear Activation Function

- The linear activation function is also known as Identity Function where the activation is proportional to the input.
- However, a linear activation function has two major problems :
  - It’s not possible to use backpropagation as the derivative of the function is a constant and has no relation to the input x.
  - All layers of the neural network will collapse into one if a linear activation function is used.
  - No matter the number of layers in the neural network, the last layer will still be a linear function of the first layer. So, essentially, a linear activation function turns the neural network into just one layer.

Linear Activation Function

$$f(x) = x$$



Non-Linear Activation Functions

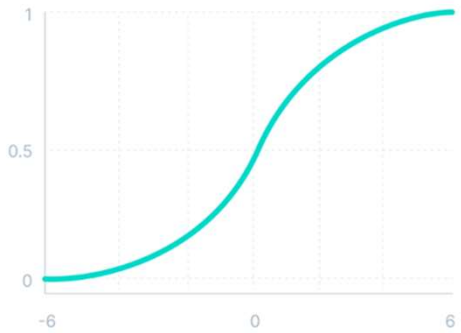
- Non-linear activation functions solve the following limitations of linear activation functions:
  - They allow backpropagation because now the derivative function would be related to the input, and it's possible to go back and understand which weights in the input neurons can provide a better prediction.
  - They allow the stacking of multiple layers of neurons as the output would now be a non-linear combination of input passed through multiple layers. Any output can be represented as a functional computation in a neural network.

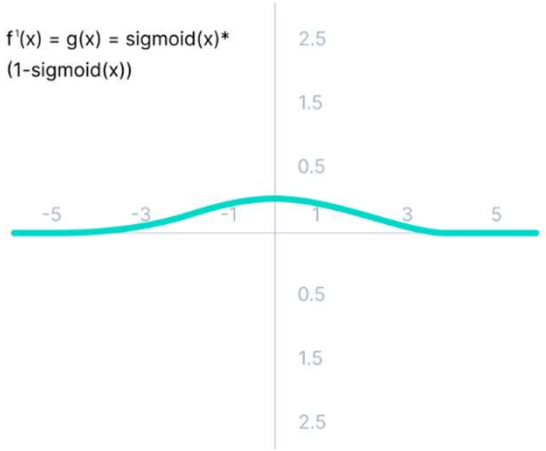
Sigmoid Activation Function

- This function takes any real value as input and outputs values in the range of 0 to 1.
- Here's why sigmoid/logistic activation function is one of the most widely used functions:
  - It is commonly used for models where we have to predict the probability as an output. Since probability of anything exists only between the range of 0 and 1, sigmoid is the right choice because of its range.
  - The function is differentiable and provides a smooth gradient, i.e., preventing jumps in output values. This is represented by an S-shape of the sigmoid activation function.

Sigmoid Activation Function

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





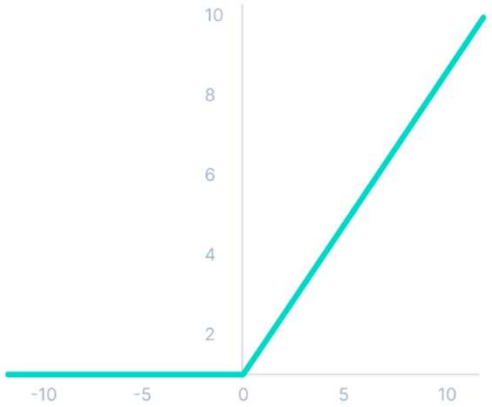
It implies that for values greater than 3 or less than -3, the function will have very small gradients. As the gradient value approaches zero, the network ceases to learn and suffers from the Vanishing gradient problem.

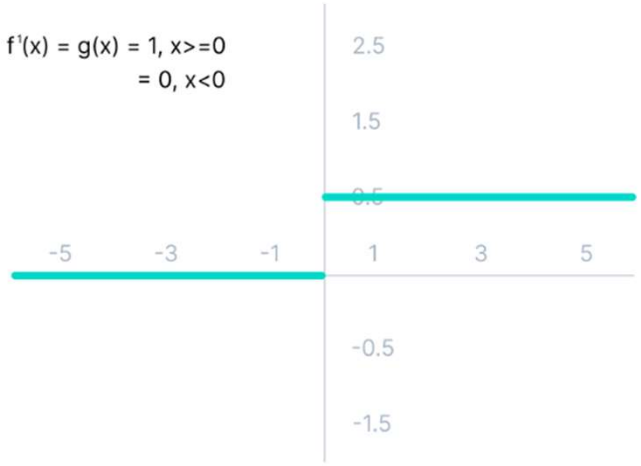
ReLU

- ReLU stands for Rectified Linear Unit.
- The main catch here is that the ReLU function does not activate all the neurons at the same time. The neurons will only be deactivated if the output of the linear transformation is less than 0.
- The advantages of using ReLU as an activation function are as follows:
  - Since only a certain number of neurons are activated, the ReLU function is far more computationally efficient when compared to the sigmoid and tanh functions.
  - ReLU accelerates the convergence of gradient descent towards the global minimum of the loss function due to its linear, non-saturating property.

ReLU

$$f(x) = max\ (0, x)$$





ReLU has a derivative function and allows for backpropagation while simultaneously making it computationally efficient.

Also, The negative side of the graph makes the gradient value zero. Due to this reason, during the backpropagation process, the weights and biases for some neurons are not updated. This can create dead neurons which never get activated.

Softmax Function

- The Softmax function is described as a combination of multiple sigmoids.
- In the sigmoid function: Let's suppose we have five output values of 0.8, 0.9, 0.7, 0.8, and 0.6, respectively. How can we move forward with it? The answer is: We can't.
- The above values don't make sense as the sum of all the classes/output probabilities should be equal to 1.
- It calculates the relative probabilities. Similar to the sigmoid/logistic activation function, the SoftMax function returns the probability of each class.
- It is most commonly used as an activation function for the last layer of the neural network in the case of multi-class classification.

Softmax Function

$$\text{softmax}(z_i) = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

Assume that you have three classes, meaning that there would be three neurons in the output layer. Now, suppose that your output from the neurons is [1.8, 0.9, 0.68].

Applying the softmax function over these values to give a probabilistic view will result in the following outcome: [0.58, 0.23, 0.19].

The function returns 1 for the largest probability index while it returns 0 for the other two array indexes. Here, giving full weight to index 0 and no weight to index 1 and index 2. So the output would be the class corresponding to the 1st neuron(index 0) out of three.

You can see now how softmax activation function make things easy for multi-class classification problems.

Choosing the Right Activation Function

- You need to match your activation function for your output layer based on the type of prediction problem that you are solving—specifically, the type of predicted variable.
- As a rule of thumb, you can begin with using the ReLU activation function and then move over to other activation functions if ReLU doesn't provide optimum results.
- And here are a few other guidelines to help you out.
  - ReLU activation function should only be used in the hidden layers.
  - Sigmoid/Logistic and Tanh functions should not be used in hidden layers as they make the model more susceptible to problems during training (due to vanishing gradients).
  - Swish function is used in neural networks having a depth greater than 40 layers.

Choosing the Right Activation Function

- Regression - Linear Activation Function
- Binary Classification - Sigmoid/Logistic Activation Function
- Multiclass Classification - Softmax
- Multilabel Classification – Sigmoid
- Convolutional Neural Network (CNN): ReLU activation function.
- Recurrent Neural Network: Tanh and/or Sigmoid activation function

Multi-class Classification and Softmax

- Mutually exclusive classes: what to do when each data point can only have a single class assigned to it?
- We can use a very clever **softmax** function
  - It calculates probabilities distribution of the event over k different events
  - It also calculates probabilities of each target class over all possible target classes
  - The range will be 0 – 1. The sum of all the probabilities will be equal to one,
  - The target class chosen will have the highest probability
  - You get this sort of output:

[Red , Green , Blue]
[0.1 , 0.6 , 0.3]

Evaluating a Model: Cost Function

- After the network creates its predictions we have two questions:
  - How do we evaluate it?
  - How to update the network's weights and biases
- We need to take estimated outputs of the network and then compare them to the real values of the label
- The cost function also known as the loss function must be a kind of average so it can output a single value
- We can keep track of out loss/cost during training to monitor network performance

Cost Function

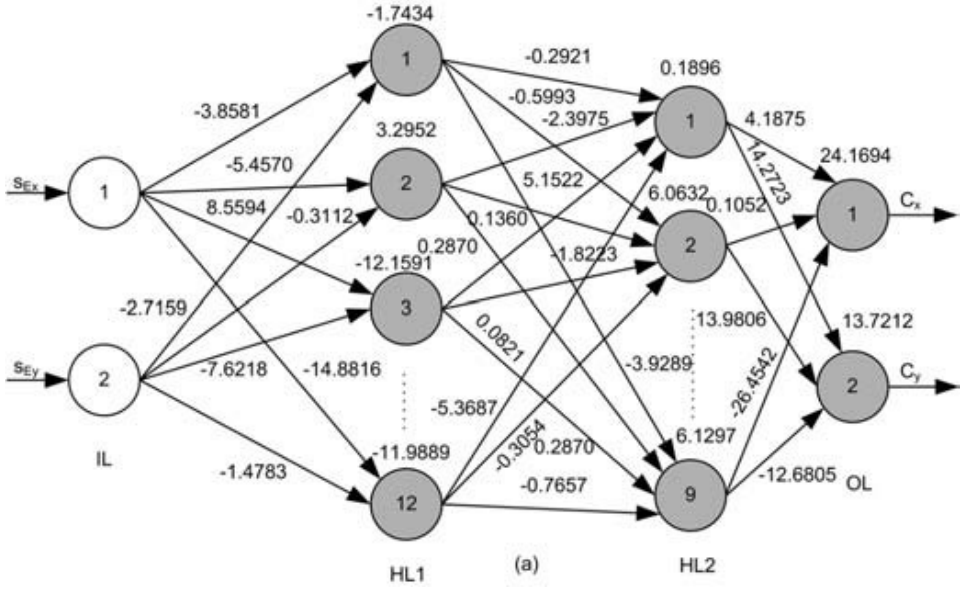
- In general a cost function is a function of weights, biases, input of a training sample and desired output of the training sample:
- One very common cost function is the quadratic cost function
- We simply calculate the difference between the real values y(x) against our predicted values a(x):

$$C = \frac{1}{2n} \sum_{x} \|y(x) - a^{L}(x)\|^{2}$$

L -> layer, a activation function which contains information about weights and biases

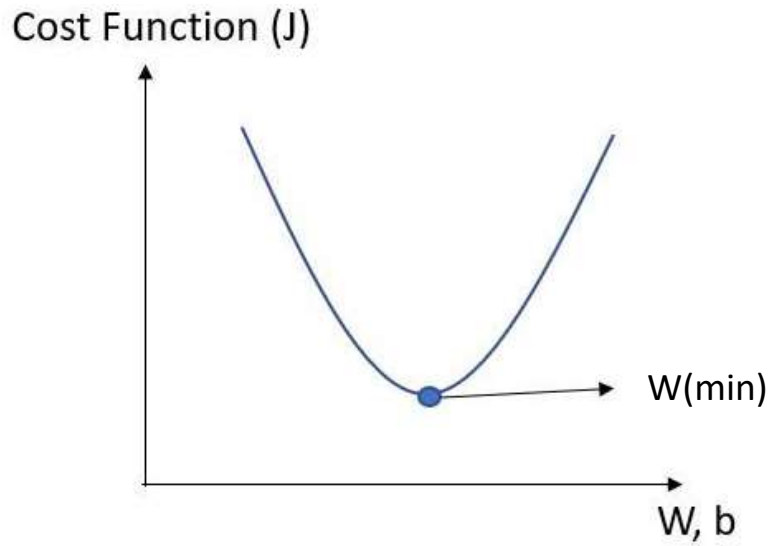
Cost Function

- Here's a small network with all it's parameters labelled
- How do we calculate the cost function and minimize it?
  - Which particular weights minimize the cost function



Cost Function

- For simplicity, let's imagine we only has one weight in our cost functions w, we want to minimize our loss
- That would mean, we need to figure out what value of w results in the minimum c(w)
- In this case, we have a plot like this:
- Real cost function is real complex
  - Use a stochastic process
  - Use Gradient descent

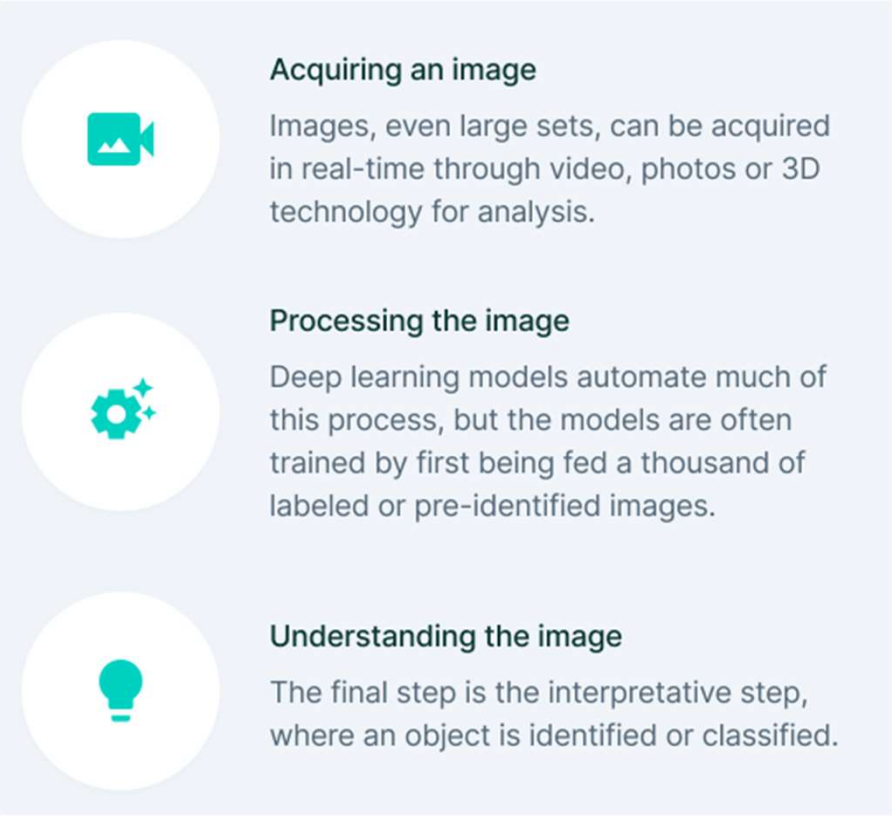


Convolutional Neural Networks

Image Processing

- Digital Image Processing, or Image Processing, in short, is a subset of Computer Vision. It deals with enhancing and understanding images through various algorithms.
- More than just a subset, Image Processing forms the precursor of modern-day computer vision, overseeing the development of numerous rule-based and optimization-based algorithms that have led machine vision to what it is today.
- Image Processing may be defined as the task of performing a set of operations on an image based on data collected by algorithms to analyze and manipulate the contents of an image or the image data.

Practical Side of Computer Vision



Convolution

- In the context of CNN's, these filters are referred to as **Convolution Kernels**
- The process of passing them over an image is known as **convolution**

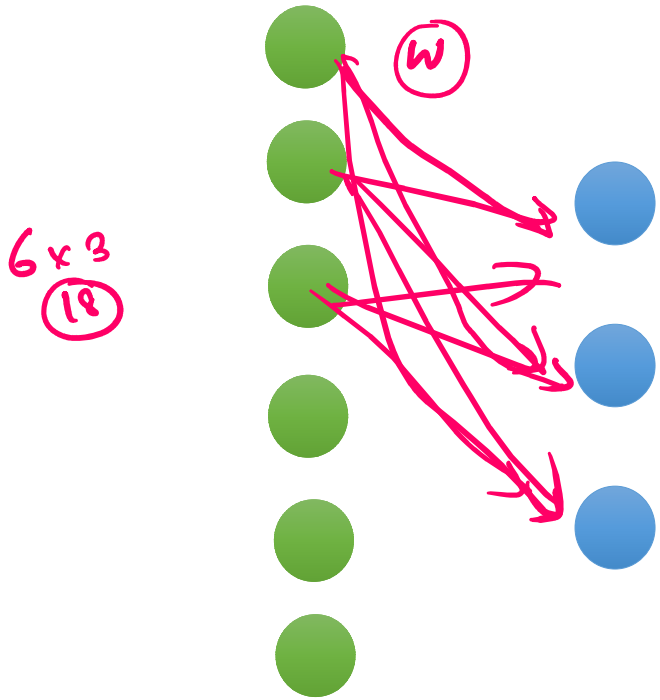
CNN

- Recall that running an ANN for the MNIST data set resulted in a network with relatively goof accuracy
- However, there are some issues with always using ANN models for the image data
- ANNs
  - Large amounts of paramters (over 100,000 for tiny 28x28 images)
  - We lose all the 2D information due to flattening
  - We only work on very similar well centered images

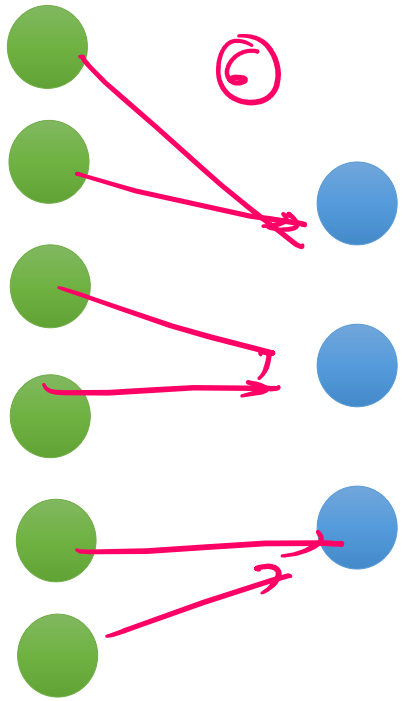
CNN

- CNN uses convolutional layers to help alleviate these issues
- A convolutional layer is created when we apply multiple image filters to the input images
- The layer will then be trained to figure out the best filter weight values
- The CNN also helps reduce paramters by focussing on local connectivity
- Not all neurons will be fully connected
- Instead, neurons are only connected to a subset of local neuraon in the next later (these end up being the filters)

1D Analysis

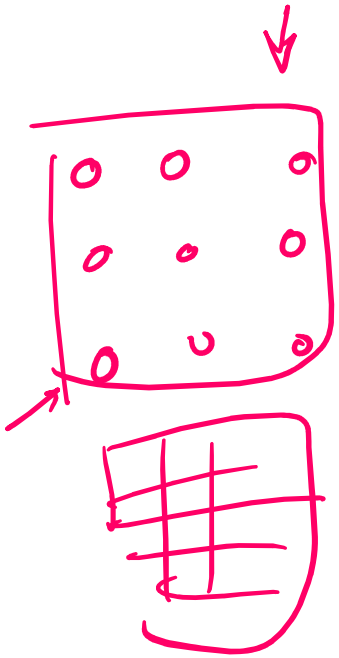


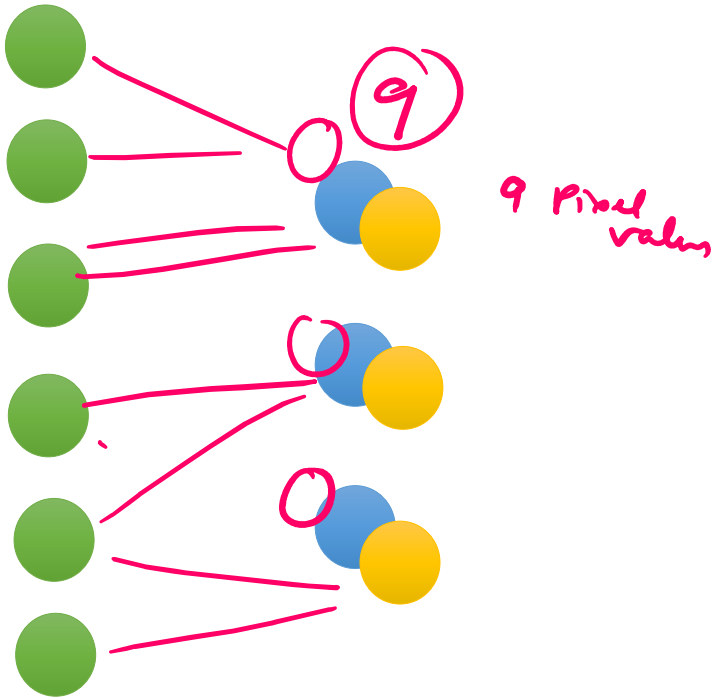
Fully Connected: Lot's of parameters



Locally Connected: Reduced parameters

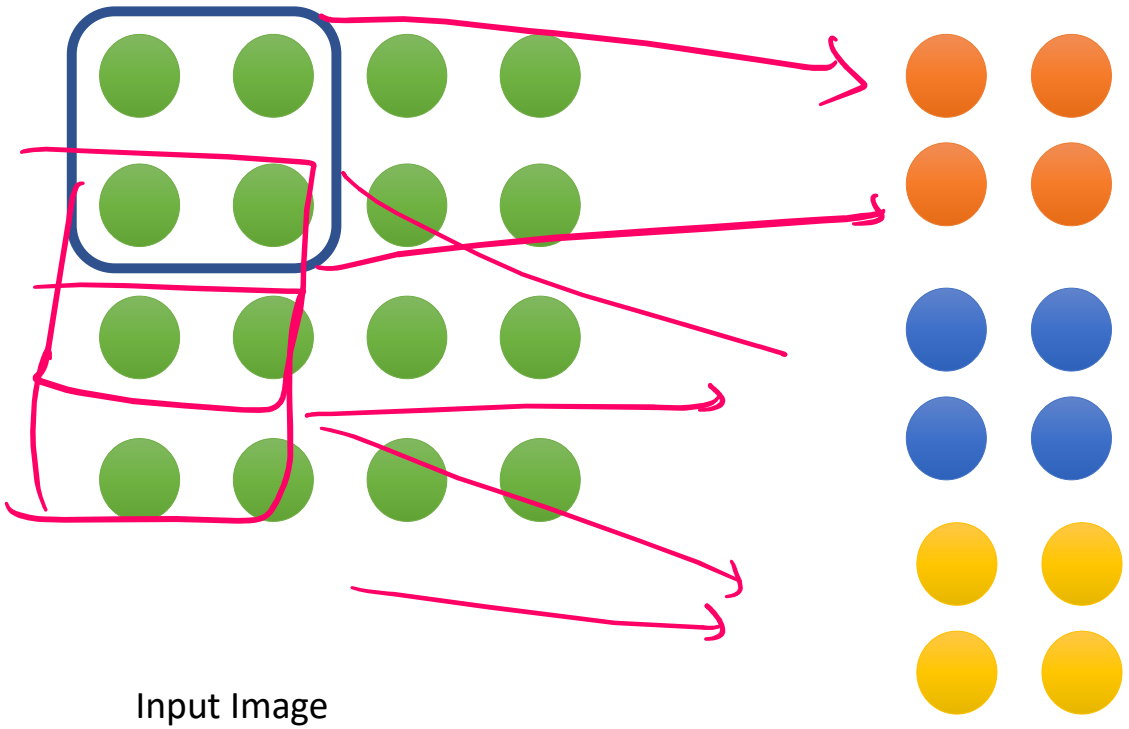
1D Analysis





Locally Connected: Reduced parameters

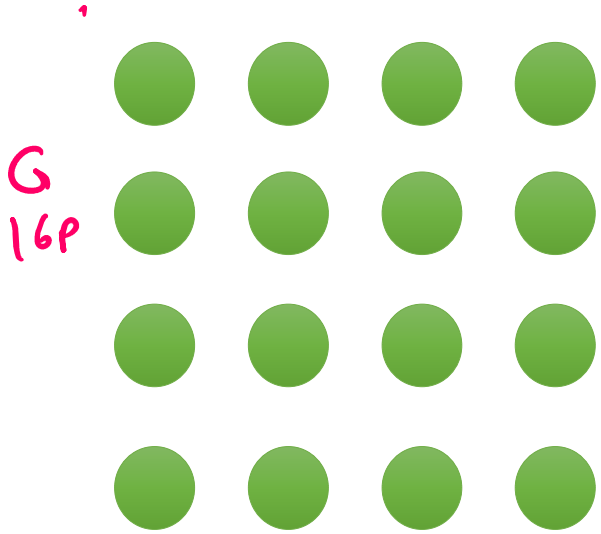
2D Analysis



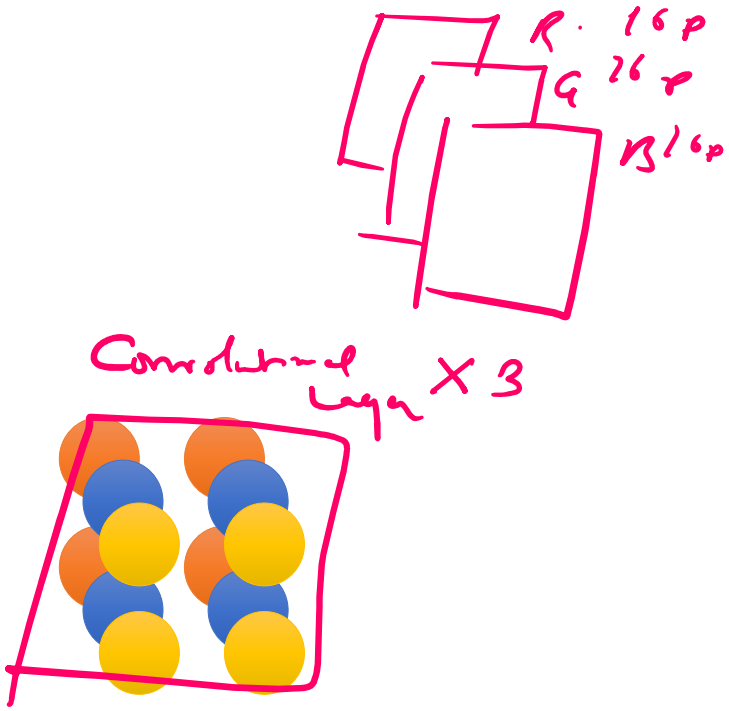
Input Image

Essentially a kernel

2D Analysis



Input Image



Convolutional Layer

3D Analysis

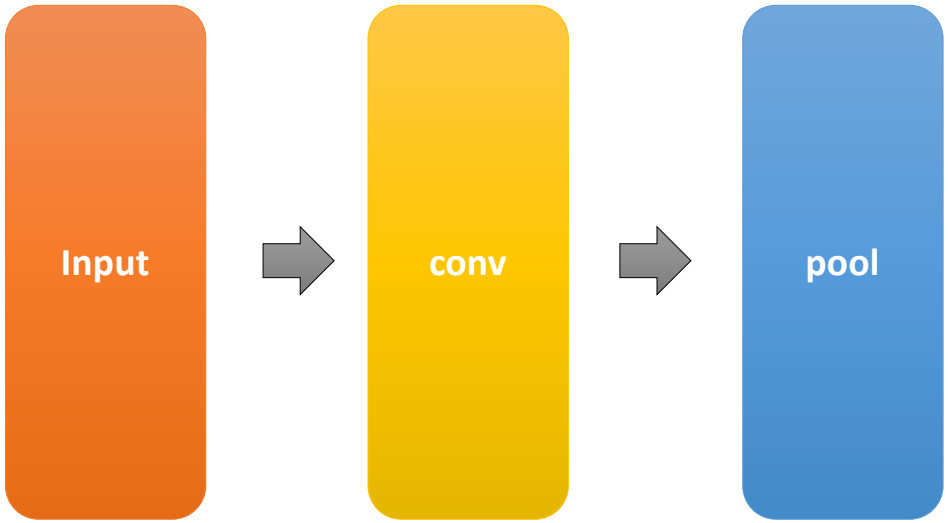
- But that was just for 2D gray scale images
- What about colour images?
- Colour images can be thought of as 3d Tensors consisting of Red, Green and Blue colour channels
- We have already see that additive colour mixing allows us to represent a wide variety of colours by simply combining different amount of R, G and B
- This means that a shape of the colour array has 3 dimensions: height, width and color channels
- So the filter we need to use should have 3 dimensions

Pooling Layers

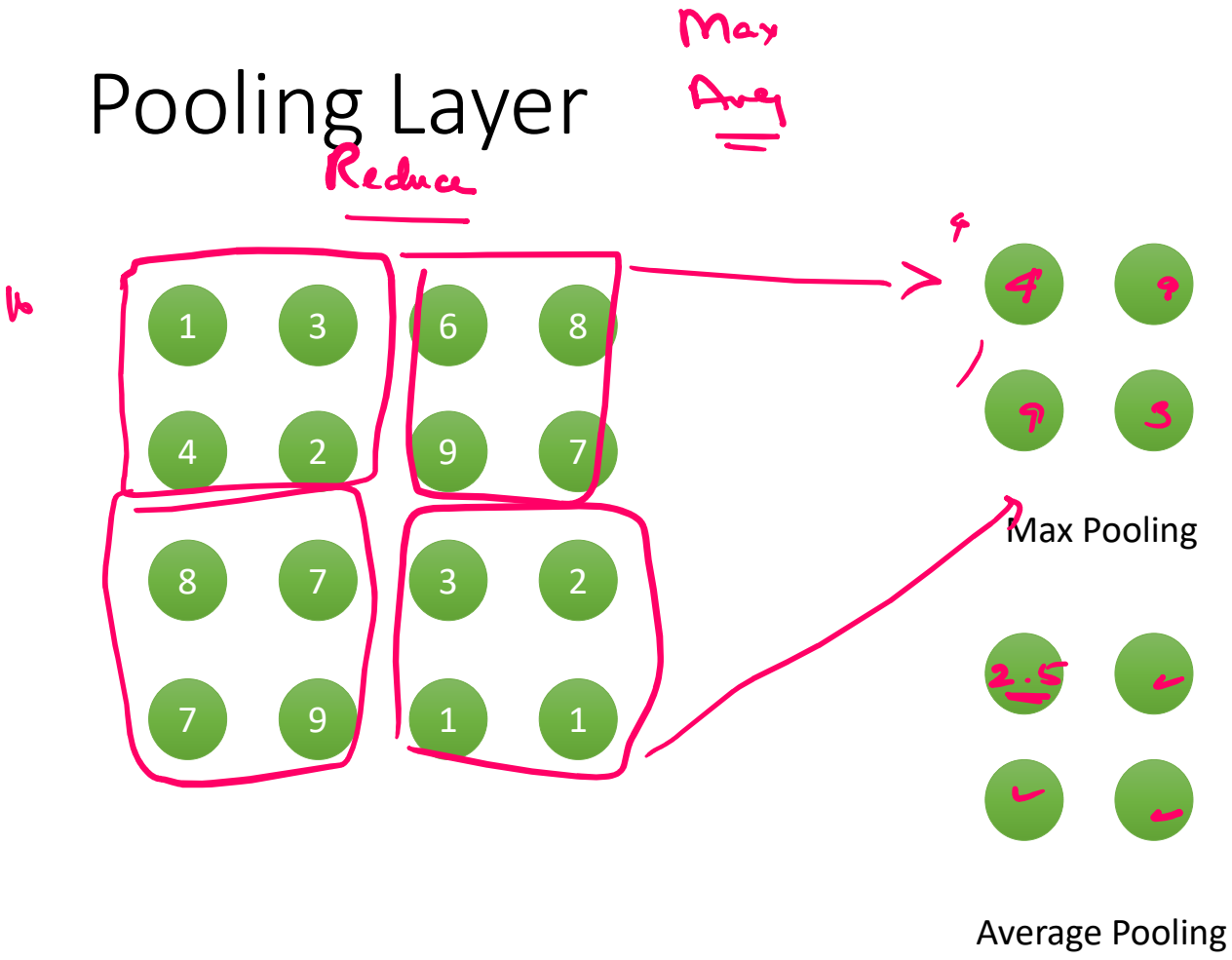
- Even with local connectivity, when dealing with color images and possibly 10s and 100s of filters we will have a large about of parameters
- We can use pooling layers to reduce this

Pooling Layers

- Pooling Layers accept convolutional layers as inputs



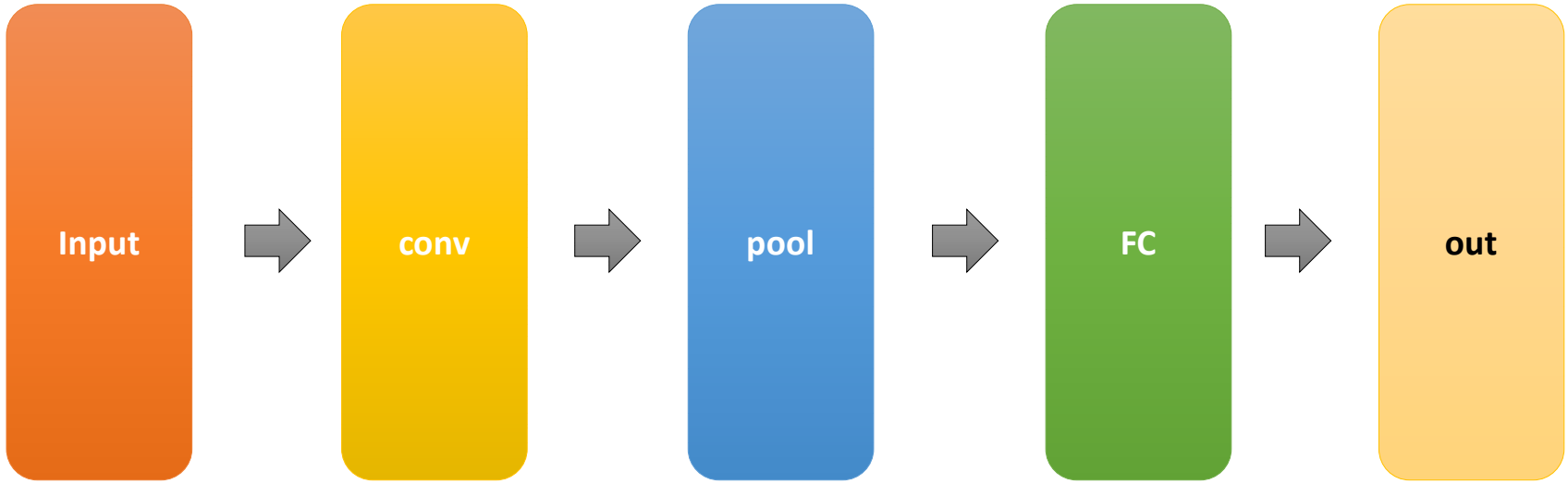
Pooling Layer



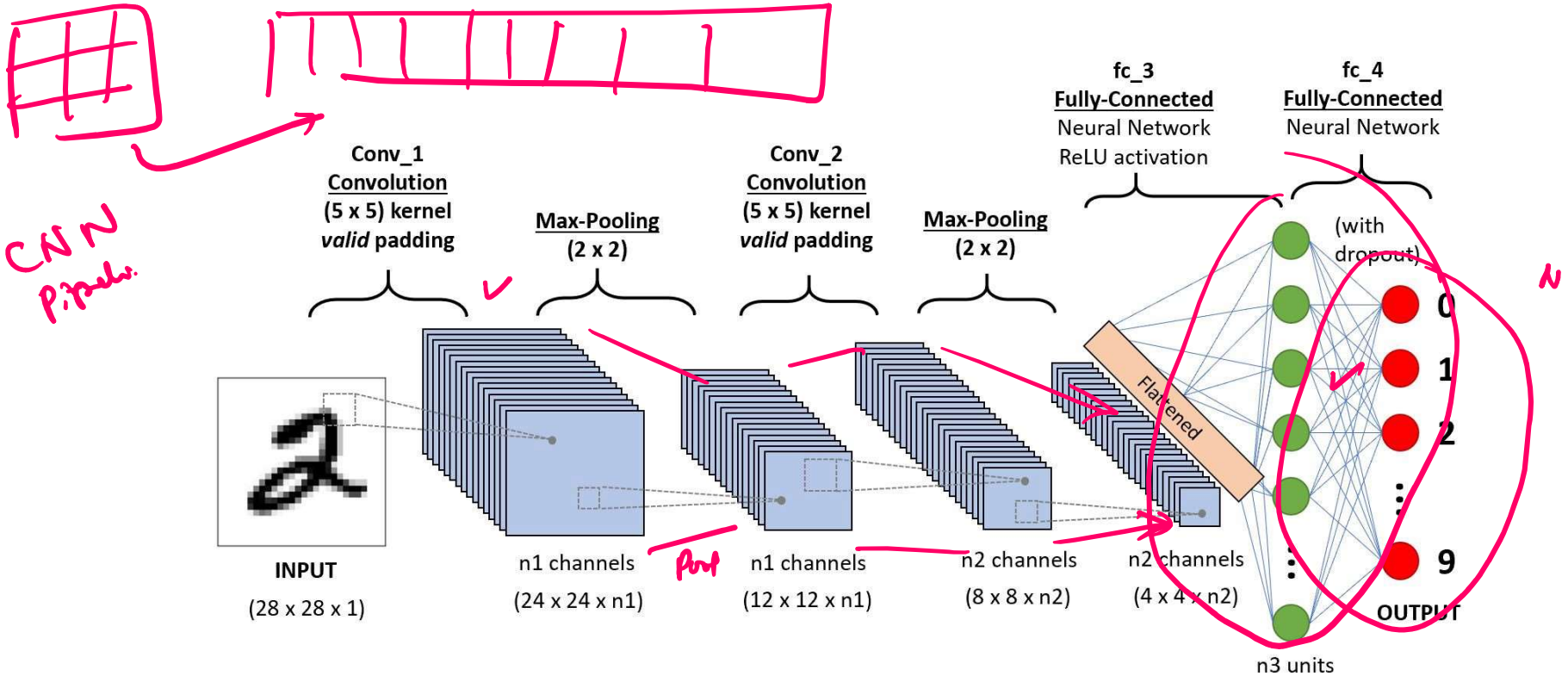
We loose quite a lot of information. But, the general trend remain the same

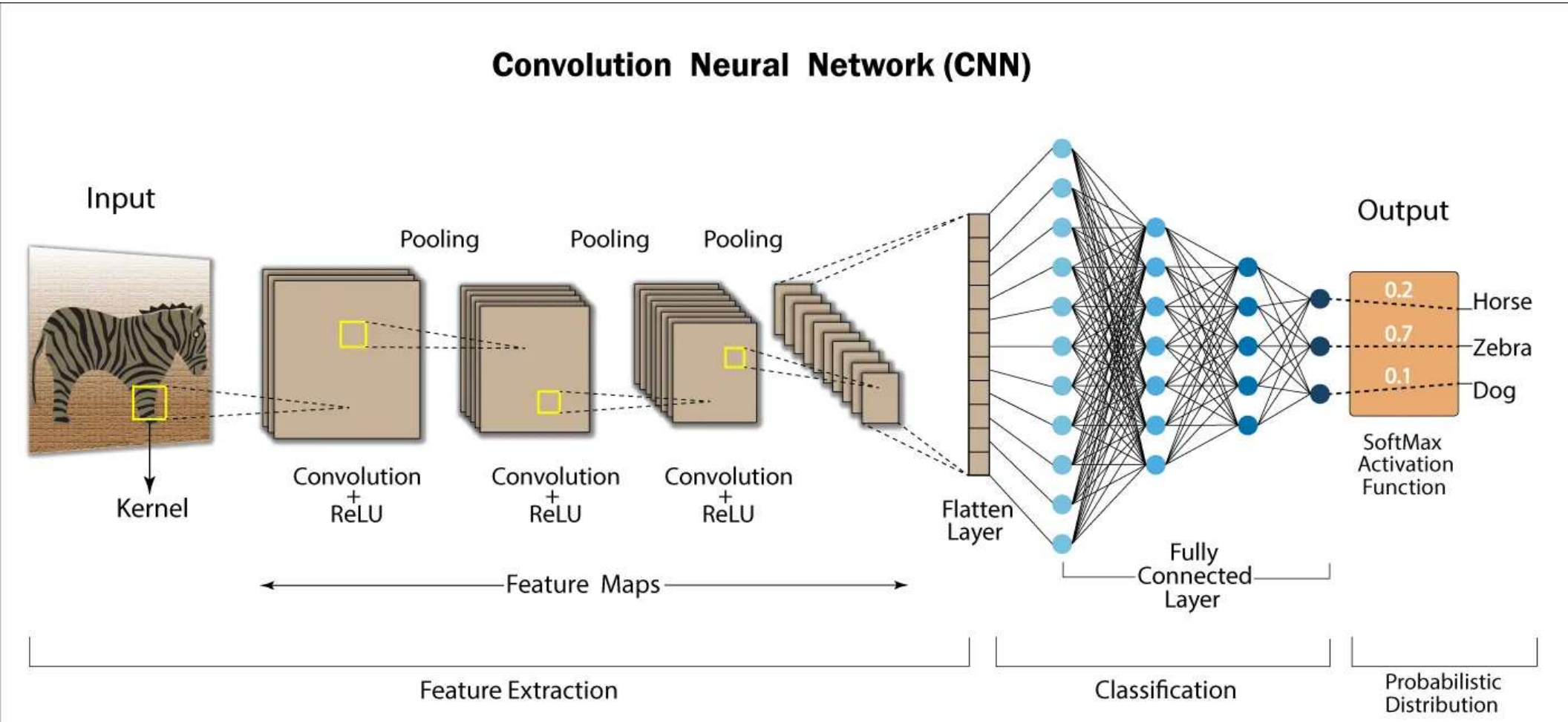
Convolutional Networks

- There is nearly no wrong way. Try to use any combination of convolutional layer, pooling layers
- But finally connect with a fully connected layer and also an output layer

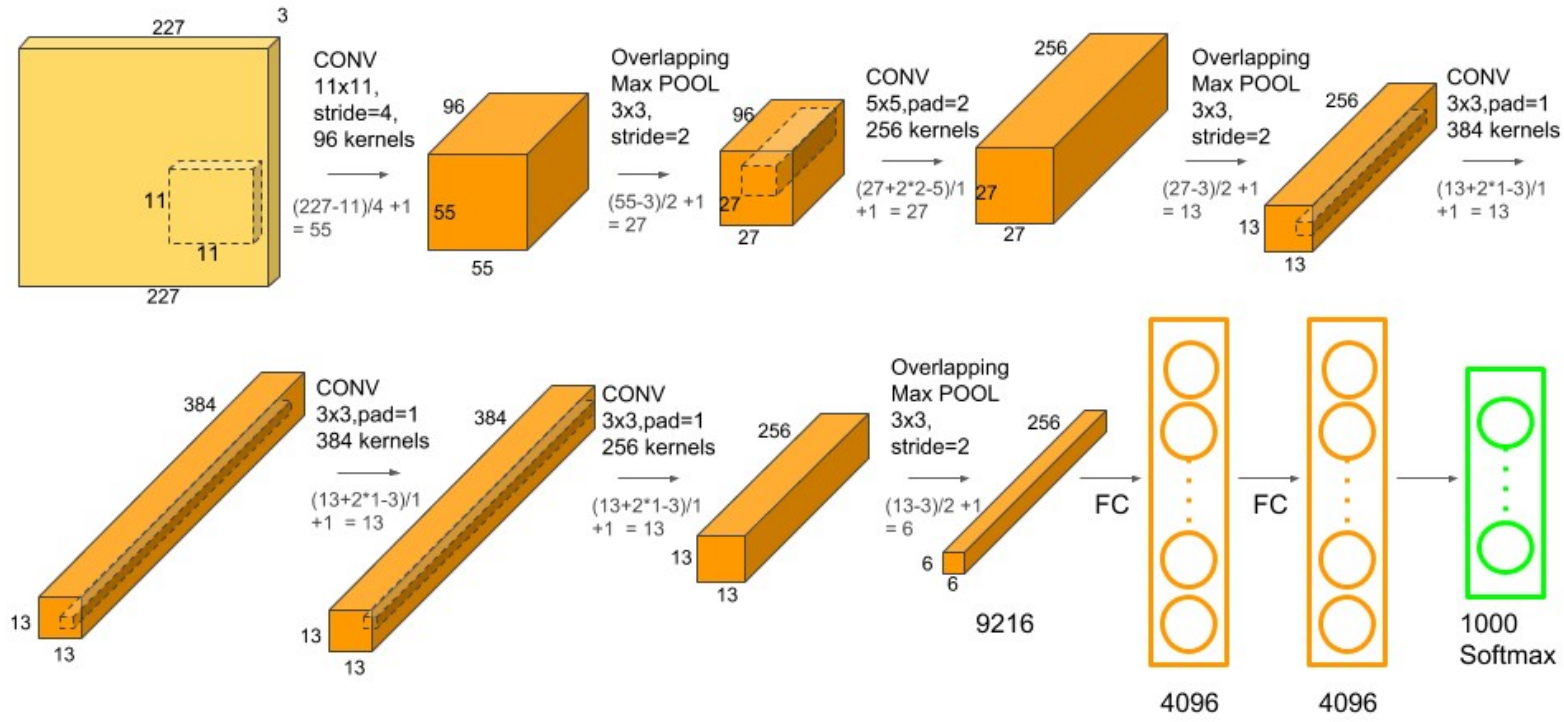


Typical CNN Architecture

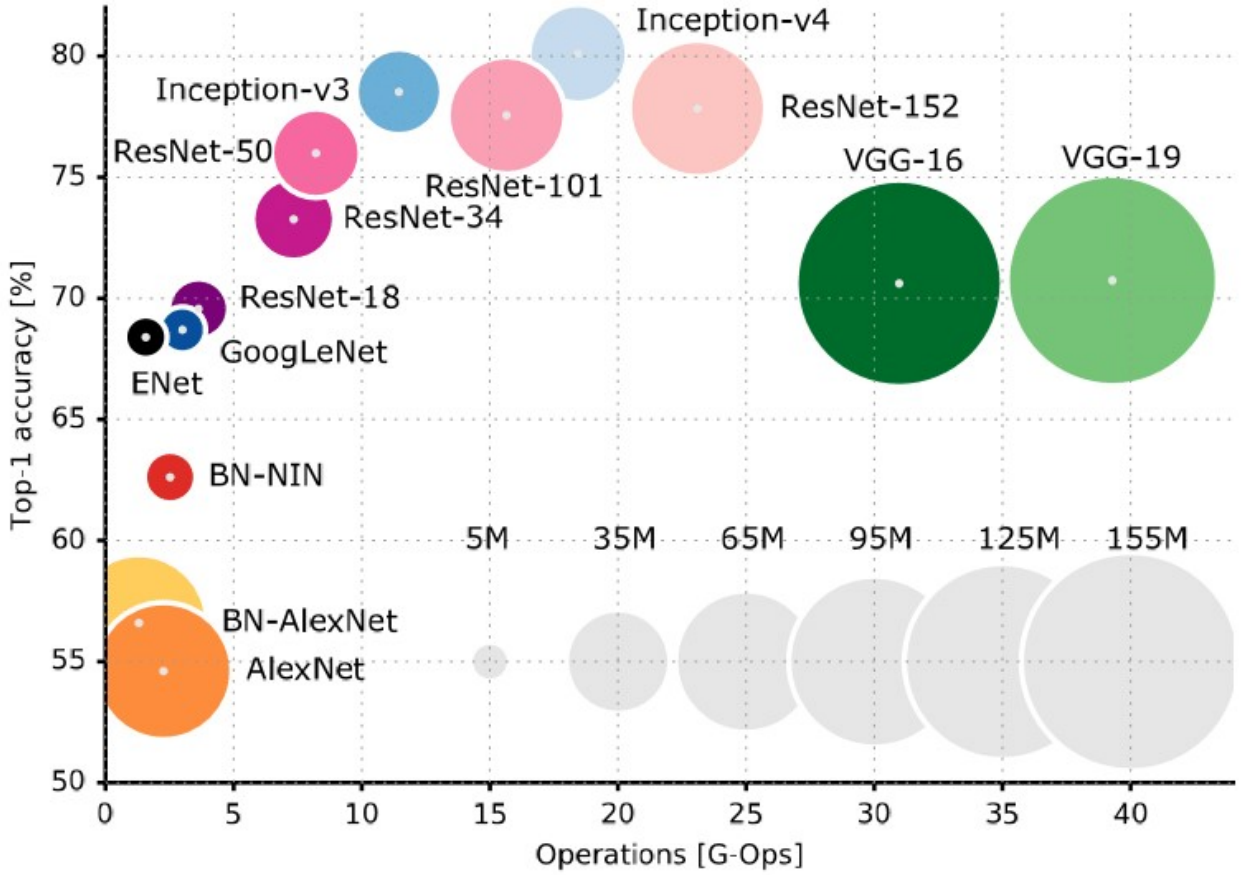



Convolution Neural Network (CNN)
Input
Pooling
Pooling
Pooling
Output
0.2
Horse
0.7
Zebra
0.1
Dog
Kernel
Convolution + ReLU
Convolution + ReLU
Convolution + ReLU
Flatten Layer
SoftMax Activation Function
Fully Connected Layer
Feature Maps
Feature Extraction
Classification
Probabilistic Distribution

Popular CNN Architectures: AlexNet



Famous CNN Architectures



Other examples for NN

- **Seq2seq** was first introduced for machine translation, by Google. Before that, the translation worked in a very naïve way. Each word that you used to type was converted to its target language giving no regard to its grammar and sentence structure.
- As the name suggests, seq2seq takes as input a sequence of words(sentence or sentences) and generates an output sequence of words. It does so by use of the recurrent neural network (RNN). Although the vanilla version of RNN is rarely used, its more advanced version i.e. LSTM or GRU is used.

Other examples

- Health Risk Analysis using Predictive Analytics
  - HRA is a predictive application that computes the likelihood of specific events. Based on patient data, this use case includes disease progression or complications. It looks for similar PHR, analyses the patient's data, looks for patterns, and calculates potential outcomes. This system can be used for routine health checks.

Other Examples

- Drug Discovery using Predictive Analytics
  - CNN optimizes and streamlines the drug discovery process at critical stages. It allows for a reduction in the time required to develop cures for emerging diseases.
- Precision Medicine