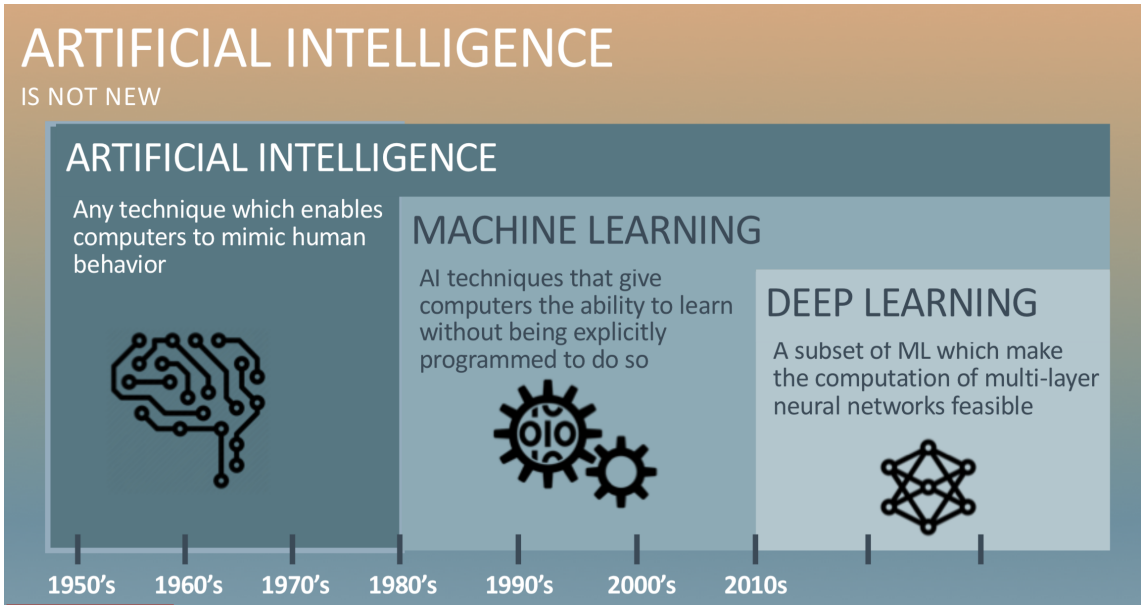


Deep Learning Introduction

- Deep learning is a branch of machine learning which is completely based on artificial neural networks, as neural network is going to mimic the human brain so deep learning is also a kind of mimic of human brain.
- it is an artificial intelligence (AI) function that imitates the workings of the human brain in processing data and creating patterns for use in decision making.

KEY TAKEAWAYS

- Deep learning is an AI function that mimics the workings of the human brain in processing data for use in detecting objects, recognizing speech, translating languages, and making decisions.
- Deep learning AI is able to learn without human supervision, drawing from data that is both unstructured and unlabeled.
- Deep learning, a form of machine learning, can be used to help detect fraud or money laundering, among other functions.

Key Difference between Machine Learning and Deep Learning :

S.No.	Machine Learning	Deep Learning
1.	Machine Learning is a superset of Deep Learning	Deep Learning is a subset of Machine Learning
2.	The data represented in Machine Learning is quite different as compared to Deep Learning as it uses structured data	The data representation is used in Deep Learning is quite different as it uses neural networks(ANN).
3.	Machine Learning is an evolution of AI	Deep Learning is an evolution to Machine Learning. Basically it is how deep is the machine learning.
4.	Machine learning consists of thousands of data points.	Big Data: Millions of data points.
5.	Outputs: Numerical Value, like classification of score	Anything from numerical values to free-form elements, such as free text and sound.
6.	Uses various types of automated algorithms that turn to model functions and predict future action from data.	Uses neural network that passes data through processing layers to the interpret data features and relations.
7.	Algorithms are detected by data analysts to examine specific variables in data sets.	Algorithms are largely self-depicted on data analysis once they're put into production.
8.	Machine Learning is highly used to stay in the competition and learn new things.	Deep Learning solves complex machine learning issues.

Types of Deep Learning Algorithms That I Coverd In Notebook

1. Multilayer Perceptrons (MLPs)
2. Convolutional Neural Networks (CNNs)
3. Recurrent Neural Networks (RNNs)
4. Long Short Term Memory Networks (LSTMs)
5. Generative Adversarial Networks (GANs)
6. Restricted Boltzmann Machines(RBMs)
7. Autoencoders
8. Self Organizing Maps (SOMs)

There are so many techniques but in my note book i will focus on this 8 topic.

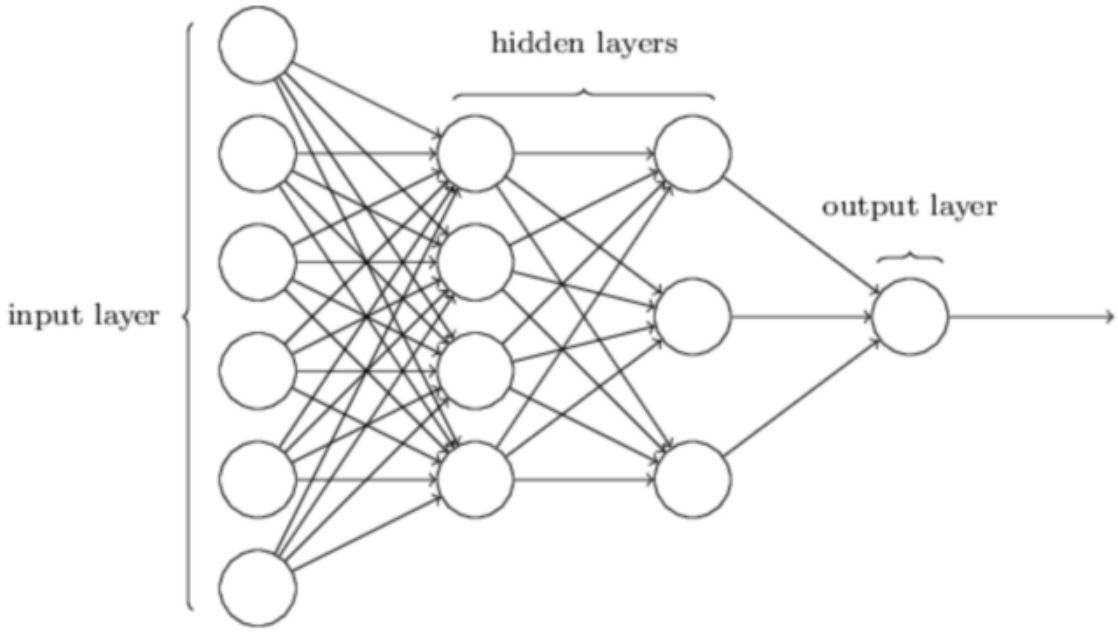
Neural Networks

- Before Deep Dive in to deep learning first see some important terminology regarding this

Q1. What are Neural networks?

- Neural networks are set of algorithms inspired by the functioning of human brian. Generally when you open your eyes, what you see is called data and is processed by the Nuerons(data processing cells) in your brain, and recognises what is around you. That's how similar the Neural Networks works. They takes a large set of data, process the data(draws out the patterns from data), and outputs what it is.

- A neural network is composed of layers, which is a collection of neurons, with connections between different layers. These layers transform data by first calculating the weighted sum of inputs and then normalizing it using the activation functions assigned to the neurons.

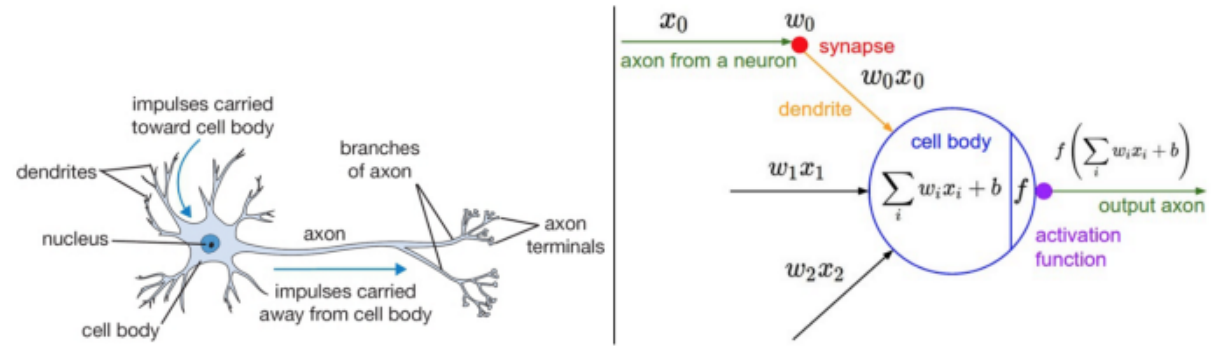


- The leftmost layer in a Neural Network is called the input layer, and the rightmost layer is called the output layer. The layers between the input and the output, are called the hidden layers. Any Neural Network has 1 input layer and 1 output layer.
- The number of hidden layers differ between different networks depending on the complexity of the problem. Also, each hidden layer can have its own activation function.

- Here 3 terms Comes in picture 1. Neuron , 2. Weights , 3. Bias , 4. Actiation_Function

1. Neuron

- Like in a human brain, the basic building block of a Neural Network is a Neuron. Its functionality is similar to a human brain, i.e, it takes in some inputs and fires an output. Each neuron is a small computing unit that takes a set of real valued numbers as input, performs some computation on them, and produces a single output value.
- The basic unit of computation in a neural network is the neuron, often called as a node or unit. It receives input from some other nodes, or from an external source and computes an output. Each input has an associated weight (w), which is assigned on the basis of its relative importance to other inputs. The node applies a activation function f (defined below) to the weighted sum of its inputs as in figure below.



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

The above network have:

- numerical inputs X1 and X2
- weights w1 and w2 associated with those inputs
- b (called the Bias) associated with it.

The Left side Picture is Neuron Of Human Brain ,The Right Side is Artificial Neuron

Biological Neuron Work:

- Information from other neurons, in the form of electrical impulses, enters the dendrites at connection points called synapses. The information flows from the dendrites to the cell where it is processed. The output signal, a train of impulses, is then sent down the axon to the synapse of other neurons.

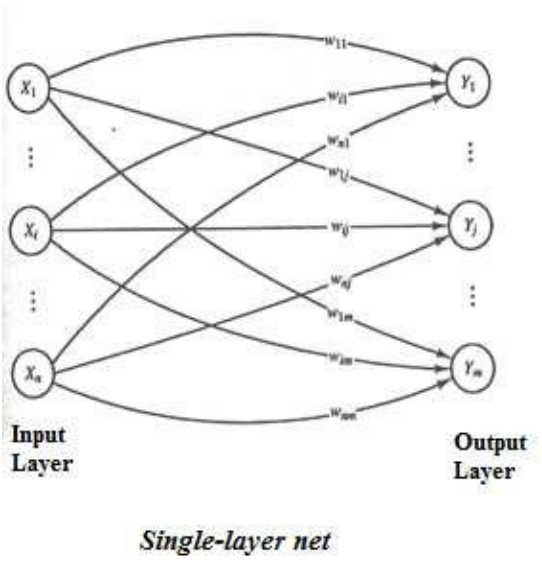
Artificial Neuron Work:

- The arrangements and connections of the neurons made up the network and have three layers.
- The first layer is called the input layer and is the only layer exposed to external signals.
- The input layer transmits signals to the neurons in the next layer, which is called a hidden layer. The hidden layer extracts relevant features or patterns from the received signals.
- Those features or patterns that are considered important are then directed to the output layer, which is the final layer of the network.

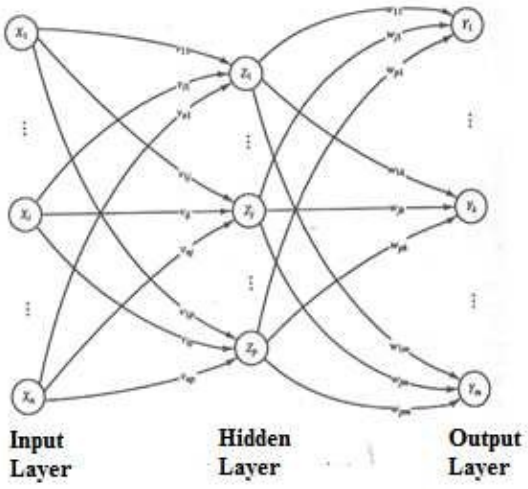
Difference between the human brain and computers in terms of how information is processed.

Human Brain(Biological Neuron Network)	Computers(Artificial Neuron Network)
The human brain works asynchronously	Computers(ANN) work synchronously.
Biological Neurons compute slowly (several ms per computation)	Artificial Neurons compute fast (<1 nanosecond per computation)
The brain represents information in a distributed way because neurons are unreliable and could die any time.	In computer programs every bit has to function as intended otherwise these programs would crash.
Our brain changes their connectivity over time to represents new information and requirements imposed on us.	The connectivity between the electronic components in a computer never change unless we replace its components.
Biological neural networks have complicated topologies.	ANNs are often in a tree structure.
Researchers are still to find out how the brain actually learns.	ANNs use Gradient Descent for learning.

Single Layers And Multi Layer Netwrok



Single-layer net



Multi-layer net

- In Multi Layer net there are many number of hidden layer in between input and output layer,but in single only one or not hidden layer.

Weight:

- Every input(x) to a neuron has an associated weight(w), which is assigned on the basis of its relative importance to other inputs.
- The way a neuron works is, if the weighted sum of inputs is greater than a specific threshold, it would give an output 1, otherwise an output 0. This is the mathematical model of a neuron, also known as the Perceptron.
- Every neural unit takes in a weighted sum of its inputs, with an additional term in the sum called a Bias.

Bias:

- Bias is a constant which is used to adjust the output along with the weighted sum of inputs, so that the model can best fit for the given data.

$$z=w.x+b$$

I defined

- weighted sum z
- weight vector w
- input vector x
- bias value b.

$$y=a=f(z)$$

- The output(y) of the neuron is a function f of the weighted sum of inputs z. The function f is non linear and is called the Activation Function.

Activation Function: ¶

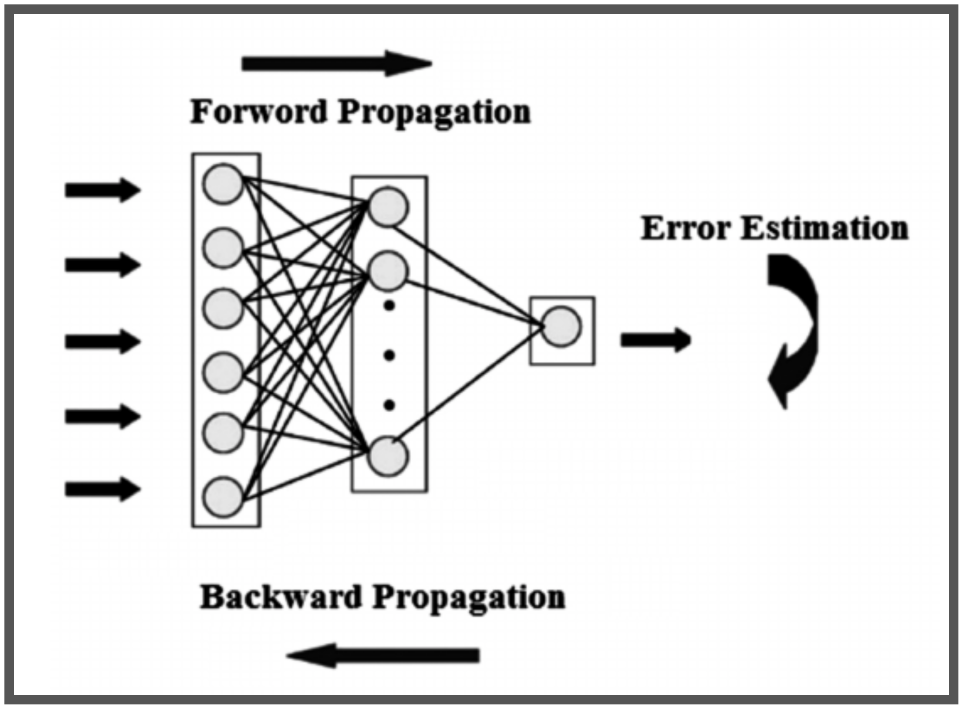
- The purpose of activation function is to introduce non-linearity into the output of neuron. It takes a single number, and performs some mathematical operation on it. There are several activation functions used in practice:

    1. Sigmoid
    2. Tanh
    3. ReLU
    4. Leaky relu
    5. Softmax function
- These Are most widly use activation function that i covered in subsequent notebook.

Forward And Backward Propogation

Every Neural Network has 2 main parts:

1. Feed Forward Propogation/Forward Propogation.
2. Backward Propogation/Back propogation.



Feed Forward Propogation:

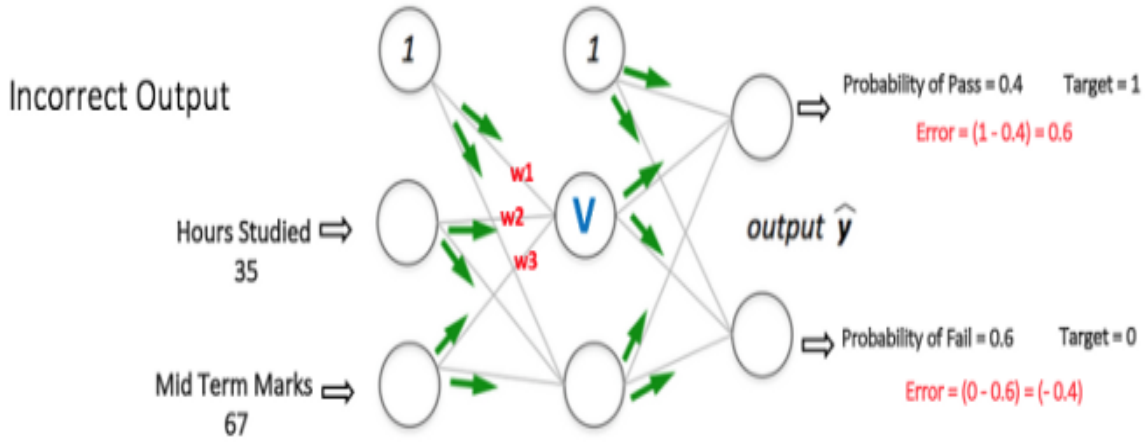
- All weights in the network are randomly assigned. Assume the weights of the connections from the inputs to that node are w1, w2 and w3.
- Here I take an example to better understand of this two concept
- Let take an example of if you study 35 Hour per day then you definitly Pass the exam with 67 Mark. Now we apply this.

    - Input to the network = [35, 67]
    - Desired output from the network (target) = [1, 0] 1 means PASS and 0 means Fail

Then output V from the node in consideration can be calculated as below (f is an activation function such as sigmoid):

$$V = f(1*w1 + 35*w2 + 67*w3)$$

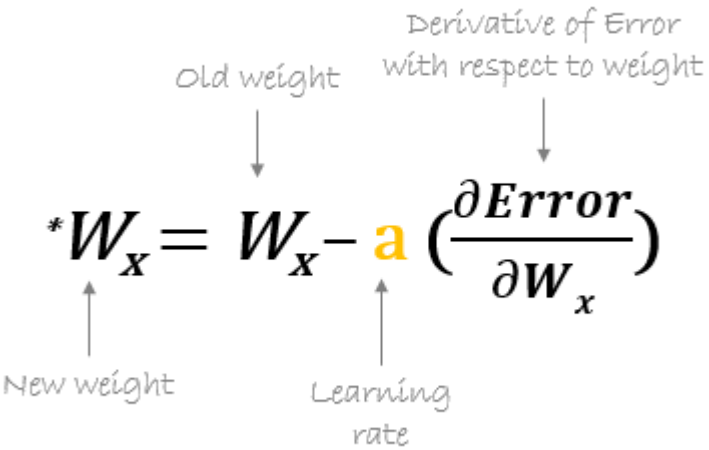
- Similarly, outputs from the other node in the hidden layer is also calculated. The outputs of the two nodes in the hidden layer act as inputs to the two nodes in the output layer. This enables us to calculate output probabilities from the two nodes in output layer.



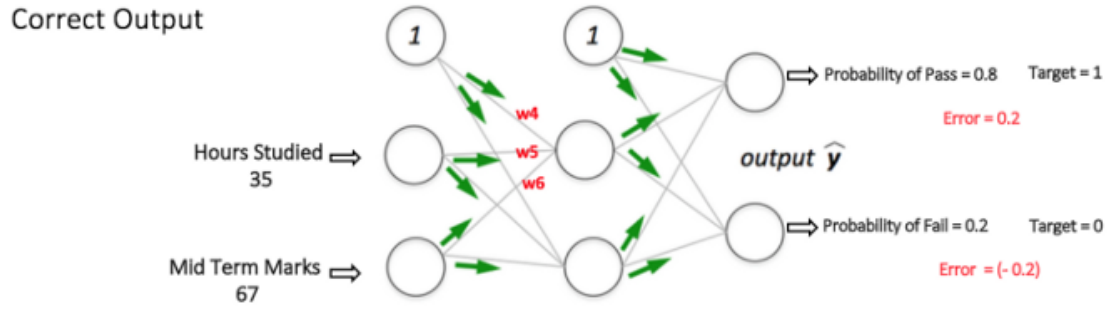
- Suppose the output probabilities from the two nodes in the output layer are 0.4 and 0.6 respectively (since the weights are randomly assigned, outputs will also be random). We can see that the calculated probabilities (0.4 and 0.6) are very far from the desired probabilities (1 and 0 respectively), hence the network in above Figure is said to have an 'Incorrect Output'. As it give output as fail but it not happen that one study 35 hr and secure 67.
- here Weight are randomly assigned so now we do Back Propagation and with Weight Updation.

Back Propagation and Weight Updation:

- Here We calculate the total error at the output nodes and propagate these errors back through the network using Backpropagation to calculate the gradients.
- Then we use an optimization method such as Gradient Descent to 'adjust' all weights in the network with an aim of reducing the error at the output layer.



- If we now input the same example to the network again, the network should perform better than before since the weights have now been adjusted to minimize the error in prediction.



- As shown in above Figure, the errors at the output nodes now reduce to [0.2, -0.2] as compared to [0.6, -0.4] earlier. This means that our network has learnt to correctly classify our first training example.
- We repeat this process with all other training examples in our dataset. Then, our network is said to have learnt those examples.

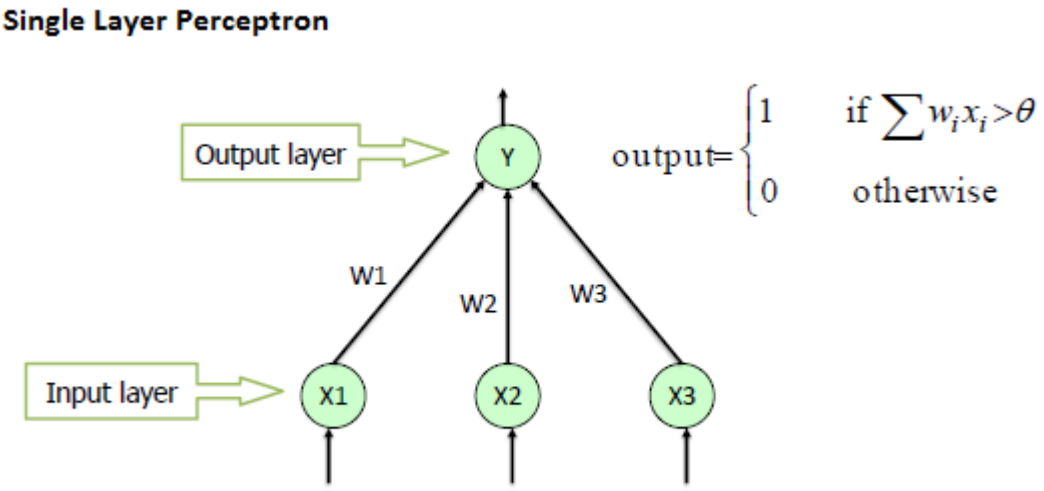
```
1. Thanks To https://medium.com/@purnasaigudikandula/a-beginner-intro-to-
neural-networks-
543267bda3c8#:~:text=Neural%20networks%20are%20set%20of,the%20functioning%20of%
20human%20brian.&text=That's%20how%20similar%20the%20Neural,and%20outputs%20wha
t%20it%20is.
2. http://home.agh.edu.pl/~vlsi/AI/backp_t_en/backprop.html
```

What is Perceptron?

Perceptron is a single layer neural network and a multi-layer perceptron is called Neural Networks.

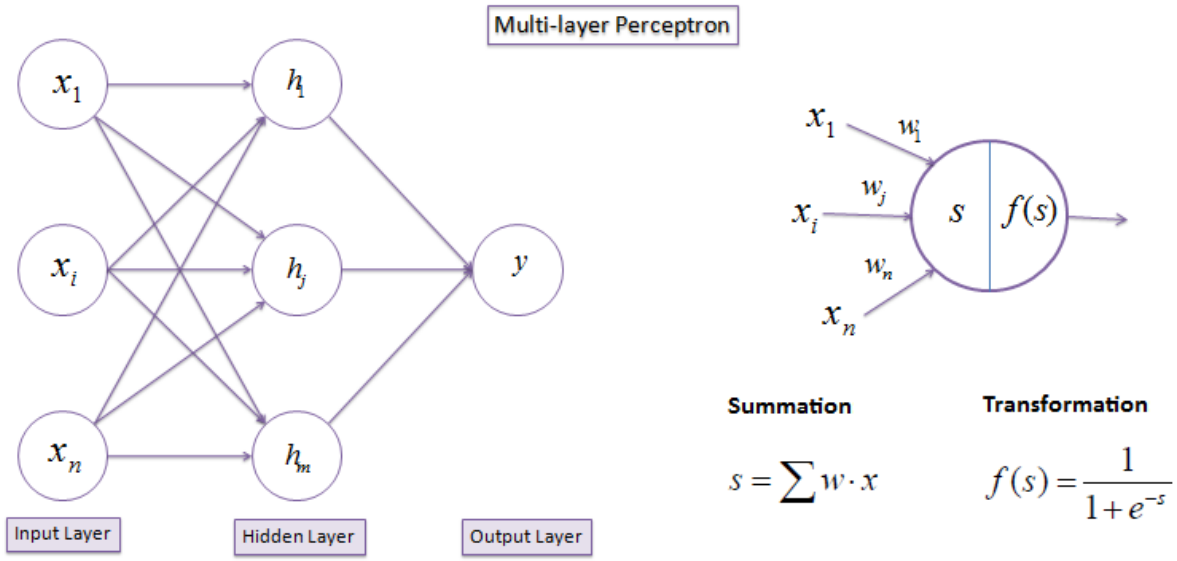
1. Single-layered perceptron model
2. Multi-layered perceptron model.

1. Single-layered perceptron model



- If you talk about the functioning of the single-layered perceptron model, its algorithm doesn't have previous information, so initially, weights are allocated inconstantly, then the algorithm adds up all the weighted inputs,
- if the added value is more than some pre-determined value(or, threshold value) then single-layered perceptron is stated as activated and delivered output as +1.
- In simple words, multiple input values feed up to the perceptron model, model executes with input values, and if the estimated value is the same as the required output, then the model performance is found out to be satisfied, therefore weights demand no changes. In fact, if the model doesn't meet the required result then few changes are made up in weights to minimize errors.

2. Multi-layered perceptron model



- In the forward stage, activation functions are originated from the input layer to the output layer, and in the backward stage, the error between the actual observed value and demanded given value is originated backward in the output layer for modifying weights and bias values.
- In simple terms, multi-layered perceptron can be treated as a network of numerous artificial neurons overhead varied layers, the activation function is no longer linear, instead, non-linear activation functions such as Sigmoid functions, TanH, ReLU activation Functions, etc are deployed for execution.

Activation Function

- Activation function decides, whether a neuron should be activated or not by calculating weighted sum and further adding bias with it. The purpose of the activation function is to introduce non-linearity into the output of a neuron.

Explanation :-

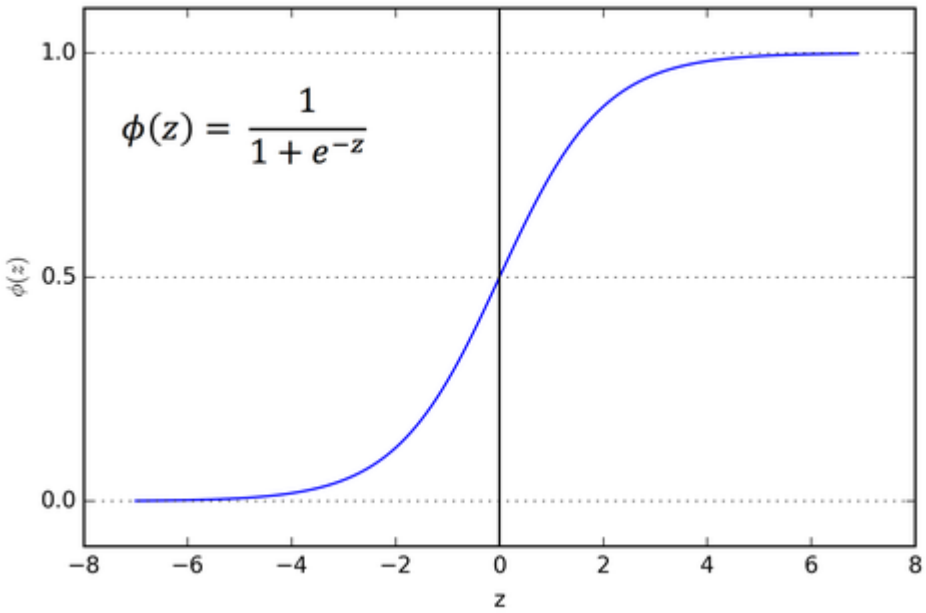
We know, neural network has neurons that work in correspondence of weight, bias and their respective activation function. In a neural network, we would update the weights and biases of the neurons on the basis of the error at the output. This process is known as back-propagation. Activation functions make the back-propagation possible since the gradients are supplied along with the error to update the weights and biases.
Why do we need Non-linear activation functions :-
A neural network without an activation function is essentially just a linear regression model. The activation function does the non-linear transformation to the input making it capable to learn and perform more complex tasks.

There are several type of Activation But here i dicuss some of them:

1. Sigmoid (Binary Classification)
2. Tanh
3. Relu
4. Leaky Relu
5. Linear
6. Softmax (Use Multiclass Classification)

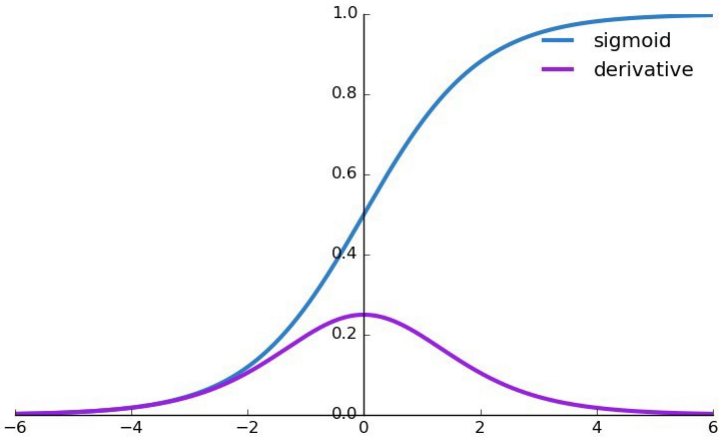
1. Sigmoid

- The Sigmoid Function curve looks like a S-shape.



- The main reason why we use sigmoid function is because it exists between (0 to 1). Therefore, it is especially 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.

Derivative Of Sigmoid Function

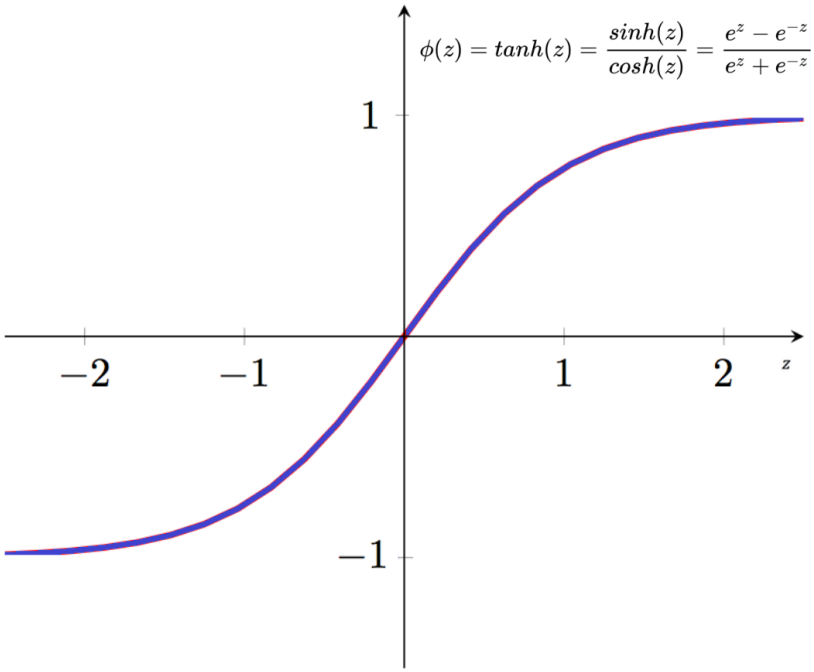


- Sigmoid Function Derivative range from 0 to 0.25

Uses : Usually used in output layer of a binary classification, where result is either 0 or 1, as value for sigmoid function lies between 0 and 1 only so, result can be predicted easily to be 1 if value is greater than 0.5 and 0 otherwise.

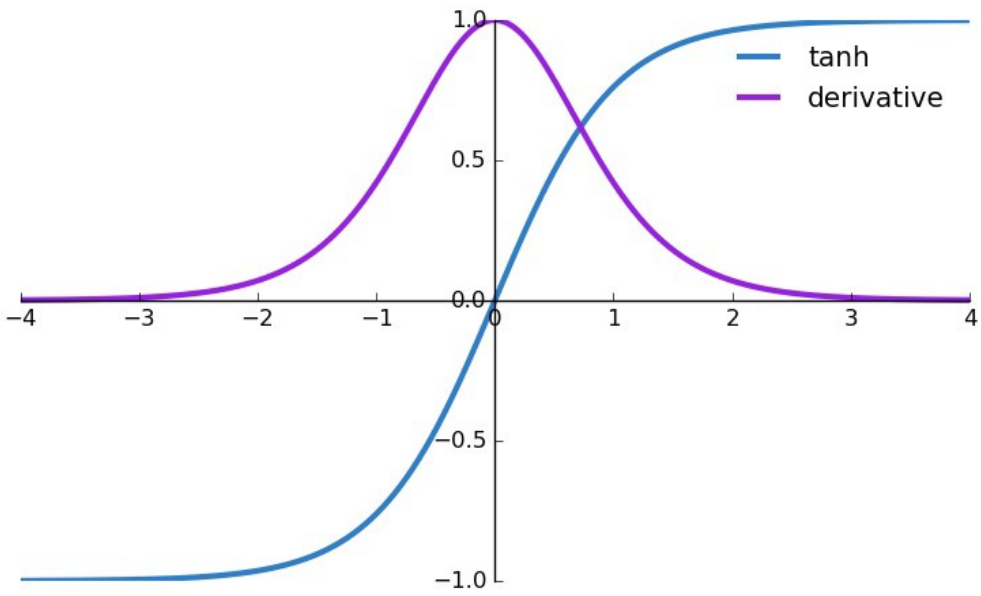
2. Tanh or hyperbolic tangent Activation Function

- tanh is also like logistic sigmoid but better. The range of the tanh function is from (-1 to 1). tanh is also sigmoidal (s - shaped).



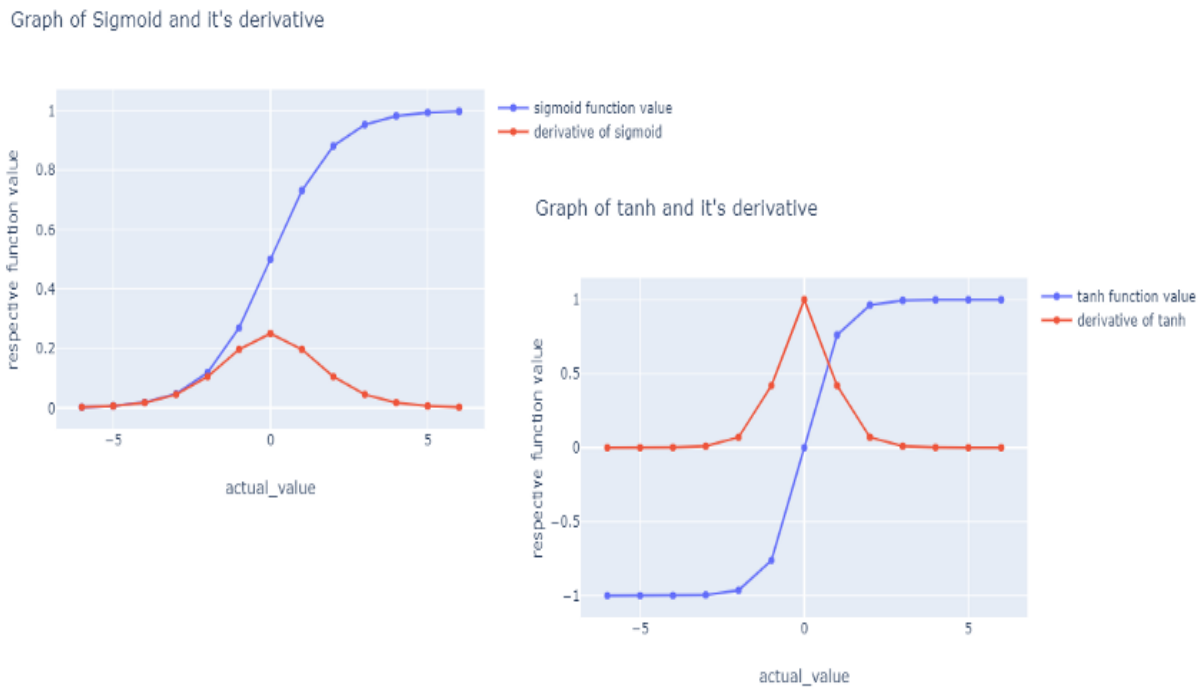
- The advantage is that the negative inputs will be mapped strongly negative and the zero inputs will be mapped near zero in the tanh graph.
- The activation that works almost always better than sigmoid function is Tanh function also knows as Tangent Hyperbolic function. It's actually mathematically shifted version of the sigmoid function. Both are similar and can be derived from each other.

Derivative Of Tanh Function:-



Sigmoid Function Derivative range from 0 to 1

Uses :- Usually used in hidden layers of a neural network as it's values lies between -1 to 1 hence the mean for the hidden layer comes out be 0 or very close to it, hence helps in centering the data by bringing mean close to 0. This makes learning for the next layer much easier.



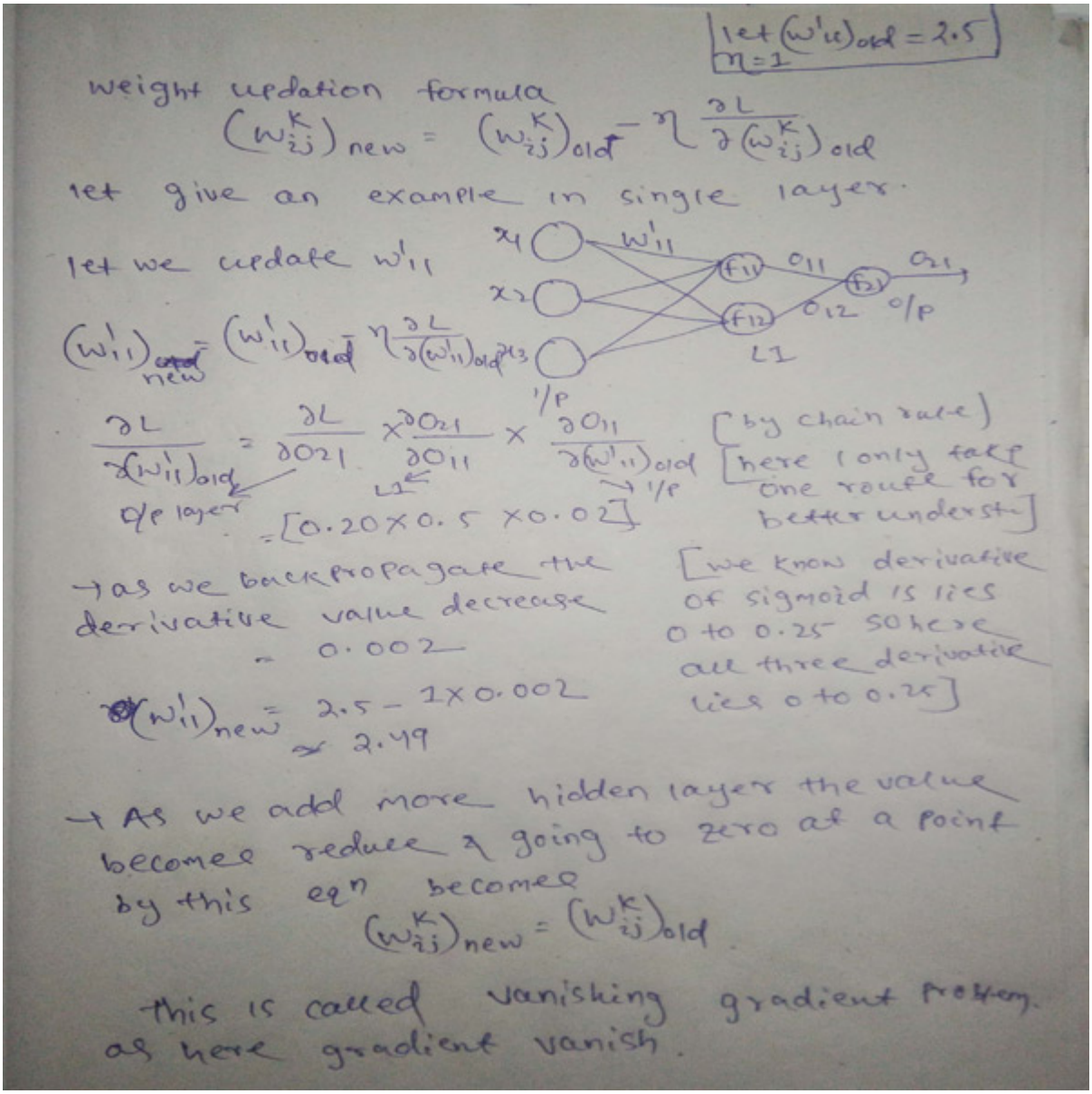
Point :-

1. tanh and logistic sigmoid are the most popular activation functions in 90's but because of their Vanishing gradient problem and sometimes Exploding gradient problem (because of weights), they aren't mostly used now.
2. These days Relu activation function is widely used. Even though, it sometimes gets into vanishing gradient problem, variants of Relu help solving such cases.
3. tanh is preferred to sigmoid for faster convergence BUT again, this might change based on data. Data will also play an important role in deciding which activation function is best to choose.

Vanishing Gradient Problem

- Vanishing gradient problem is a common problem that we face while training deep neural networks.Gradients of neural networks are found during back propagation.
- Generally, adding more hidden layers will make the network able to learn more complex arbitrary functions, and thus do a better job in predicting future outcomes. This is where Deep Learning is making a big difference.

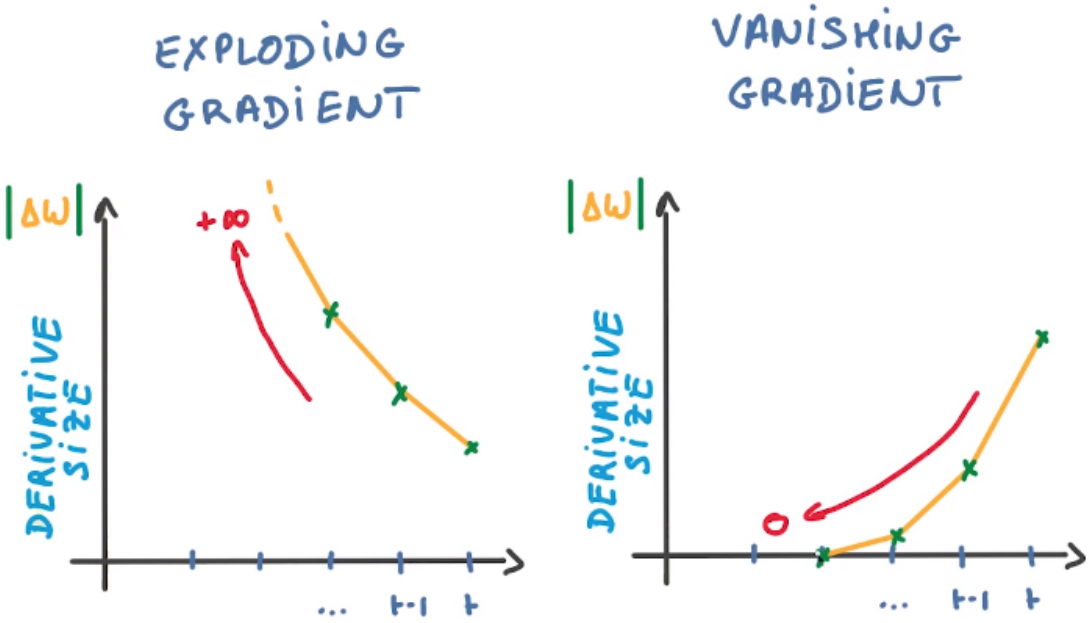
Now during back-propagation i.e moving backward in the Network and calculating gradients, it tends to get smaller and smaller as we keep on moving backward in the Network . Below is Just a simple demonstration of Vanishing Gradient Problem in single layer.



- This Happen because of we use sigmoid and tanh activation function in hidden layer. As sigmoid and tanh deriative 0.25,1 respectively. so by calculating number of hidden layer the derivative becomes 0 so avoid it we use RELU activation function in hidden layer.

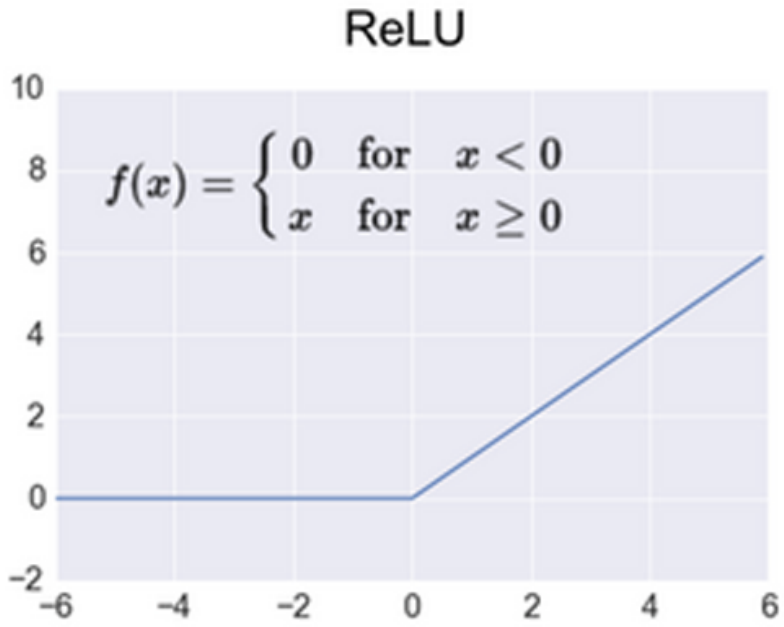
Exploding gradient Problem

- We have discussed about vanishing gradient problem.Now we will get in to exploding gradient problem.Earlier we discussed what happens when our gradient becomes very small.Now we will discuss what will happen if it gets large.
- In deep networks or recurrent neural networks, error gradients can accumulate during an update and result in very large gradients.
- These in turn result in large updates to the network weights, and in turn, an unstable network.The explosion occurs through exponential growth by repeatedly multiplying gradients through the network layers that have values larger than 1.0.This will ultimately led to an total unstable network.



3. Relu

- By Using of Sigmoid And Tanh function there is vanishing gradient problem occure so the convergence rate slow down.
- To overcome slighlty we use Relu Function. it not totally overcome this problem but here the convergence rate is faster than sigmoid and tanh.
- Value Range :- [0, inf)
- Its Nature non-linear, which means we can easily backpropagate the errors and have multiple layers of neurons being activated by the ReLU function.



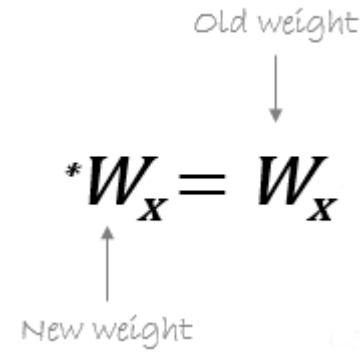
Point:

1. Vanishing Gradient Problem occure due to multiple number of derivative.
2. As sigmoid derivative 0 to 0.25 so by multiple by this sigmoid derivative the result might vanishing gradient as number of hidden layer increase.
3. But in Relu here its derivative 0 to 1 so here no problem of any vanishing gradient problem as its derivative cant be 0.2,0.3 like.
4. But as its derivative can be 0 so here a problem aries that called Dead_Activation

Uses :- ReLu is less computationally expensive than tanh and sigmoid because it involves simpler mathematical operations. At a time only a few neurons are activated making the network sparse making it efficient and easy for computation.

What is Dead_Activation ?

- When Derivative equal to 0 in Relu Then New Weight = Old Weight :



which is not good for any model.Here Weight Cant be update. it occure when value of z is negative.This state called Dead Activation state. to overcome this we use Leaky Relu.

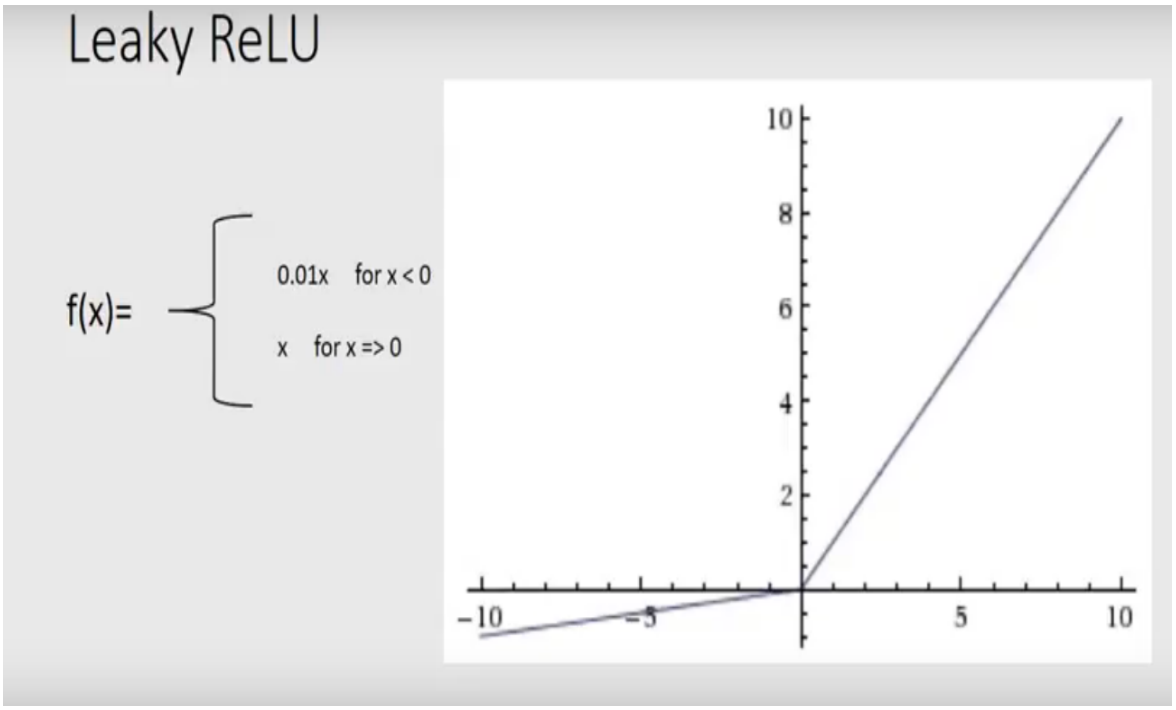
```
Epoch 1/25
32428/32428 [==============================] - 41s 1ms/step - loss: nan - accuracy: 0.5900 - val_loss: nan - val_accuracy: 0.59
44
Epoch 2/25
32428/32428 [==============================] - 39s 1ms/step - loss: nan - accuracy: 0.5907 - val_loss: nan - val_accuracy: 0.59
44
Epoch 3/25
32428/32428 [==============================] - 36s 1ms/step - loss: nan - accuracy: 0.5907 - val_loss: nan - val_accuracy: 0.59
44
Epoch 4/25
32428/32428 [==============================] - 34s 1ms/step - loss: nan - accuracy: 0.5907 - val_loss: nan - val_accuracy: 0.59
44
Epoch 5/25
32428/32428 [==============================] - 36s 1ms/step - loss: nan - accuracy: 0.5907 - val_loss: nan - val_accuracy: 0.59
44
Epoch 6/25
32428/32428 [==============================] - 37s 1ms/step - loss: nan - accuracy: 0.5907 - val_loss: nan - val_accuracy: 0.59
44
Epoch 7/25
32428/32428 [==============================] - 36s 1ms/step - loss: nan - accuracy: 0.5907 - val_loss: nan - val_accuracy: 0.59
44
Epoch 8/25
32428/32428 [==============================] - 38s 1ms/step - loss: nan - accuracy: 0.5907 - val_loss: nan - val_accuracy: 0.59
```

- See here no updation happen the value remain same and ‘nan’ for validation loss is an unexpected very large or very small number. This is dead activation state to overcome this we use leaky relu

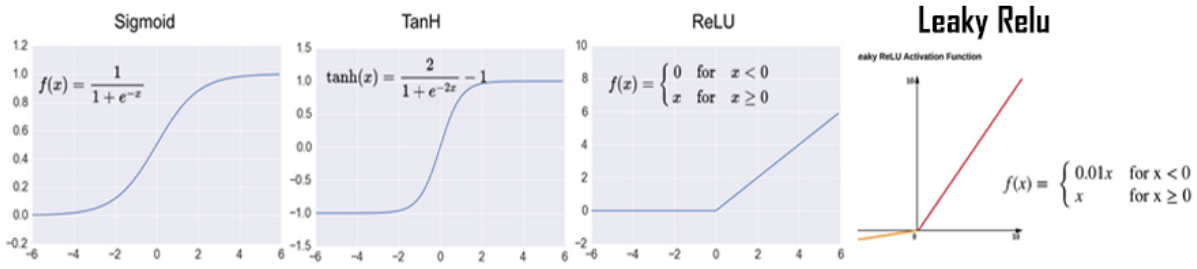
4. Leaky Relu

- Leaky ReLU is an improved version of the ReLU function.
- ReLU function, the gradient is 0 for x < 0(-ve), which made the neurons die for activations in that region.
- Leaky ReLU is defined to address this problem. Instead of defining the Relu function as 0 for x less than 0, we define it as a small linear component of x.
- Leaky ReLUs are one attempt to fix the Dying ReLU problem. Instead of the function being zero when x < 0, a leaky ReLU will instead have a small negative slope (of 0.01, or so). That is, the function computes:

$$f(x) = 1(x < 0)(\alpha x) + 1(x >= 0)(x)$$



Sigmoid,Tanh,Relu,Leaky Relu



Softmax

- Softmax is used as the activation function for multi-class classification tasks, usually the last layer.
- We talked about its role transforming numbers (aka logits) into probabilities that sum to one.
- Let's not forget it is also an activation function which means it helps our model achieve non-linearity. Linear combinations of linear combinations will always be linear but adding activation function helps gives our model ability to handle non-linear data.
- Output of other activation functions such as sigmoid does not necessarily sum to one. Having outputs summing to one makes softmax function great for probability analysis.
- The function is great for classification problems, especially if you're dealing with multi-class classification problems, as it will report back the "confidence score" for each class. Since we're dealing with probabilities here, the scores returned by the softmax function will add up to 1.

Mathematical representation

$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

Where:

σ = softmax

$\vec{z}$=input vector

e^{z_i} =standard exponential function for input vector

K = number of classes in the multi-class classifier

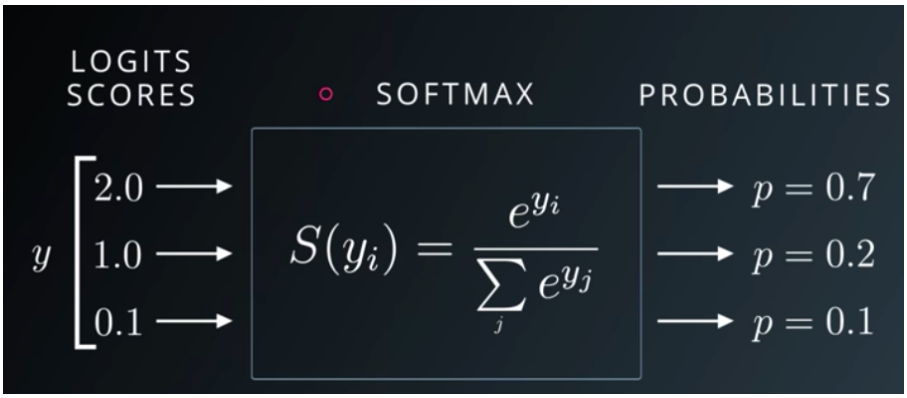
e^{z_j} =standard exponential function for output vector

- It states that we need to apply a standard exponential function to each element of the output layer, and then normalize these values by dividing by the sum of all the exponentials. Doing so ensures the sum of all exponentiated values adds up to 1.

Here are the steps For Softmax:

1. Exponentiate every element of the output layer and sum the results
2. Take each element of the output layer, exponentiate it and divide by the sum obtained in step 1

Example Implementation



To start, let's declare an array which imitates the output layer of a neural network:

In [1]:
```
## as we know softmax used in output layer so here i take a outputlayer value
import numpy as np
output_layer = np.array([2.0,1.0,0.1])
output_layer
```

Out[1]:
```
array([2. , 1. , 0.1])
```

By step 1 we need to exponentiate each of the elements of the output layer:

In [2]:
```
exponentiated = np.exp(output_layer)
exponentiated
```

Out[2]:
```
array([7.3890561 , 2.71828183, 1.10517092])
```

According to step 2 calculate probabilities! We can use Numpy to divide each element by exponentiated sum and store results in another array

In [3]:
```
probabilities = exponentiated / np.sum(exponentiated)
print(probabilities)
print(sum(probabilities))
print(np.argmax(probabilities))
```

```
[0.65900114 0.24243297 0.09856589]
1.0
0
```

Here see the output are formed. If we sum three then we get probability 1. after this you use argmax function which return highest value index number. See here return 0 as 0 index have 0.65 value which is highest among three value.

When you use softmax in your dataset you should use argmax function to predict output.

- From a probabilistic perspective, if the argmax() function returns 1 in the large value, it returns 0 for the other two array indexes.here it giving full weight to index 0 and no weight to index 1

and index 2 for the largest value in the list [0.65,0.24,0.09].

In the Keras deep learning library with a three-class classification task, use of softmax in the output layer may look as follows:

model.add(Dense(no.of output layer, activation='softmax'))

- It apply when you have multiclass problem aries

The Differences between Sigmoid and Softmax Activation Functions

In [4]:
```
import numpy as np
### sigmoid function
def sigmoid(x):
    s = 1 / (1 + np.exp(-x))
    return s

## softmax function
def softmax(x):
    exponentiated = np.exp(x)
    probabilities = exponentiated / np.sum(exponentiated)
    return probabilities
```

In [5]:
```
x = np.array([-0.5, 1.2, -0.1, 2.4])
a = sigmoid(x)
print("--- Sigmoid---")
print(a.round(2))
print(sum(a).round(2))

print(50*"*")

output_layer = np.array([-0.5, 1.2, -0.1, 2.4])
b = softmax(output_layer)
print("---Softmax---")
print(b.round(2))
print(sum(b))
```

```
--- Sigmoid---
[0.38 0.77 0.48 0.92]
2.54
**************************************************
---Softmax---
[0.04 0.21 0.06 0.7 ]
1.0
```

The key takeaway from this example is:

- **Sigmoid:** probabilities produced by a Sigmoid are independent. Furthermore, they are not constrained to sum to one: 0.38 + 0.77 + 0.48 + 0.92 = 2.54. The reason for this is because the Sigmoid looks at each raw output value separately.
- **Softmax:** the outputs are interrelated. The Softmax probabilities will always sum to one by design: 0.04 + 0.21 + 0.06 + 0.7 = 1.00. In this case, if we want to increase the likelihood of one class, the other has to decrease by an equal amount.

Summary..

Characteristics of a Sigmoid Activation Function:

1. Used for Binary Classification in the Logistic Regression model
2. The probabilities sum does not need to be 1
3. Used as an Activation Function while building a Neural Network

Characteristics of a Softmax Activation Function

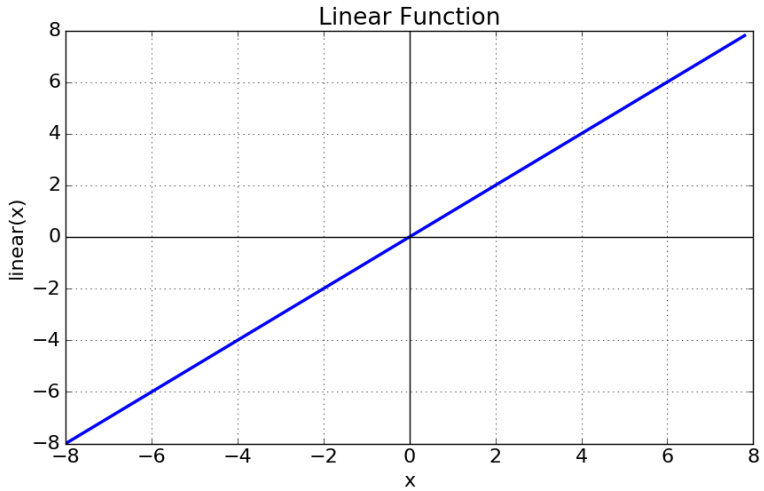
1. Used for Multi-classification in the Logistics Regression model
2. The probabilities sum will be 1
3. Used in the different layers of Neural Networks

Activation Function For Regression Problem

- Linear Activation Function:-

Equation : f(x) = x

Range : (-infinity to infinity)



- No matter how many layers we have, if all are linear in nature, the final activation function of last layer is nothing but just a linear function of the input of first layer.
- Linear activation function is used at just one place i.e. output layer.
- If we will differentiate linear function to bring non-linearity, result will no more depend on input “x” and function will become constant, it won’t introduce any ground-breaking behavior to our algorithm.

algorithm.

Uses: Calculation of price of a house is a regression problem. House price may have any big/small value, so we can apply linear activation at output layer. Even in this case neural net must have any non-linear function at hidden layers.

Loss Functions

- The loss function is the function that computes the distance between the current output of the algorithm and the expected output. It's a method to evaluate how your algorithm models the data. It can be categorized into two groups. One for classification (discrete values, 0,1,2…) and the other for regression (continuous values).

TYPES OF LOSS FUNCTION:

1. Regression Loss Functions

```
Mean Absolute Error
Mean Squared Error
Root Mean Square error (RMSE)
```

2. Binary Classification Loss Functions

```
Binary Cross-Entropy
```

3. Multi-Class Classification Loss Functions

```
Multi-Class Cross-Entropy Loss
Sparse Multiclass Cross-Entropy Loss
```

Regression Losses

We know all of this regression loss function but here i discuss a brief

Mean Absolute Error

- Regression metric which measures the average magnitude of errors in a group of predictions, without considering their directions. In other words, it's a mean of absolute differences among predictions and expected results where all individual deviations have even importance.

$$\mathrm{MAE} = \underbrace{\frac{1}{n}\sum_{i=1}^{n}}_{\text{test set}} \underbrace{|y_i}_{\text{predicted vaue}} - \underbrace{\hat{y}_i|}_{\text{actual value}}$$

where:

i — index of sample,

ŷ — predicted value,

y — expected value,

m — number of samples in dataset.

Sometimes it is possible to see the form of formula with swapped predicted value and expected value, but it works the same.

Mean Squared Error

- One of the most commonly used and firstly explained regression metrics. Average squared difference between the predictions and expected results. In other words, an alteration of MAE where instead of taking the absolute value of differences, they are squared.
- In MAE, the partial error values were equal to the distances between points in the coordinate system. Regarding MSE, each partial error is equivalent to the area of the square created out of the geometrical distance between the measured points. All region areas are summed up and averaged.

$$\underbrace{\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}}_{\text{test set}} \underbrace{(y_i}_{\text{predicted vaue}} - \underbrace{\hat{y}_i}_{\text{actual value}})^2$$

Where

i — index of sample,

ŷ — predicted value,

y — expected value,

m — number of samples in dataset.

Root Mean Square error (RMSE)

- Root Mean Square error is the extension of MSE — measured as the average of square root of sum of squared differences between predictions and actual observations.

$$RMSE = \sqrt{\frac{\sum_{i=1}^{N}(Predicted_i - Actual_i)^2}{N}}$$

Classification Losses

Binary Classification Loss Functions

Binary Cross Entropy

- Also called Sigmoid Cross-Entropy loss. It is a Sigmoid activation plus a Cross-Entropy loss. Unlike Softmax loss it is independent for each vector component (class), meaning that the loss computed for every CNN output vector component is not affected by other component values.
- Binary cross entropy measures how far away from the true value (which is either 0 or 1) the prediction is for each of the classes and then averages these class-wise errors to obtain the final loss.
- We can define cross entropy as the difference between two probability distributions p and q, where p is our true output and q is our estimate of this true output.
- it Only use for binary classification problem

$$H(x) = \sum_{i=1}^{N} \overset{\text{true label}}{p}(x) \log \underset{\text{estimate}}{q}(x)$$

Multi-Class Classification Loss Functions

Categorical cross-entropy

- Used binary and multiclass problem, the label needs to be encoded as categorical, one-hot encoding representation (for 3 classes: [0, 1, 0], [1,0,0]…)
- It is a loss function that is used for single label categorization. This is when only one category is applicable for each data point. In other words, an example can belong to one class only.

$$L(y, \hat{y}) = - \sum_{j=0}^{M} \sum_{i=0}^{N} (y_{ij} * log(\hat{y}_{ij}))$$

- Use categorical crossentropy in classification problems where only one result can be correct.
- Example: In the MNIST problem where you have images of the numbers 0,1, 2, 3, 4, 5, 6, 7, 8, and 9. Categorical crossentropy gives the probability that an image of a number is, for example, a 4 or a 9.

- Categorical cross-entropy will compare the distribution of the predictions (the activations in the output layer, one for each class) with the true distribution, where the probability of the true class is set to 1 and 0 for the other classes. To put it in a different way, the true class is represented as a one-hot encoded vector, and the closer the model's outputs are to that vector, the lower the loss.

Sparse Categorical cross-entropy

- Used binary and multiclass problem (the label is an integer — 0 or 1 or … n, depends on the number of labels)

If your targets are **one-hot encoded**, use `categorical_crossentropy`.

- Examples of one-hot encodings:
  - `[1,0,0]`
  - `[0,1,0]`
  - `[0,0,1]`

But if your targets are **integers**, use `sparse_categorical_crossentropy`.

- Examples of integer encodings (*for the sake of completion*):
  - `1`
  - `2`
  - `3`

All Losses : [https://keras.io/api/losses/ (https://keras.io/api/losses/)](https://keras.io/api/losses/)

Summary

There are three kinds of classification tasks:

```
1. Binary classification: two exclusive classes
2. Multi-class classification: more than two exclusive classes
3. Multi-label classification: just non-exclusive classes
```

Here, we can say

```
1. In the case of (1), you need to use binary cross entropy.
2. In the case of (2), you need to use categorical cross entropy.
3.In the case of (3), you need to use binary cross entropy.
```

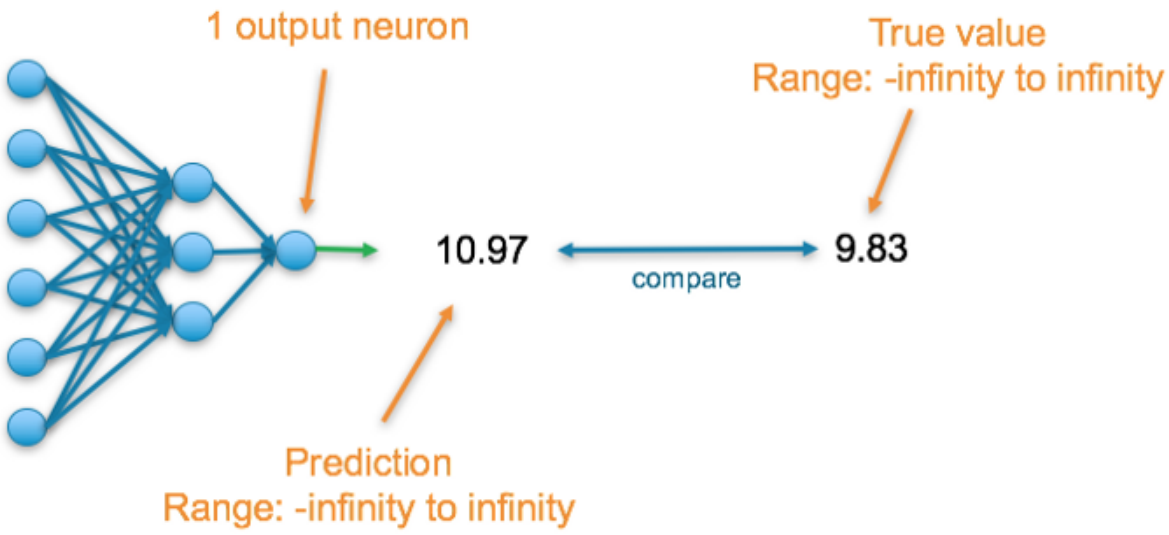
Which Loss and Activation Functions should I use?

- The motive of the blog is to give you some ideas on the usage of “Activation Function” & “Loss function” in different scenarios.
- Choosing an activation function and loss function is directly dependent upon the output you want to predict. There are different cases and different outputs of a predictive model. Before I introduce you to such cases let see an introduction to the activation function and loss function.
- The activation function activates the neuron that is required for the desired output, converts linear input to non-linear output. If you are not aware of the different activation functions I would recommend you visit my activation pdf to get an in-depth explanation of different activation functions click here : https://github.com/pratyusa98/ML_Algo_pdf/tree/main/01_Deep_Learning_PDF (https://github.com/pratyusa98/ML_Algo_pdf/tree/main/01_Deep_Learning_PDF).
- Loss function helps you figure out the performance of your model in prediction, how good the model is able to generalize. It computes the error for every training. You can read more about loss functions and how to reduce the loss https://github.com/pratyusa98/ML_Algo_pdf/tree/main/01_Deep_Learning_PDF (https://github.com/pratyusa98/ML_Algo_pdf/tree/main/01_Deep_Learning_PDF)..

Let’s see the different cases:

CASE 1: When the output is a numerical value that you are trying to predict

- Ex:- Consider predicting the prices of houses provided with different features of the house. A neural network structure where the final layer or the output later will consist of only one neuron that reverts the numerical value. For computing the accuracy score the predicted values are compared to true numeric values.



- Activation Function to be used in Output layer such cases,

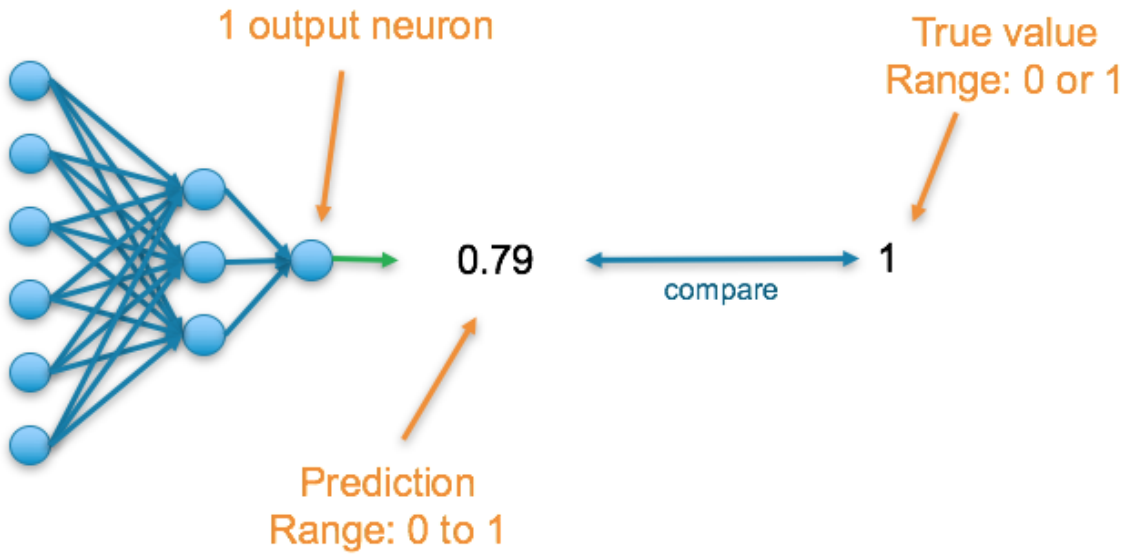
```
        * Linear Activation - it gives output in a numeric form th
at is the demand for this case. Or
        * ReLU Activation - This activation function gives you pos
itive numeric outputs as a result.
```

- Loss function to be used in such cases,

```
        * Mean Squared Error (MSE) - This loss function is respons
ible to compute the average squared difference    between the true v
alues and the predicted values.
```

CASE 2: When the output you are trying to predict is Binary

- Ex:- Consider a case where the aim is to predict whether a loan applicant will default or not. In these types of cases, the output layer consists of only one neuron that is responsible to result in a value that is between 0 and 1 that can be also called probabilistic scores.
- For computing the accuracy of the prediction, it is again compared with the true labels. The true value is 1 if the data belongs to that class or else it is 0.



- Activation Function to be used in Output layer such cases,

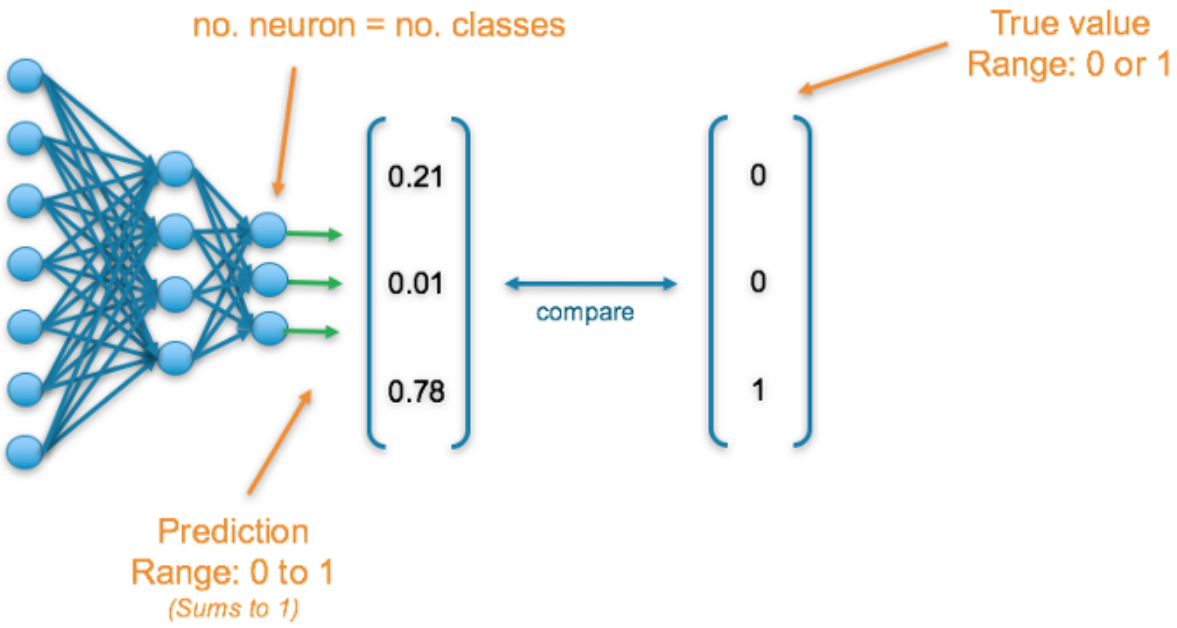
```
        * Sigmoid Activation -  This activation function gives
 the output as 0 and 1.
```

- Loss function to be used in such cases,

```
        * Binary Cross Entropy - The difference between the tw
o probability distributions is given by binary  cross-entropy. (p,1-
p) is the model distribution predicted by the model, to compare it w
ith true distribution, the   binary cross-entropy is used.
```

CASE 3: Predicting a single class from many classes

- Ex:- Consider a case where you are predicting the name of the fruit amongst 5 different fruits. In the case, the output layer will consist of only one neuron for every class and it will revert a value between 0 and 1, the output is the probability distribution that results in 1 when all are added.
- Each output is checked with its respective true value to get the accuracy. These values are one-hot-encoded which means if will be 1 for the correct class or else for others it would be zero.



- Activation Function to be used in Output layer such cases,

```
* Softmax Activation -  This activation function gives
the output between 0 and 1 that are the probability scores which if
 added gives the result as 1.
```

- Loss function to be used in such cases,

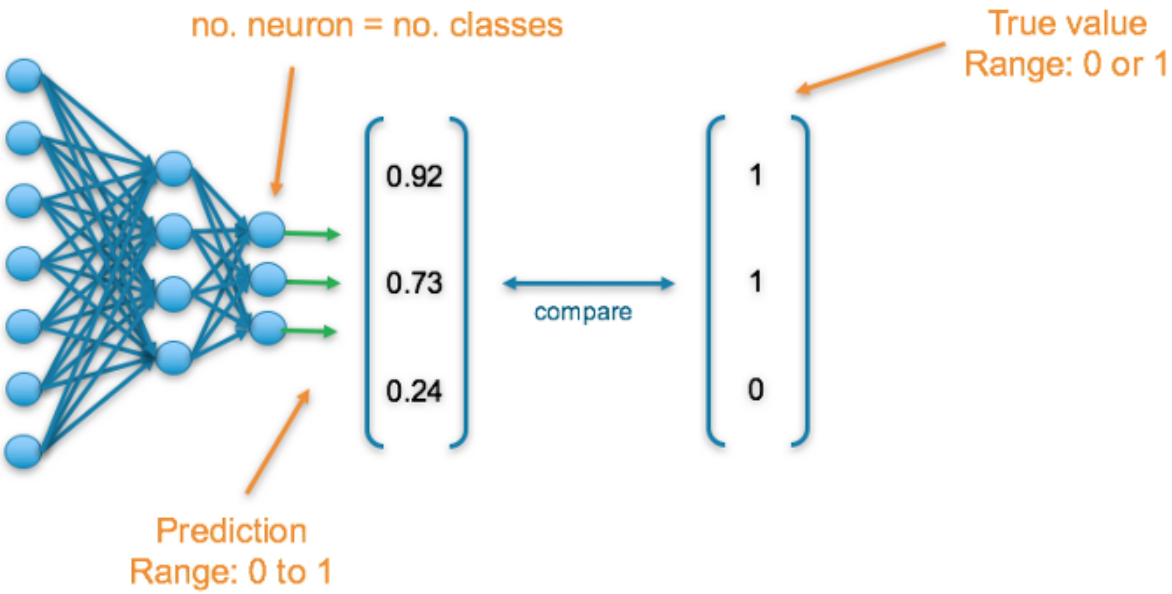
```
* Cross-Entropy - It computes the difference between t
wo probability distributions.
* (p1,p2,p3) is the model distribution that is predict
ed by the model where p1+p2+p3=1. This is compared with the true dis
tribution using cross-entropy.
```

CASE 4: Predicting multiple labels from multiple class

- Ex:- Consider the case of predicting different objects in an image having multiple objects. This is termed as multiclass classification. In these types of cases, the output layer consists of only one neuron that is responsible to result in a value that is between 0 and 1 that can be also

called probabilistic scores.

- For computing the accuracy of the prediction, it is again compared with the true labels. The true value is 1 if the data belongs to that class or else it is 0.



- Activation Function to be used in Output layer such cases,

```
                * Sigmoid Activation -  This activation function giv
es the output as 0 and 1.
```

- Loss function to be used in such cases,

```
                * Binary Cross Entropy - The difference between the
 two probability distributions is given by binary cross-entropy. (p,
1-p) is the model distribution predicted by the model, to compare it
with true distribution, the binary cross-entropy is used.
```

All Losses : https://keras.io/api/losses/ (https://keras.io/api/losses/)
All Activation : https://keras.io/api/layers/activations/ (https://keras.io/api/layers/activations/)

Summary

- This activation use only output layer and in hidden layer you can use Relu or Leaky Relu.
- The following table summarizes the above information to allow you to quickly find the final layer activation function and loss function that is appropriate to your use-case

Problem Type	Output Type	Final Activation Function	Loss Function
Regression	Numerical value	Linear	Mean Squared Error (MSE)
Classification	Binary outcome	Sigmoid	Binary Cross Entropy
Classification	Single label, multiple classes	Softmax	Cross Entropy
Classification	Multiple labels, multiple classes	Sigmoid	Binary Cross Entropy

Weight Initialization

- The weight initialization technique you choose for your neural network can determine how quickly the network converges or whether it converges at all. Although the initial values of these weights are just one parameter among many to tune, they are incredibly important. Their distribution affects the gradients and, therefore, the effectiveness of training.

Why is weight initialization important? ¶

- Improperly initialized weights can negatively affect the training process by contributing to the vanishing or exploding gradient problem.
- With the vanishing gradient problem, the weight update is minor and results in slower convergence — this makes the optimization of the loss function slow and in a worst case scenario, may stop the network from converging altogether.
- Conversely, initializing with weights that are too large may result in exploding gradient values during forward propagation or back-propagation.

1. Zero initialization :

- If all the weights are initialized with 0, the derivative with respect to loss function is the same for every weight(w), thus all weights have the same value in subsequent iterations.
- This makes hidden units symmetric and continues for all the n iterations i.e. setting weights to 0 does not make it better than a linear model.
- An important thing to keep in mind is that biases have no effect what so ever when initialized with 0.
- It also gives problems like vanishing gradient problem.

2. initialization With -ve Number :

- If all weigth can be negative then it affect Relu Activation Function.As in -ve Relu comes under dead activation problem. so we cant use this technique.
- Weights can’t be too high as gives problems like exploding Gradient problem(weights of the model explode to infinity), which means that a large space is made available to search for global minima hence convergence becomes slow.

To prevent the gradients of the network’s activations from vanishing or exploding, we need to have following rules:

```
        1. The mean of the activations should be zero.
        2. The variance of the activations should stay the same across e
very layer.
```

Idea 1 : Normal or Naïve Initialization:

- In normal distribution weights can be a part of normal or gaussian distribution with mean as zero and a unit standard deviation.

$$W \approx N(0, \sigma) \quad \mu = 0 \qquad \sigma = \text{small number}$$

- Random initialization is done so that convergence is not to a false minima.
- In Keras it can be simply written as hyperparameter as - kernel_initializer='random_normal'

Idea 2: Uniform Initialization:

- In uniform initialization of weights , weights belong to a uniform distribution in range a,b with values of a and b as below:

$$W \approx U(a,b) \qquad a = \frac{-1}{\sqrt{f\_in}} \quad , \quad b = \frac{1}{\sqrt{f\_in}}$$

- Whenever **sigmoid** activation function is used as , Uniform works well.
- In Keras it can be simply written as hyperparameter as - kernel_initializer='random_uniform'

Idea 3: Xavier/ Glorot Weight Initialization:

- The variance of weights in the case normal distribution was not taken care of which resulted in too large or too small activation values which again led to exploding gradient and vanishing gradient problems respectively, when back propagation was done.
- In order to overcome this problem Xavier Initialization was introduced. It keeps the variance the same across every layer. We will assume that our layer's activations are normally distributed around zero.
- Glorot or Xavier had a belief that if they maintain variance of activations in all the layers going forward and backward convergence will be fast as compared to using standard initialization where gap was larger.
- It have Two Varient

```
a. Normal Distribution - kernel_initializer='glorot_normal'
b. Uniform Distribution - kernel_initializer='glorot_uniform'
```

Point : Works well with **tanh , sigmoid** activation functions.

a. Normal Distribution:

- In Normal Distribution, weights belong to normal distribution where mean is zero and standard deviation is as below:

$$W \approx N(\mu, \sigma)$$

$$\mu = 0 \qquad \sigma = \sqrt{\frac{2}{f\_in + f\_out}}$$

b. Uniform Distribution:

- Uniform Distribution , weights belong to uniform distribution in range of a and b defined as below:

$$W \approx U(a,b)$$

$$a = -\sqrt{\frac{6}{f\_in + f\_out}}, \qquad b = \sqrt{\frac{6}{f\_in + f\_out}}$$

Idea 4: He-Initialization:

- When using activation functions that were zero centered and have output range between-1,1 for activation functions like tanh and softsign, activation outputs were having mean of 0 and standard deviation around 1 average wise.
- But if ReLu is used instead of tanh, it was observed that on average it has standard deviation very close to square root of 2 divided by input connections.
  - It have Two Varient

```
a. Normal Distribution - kernel_initializer='he_normal'
b. Uniform Distribution - kernel_initializer='he_uniform'
```

Point : Works well with **Relu And Leaky Relu** activation functions.

a. Normal Distribution:

- In He-Normal initialization method, weights belong to normal distribution where mean is zero and standard deviation is as below:

$$W \approx N(\mu, \sigma)$$

$$\mu = 0 \qquad \sigma = \sqrt{\frac{2}{f\_in}}$$

b. Uniform Initialization :

- In He Uniform Initialization weights belong to uniform distribution in range as shown below:

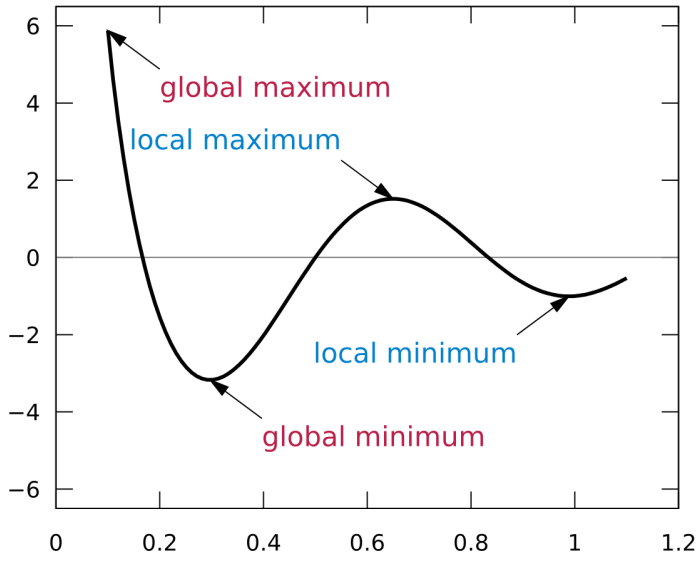
$$W \approx U(a,b) \qquad a = -\sqrt{\frac{6}{f\_in}} \quad , \quad b = \sqrt{\frac{6}{f\_in}}$$

Optimization Techniques

- Optimization algorithms are responsible for reducing losses and provide most accurate results possible.
- The weight is initialized using some initialization strategies and is updated with each epoch according to the equation. The best results are achieved using some optimization strategies or algorithms called Optimizer.

- Some of the techniques that we will be discussing in this article is-

```
* Gradient Descent
* Stochastic Gradient Descent (SGD)
* Mini-Batch Stochastic Gradient Descent (MB – SGD)
* SGD with Momentum
* Nesterov Accelerated Gradient (NAG)
* Adaptive Gradient (AdaGrad)
* AdaDelta
* RMSProp
* Adam
```



1. Gradient Descent or Batch Gradient Descent

- A Gradient Descent is an iterative algorithm, that starts from a random point on the function and traverses down its slope in steps until it reaches lowest point (global minima) of that function.
- This algorithm is apt for cases where optimal points cannot be found by equating the slope of the function to 0. For the function to reach minimum value, the weights should be altered.
- With the help of back propagation, loss is transferred from one layer to another and “weights” parameter are also modified depending on loss so that loss can be minimized.

Point :

1. Use all training Sample for a forward pass and adjust the weights.
2. This makes it computationally intensive. 3. Another drawback is there are chances the iteration values may get stuck at local minima or saddle point and never converge to minima. To obtain the best solution, the must reach global minima. 4. Good For Small training data.

Cost function: $\theta=\theta-\alpha\cdot\nabla J(\theta)$

Advantages:

- Easy computation.
- Easy to implement.
- Easy to understand.

Disadvantages:

- May trap at local minima.
- Weights are changed after calculating gradient on the whole dataset. So, if the dataset is too large than this may take years to converge to the minima.
- Requires large memory to calculate gradient on the whole dataset.

2. Stochastic Gradient Descent

- Stochastic Gradient Descent is an extension of Gradient Descent, where it overcomes some of the disadvantages of Gradient Descent algorithm.
- SGD tries to overcome the disadvantage of computationally intensive by computing the derivative of one point at a time.
- Due to this fact, SGD takes more number of iterations compared to GD to reach minimum and also contains some noise when compared to Gradient Descent.
- As SGD computes derivatives of only 1 point at a time, the time taken to complete one epoch is large compared to Gradient Descent algorithm.

Point :

1. Use One (Randomly Picked) Sample for a forward pass and adjust the weights.
2. Good when training set is very big and we dont want too much computation.

cost function $\theta=\theta-\alpha\cdot\nabla J(\theta;x(i);y(i))$, where {x(i) ,y(i)} are the training examples.

Advantages:

- Frequent updates of model parameters hence, converges in less time.
- Requires less memory as no need to store values of loss functions.
- May get new minima’s.

Disadvantages:

- High variance(noisey) in model parameters.

- May shoot even after achieving global minima.
- To get the same convergence as gradient descent needs to slowly reduce the value of learning rate.

3. Mini Batch — Stochastic Gradient Descent

- MB-SGD is an extension of SGD algorithm. It overcomes the time-consuming complexity of SGD by taking a batch of points / subset of points from dataset to compute derivative.
- It's best among all the variations of gradient descent algorithms. It is an improvement on both SGD and standard gradient descent. It updates the model parameters after every batch. So, the dataset is divided into various batches and after every batch, the parameters are updated.
- This is a mixture of both stochastic and batch gradient descent.
- The training set is divided into multiple groups called batches. Each batch has a number of training samples in it.
- At a time a single batch is passed through the network which computes the loss of every sample in the batch and uses their average to update the parameters of the neural network.
- For example, say the training set has 100 training examples which is divided into 5 batches with each batch containing 20 training examples. This means that the equation in figure2 will be iterated over 5 times (number of batches).

Point:

1. Use a Batch Of (Randomly Picked) Sample for a forward pass and adjust the weights.
2. It is observed that the derivative of loss function of MB-SGD is similar to the loss function of GD after some iterations. But the number iterations to achieve minima in MB-SGD is large compared to GD and is computationally expensive. The update of weights in much noisier because the derivative is not always towards minima.

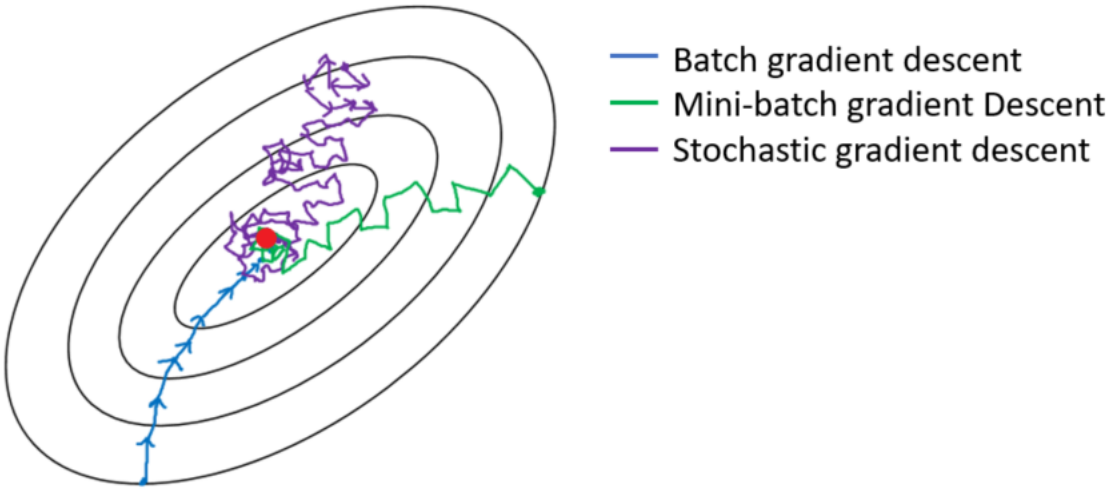
$\theta=\theta-\alpha\cdot\nabla J(\theta; B(i))$, where $\{B(i)\}$ are the batches of training examples.

Advantages:

- Frequently updates the model parameters and also has less variance.
- Requires medium amount of memory.
- Easily fits in the memory
- It is computationally efficient
- Benefit from vectorization
- If stuck in local minimums, some noisy steps can lead the way out of them
- Average of the training samples produces stable error gradients and convergence

!!!! This ensures the following advantages of both stochastic and batch gradient descent are used due to which Mini Batch Gradient Descent is most commonly used in practice.

See How in this above three convergence Occure towards minima point



Here We see in SGD Due to frequent updates the steps taken towards the minima are very noisy. This can often lead the gradient descent into other directions. Also, due to noisy steps it may take longer to achieve convergence to the minima of the loss function. to reduce this we can use SGD with Momentum.

4. SGD with Momentum

- Momentum was invented for reducing high variance in SGD and softens the convergence.
- It accelerates the convergence towards the relevant direction and reduces the fluctuation to the irrelevant direction. One more hyperparameter is used in this method known as momentum symbolized by 'γ'(gamma).
- It is an adaptive optimization algorithm which exponentially uses weighted average gradients over previous iterations to stabilize the convergence, resulting in quicker optimization.
- This is done by adding a fraction (gamma) to the previous iteration values.
- Essentially the momentum term increase when the gradient points are in the same directions and reduce when gradients fluctuate. As a result, the value of loss function converges faster than expected.

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

$$w_t = w_{t-1} - v_t$$

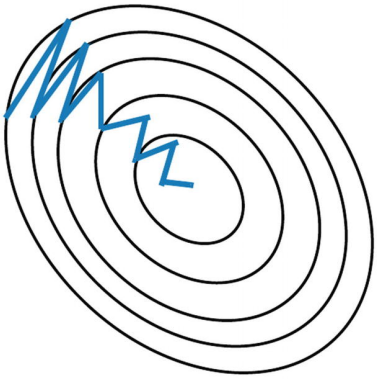
Advantages:

- Reduces the oscillations and high variance of the parameters.
- Converges faster than gradient descent.

Disadvantages:

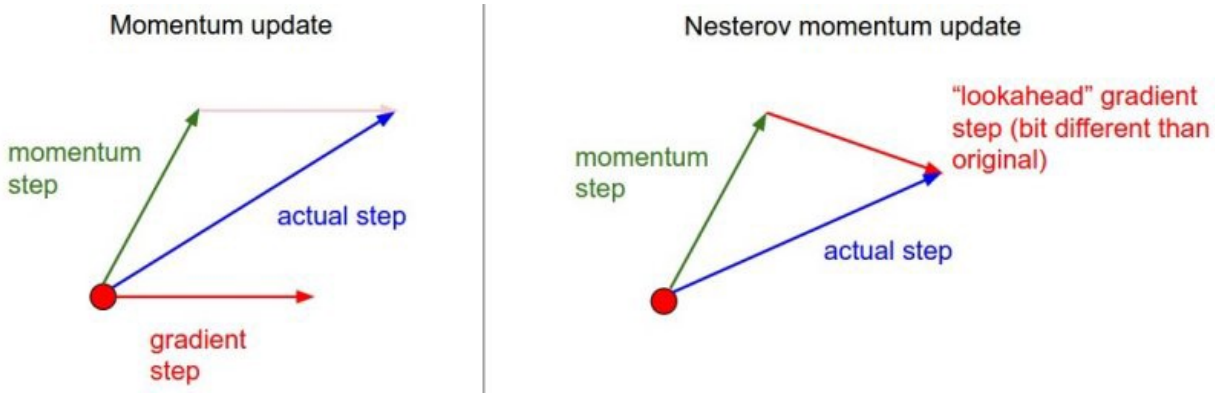
- One more hyper-parameter is added which needs to be selected manually and accurately.

Stochastic Gradient Descent **withhout** Momentum



Stochastic Gradient Descent **with** Momentum

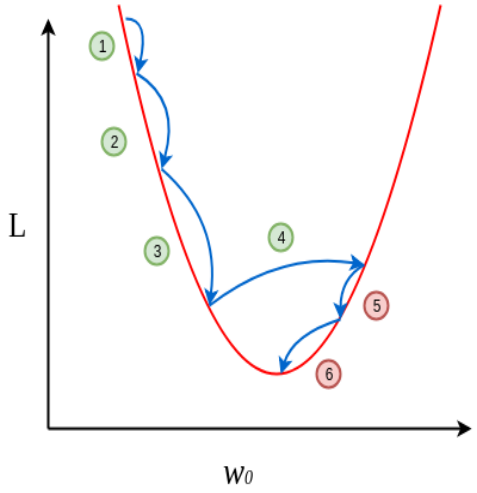
5. Nesterov accelerated gradient(NAG)



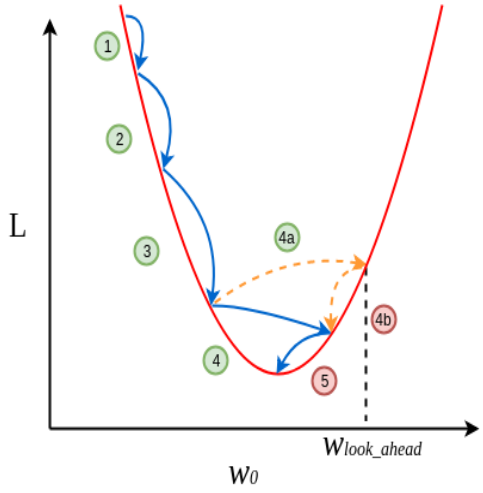
- Momentum may be a good method but if the momentum is too high the algorithm may miss the local minima and may continue to rise up. So, to resolve this issue the NAG algorithm was developed.
- Nesterov accelerated gradient (NAG) is a way to give momentum more precision.
- The idea of the NAG algorithm is very similar to SGD with momentum with a slight variant. In the case of SGD with momentum algorithm, the momentum and gradient are computed on previous updated weight.
- Both NAG and SGD with momentum algorithms work equally well and share the same advantages and disadvantages.

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

$$w_t = w_{t-1} - v_t$$



(a) Momentum-Based Gradient Descent



(b) Nesterov Accelerated Gradient Descent

$$\bigcirc \Longrightarrow \frac{\partial L}{\partial w_0} = \frac{Negative(-)}{Positive(+)} \qquad \bigcirc \Longrightarrow \frac{\partial L}{\partial w_0} = \frac{Negative(-)}{Negative(-)}$$

figure (a) :

- In figure (a), update 1 is positive i.e., the gradient is negative because as w_0 increases L decreases. Even update 2 is positive as well and you can see that the update is slightly larger than update 1 because of momentum.
- By now, you should be convinced that update 3 will be bigger than both update 1 and 2 simply because of momentum and the positive update history.
- Update 4 is where things get interesting. In SGD with Momentum case, due to the positive history, the update overshoots and the descent recovers by doing negative updates.

figure (b) :

- But in NAG's case, every update happens in two steps — first, a partial update, where we get to the look_ahead point and then the final update (see the NAG update rule), see figure (b).
- First 3 updates of NAG are pretty similar to the momentum-based method as both the updates (partial and final) are positive in those cases. But the real difference becomes apparent during update 4.
- As usual, each update happens in two stages, the partial update (4a) is positive, but the final update (4b) would be negative as the calculated gradient at w_lookahead would be negative (convince yourself by observing the graph).
- This negative final update slightly reduces the overall magnitude of the update, still resulting in an overshoot but a smaller one when compared to the vanilla momentum-based gradient descent. And that my friend, is how NAG helps us in reducing the overshoots, i.e. making us take shorter U-turns.

Advantages:

- Does not miss the local minima.
- Slows if minima's are occurring.

Disadvantages:

- Still, the hyperparameter needs to be selected manually.

Point :

- By using NAG technique, we are now able to adapt error function with the help of previous and future values and thus eventually speed up the convergence. Now, in the next techniques we will try to adapt alter or vary the individual parameters depending on the importance factor it plays in each case.

6. Adaptive Gradient (AdaGrad)

- Adaptive Gradient as the name suggests adopts the learning rate of parameters by updating it at each iteration depending on the position it is present, i.e- by adapting slower learning rates when features are occurring frequently and adapting higher learning rate when features are infrequent.
- The motivation behind Adagrad is to have different learning rates for each neuron of each hidden layer for each iteration.

But why do we need different learning rates?

Data sets have two types of features:

- Dense features, e.g. House Price Data set (Large number of non-zero valued features), where we should perform smaller updates on such features; and
- Sparse Features, e.g. Bag of words (Large number of zero valued features), where we should perform larger updates on such features.

It has been found that Adagrad greatly improved the robustness of SGD, and is used for training large-scale neural nets at Google.

$$\text{For Gradient Descent: } w_t = w_{t-1} - \eta \nabla J(w)$$

$$\text{For ADA Grad : } w_t = w_{t-1} - \eta'_t \nabla J(w)$$

$$\eta'_t = \frac{\eta}{\sqrt{(\alpha_t + \varepsilon)}} \qquad \alpha_t = \sum_{i=1}^{t-1} (\nabla J(w))^2$$

η : initial Learning rate

ϵ : smoothing term that avoids division by zero

w: Weight of parameters

- In SGD learning Rate same for all weight but in Adagrad this is different for all.

Advantage:

- No need to update the learning rate manually as it changes adaptively with iterations.
- If we have some Sparse and Dense feature it automatically takes out what learning rate is suitable.

Disadvantage:

- As the number of iteration becomes very large learning rate decreases to a very small number which leads to slow convergence.
- Computationally expensive as a need to calculate the second order derivative.

Adadelta, RMSProp, and adam tries to resolve Adagrad's radically diminishing learning rates.

7. AdaDelta

- It is simply an extension of AdaGrad that seeks to reduce its monotonically decreasing learning rate.
- Instead of summing all the past gradients, AdaDelta restricts the no. of summation values to a limit (w).
- In AdaDelta, the sum of past gradients (w) is defined as "Decaying Average of all past squared gradients". The current average at the iteration then depends only on the previous average and current gradient.

$$\text{For Gradient Descent: } w_t = w_{t-1} - \eta \nabla J(w)$$

$$\text{For ADA Grad : } w_t = w_{t-1} - \eta'_t \nabla J(w)$$

$$v_t = \gamma v_{t-1} + (1-\gamma)\nabla J(w_{t-1})^2$$

$$\eta'_t = \frac{\eta_{t-1}}{\sqrt{(V_t + \varepsilon)}}$$

- Instead of inefficiently storing all previous squared gradients, we recursively define a decaying average of all past squared gradients. The running average at each time step then depends (as a fraction γ , similarly to the Momentum term) only on the previous average and the current gradient.

Advantages:

Now the learning rate does not decay and the training does not stop.

Disadvantages:

Computationally expensive.

8. RMSProp

- RMSProp is Root Mean Square Propagation. It was devised by Geoffrey Hinton.
- RMSProp tries to resolve Adagrad's radically diminishing learning rates by using a moving average of the squared gradient. It utilizes the magnitude of the recent gradient descents to normalize the gradient.
- In RMSProp learning rate gets adjusted automatically and it chooses a different learning rate for each parameter.
- RMSProp divides the learning rate by the average of the exponential decay of squared gradients
- Its cost function same as Adadelta

9. Adam — Adaptive Moment Estimation

- It is a combination of RMSProp and Momentum.
- This method computes adaptive learning rate for each parameter.
- In addition to storing the previous decaying average of squared gradients, it also holds the average of past gradient similar to Momentum. Thus, Adam behaves like a heavy ball with friction which prefers flat minima in error surface.
- Another method that calculates the individual adaptive learning rate for each parameter from estimates of first (Momentum) and second (RMSProp) moments of the gradients.

Momentum

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

$$w_t = w_{t-1} - v_t$$

RMS Prop

$$v_t = \gamma v_{t-1} + (1-\gamma) \nabla J(w_{t-1})^2$$

$$\eta'_t = \frac{\eta_{t-1}}{\sqrt{(V_t + \varepsilon)}}$$

ADAM

Momentum Component ⟶ $m_t = \beta_1 m_{t-1} + (1-\beta_1) \nabla J(w)$

RMS Prop Component ⟶ $v_t = \beta_2 v_{t-1} + (1-\beta_2) \nabla J(w)^2$

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{v_t + \varepsilon}} * m_t$$

Advantages:

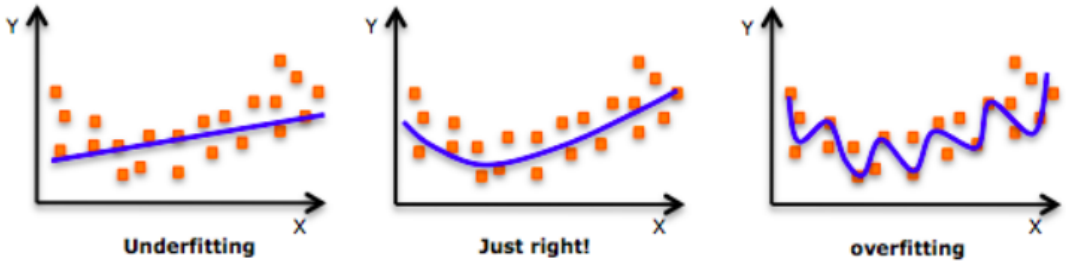
- The method is too fast and converges rapidly.
- Rectifies vanishing learning rate, high variance.

Disadvantages:

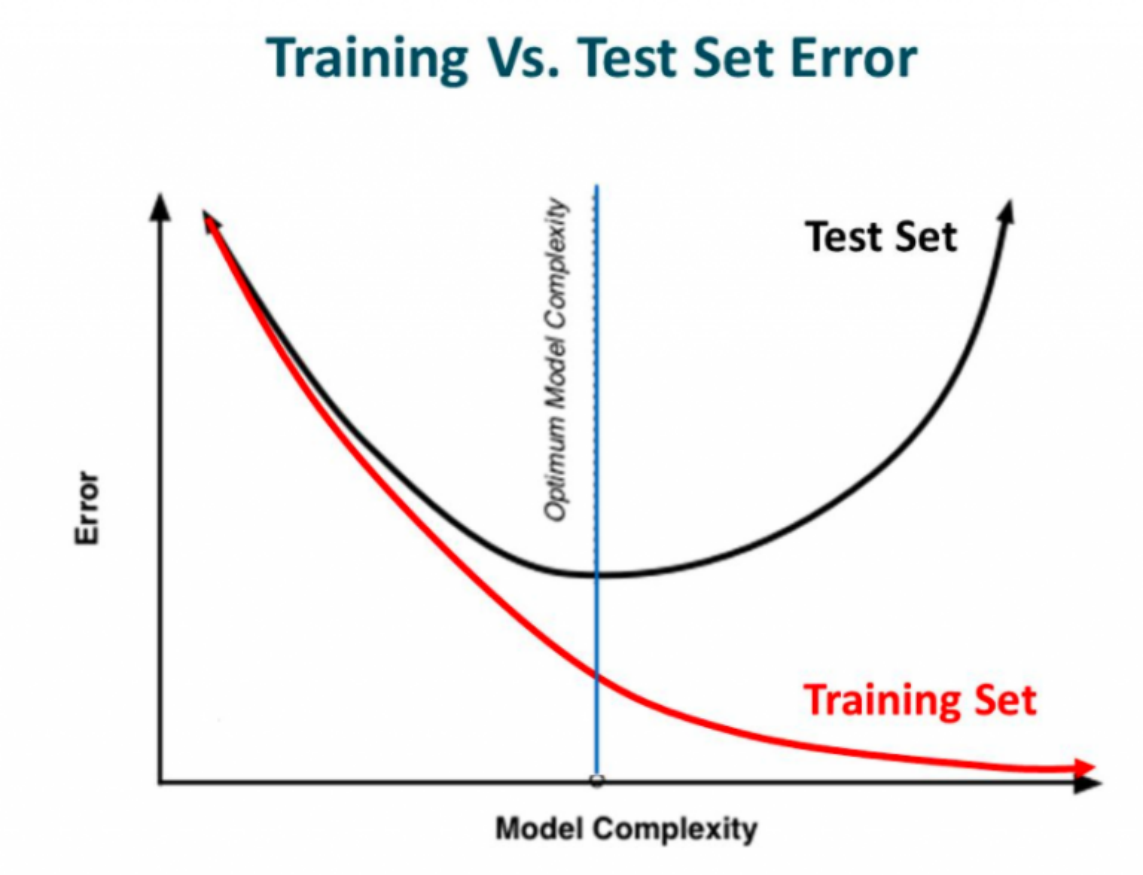
- Computationally costly.

Regularization Techniques

- One of the most common problem data science professionals face is to avoid overfitting. Have you come across a situation where your model performed exceptionally well on train data, but was not able to predict test data.



- Have you seen this image before? As we move towards the right in this image, our model tries to learn too well the details and the noise from the training data, which ultimately results in poor performance on the unseen data.
- In other words, while going towards the right, the complexity of the model increases such that the training error reduces but the testing error doesn't. This is shown in the image below.



- If you've built a neural network before, you know how complex they are. This makes them more prone to overfitting. Regularization is a technique which makes slight modifications to the learning algorithm such that the model generalizes better. This in turn improves the model's performance on the unseen data as well.

.

Different Regularization Techniques in Deep Learning

1. L1 & L2 regularization

- L1 and L2 are the most common types of regularization. These update the general cost function by adding another term known as the regularization term.

```
                                Cost function = Loss (say, binary cross entrop
    y) + Regularization term
```

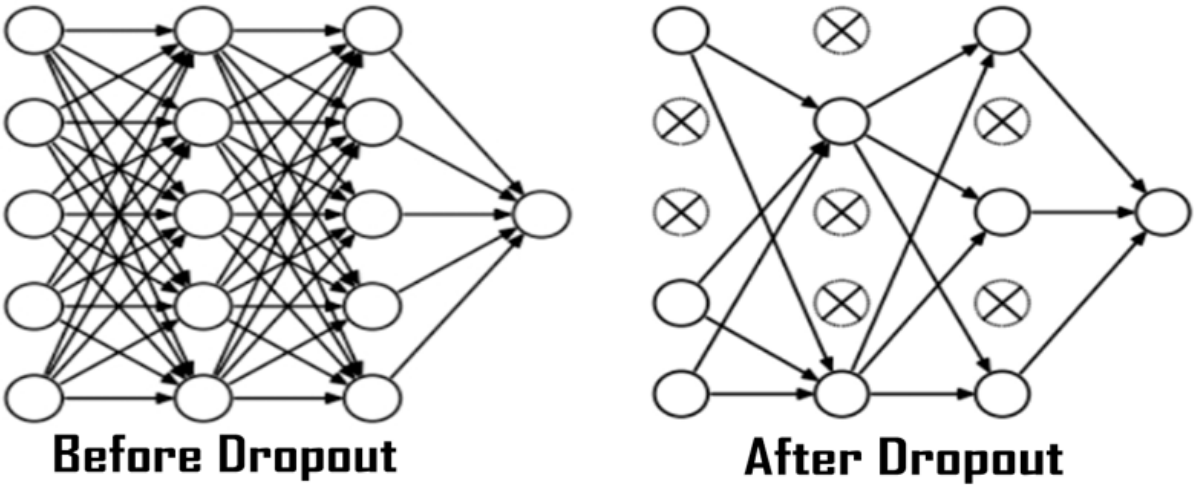
- Due to the addition of this regularization term, the values of weight matrices decrease because it assumes that a neural network with smaller weight matrices leads to simpler models. Therefore, it will also reduce overfitting to quite an extent.

```
## Below is the sample code to apply L2 regularization to a Dense layer.

from keras import regularizers
model.add(Dense(64, input_dim=64,
                kernel_regularizer=regularizers.l2(0.01)


## Below is the sample code to apply L1 regularization to a Dense layer.

from keras import regularizers
model.add(Dense(64, input_dim=64,
                kernel_regularizer=regularizers.l1(0.01)
```

Note: Here the value 0.01 is the value of regularization parameter, i.e., lambda, which we need to optimize further. We can optimize it using the hyper parameter tuning method.

2. Dropout

- This is the one of the most interesting types of regularization techniques. It also produces very good results and is consequently the most frequently used regularization technique in the field of deep learning.
- To understand dropout, let’s say our neural network structure is akin to the one shown below:



- So what does dropout do? At every iteration, it randomly selects some nodes and removes them along with all of their incoming and outgoing connections as shown below. So each iteration has a different set of nodes and this results in a different set of outputs. It can also be thought of as an ensemble technique in machine learning.

Point:-

- dropout is usually preferred when we have a large neural network structure in order to introduce more randomness.

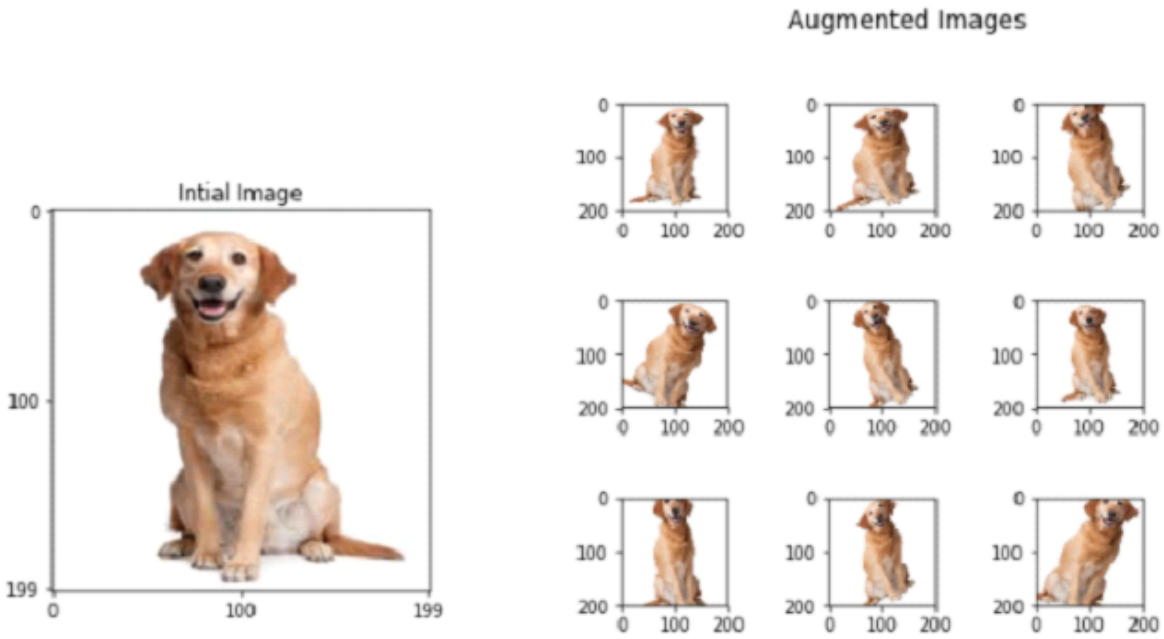
```
## In keras, we can implement dropout using the keras core layer. Below is the
python code for it:

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'),
 ])
```

Note : As you can see, we have defined 0.25 as the probability of dropping. We can tune it further for better results using the Hyper parameter tuning method.

3. Data Augmentation (For Image Data)

- The simplest way to reduce overfitting is to increase the size of the training data. In machine learning, we were not able to increase the size of training data as the labeled data was too costly.
- But, now let's consider we are dealing with images. In this case, there are a few ways of increasing the size of the training data – rotating the image, flipping, scaling, shifting, etc. In the below image, some transformation has been done on the handwritten digits dataset.

- This technique is known as data augmentation. This usually provides a big leap in improving the accuracy of the model. It can be considered as a mandatory trick in order to improve our predictions.
- In keras, we can perform all of these transformations using ImageDataGenerator. It has a big list of arguments which you you can use to pre-process your training data.



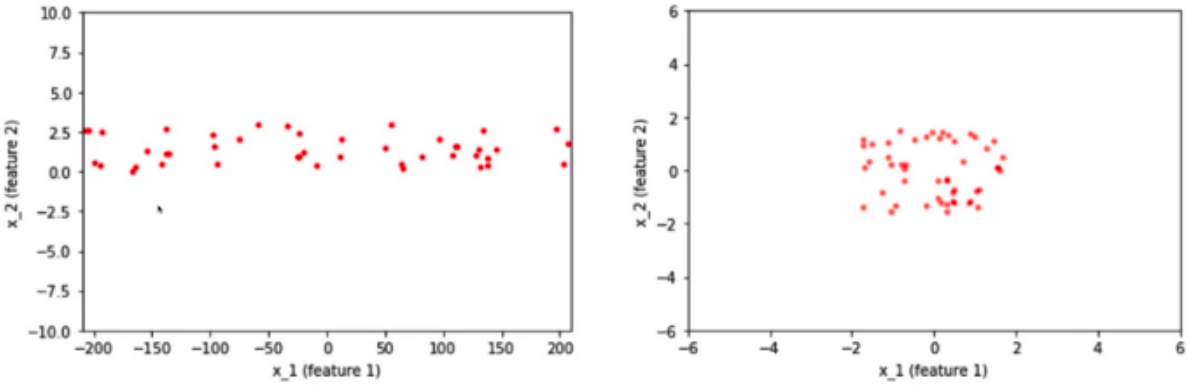

```
## Below is the sample code to implement it.
from keras.preprocessing.image import ImageDataGenerator
datagen = ImageDataGenerator(horizontal flip=True)
datagen.fit(train)
```

4. Batch Normalization

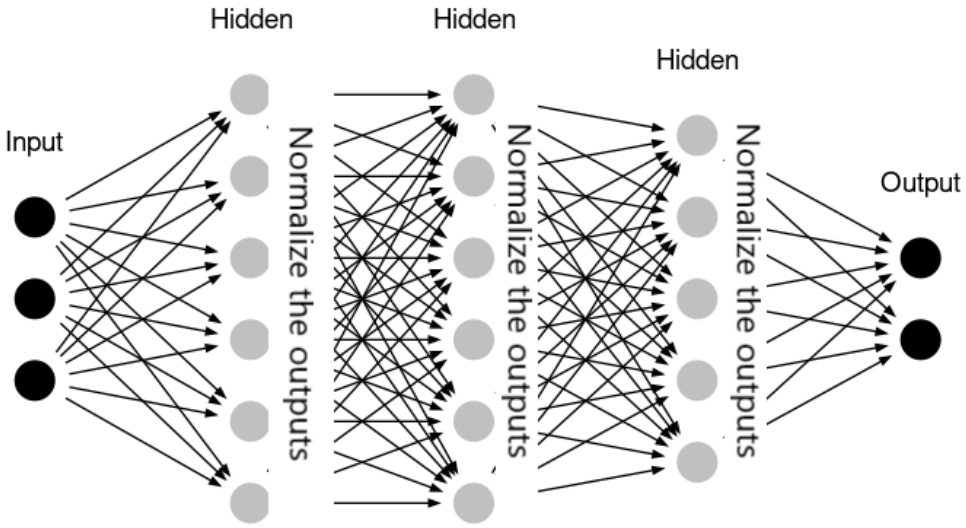
- Batch normalization is a technique for improving the speed, performance, and stability of artificial neural networks, also known as batch norm. The idea is to normalize the inputs of each layer in such a way that, they have a mean activation output zero and a unit standard deviation.
- The reason for the ‘batch’ in the term Batch Normalization is because neural networks are usually trained with a collated set of data at a time, this set or group of data is referred to as a batch. The operation within the BN technique occurs to an entire batch of input values as opposed to a single input value.

Why should we normalize the input?

- Let say we have 2D data, X1, and X2. X1 feature has a very wider spread between 200 to -200 whereas the X2 feature has a very narrow spread. The left graph shows the variance of the data which has different ranges. The right graph shows data lies between -2 to 2 and it’s normally distributed with 0 mean and unit variance.



- Essentially, scaling the inputs through normalization gives the error surface a more spherical shape, where it would otherwise be a very high curvature ellipse. Having an error surface with high curvature will mean that we take many steps that aren't necessarily in the optimal direction.
- When we scale the inputs, we reduce the curvature, which makes methods that ignore curvature like gradient descent work much better. When the error surface is circular or spherical, the gradient points right at the minimum.



Benefits of Batch Normalization

- Inclusion of Batch Normalization technique in deep neural networks improves training time
- BN enables the utilization of larger learning rates, this shortness the time of convergence when training neural networks
- Reduces the common problem of vanishing gradients
- Covariate shift within neural network is reduced

Point: In Batch normalization just as we standardize the inputs, the same way we standardize the activation at all the layers so that, at each layer we have 0 mean and unit standard deviation.

```
## In keras, we can implement BatchNormalization using the keras layer. Below
is the python code for it:
model = Sequential([
 Dense(output_dim=hidden1_num_units, input_dim=input_num_units,
activation='relu'),
 keras.layers.BatchNormalization(),

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

In [1]:

```
import pandas as pd
from matplotlib import pyplot as plt
import numpy as np
%matplotlib inline

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import StandardScaler

from sklearn.metrics import mean_absolute_error,mean_squared_error
from sklearn.metrics import confusion_matrix , classification_report,accuracy_score


import tensorflow as tf
from tensorflow import keras
from keras.models import Sequential
from keras.layers import Dense
```

Binary Classifier

In [2]:

```
df = pd.read_csv("kaggle_diabetes.csv")
df.head()
```

Out[2]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunctio
0	2	138	62	35	0	33.6	0.12
1	0	84	82	31	125	38.2	0.23
2	0	145	0	0	0	44.2	0.63
3	0	135	68	42	250	42.3	0.36
4	1	139	62	41	480	40.7	0.53

In [3]:

```
df.shape
```

Out[3]:

```
(2000, 9)
```

In [4]:

```
X = df.drop('Outcome',axis=1)
y = df['Outcome']

X_train, X_test, y_train, y_test = train_test_split(X,y,test_size=0.2,random_state=5)
```

Apply ANN

In [5]:

```
classifier = Sequential()
##input 1st layer
classifier.add(Dense(16,activation='relu',input_dim=8))


## second hidden layer
classifier.add(Dense(8,activation='relu'))


## output layer
classifier.add(Dense(1,activation='sigmoid'))
```

In [6]:

```
classifier.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

In [7]:

```
classifier.fit(X_train, y_train, epochs=150)
```

```
Epoch 1/150
50/50 [==============================] - 1s 1ms/step - loss: 6.1302 - accu
racy: 0.4691
Epoch 2/150
50/50 [==============================] - 0s 1ms/step - loss: 1.5064 - accu
racy: 0.6145
Epoch 3/150
50/50 [==============================] - 0s 1ms/step - loss: 1.1917 - accu
racy: 0.6264
Epoch 4/150
50/50 [==============================] - 0s 1ms/step - loss: 0.9443 - accu
racy: 0.6452
Epoch 5/150
50/50 [==============================] - 0s 1ms/step - loss: 0.8631 - accu
racy: 0.6444
Epoch 6/150
50/50 [==============================] - 0s 2ms/step - loss: 0.8198 - accu
racy: 0.6555
Epoch 7/150
```

In [8]:

```
classifier.evaluate(X_test, y_test)
```

```
13/13 [==============================] - 0s 1ms/step - loss: 0.4997 - accura
cy: 0.7775
```

Out[8]:

```
[0.4997430741786957, 0.7774999737739563]
```

In [9]:

```
y_pred = classifier.predict(X_test)
```

In [10]:

```
yp = classifier.predict(X_test)
yp[:5]
```

Out[10]:

```
array([[0.1072953 ],
       [0.03356129],
       [0.20200807],
       [0.16860384],
       [0.50361   ]], dtype=float32)
```

In [11]:

```
y_pred = []
for element in yp:
    if element > 0.5:
        y_pred.append(1)
    else:
        y_pred.append(0)
```

In [12]:

```
print(classification_report(y_test,y_pred))
```

```
              precision    recall  f1-score   support

           0       0.79      0.92      0.85       269
           1       0.74      0.49      0.59       131

    accuracy                           0.78       400
   macro avg       0.77      0.70      0.72       400
weighted avg       0.77      0.78      0.76       400
```

Multi Classification Using ANN

In [13]:

```
data = pd.read_csv('https://gist.githubusercontent.com/curran/a08a1080b88344b0c8a7/raw/0e7a
data.shape
```

Out[13]:

```
(150, 5)
```

In [14]:

```
data.head()
```

Out[14]:

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

Split in to X and y

In [15]:

```
X = data.drop('species',axis=1)
y = data['species']
```

Encoding target variable Using Label or Dummy

imp1:- If you use dummy then in loss function you use categorical_crossentropy

imp2:- if you use label_encoding then in loss function you use sparse_categorical_crossentropy

If your targets are **one-hot encoded**, use `categorical_crossentropy`.

- Examples of one-hot encodings:
  - `[1,0,0]`
  - `[0,1,0]`
  - `[0,0,1]`

But if your targets are **integers**, use `sparse_categorical_crossentropy`.

- Examples of integer encodings (*for the sake of completion*):
  - `1`
  - `2`
  - `3`

In [16]:

```
## label

lb = LabelEncoder()
y_enc = lb.fit_transform(y)
y_enc
```

Out[16]:

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

In [17]:

```
# dummy
# y_dummy = pd.get_dummies(y).values
```

In [18]:

```
X_train,X_test,y_train,y_test = train_test_split(X,y_enc,test_size=0.25,random_state=4)
```

In [19]:

```
sc = StandardScaler()
X_train_scaled = sc.fit_transform(X_train)
X_test_scaled = sc.transform(X_test)
```

In [20]:

```
X.shape
```

Out[20]:

```
(150, 4)
```

Apply ANN

In [21]:

```
classifier = Sequential()
classifier.add(Dense(10,input_dim = 4,activation = "relu"))

classifier.add(Dense(3,activation = "softmax"))
```

In [22]:

```
## if target dummy encoding use categorical_crossentropy if use label encoding use sparse_c
classifier.compile(optimizer = 'adam' , loss = 'sparse_categorical_crossentropy',
                   metrics = ['accuracy'] )
```

In [23]:

```
classifier.fit(X_train_scaled , y_train ,epochs = 100)
```

```
Epoch 1/100
4/4 [==============================] - 0s 2ms/step - loss: 0.8949 - accura
cy: 0.6033
Epoch 2/100
4/4 [==============================] - 0s 2ms/step - loss: 0.9165 - accura
cy: 0.5705
Epoch 3/100
4/4 [==============================] - 0s 2ms/step - loss: 0.8442 - accura
cy: 0.6268
Epoch 4/100
4/4 [==============================] - 0s 4ms/step - loss: 0.8412 - accura
cy: 0.6121
Epoch 5/100
4/4 [==============================] - 0s 4ms/step - loss: 0.8268 - accura
cy: 0.5933
Epoch 6/100
4/4 [==============================] - 0s 3ms/step - loss: 0.8125 - accura
cy: 0.6260
Epoch 7/100
```

In [24]:

```
y_pred = classifier.predict(X_test_scaled)
# y_test= np.argmax(y_test,axis=1) # when use dummy encoding in target
y_pred = np.argmax(y_pred,axis=1)
```

In [25]:

```
print(classification_report(y_test,y_pred))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        18
           1       1.00      0.88      0.93         8
           2       0.92      1.00      0.96        12

    accuracy                           0.97        38
   macro avg       0.97      0.96      0.96        38
weighted avg       0.98      0.97      0.97        38
```

In [ ]:

```

```

In [ ]:

```

```

Regression

In [26]:

```
df=pd.read_csv('https://raw.githubusercontent.com/krishnaik06/Keras-Tuner/main/Real_Combine
```

In [27]:

```
df.head()
```

Out[27]:

	T	TM	Tm	SLP	H	VV	V	VM	PM 2.5
0	7.4	9.8	4.8	1017.6	93.0	0.5	4.3	9.4	219.720833
1	7.8	12.7	4.4	1018.5	87.0	0.6	4.4	11.1	182.187500
2	6.7	13.4	2.4	1019.4	82.0	0.6	4.8	11.1	154.037500
3	8.6	15.5	3.3	1018.7	72.0	0.8	8.1	20.6	223.208333
4	12.4	20.9	4.4	1017.3	61.0	1.3	8.7	22.2	200.645833

In [28]:

```
df.dropna(inplace=True)
```

In [29]:

```
X=df.drop('PM 2.5',axis=1) ## independent features
y=df['PM 2.5'] ## dependent features
```

In [30]:

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=0)
```

In [31]:

```
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)
```

In [32]:

```
X.shape
```

Out[32]:

```
(1092, 8)
```

Apply ANN

In [33]:

```
classifier = Sequential()
classifier.add(Dense(10,input_dim = 8,activation = "relu"))

## second hidden layer
classifier.add(Dense(8,activation='relu'))

classifier.add(Dense(1,activation = "linear"))
```

In [34]:

```
## if target dummy encoding use categorical_crossentropy if use label encoding use sparse_c
classifier.compile(optimizer = 'adam' , loss = 'mse',
                   metrics = ['mse'] )
```

In [35]:

```
classifier.fit(X_train , y_train ,epochs = 100)
```

```
Epoch 1/100
24/24 [==============================] - 1s 2ms/step - loss: 18824.7756 -
mse: 18824.7756
Epoch 2/100
24/24 [==============================] - 0s 2ms/step - loss: 20844.0150 -
mse: 20844.0150
Epoch 3/100
24/24 [==============================] - 0s 2ms/step - loss: 21034.4066 -
mse: 21034.4066
Epoch 4/100
24/24 [==============================] - 0s 2ms/step - loss: 21785.1067 -
mse: 21785.1067
Epoch 5/100
24/24 [==============================] - 0s 2ms/step - loss: 16766.3743 -
mse: 16766.3743
Epoch 6/100
24/24 [==============================] - 0s 2ms/step - loss: 19805.6353 -
mse: 19805.6353
Epoch 7/100
```

In [36]:

```
classifier.evaluate(X_test, y_test)
```

```
11/11 [==============================] - 0s 2ms/step - loss: 3349.8369 - ms
e: 3349.8369
```

Out[36]:

```
[3349.8369140625, 3349.8369140625]
```

In [37]:

```
y_pred = classifier.predict(X_test)
```

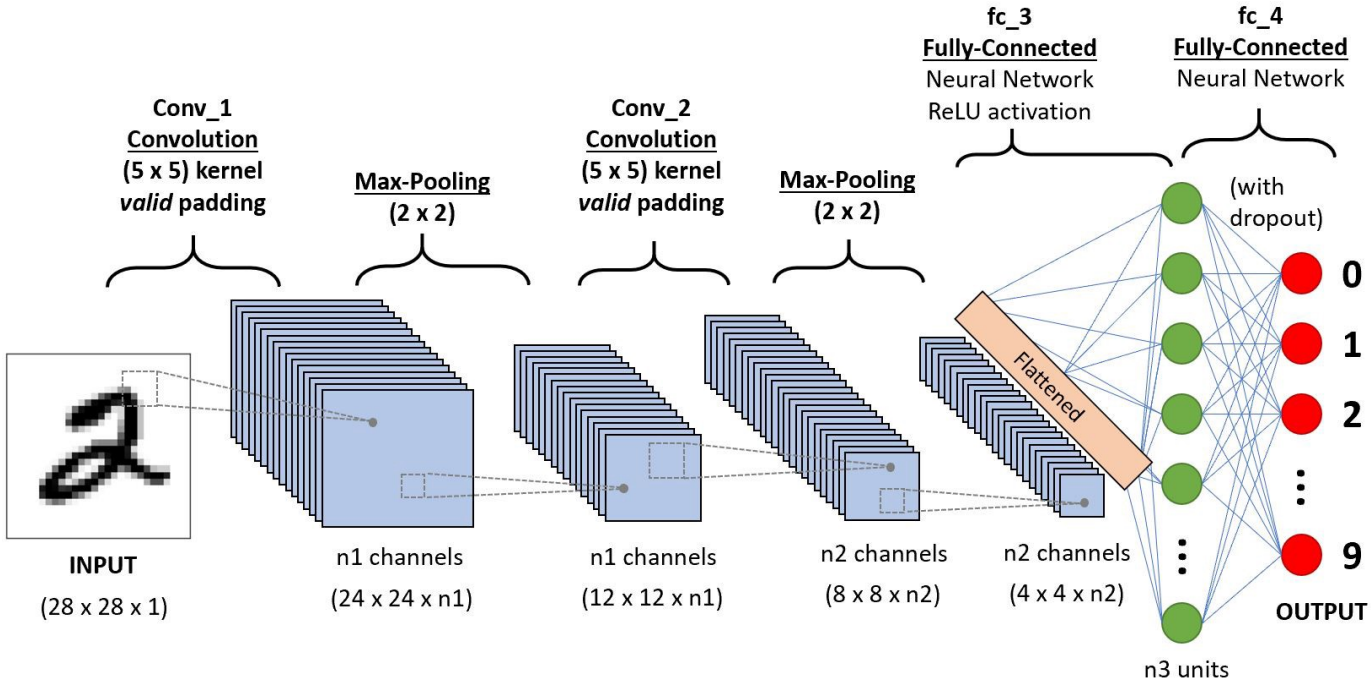
In [38]:

```
print("MAE",mean_absolute_error(y_test,y_pred))
print("MSE",mean_squared_error(y_test,y_pred))
print("RMSE",mean_squared_error(y_test,y_pred,squared=False))
```

```
MAE 40.52856917885261
MSE 3349.837197332338
RMSE 57.87777809602177
```

Convolutional neural networks (CNN)

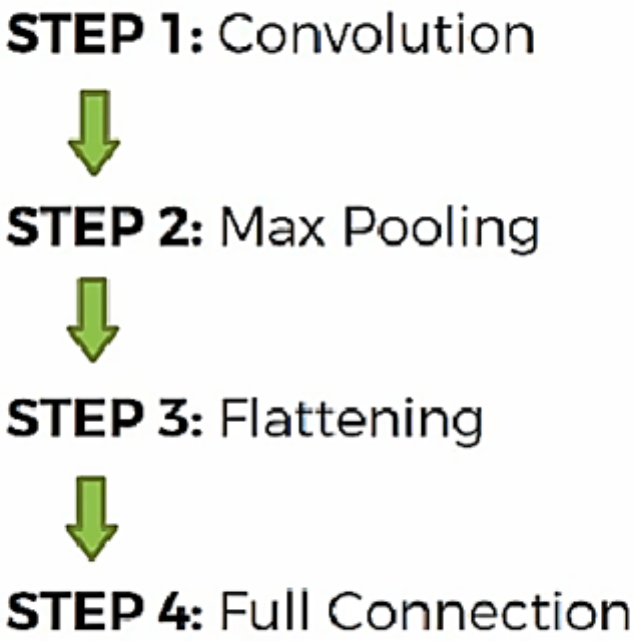
- Cnn are one of the most popular models used today. This neural network computational model uses a variation of multilayer perceptrons and contains one or more convolutional layers that can be either entirely connected or pooled.
- These convolutional layers create feature maps that record a region of image which is ultimately broken into rectangles and sent out for nonlinear processing.



- Let us suppose this in the input matrix of 5×5 and a filter of matrix 3X3, for those who don't know what a filter is a set of weights in a matrix applied on an image or a matrix to obtain the required features, please search on convolution if this is your first time!

Note: We always take the sum or average of all the values while doing a convolution.

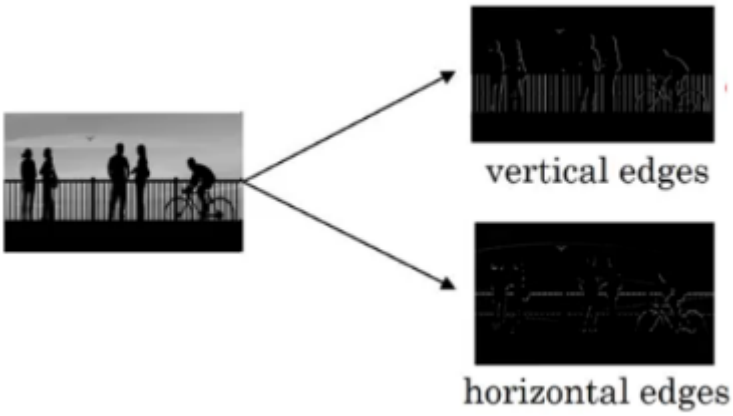
Steps Involve in CNN

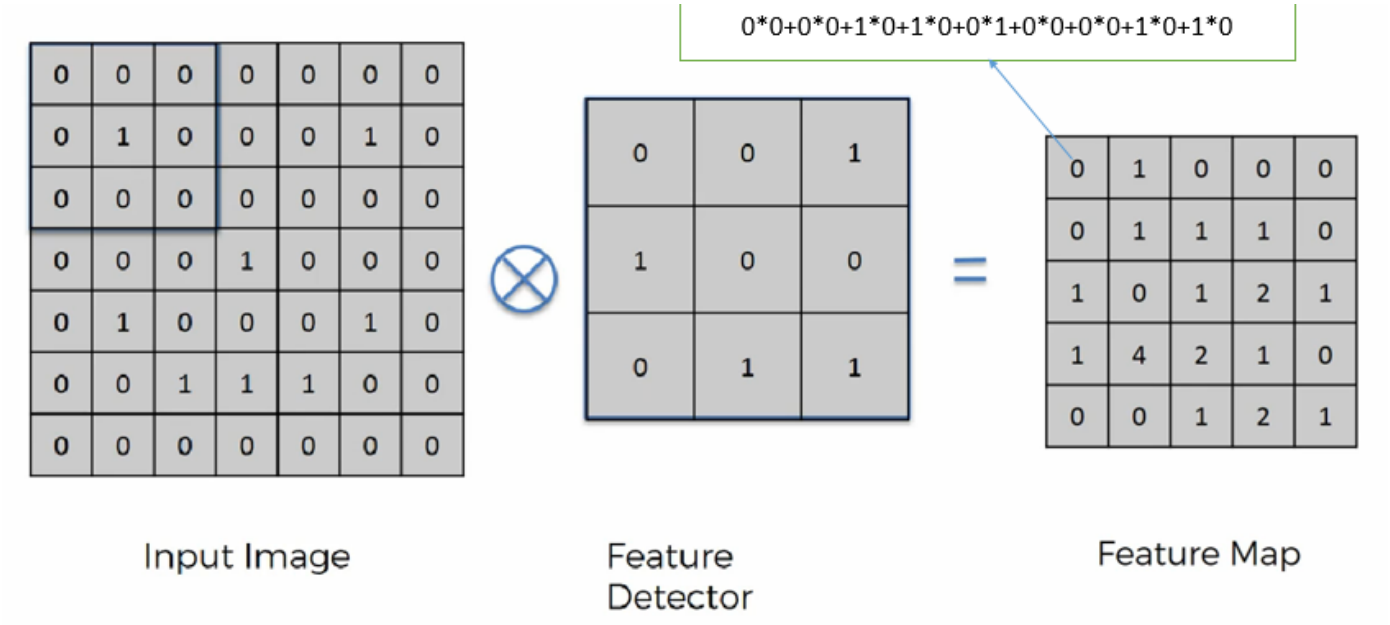


1. Edge Detection (Convolution)

- In the previous article, we saw that the early layers of a neural network detect edges from an image. Deeper layers might be able to detect the cause of the objects and even more deeper layers might detect the cause of complete objects (like a person's face).

In this section, we will focus on how the edges can be detected from an image. Suppose we are given the below image: As you can see, there are many vertical and horizontal edges in the image. The first thing to do is to detect these edges:





- So, we take the first 3 X 3 matrix from the 7 X 7 image and multiply it with the filter. Now, the first element of the (n-k+1 x n-k+1) i.e (7-3+1 X 7-3+1) 5 X 5 output will be the sum of the element-wise product of these values, i.e. 0*0*+0*0*+1*0*+*1*0+0*1*+*0*0+0*0*+*1*0+1*0 =0. To calculate the second element of the 5 X 5 output, we will shift our filter one step towards the right and again get the sum of the element-wise product:

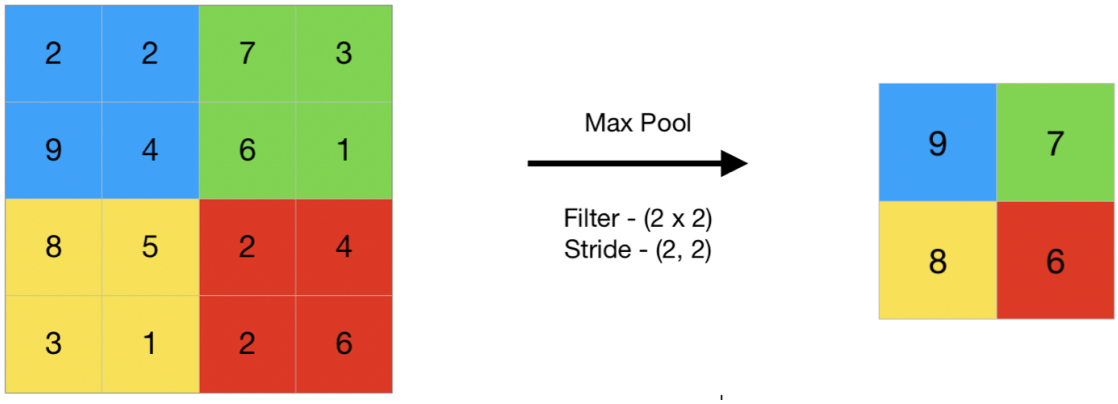
2. Pooling

- A pooling layer is another building block of a CNN. Its function is to progressively reduce the spatial size of the representation to reduce the amount of parameters and computation in the network. Pooling layer operates on each feature map independently. The most common approach used in pooling is max pooling.

Types of Pooling Layers :-

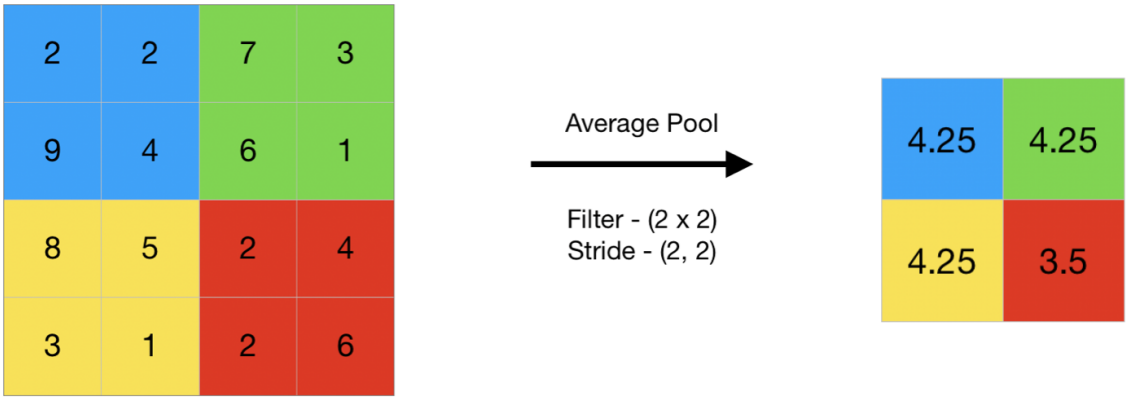
`1. Max Pooling`

Max pooling is a pooling operation that selects the maximum element from the region of the feature map covered by the filter. Thus, the output after max-pooling layer would be a feature map containing the most prominent features of the previous feature map.



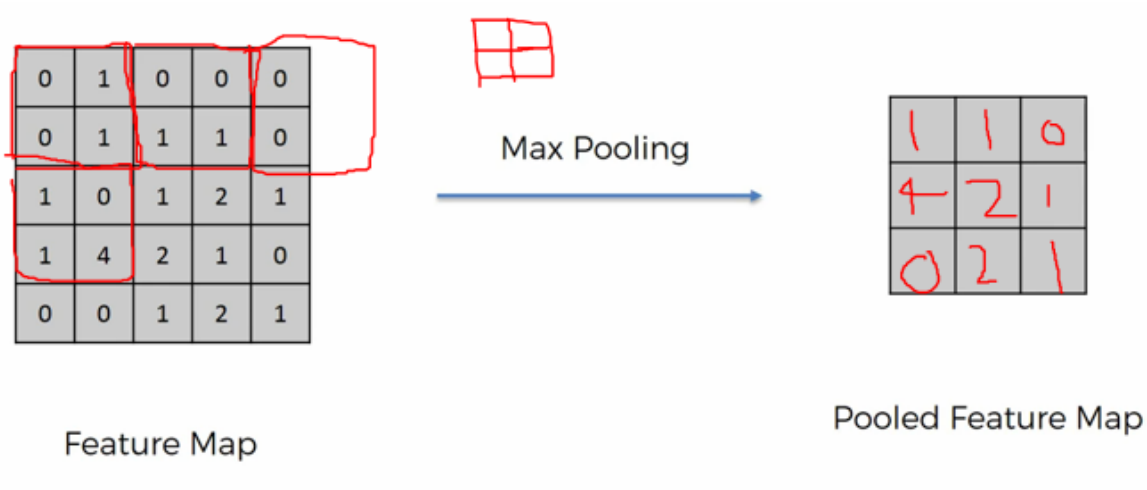
`2. Average Pooling`

Average pooling computes the average of the elements present in the region of feature map covered by the filter. Thus, while max pooling gives the most prominent feature in a particular patch of the feature map, average pooling gives the average of features present in a patch.



- More On Pooling https://www.geeksforgeeks.org/cnn-introduction-to-pooling-layer/ (https://www.geeksforgeeks.org/cnn-introduction-to-pooling-layer/)

Now Apply Pooling in our above Feature Map

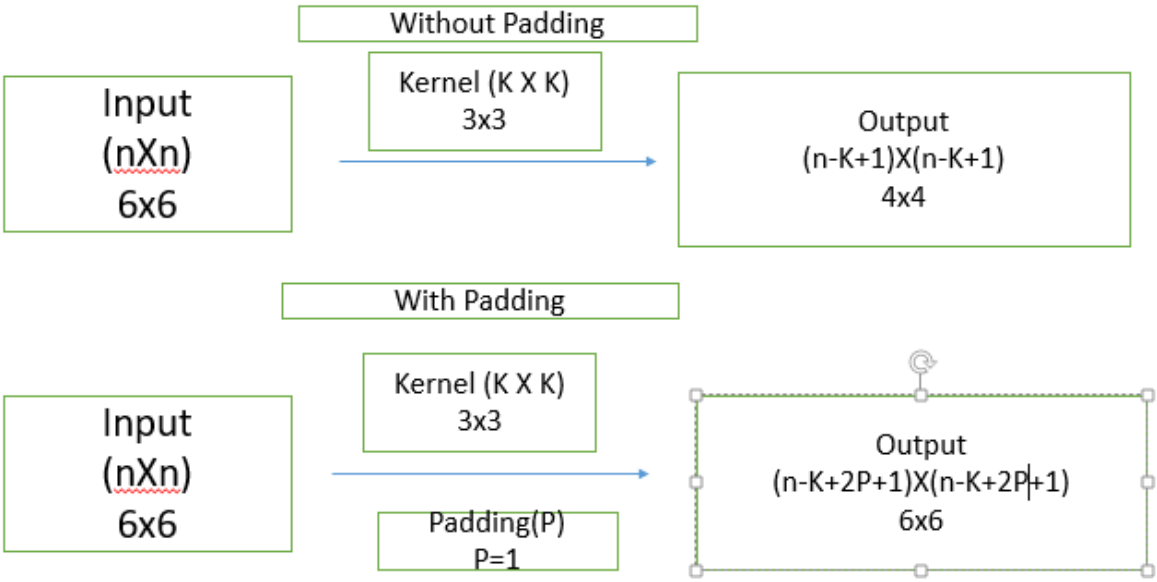


Problem with Simple Convolution Layers

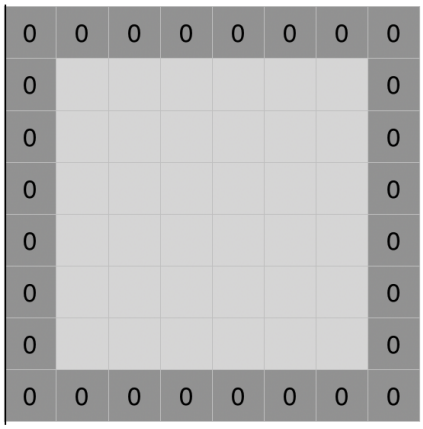
- While applying convolutions we will not obtain the output dimensions the same as input we will lose data over borders so we append a border of zeros and recalculate the convolution covering all the input values.

    1. Padding
    2. Striding

1. Padding



- See In without padding our input is 6x6 but output image goes down into 4x4 . so by using padding we got the same result.Padding is simply a process of adding layers of zeros to our input images so as to avoid the problems mentioned above.



Zero-padding added to image

- So padding prevents shrinking as, if p = number of layers of zeros added to the border of the image, then our (n x n) image becomes (n + 2p) x (n + 2p) image after padding. So, applying convolution-operation (with (f x f) filter) outputs (n + 2p – f + 1) x (n + 2p – f + 1) images. For example, adding one layer of padding to an (8 x 8) image and using a (3 x 3) filter we would get an (8 x 8) output after performing convolution operation.

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

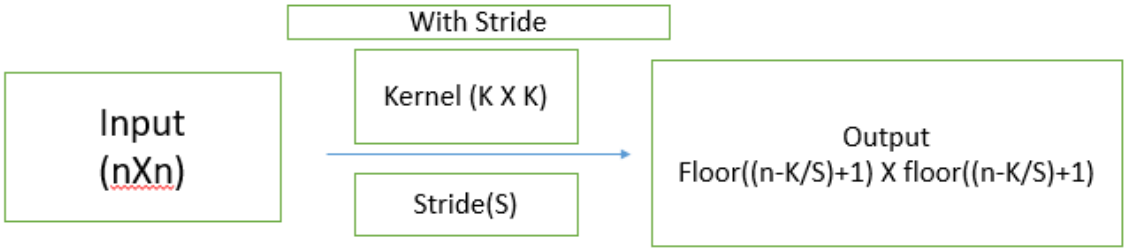
Kernel

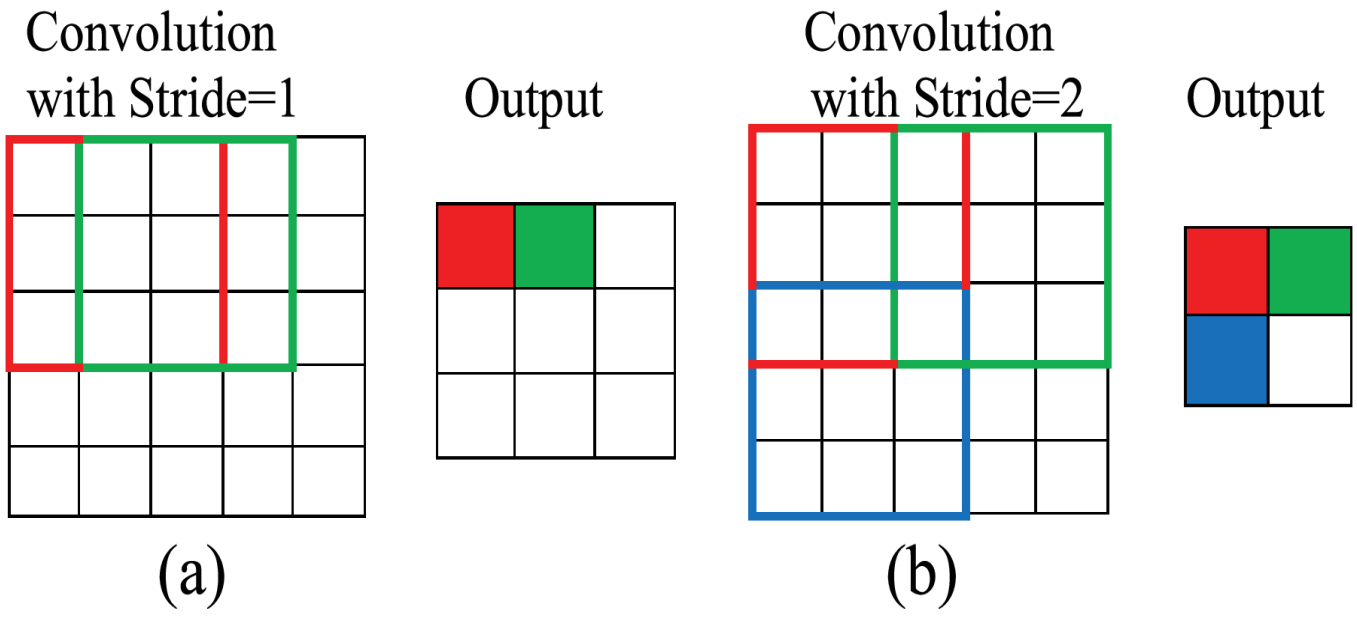
0	-1	0
-1	5	-1
0	-1	0

114				

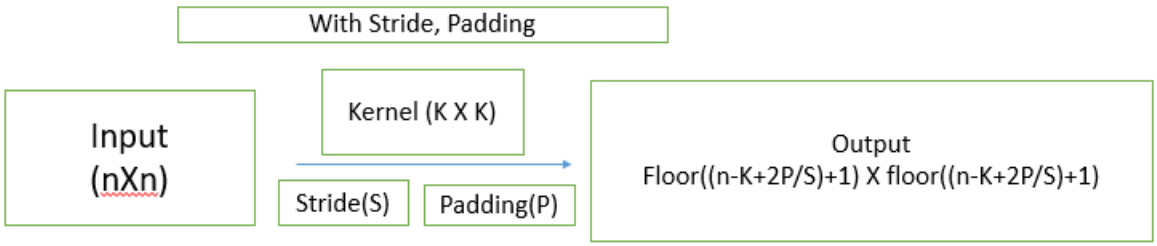
2. Strides

- It uses to reduce the size of matrix. if we sfited by 1 then we called stride=1 and if we sfited by 2 means stride = 2 so on.



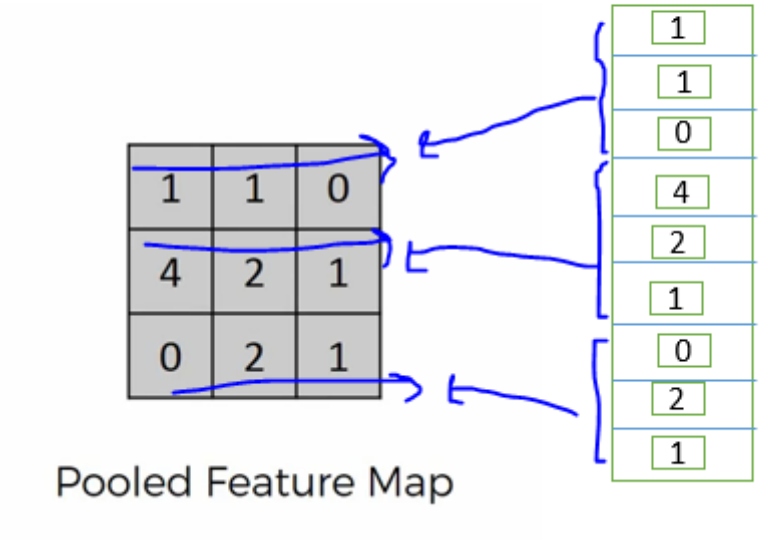


Padding,Stride Put in One Equation



Step3 : Flattening

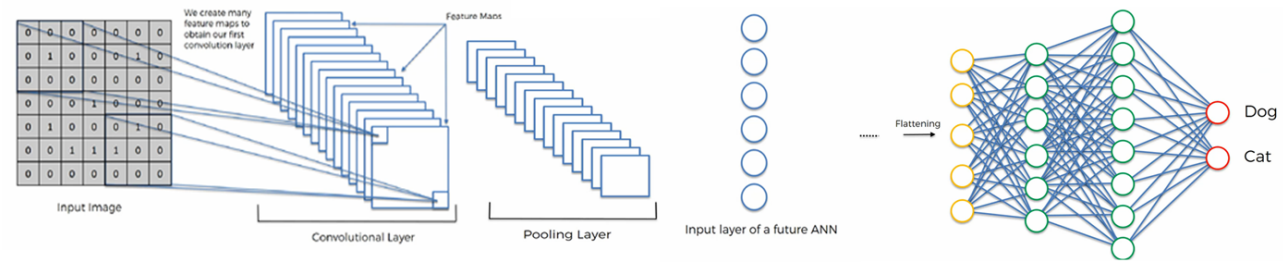
- Flattening is converting the data into a 1-dimensional array for inputting it to the next layer. We flatten the output of the convolutional layers to create a single long feature vector. And it is connected to the final classification model, which is called a fully-connected layer.



Step 4

Complete CNN in one View

here in last step we use full connection network



This is a simple CNN Network

In [1]:

```
# Importing the libraries
import tensorflow as tf
from tensorflow.keras.preprocessing.image import ImageDataGenerator
```

In [2]:

```
#data sugmentation
# Preprocessing the Training set
train_datagen = ImageDataGenerator(rescale=1./255,
                                   rotation_range=40,
                                   width_shift_range=0.2,
                                   height_shift_range=0.2,
                                   shear_range=0.2,
                                   zoom_range=0.2,
                                   horizontal_flip=True,
                                   fill_mode='nearest')
training_set = train_datagen.flow_from_directory('image_data/training',
                                                 target_size = (64, 64),
                                                 batch_size = 32,
                                                 class_mode = 'binary')
```

```
Found 198 images belonging to 2 classes.
```

In [3]:

```
# Preprocessing the Test set
test_datagen = ImageDataGenerator(rescale = 1./255)
test_set = test_datagen.flow_from_directory('image_data/validation',
                                            target_size = (64, 64),
                                            batch_size = 32,
                                            class_mode = 'binary')
```

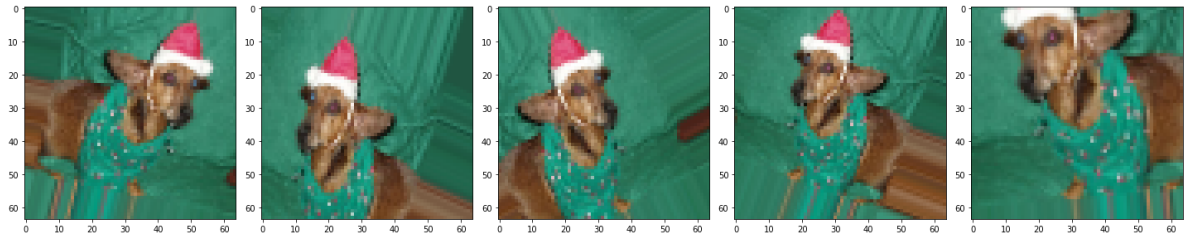
```
Found 100 images belonging to 2 classes.
```

In [4]:

```
## showing some image from training
import matplotlib.pyplot as plt
def plotImages(images_arr):
    fig, axes = plt.subplots(1, 5, figsize=(20, 20))
    axes = axes.flatten()
    for img, ax in zip(images_arr, axes):
        ax.imshow(img)
    plt.tight_layout()
    plt.show()
```

In [5]:

```
images = [training_set[0][0][0] for i in range(5)]
plotImages(images)
```



Model Build Use Only CNN

In [6]:

```
from tensorflow.keras.layers import Conv2D
```

In [7]:

```
# Part 2 - Building the CNN

# Initialising the CNN
cnn = tf.keras.models.Sequential()

# Step 1 - # Adding a first convolutional layer
cnn.add(tf.keras.layers.Conv2D(filters=32,padding="same",kernel_size=3, activation='relu',
## step 2 - #apply maxpool
cnn.add(tf.keras.layers.MaxPool2D(pool_size=2, strides=2)) ## Apply pooing stride

# Adding a second convolutional layer
cnn.add(tf.keras.layers.Conv2D(filters=32,padding='same',kernel_size=3, activation='relu'))
cnn.add(tf.keras.layers.MaxPool2D(pool_size=2, strides=2))


# Step 3 - Flattening
cnn.add(tf.keras.layers.Flatten())

# Step 4 - Full Connection
cnn.add(tf.keras.layers.Dense(units=128, activation='relu'))
tf.keras.layers.Dropout(0.5)

# Step 5 - Output Layer
cnn.add(tf.keras.layers.Dense(units=1, activation='sigmoid'))
```

In [8]:

```
# Part 3 - Training the CNN

# Compiling the CNN
cnn.compile(optimizer = 'adam', loss = 'binary_crossentropy', metrics = ['accuracy'])
```

In [9]:

```
# Training the CNN on the Training set and evaluating it on the Test set
history = cnn.fit(x = training_set, validation_data = test_set, epochs = 2)
```

```
Epoch 1/2
7/7 [==============================] - 4s 410ms/step - loss: 0.7529 - accura
cy: 0.4600 - val_loss: 0.6988 - val_accuracy: 0.5000
Epoch 2/2
7/7 [==============================] - 2s 308ms/step - loss: 0.6898 - accura
cy: 0.5349 - val_loss: 0.6932 - val_accuracy: 0.5100
```

Save And Load Model

In [10]:

```
#save model
from tensorflow.keras.models import load_model
cnn.save('model_rcat_dog.h5')
```

In [11]:

```
from tensorflow.keras.models import load_model
# load model
model = load_model('model_rcat_dog.h5')
```

In [12]:

```
# Part 4 - Making a single prediction

import numpy as np
from tensorflow.keras.preprocessing import image
test_image = image.load_img('image_data/test/3285.jpg', target_size = (64,64))
test_image = image.img_to_array(test_image)
test_image=test_image/255
test_image = np.expand_dims(test_image, axis = 0)
result = cnn.predict(test_image)
result
```

Out[12]:

```
array([[0.5059088]], dtype=float32)
```

In [13]:

```
if result[0]<=0.5:
    print("The image classified is cat")
else:
    print("The image classified is dog")

from IPython.display import Image
Image(filename='image_data/test/3285.jpg',height='200',width='200')
```

```
The image classified is dog
```

Out[13]:

Basic Introduction

LeNet-5, from the paper Gradient-Based Learning Applied to Document Recognition, is a very efficient convolutional neural network for handwritten character recognition.

Paper: Gradient-Based Learning Applied to Document Recognition (http://yann.lecun.com/exdb/publis/pdf/lecun-98.pdf)

Authors: Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner

Published in: Proceedings of the IEEE (1998)

Structure of the LeNet network

LeNet5 is a small network, it contains the basic modules of deep learning: convolutional layer, pooling layer, and full link layer. It is the basis of other deep learning models. Here we analyze LeNet5 in depth. At the same time, through example analysis, deepen the understanding of the convolutional layer and pooling layer.

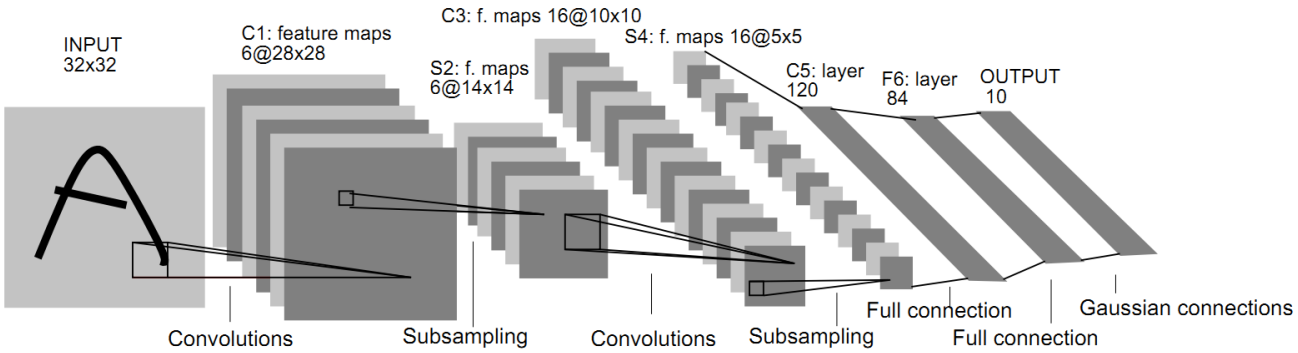


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

LeNet-5 Total seven layer , does not comprise an input, each containing a trainable parameters; each layer has a plurality of the Map the Feature , a characteristic of each of the input FeatureMap extracted by means of a convolution filter, and then each FeatureMap There are multiple neurons.

Layer		Feature Map	Size	Kernel Size	Stride	Activation
Input	Image	1	32x32	-	-	-
1	Convolution	6	28x28	5x5	1	tanh
2	Average Pooling	6	14x14	2x2	2	tanh
3	Convolution	16	10x10	5x5	1	tanh
4	Average Pooling	16	5x5	2x2	2	tanh
5	Convolution	120	1x1	5x5	1	tanh
6	FC	-	84	-	-	tanh
Output	FC	-	10	-	-	softmax

Detailed explanation of each layer parameter:

INPUT Layer

The first is the data INPUT layer. The size of the input image is uniformly normalized to 32 * 32.

> Note: This layer does not count as the network structure of LeNet-5. Traditionally, the input layer is not considered as one of the network hierarchy.

C1 layer-convolutional layer

> **Input picture**: 32 * 32
>
> **Convolution kernel size**: 5 * 5
>
> **Convolution kernel types**: 6
>
> **Output featuremap size**: 28 * 28 (32-5 + 1) = 28
>
> **Number of neurons**: 28 *28* 6
>
> **Trainable parameters**: (5 *5* + *1)* 6 (5 * 5 = 25 unit parameters and one bias parameter per filter, a total of 6 filters)
>
> **Number of connections**: (5 *5* + *1)* 6 *28* 28 = 122304

Detailed description:

1. The first convolution operation is performed on the input image (using 6 convolution kernels of size 5 *5) to obtain 6 C1 feature maps (6 feature maps of size 28* 28, 32-5 + 1 = 28).
2. Let's take a look at how many parameters are needed. The size of the convolution kernel is 5 *5, and there are 6* (5 * 5 + 1) = 156 parameters in total, where +1 indicates that a kernel has a bias.
3. For the convolutional layer C1, each pixel in C1 is connected to 5 *5 pixels and 1 bias in the input image, so there are 156* 28 * 28 = 122304 connections in total. There are 122,304 connections, but we only need to learn 156 parameters, mainly through weight sharing.

S2 layer-pooling layer (downsampling layer)

> **Input**: 28 * 28
>
> **Sampling area**: 2 * 2
>
> **Sampling method**: 4 inputs are added, multiplied by a trainable parameter, plus a trainable offset. Results via sigmoid
>
> **Sampling type**: 6
>
> **Output featureMap size**: 14 * 14 (28/2)
>
> **Number of neurons**: 14 *14* 6
>
> **Trainable parameters**: 2 * 6 (the weight of the sum + the offset)
>
> **Number of connections**: (2 *2* + *1)* 6 *14* 14
>
> The size of each feature map in S2 is 1/4 of the size of the feature map in C1.

Detailed description:

The pooling operation is followed immediately after the first convolution. Pooling is performed using 2 *2 kernels, and S2, 6 feature maps of 14* 14 (28/2 = 14) are obtained.

The pooling layer of S2 is the sum of the pixels in the 2 * 2 area in C1 multiplied by a weight coefficient plus an offset, and then the result is mapped again.

So each pooling core has two training parameters, so there are 2x6 = 12 training parameters, but there are 5x14x14x6 = 5880 connections.

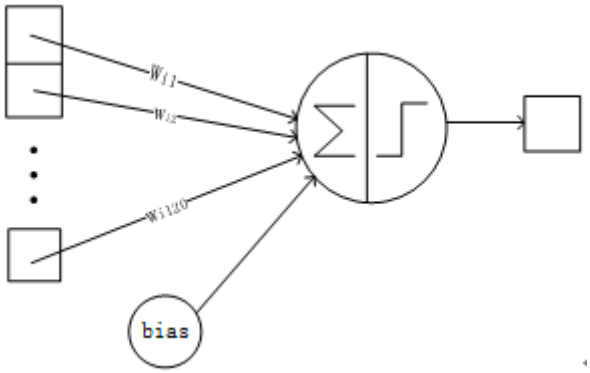
C3 layer-convolutional layer

> **Input**: all 6 or several feature map combinations in S2
>
> **Convolution kernel size**: 5 * 5
>
> **Convolution kernel type**: 16
>
> **Output featureMap size**: 10 * 10 (14-5 + 1) = 10
>
> Each feature map in C3 is connected to all 6 or several feature maps in S2, indicating that the feature map of this layer is a different combination of the feature maps extracted from the previous layer.
>
> One way is that the first 6 feature maps of C3 take 3 adjacent feature map subsets in S2 as input. The next 6 feature maps take 4 subsets of neighboring feature maps in S2 as input. The next three take the non-adjacent 4 feature map subsets as input. The last one takes all the feature maps in S2 as input.
>
> **The trainable parameters are**: 6 *(3* 5 *5* + *1)* + *6* (4 *5* 5 + 1) + 3 *(4* 5 *5* + *1)* + *1* (6 *5* 5 +1) = 1516
>
> **Number of connections**: 10 *10* 1516 = 151600

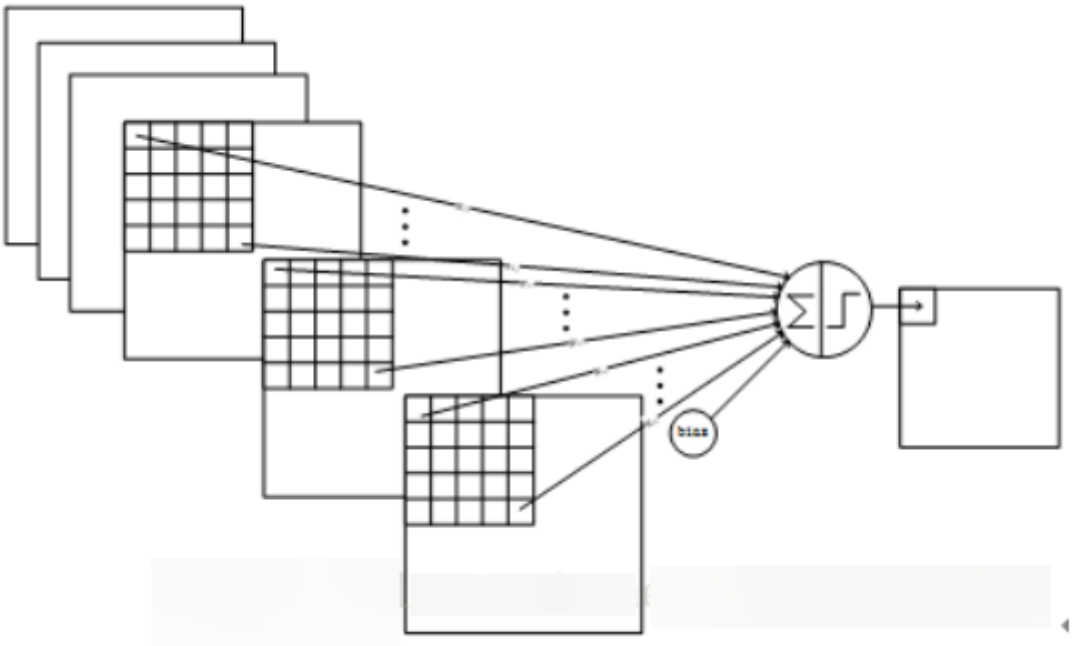
Detailed description:

After the first pooling, the second convolution, the output of the second convolution is C3, 16 10x10 feature maps, and the size of the convolution kernel is 5 *5. We know that S2 has 6 14* 14 feature maps, how to get 16 feature maps from 6 feature maps? Here are the 16 feature maps calculated by the special combination of the feature maps of S2. details as follows:

The first 6 feature maps of C3 (corresponding to the 6th column of the first red box in the figure above) are connected to the 3 feature maps connected to the S2 layer (the first red box in the above figure), and the next 6 feature maps are connected to the S2 layer The 4 feature maps are connected (the second red box in the figure above), the next 3 feature maps are connected with the 4 feature maps that are not connected at the S2 layer, and the last is connected with all the feature maps at the S2 layer. The convolution kernel size is still 5 *5, so there are 6* (3 *5* 5 + 1) + 6 *(4* 5 *5* + *1)* + *3* (4 *5* 5 + 1) +1 *(6* 5 *5* + *1) = 1516 parameters. The image size is 10* 10, so there are 151600 connections.



The convolution structure of C3 and the first 3 graphs in S2 is shown below:



S4 layer-pooling layer (downsampling layer)

> **Input**: 10 * 10
>
> **Sampling area**: 2 * 2
>
> **Sampling method**: 4 inputs are added, multiplied by a trainable parameter, plus a trainable offset. Results via sigmoid
>
> **Sampling type**: 16
>
> **Output featureMap size**: 5 * 5 (10/2)
>
> **Number of neurons**: 5 *5* 16 = 400
>
> **Trainable parameters**: 2 * 16 = 32 (the weight of the sum + the offset)
>
> **Number of connections**: 16 *(2* 2 + 1) *5* 5 = 2000
>
> The size of each feature map in S4 is 1/4 of the size of the feature map in C3

Detailed description:

S4 is the pooling layer, the window size is still 2 * 2, a total of 16 feature maps, and the 16 10x10 maps of the C3 layer are pooled in units of 2x2 to obtain 16 5x5 feature maps. This layer has a total of 32 training parameters of 2x16, 5x5x5x16 = 2000 connections.

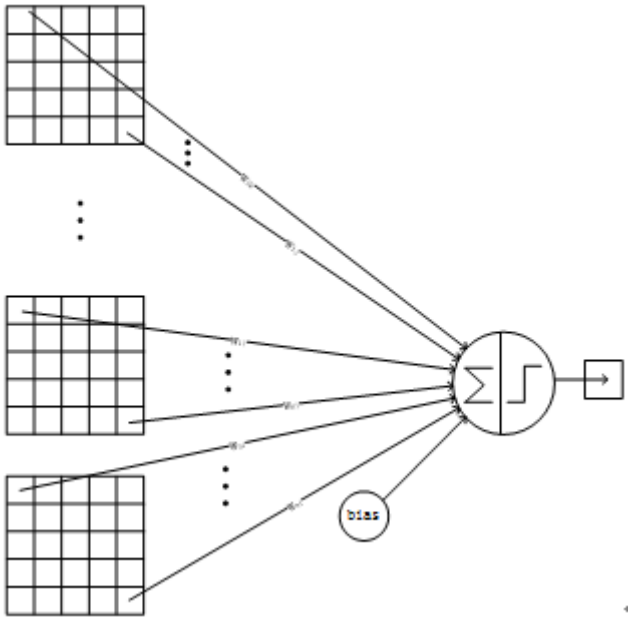
The connection is similar to the S2 layer.

C5 layer-convolution layer

> **Input**: All 16 unit feature maps of the S4 layer (all connected to s4)
>
> **Convolution kernel size**: 5 * 5
>
> **Convolution kernel type**: 120
>
> **Output featureMap size**: 1 * 1 (5-5 + 1)
>
> **Trainable parameters / connection**: 120 *(16* 5 * 5 + 1) = 48120

Detailed description:

The C5 layer is a convolutional layer. Since the size of the 16 images of the S4 layer is 5x5, which is the same as the size of the convolution kernel, the size of the image formed after convolution is 1x1. This results in 120 convolution results. Each is connected to the 16 maps on the previous level. So there are (5x5x16 + 1) x120 = 48120 parameters, and there are also 48120 connections. The network structure of the C5 layer is as follows:



F6 layer-fully connected layer

> **Input**: c5 120-dimensional vector
>
> **Calculation method**: calculate the dot product between the input vector and the weight vector, plus an offset, and the result is output through the sigmoid function.
>
> **Trainable parameters**: 84 * (120 + 1) = 10164

Detailed description:

Layer 6 is a fully connected layer. The F6 layer has 84 nodes, corresponding to a 7x12 bitmap, -1 means white, 1 means black, so the black and white of the bitmap of each symbol corresponds to a code. The training parameters and number of connections for this layer are (120 + 1) x84 = 10164. The ASCII encoding diagram is as follows:

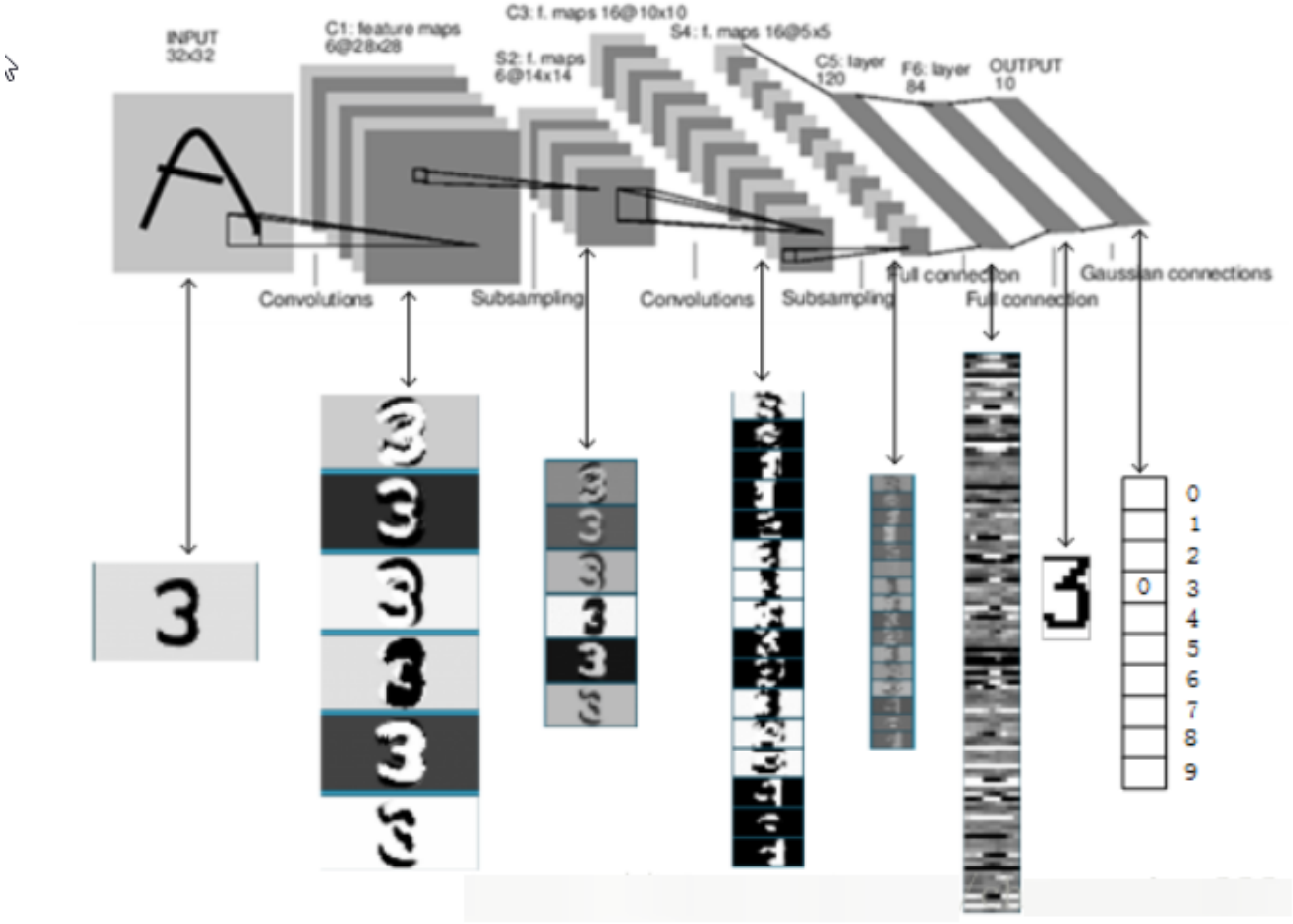
The connection method of the F6 layer is as follows:

Output layer-fully connected layer

The output layer is also a fully connected layer, with a total of 10 nodes, which respectively represent the numbers 0 to 9, and if the value of node i is 0, the result of network recognition is the number i. A radial basis function (RBF) network connection is used. Assuming x is the input of the previous layer and y is the output of the RBF, the calculation of the RBF output is:

$$y_i = \sum_j (x_j - w_{ij})^2$$

The value of the above formula w_ij is determined by the bitmap encoding of i, where i ranges from 0 to 9, and j ranges from 0 to 7 * 12-1. The closer the value of the RBF output is to 0, the closer it is to i, that is, the closer to the ASCII encoding figure of i, it means that the recognition result input by the current network is the character i. This layer has 84x10 = 840 parameters and connections.



Summary

- LeNet-5 is a very efficient convolutional neural network for handwritten character recognition.
- Convolutional neural networks can make good use of the structural information of images.
- The convolutional layer has fewer parameters, which is also determined by the main characteristics of the convolutional layer, that is, local connection and shared weights.

Code Implementation

In [4]:

```
import keras
from keras.datasets import mnist
from keras.layers import Conv2D, MaxPooling2D,AveragePooling2D
from keras.layers import Dense, Flatten
from keras.models import Sequential
```

In [2]:

```
# Loading the dataset and perform splitting
(x_train, y_train), (x_test, y_test) = mnist.load_data()
# Peforming reshaping operation
x_train = x_train.reshape(x_train.shape[0], 28, 28, 1)
x_test = x_test.reshape(x_test.shape[0], 28, 28, 1)
```

In [3]:

```
# Normalization
x_train = x_train / 255
x_test = x_test / 255

# One Hot Encoding
y_train = keras.utils.to_categorical(y_train, 10)
y_test = keras.utils.to_categorical(y_test, 10)
# Building the Model Architecture
```

LeNet Model

In [8]:

```
model = Sequential()
# Select 6 feature convolution kernels with a size of 5 * 5 (without offset), and get 66 fe
# That is, the number of neurons has been reduced from 10241024 to 28 * 28 = 784 28 * 28 =
# Parameters between input layer and C1 layer: 6 * (5 * 5 + 1)
model.add(Conv2D(6, kernel_size=(5, 5), activation='tanh', input_shape=(28, 28, 1)))
# The input of this layer is the output of the first layer, which is a 28 * 28 * 6 node mat
# The size of the filter used in this layer is 2 * 2, and the step length and width are bot
model.add(MaxPooling2D(pool_size=(2, 2)))
# The input matrix size of this layer is 14 * 14 * 6, the filter size used is 5 * 5, and th
# The output matrix size of this layer is 10 * 10 * 16. This layer has 5 * 5 * 6 * 16 + 16
model.add(Conv2D(16, kernel_size=(5, 5), activation='tanh'))
# The input matrix size of this layer is 10 * 10 * 16. The size of the filter used in this
model.add(MaxPooling2D(pool_size=(2, 2)))
# The input matrix size of this layer is 5 * 5 * 16. This layer is called a convolution lay
# So it is not different from the fully connected layer. If the nodes in the 5 * 5 * 16 mat
# The number of output nodes in this layer is 120, with a total of 5 * 5 * 16 * 120 + 120 =
model.add(Flatten())
model.add(Dense(120, activation='tanh'))
# The number of input nodes in this layer is 120 and the number of output nodes is 84. The
model.add(Dense(84, activation='tanh'))
# The number of input nodes in this layer is 84 and the number of output nodes is 10. The t
model.add(Dense(10, activation='softmax'))
```

OR

In [ ]:

```
model = keras.Sequential()

model.add(Conv2D(filters=6, kernel_size=(5, 5), activation='tanh', input_shape=(32,32,1)))
model.add(AveragePooling2D(2,2))

model.add(Conv2D(filters=16, kernel_size=(5, 5), activation='tanh'))
model.add(AveragePooling2D())

model.add(Flatten())

model.add(Dense(units=120, activation='tanh'))

model.add(Dense(units=84, activation='tanh'))

model.add(Dense(units=10, activation = 'softmax'))
```

In [9]:

```
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_2 (Conv2D)            (None, 24, 24, 6)         156
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 12, 12, 6)         0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 8, 8, 16)          2416
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 4, 4, 16)          0
_________________________________________________________________
flatten_1 (Flatten)          (None, 256)               0
_________________________________________________________________
dense_3 (Dense)              (None, 120)               30840
_________________________________________________________________
dense_4 (Dense)              (None, 84)                10164
_________________________________________________________________
dense_5 (Dense)              (None, 10)                850
=================================================================
Total params: 44,426
Trainable params: 44,426
Non-trainable params: 0
_________________________________________________________________
```

In [10]:

```
model.compile(loss=keras.metrics.categorical_crossentropy, optimizer=keras.optimizers.Adam(
model.fit(x_train, y_train, batch_size=128, epochs=1, verbose=1, validation_data=(x_test, y
```

```
469/469 [==============================] - 22s 46ms/step - loss: 0.6072 - ac
curacy: 0.8285 - val_loss: 0.0840 - val_accuracy: 0.9737
```

Out[10]:

```
<tensorflow.python.keras.callbacks.History at 0x195a9244748>
```

In [11]:

```
score = model.evaluate(x_test, y_test)
print('Test Loss:', score[0])
print('Test accuracy:', score[1])
```

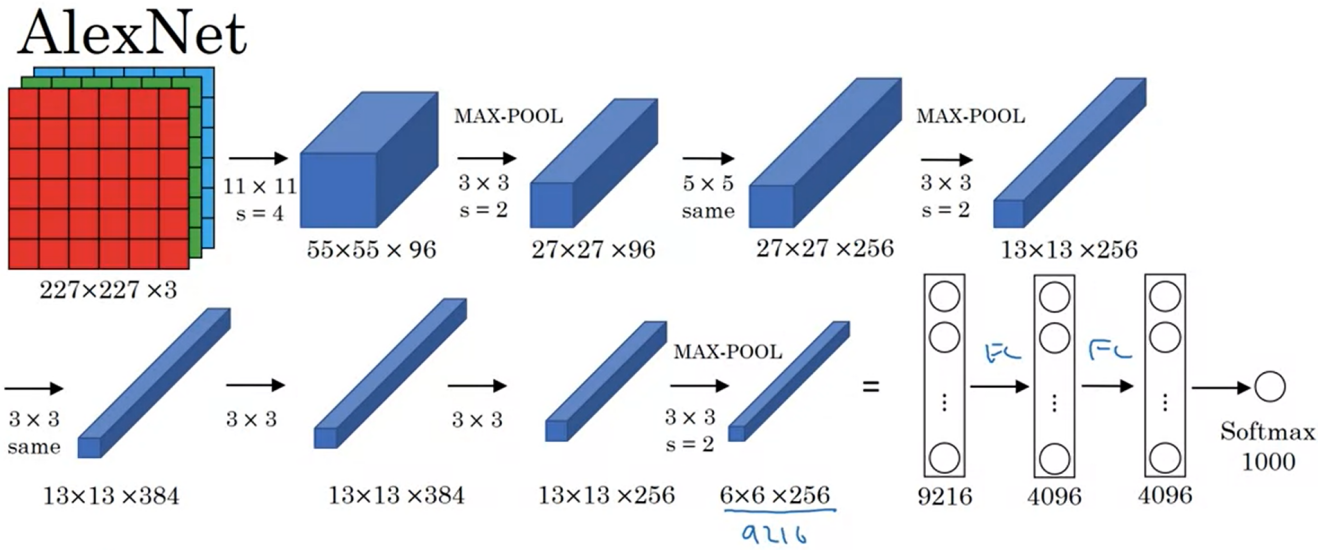
```
313/313 [==============================] - 2s 7ms/step - loss: 0.0840 - accu
racy: 0.9737
Test Loss: 0.08401492983102798
Test accuracy: 0.9736999869346619
```

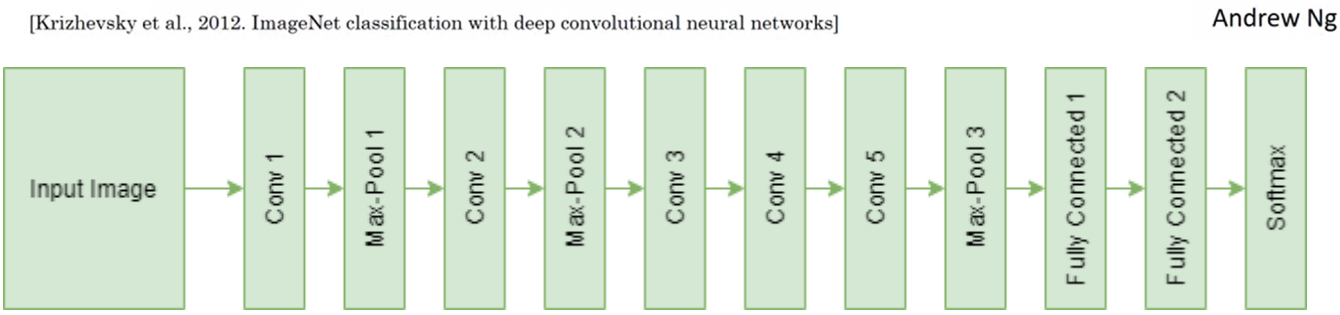
ALEXNET

The most important features of the AlexNet paper are:

- As the model had to train 60 million parameters (which is quite a lot), it was prone to overfitting. According to the paper, the usage of Dropout and Data Augmentation significantly helped in reducing overfitting. The first and second fully connected layers in the architecture thus used a dropout of 0.5 for the purpose. Artificially increasing the number of images through data augmentation helped in the expansion of the dataset dynamically during runtime, which helped the model generalize better.
- Another distinct factor was using the ReLU activation function instead of tanh or sigmoid, which resulted in faster training times (a decrease in training time by 6 times). Deep Learning Networks usually employ ReLU non-linearity to achieve faster training times as the others start saturating when they hit higher activation values.

- AlexNet is a Classic type of Convolutional Neural Network, and it came into existence after the 2012 ImageNet challenge. The network architecture is given below :





Model Explanation :

- The Input to this model have the dimensions 227x227x3 follwed by a Convolutional Layer with 96 filters of 11x11 dimensions and having a ‘same’ padding and a stride of 4. The resulting output dimensions are given as :

```
floor(((n + 2*padding - filter)/stride) + 1 ) * floor(((n + 2*padding – filter)/stride) + 1)
```

Note : This formula is for square input with height = width = n

- Explaining the first Layer with input 227x227x3 and Convolutional layer with 96 filters of 11x11 , 'valid' padding and stride = 4 , output dims will be

= floor(((227 + 0–11)/4) + 1) * floor(((227 + 0–11)/4) + 1)

= floor((216/4) + 1) * floor((216/4) + 1)

= floor(54 + 1) * floor(54 + 1)

= 55 * 55

- Since number of filters = 96 , thus output of first Layer is : 55x55x96
- Continuing we have the MaxPooling layer (3, 3) with the stride of 2,making the output size decrease to 27x27x96, followed by another Convolutional Layer with 256, (5,5) filters and 'same' padding, that is, the output height and width are retained as the previous layer thus output from this layer is 27x27x256.
- Next we have the MaxPooling again ,reducing the size to 13x13x256. Another Convolutional Operation with 384, (3,3) filters having same padding is applied twice giving the output as 13x13x384, followed by another Convulutional Layer with 256 , (3,3) filters and same padding resulting in 13x13x256 output.
- This is MaxPooled and dimensions are reduced to 6x6x256. Further the layer is Flatten out and 2 Fully Connected Layers with 4096 units each are made which is further connected to 1000 units softmax layer.
- The network is used for classifying much large number of classes as per our requirement. However in our case, we will make the output softmax layer with 6 units as we ahve to classify into 6 classes. The softmax layer gives us the probablities for each class to which an Input Image might belong.

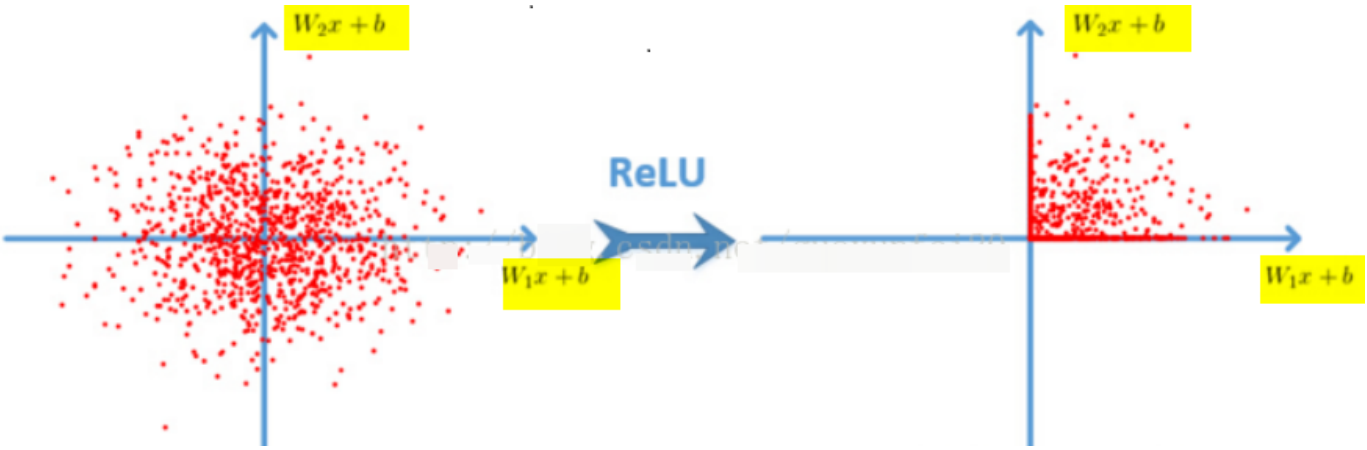
Size / Operation	Filter	Depth	Stride	Padding	Number of Parameters	Forward Computation
3 *227* 227						
Conv1 + Relu	11 *11*	96	4		(1111*3 + 1)* 96=34944	(11*11*3 + 1) *96* 55 *55=105705600*
96 55 *55*						
Max Pooling	3 3		2			
96 *27* 27						
Norm						
Conv2 + Relu	5 *5*	256	1	2	(5 5 *96 + 1)* 256=614656	(5 *5* 96 + 1) *256* 27 *27=448084224*
256 27 *27*						
Max Pooling	3 3		2			
256 *13* 13						
Norm						
Conv3 + Relu	3 *3*	384	1	1	(3 3 *256 + 1)* 384=885120	(3 *3* 256 + 1) *384* 13 *13=149585280*
384 13 *13*						
Conv4 + Relu	3 3	384	1	1	(3 *3* 384 + 1) *384=1327488*	(3 3 *384 + 1)* 384 *13* 13=224345472
384 *13* 13						
Conv5 + Relu	3 *3*	256	1	1	(3 3 *384 + 1)* 256=884992	(3 *3* 384 + 1) *256* 13 *13=149563648*
256 13 *13*						

Size / Operation	Filter	Depth	Stride	Padding	Number of Parameters	Forward Computation
Max Pooling	3 3		2			
256 *6* 6						
Dropout (rate 0.5)						
FC6 + Relu					256 *6* 6 *4096=37748736*	256 6 *6* 4096=37748736
4096						
Dropout (rate 0.5)						
FC7 + Relu					4096 *4096=16777216*	4096 4096=16777216
4096						
FC8 + Relu					4096 *1000=4096000*	4096 1000=4096000
1000 classes						
Overall					62369152=62.3 million	1135906176=1.1 billion
Conv VS FC					Conv:3.7million (6%) , FC: 58.6 million (94%)	Conv: 1.08 billion (95%) , FC: 58.6 million (5%)

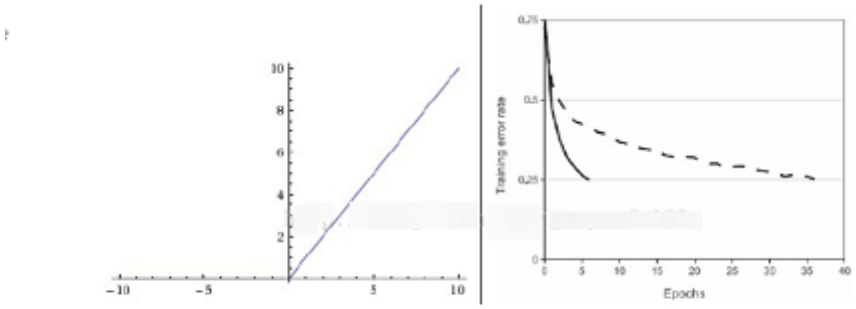
Why does AlexNet achieve better results?

1. **Relu activation function is used.**

Relu function: f (x) = max (0, x)



ReLU-based deep convolutional networks are trained several times faster than tanh and sigmoid- based networks. The following figure shows the number of iterations for a four-layer convolutional network based on CIFAR-10 that reached 25% training error in tanh and ReLU:



Left: Rectified Linear Unit (ReLU) activation function, which is zero when x < 0 and then linear with slope 1 when x > 0. **Right:** A plot from Krizhevsky et al. (pdf) paper indicating the 6x improvement in convergence with the ReLU unit compared to the tanh unit.

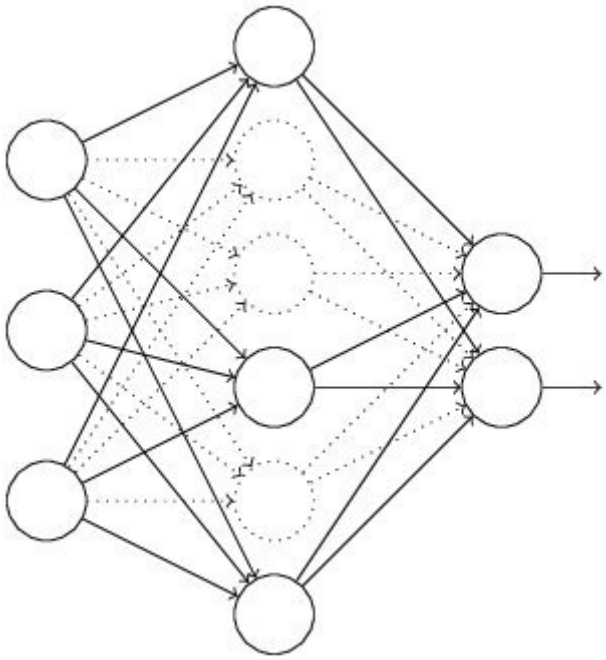
1. **Standardization (Local Response Normalization)**

After using ReLU f (x) = max (0, x), you will find that the value after the activation function has no range like the tanh and sigmoid functions, so a normalization will usually be done after ReLU, and the LRU is a steady proposal (Not sure here, it should be proposed?) One method in neuroscience is called "Lateral inhibition", which talks about the effect of active neurons on its surrounding neurons.

$$b^i_{x,y} = a^i_{x,y} / \left(k + \alpha \sum_{j=\max(0,i-n/2)}^{\min(N-1,i+n/2)} (a^j_{x,y})^2 \right)^{\beta}$$

1. **Dropout**

Dropout is also a concept often said, which can effectively prevent overfitting of neural networks. Compared to the general linear model, a regular method is used to prevent the model from overfitting. In the neural network, Dropout is implemented by modifying the structure of the neural network itself. For a certain layer of neurons, randomly delete some neurons with a defined probability, while keeping the individuals of the input layer and output layer neurons unchanged, and then update the parameters according to the learning method of the neural network. In the next iteration, rerandom Remove some neurons until the end of training.



1. **Enhanced Data (Data Augmentation)**

In deep learning, when the amount of data is not large enough, there are generally 4 solutions:

Data augmentation- artificially increase the size of the training set-create a batch of "new" data from existing data by means of translation, flipping, noise

Regularization——The relatively small amount of data will cause the model to overfit, making the training error small and the test error particularly large. By adding a regular term after the Loss Function , the overfitting can be suppressed. The disadvantage is that a need is introduced Manually adjusted hyper-parameter.

Dropout- also a regularization method. But different from the above, it is achieved by randomly setting the output of some neurons to zero

Unsupervised Pre-training- use Auto-Encoder or RBM's convolution form to do unsupervised pre-training layer by layer, and finally add a classification layer to do supervised Fine-Tuning

In [1]:

```
#Importing library
import keras
from keras.models import Sequential
from keras.layers import Dense, Activation, Dropout, Flatten, Conv2D, MaxPooling2D
from keras.layers.normalization import BatchNormalization
import numpy as np

np.random.seed(1000)
```

In [6]:

```
model = Sequential()

# 1st Convolutional Layer
model.add(Conv2D(filters = 96, input_shape = (224, 224, 3), kernel_size = (11, 11), strides
model.add(Activation('relu'))
# Max-Pooling
model.add(MaxPooling2D(pool_size = (2, 2), strides = (2, 2), padding = 'valid'))
# Batch Normalisation
model.add(BatchNormalization())

# 2nd Convolutional Layer
model.add(Conv2D(filters = 256, kernel_size = (11, 11), strides = (1, 1), padding = 'valid'
model.add(Activation('relu'))
# Max-Pooling
model.add(MaxPooling2D(pool_size = (2, 2), strides = (2, 2), padding = 'valid'))
# Batch Normalisation
model.add(BatchNormalization())

# 3rd Convolutional Layer
model.add(Conv2D(filters = 384, kernel_size = (3, 3), strides = (1, 1), padding = 'valid'))
model.add(Activation('relu'))
# Batch Normalisation
model.add(BatchNormalization())

# 4th Convolutional Layer
model.add(Conv2D(filters = 384, kernel_size = (3, 3), strides = (1, 1), padding = 'valid'))
model.add(Activation('relu'))
# Batch Normalisation
model.add(BatchNormalization())

# 5th Convolutional Layer
model.add(Conv2D(filters = 256, kernel_size = (3, 3), strides = (1, 1), padding = 'valid'))
model.add(Activation('relu'))
# Max-Pooling
model.add(MaxPooling2D(pool_size = (2, 2), strides = (2, 2),padding = 'valid'))
# Batch Normalisation
model.add(BatchNormalization())

# Flattening
model.add(Flatten())

# 1st Dense Layer
model.add(Dense(4096, input_shape = (224*224*3, )))
model.add(Activation('relu'))
# Add Dropout to prevent overfitting
model.add(Dropout(0.4))
# Batch Normalisation
model.add(BatchNormalization())

# 2nd Dense Layer
model.add(Dense(4096))
model.add(Activation('relu'))
# Add Dropout
model.add(Dropout(0.4))
# Batch Normalisation
model.add(BatchNormalization())

# Output Softmax Layer
model.add(Dense(10))
```

```
model.add(Activation('softmax'))
```

In [7]:

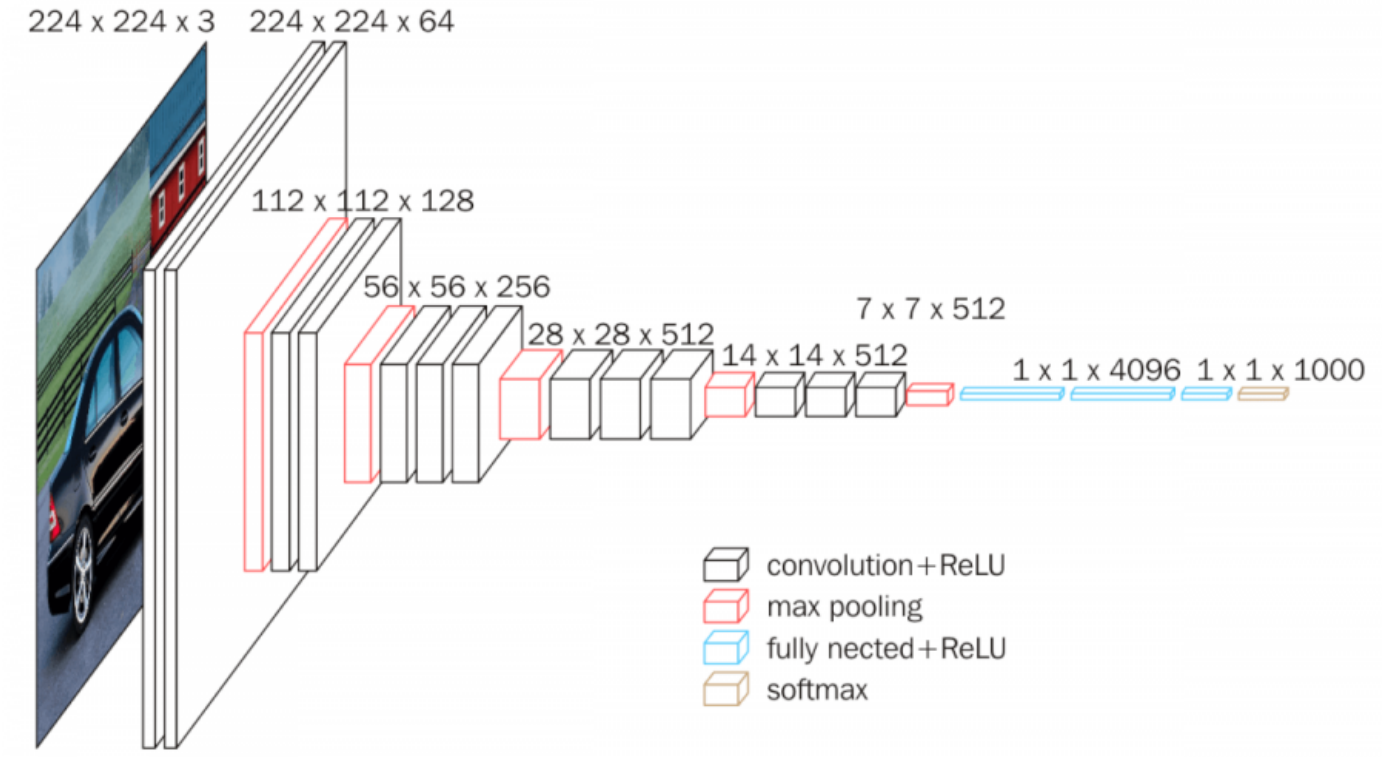
```
#Model Summary
model.summary()
```

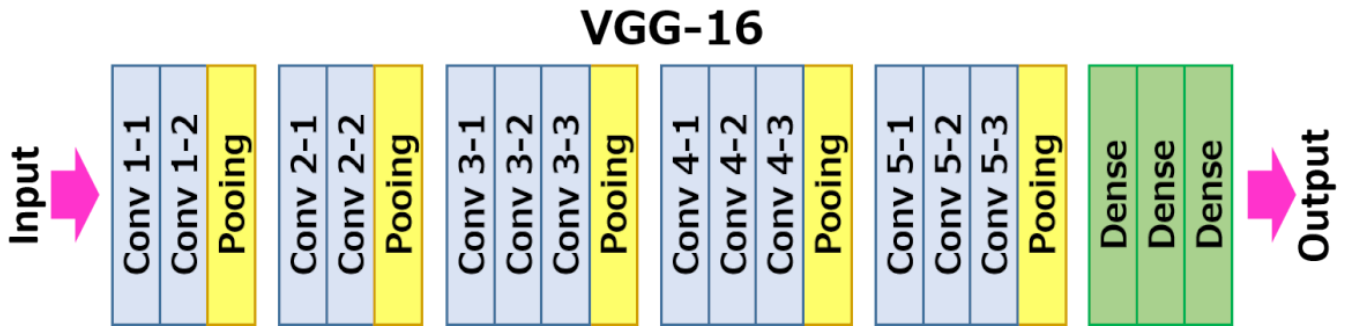
```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 54, 54, 96)        34944
_________________________________________________________________
activation (Activation)      (None, 54, 54, 96)        0
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 27, 27, 96)        0
_________________________________________________________________
batch_normalization (BatchNo (None, 27, 27, 96)        384
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 17, 17, 256)       2973952
_________________________________________________________________
activation_1 (Activation)    (None, 17, 17, 256)       0
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 8, 8, 256)         0
_________________________________________________________________
batch_normalization_1 (Batch (None, 8, 8, 256)         1024
```

In [ ]:

VGG16

- VGG16 is a convolution neural net (CNN) architecture which was used to win ILSVR(Imagenet) competition in 2014.
- It is considered to be one of the excellent vision model architecture till date. Most unique thing about VGG16 is that instead of having a large number of hyper-parameter they focused on having convolution layers of 3x3 filter with a stride 1 and always used same padding and maxpool layer of 2x2 filter of stride 2.
- It follows this arrangement of convolution and max pool layers consistently throughout the whole architecture. In the end it has 2 FC(fully connected layers) followed by a softmax for output. The 16 in VGG16 refers to it has 16 layers that have weights. This network is a pretty large network and it has about 138 million (approx) parameters.





The following are the layers of the model:

- Convolutional Layers = 13
- Pooling Layers = 5
- Dense Layers = 3

Let us explore the layers in detail:

1. **Input:** Image of dimensions (224, 224, 3).

2. **Convolution Layer Conv1:**

    ```
    - Conv1-1: 64 filters
    - Conv1-2: 64 filters and Max Pooling
    - Image dimensions: (224, 224)
    ```

3. **Convolution layer Conv2:** Now, we increase the filters to 128

    ```
    - Input Image dimensions: (112,112)
    - Conv2-1: 128 filters
    - Conv2-2: 128 filters and Max Pooling
    ```

4. **Convolution Layer Conv3:** Again, double the filters to 256, and now add another convolution layer

    ```
    - Input Image dimensions: (56,56)
    - Conv3-1: 256 filters
    - Conv3-2: 256 filters
    - Conv3-3: 256 filters and Max Pooling
    ```

5. **Convolution Layer Conv4:** Similar to Conv3, but now with 512 filters

    ```
    - Input Image dimensions: (28, 28)
    - Conv4-1: 512 filters
    - Conv4-2: 512 filters
    - Conv4-3: 512 filters and Max Pooling
    ```

6. **Convolution Layer Conv5:** Same as Conv4

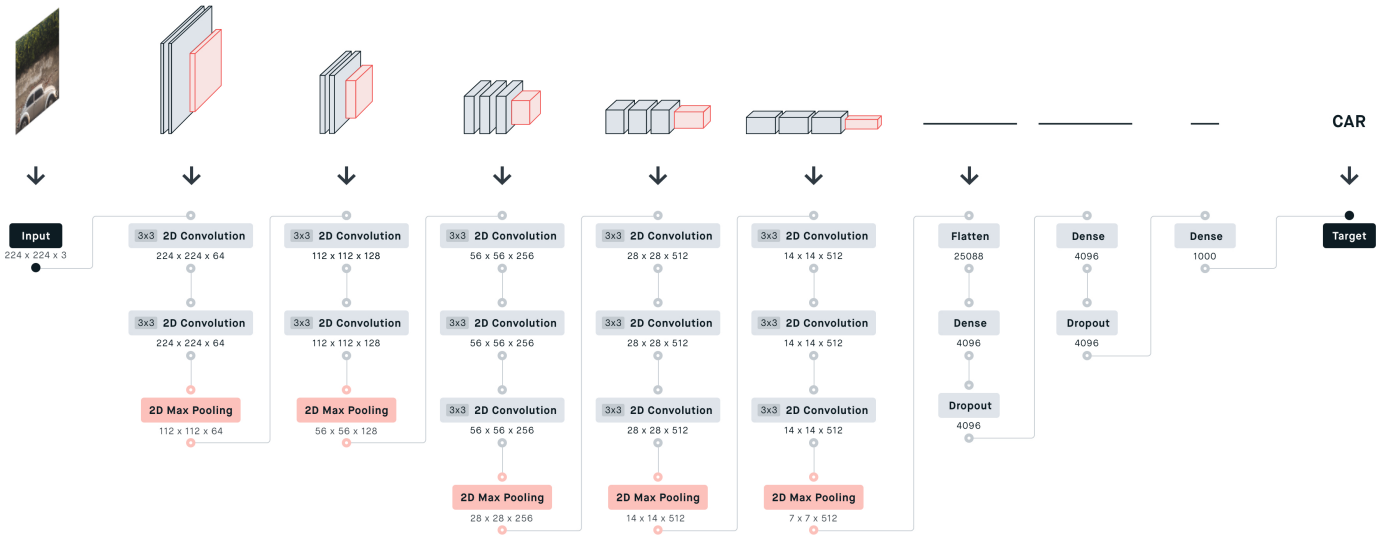
    ```
    - Input Image dimensions: (14, 14)
    - Conv5-1: 512 filters
    - Conv5-2: 512 filters
    - Conv5-3: 512 filters and Max Pooling
    - The output dimensions here are (7, 7). At this point, we flatten the output
    of this layer to generate a feature vector
    ```

7. **Fully Connected/Dense FC1:** 4096 nodes, generating a feature vector of size(1, 4096)
8. **Fully ConnectedDense FC2:** 4096 nodes generating a feature vector of size(1, 4096)
9. **Fully Connected /Dense FC3:** 4096 nodes, generating 1000 channels for 1000 classes. This is then passed on to a Softmax activation function
10. **Output layer**

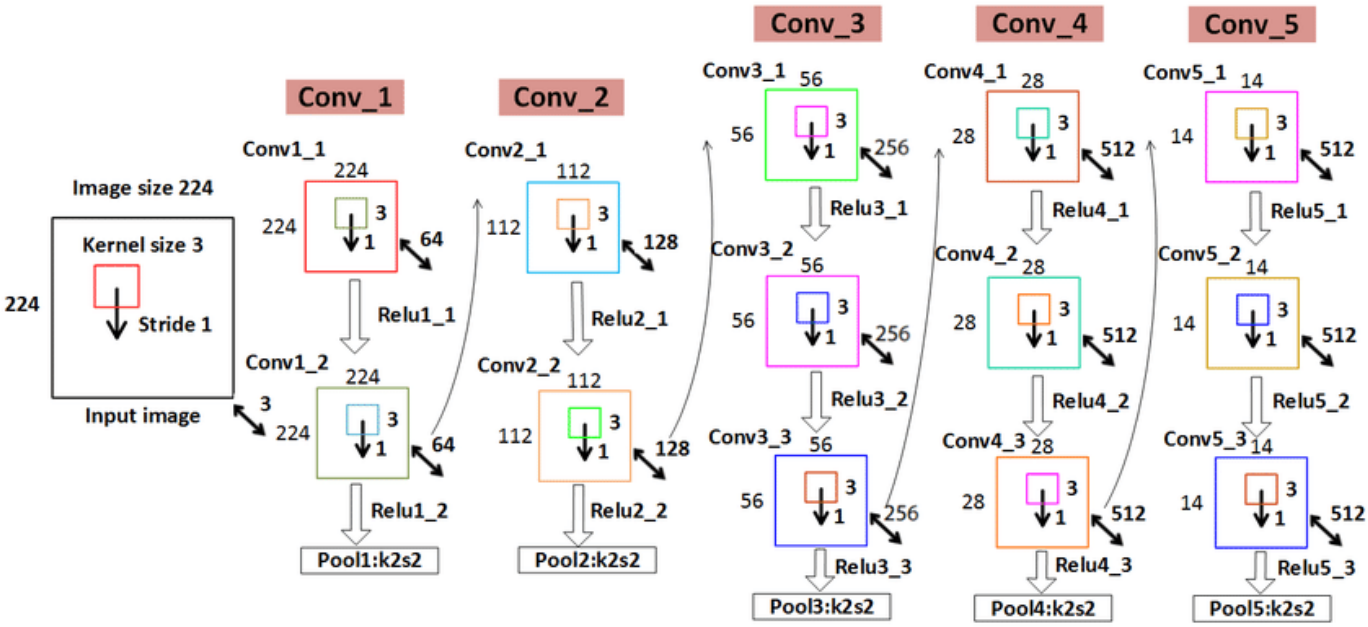
> VGG16 contains 16 layers and VGG19 contains 19 layers. A series of VGGs are exactly the same in the last three fully connected layers. The overall structure includes 5 sets of convolutional layers, followed by a MaxPool. The difference is that more and more cascaded convolutional layers are included in the five sets of convolutional layers .

Layer		Feature Map	Size	Kernel Size	Stride	Activation
Input	Image	1	224 x 224 x 3	-	-	-
1	2 X Convolution	64	224 x 224 x 64	3x3	1	relu
	Max Pooling	64	112 x 112 x 64	3x3	2	relu
3	2 X Convolution	128	112 x 112 x 128	3x3	1	relu
	Max Pooling	128	56 x 56 x 128	3x3	2	relu
5	2 X Convolution	256	56 x 56 x 256	3x3	1	relu
	Max Pooling	256	28 x 28 x 256	3x3	2	relu
7	3 X Convolution	512	28 x 28 x 512	3x3	1	relu
	Max Pooling	512	14 x 14 x 512	3x3	2	relu
10	3 X Convolution	512	14 x 14 x 512	3x3	1	relu
	Max Pooling	512	7 x 7 x 512	3x3	2	relu
13	FC	-	25088	-	-	relu
14	FC	-	4096	-	-	relu
15	FC	-	4096	-	-	relu
Output	FC	-	1000	-	-	Softmax

Each convolutional layer in AlexNet contains only one convolution, and the size of the convolution kernel is 7 *7 ,. In VGGNet, each convolution layer contains 2 to 4 convolution operations. The size of the convolution kernel is 3* 3, the convolution step size is 1, the pooling kernel is 2 * 2, and the step size is 2. The most obvious improvement of VGGNet is to reduce the size of the convolution kernel and increase the number of convolution layers.



Using multiple convolution layers with smaller convolution kernels instead of a larger convolution layer with convolution kernels can reduce parameters on the one hand, and the author believes that it is equivalent to more non-linear mapping, which increases the Fit expression ability.



> Two consecutive 3 *3 convolutions are equivalent to a 5* 5 receptive field, and three are equivalent to 7 *7. The advantages of using three 3* 3 convolutions instead of one 7 *7 convolution are twofold : one, including three ReLu layers instead of one , makes the decision function more discriminative; and two, reducing parameters . For example, the input and output are all C channels. 3 convolutional layers using 3* 3 require 3 (3 *3* C *C) = 27* C *C, and 1 convolutional layer using 7* 7 requires 7 *7* C *C = 49C* C. This can be seen as applying a kind of regularization to the 7 *7 convolution, so that it is decomposed into three 3* 3 convolutions.
>
> The 1 *1 convolution layer is mainly to increase the non-linearity of the decision function without affecting the receptive field of the convolution layer. Although the 1* 1 convolution operation is linear, ReLu adds non-linearity.

Some basic questions

Q1: Why can 3 3x3 convolutions replace 7x7 convolutions?

Answer 1

3 3x3 convolutions, using 3 non-linear activation functions, increasing non-linear expression capabilities, making the segmentation plane more separable Reduce the number of parameters. For the convolution kernel of C channels, 7x7 contains parameters , and the number of 3 3x3 parameters is greatly reduced.

Q2: The role of 1x1 convolution kernel

Answer 2

Increase the nonlinearity of the model without affecting the receptive field 1x1 winding machine is equivalent to linear transformation, and the non-linear activation function plays a non-linear role

Q3: The effect of network depth on results (in the same year, Google also independently released the network GoogleNet with a depth of 22 layers)

Answer 3

VGG and GoogleNet models are deep Small convolution VGG only uses 3x3, while GoogleNet uses 1x1, 3x3, 5x5, the model is more complicated (the model began to use a large convolution kernel to reduce the calculation of the subsequent machine layer)

Implement On Keras

In [3]:

```
#Importing library
import keras
from keras.models import Sequential
from keras.layers import Dense, Activation, Dropout, Flatten, Conv2D, MaxPooling2D,MaxPool2
from keras.layers.normalization import BatchNormalization
import numpy as np

np.random.seed(1000)
```

In [4]:

```
model = Sequential()
model.add(Conv2D(input_shape=(224,224,3),filters=64,kernel_size=(3,3),padding="same", activ
model.add(Conv2D(filters=64,kernel_size=(3,3),padding="same", activation="relu"))
model.add(MaxPool2D(pool_size=(2,2),strides=(2,2)))


model.add(Conv2D(filters=128, kernel_size=(3,3), padding="same", activation="relu"))
model.add(Conv2D(filters=128, kernel_size=(3,3), padding="same", activation="relu"))
model.add(MaxPool2D(pool_size=(2,2),strides=(2,2)))


model.add(Conv2D(filters=256, kernel_size=(3,3), padding="same", activation="relu"))
model.add(Conv2D(filters=256, kernel_size=(3,3), padding="same", activation="relu"))
model.add(Conv2D(filters=256, kernel_size=(3,3), padding="same", activation="relu"))
model.add(MaxPool2D(pool_size=(2,2),strides=(2,2)))


model.add(Conv2D(filters=512, kernel_size=(3,3), padding="same", activation="relu"))
model.add(Conv2D(filters=512, kernel_size=(3,3), padding="same", activation="relu"))
model.add(Conv2D(filters=512, kernel_size=(3,3), padding="same", activation="relu"))
model.add(MaxPool2D(pool_size=(2,2),strides=(2,2)))


model.add(Conv2D(filters=512, kernel_size=(3,3), padding="same", activation="relu"))
model.add(Conv2D(filters=512, kernel_size=(3,3), padding="same", activation="relu"))
model.add(Conv2D(filters=512, kernel_size=(3,3), padding="same", activation="relu"))
model.add(MaxPool2D(pool_size=(2,2),strides=(2,2)))

model.add(Flatten())
model.add(Dense(units=4096,activation="relu"))
model.add(Dense(units=4096,activation="relu"))
model.add(Dense(units=2, activation="softmax"))
```

In [6]:

```
print("The output of this will be the summary of the model which I just created.")
model.summary()
```

```
The output of this will be the summary of the model which I just created.
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_2 (Conv2D)            (None, 224, 224, 64)      1792
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 224, 224, 64)      36928
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 112, 112, 64)      0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 112, 112, 128)     73856
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 112, 112, 128)     147584
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 56, 56, 128)       0
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 56, 56, 256)       295168
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 56, 56, 256)       590080
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 56, 56, 256)       590080
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 28, 28, 256)       0
_________________________________________________________________
conv2d_9 (Conv2D)            (None, 28, 28, 512)       1180160
_________________________________________________________________
conv2d_10 (Conv2D)           (None, 28, 28, 512)       2359808
_________________________________________________________________
conv2d_11 (Conv2D)           (None, 28, 28, 512)       2359808
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 14, 14, 512)       0
_________________________________________________________________
conv2d_12 (Conv2D)           (None, 14, 14, 512)       2359808
_________________________________________________________________
conv2d_13 (Conv2D)           (None, 14, 14, 512)       2359808
_________________________________________________________________
conv2d_14 (Conv2D)           (None, 14, 14, 512)       2359808
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 7, 7, 512)         0
_________________________________________________________________
flatten (Flatten)            (None, 25088)             0
_________________________________________________________________
dense (Dense)                (None, 4096)              102764544
_________________________________________________________________
dense_1 (Dense)              (None, 4096)              16781312
_________________________________________________________________
dense_2 (Dense)              (None, 2)                 8194
=================================================================
Total params: 134,268,738
Trainable params: 134,268,738
Non-trainable params: 0
_________________________________________________________________
```

Here I have started with initialising the model by specifying that the model is a sequential model. After

initialising the model I add

→ 2 x convolution layer of 64 channel of 3x3 kernal and same padding

→ 1 x maxpool layer of 2x2 pool size and stride 2x2

→ 2 x convolution layer of 128 channel of 3x3 kernal and same padding

→ 1 x maxpool layer of 2x2 pool size and stride 2x2

→ 3 x convolution layer of 256 channel of 3x3 kernal and same padding

→ 1 x maxpool layer of 2x2 pool size and stride 2x2

→ 3 x convolution layer of 512 channel of 3x3 kernal and same padding

→ 1 x maxpool layer of 2x2 pool size and stride 2x2

→ 3 x convolution layer of 512 channel of 3x3 kernal and same padding

→ 1 x maxpool layer of 2x2 pool size and stride 2x2

I also add relu(Rectified Linear Unit) activation to each layers so that all the negative values are not passed to the next layer.

After creating all the convolution I pass the data to the dense layer so for that I flatten the vector which comes out of the convolutions and add

→ 1 x Dense layer of 4096 units

→ 1 x Dense layer of 4096 units

→ 1 x Dense Softmax layer of 2 units

I will use RELU activation for both the dense layer of 4096 units so that I stop forwarding negative values through the network. I use a 2 unit dense layer in the end with softmax activation as I have 2 classes to predict from in the end which are dog and cat. The softmax layer will output the value between 0 and 1 based on the confidence of the model that which class the images belongs to.

After the creation of softmax layer the model is finally prepared.

In [ ]:

Introduction

ResNet is a network structure proposed by the He Kaiming, Sun Jian and others of Microsoft Research Asia in 2015, and won the first place in the ILSVRC-2015 classification task. At the same time, it won the first place in ImageNet detection, ImageNet localization, COCO detection, and COCO segmentation tasks. It was a sensation at the time.

ResNet, also known as residual neural network, refers to the idea of adding residual learning to the traditional convolutional neural network, which solves the problem of gradient dispersion and accuracy degradation (training set) in deep networks, so that the network can get more and more The deeper, both the accuracy and the speed are controlled.

> Deep Residual Learning for Image Recognition Original link : ResNet Paper (https://arxiv.org/pdf/1512.03385.pdf)

The problem caused by increasing depth

- The first problem brought by increasing depth is the problem of gradient explosion / dissipation . This is because as the number of layers increases, the gradient of backpropagation in the network will become unstable with continuous multiplication, and become particularly large or special. small. Among them , the problem of gradient dissipation often occurs .
- In order to overcome gradient dissipation, many solutions have been devised, such as using BatchNorm, replacing the activation function with ReLu, using Xaiver initialization, etc. It can be said that gradient dissipation has been well solved
- Another problem of increasing depth is the problem of network degradation, that is, as the depth increases, the performance of the network will become worse and worse, which is directly reflected in the decrease in accuracy on the training set. The residual network article solves this problem. And after this problem is solved, the depth of the network has increased by several orders of magnitude.

Degradation of deep network

> With network depth increasing, accuracy gets saturated (which might be unsurprising) and then degrades rapidly. Unexpectedly, such degradation is not caused by overfitting, and adding more layers to a favored deep model leads to higher training error.

Degradation problem

"with the network depth increasing, accuracy gets saturated"

Not caused by overfitting:

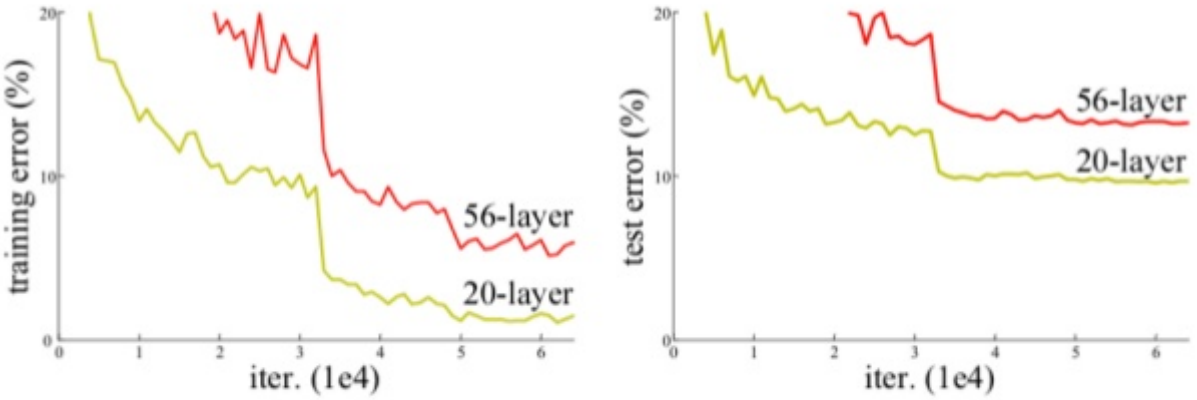


Figure 1. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer "plain" networks. The deeper network has higher training error, and thus test error.

- The above figure is the error rate of the training set classified by the network on the CIFAR10-data set with the increase of the network depth . It can be seen that if we directly stack the convolutional layers, as the number of layers increases, the error rate increases significantly. The trend is that the deepest 56-layer network has the worst accuracy . We verified it on the VGG network. For the CIFAR-10 dataset, it took 5 minutes on the 18-layer VGG network to get the full network training. The 80% accuracy rate was achieved, and the 34-layer VGG model took 8 minutes to get the 72% accuracy rate. The problem of network degradation does exist.
- The decrease in the training set error rate indicates that the problem of degradation is not caused by overfitting. The specific reason is that it is left for further study. The author's other paper "Identity Mappings in Deep Residual Networks" proved the occurrence of degradation. It is because the optimization performance is not good, which indicates that the deeper the network, the more difficult the reverse gradient is to conduct.

Deep Residual Networks

From 10 to 100 layers

We can imagine that *when we simply stack the network directly to a particularly long length, the internal characteristics of the network have reached the best situation in one of the layers. At this time, the remaining layers should not make any changes to the characteristics and learn automatically. The form of identity mapping*. That is to say, for a particularly deep deep network, the solution space of the shallow form of the network should be a subset of the solution space of the deep network, in other words, a network deeper than the shallow network will not have at least Worse effect, but this is not true because of network degradation.

Then, we settle for the second best. In the case of network degradation, if we do not add depth, we can improve the accuracy. Can we at least make the deep network achieve the same performance as the shallow network, that is, let the layers behind the deep network achieve at least The role of identity mapping . Based on this idea, the author proposes a residual module to help the network achieve identity mapping.

To understand ResNet, we must first understand what kind of problems will occur when the network becomes deeper.

The first problem brought by increasing the network depth is the disappearance and explosion of the gradient.

This problem was successfully solved after Szegedy proposed the **BN (Batch Normalization)** structure. The BN layer can normalize the output of each layer. The size can still be kept stable after the reverse layer transfer, and it will not be too small or too large.

Is it easy to converge after adding BN and then increasing the depth?

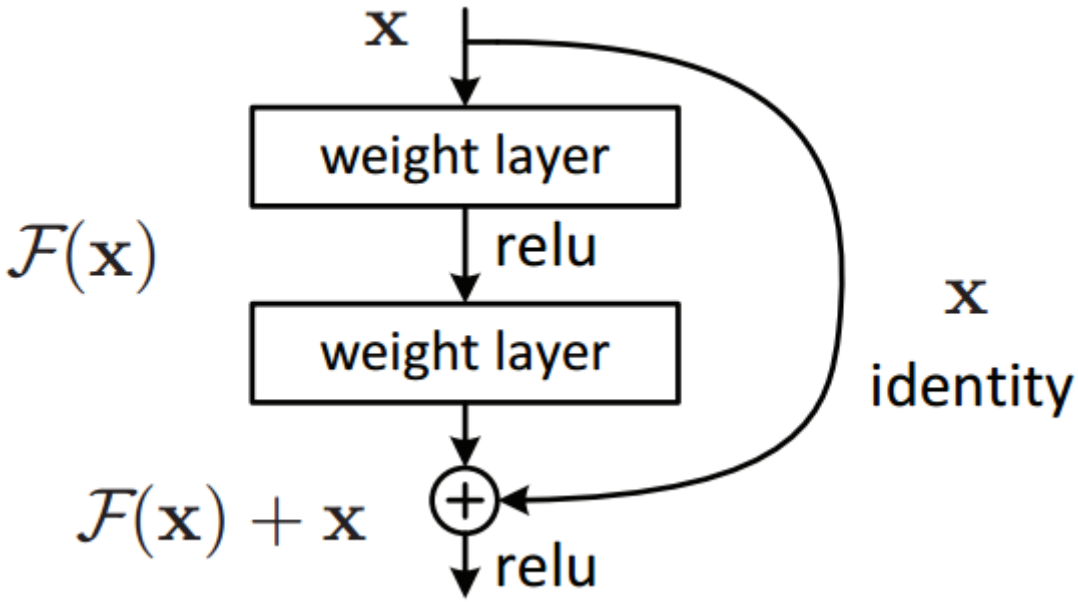
The answer is still **negative**. The author mentioned the second problem-**the degradation problem**: when the level reaches a certain level, the accuracy will saturate and then decline rapidly. This decline is not caused by the disappearance of the gradient. It is not caused by overfit, but because the network is so complicated that it is difficult to achieve the ideal error rate by unconstrained stocking training alone.

The degradation problem is not a problem of the network structure itself, but is caused by the current insufficient training methods. The currently widely used training methods, whether it is SGD, AdaGrad, or RMSProp, cannot reach the theoretically optimal convergence result after the network depth becomes larger.

We can also prove that as long as there is an ideal training method, deeper networks will definitely perform better than shallow networks.

The proof process is also very simple: Suppose that several layers are added behind a network A to form a new network B. If the added level is just an identity mapping of the output of A, that is, the output of A is after the level of B becomes the output of B, there is no change, so the error rates of network A and network B are equal, which proves that the deepened network will not be worse than the network before deepening.

He Kaiming proposed a residual structure to implement the above identity mapping (Below Figure): In addition to the normal convolution layer output, the entire module has a branch directly connecting the input to the output. The output and the output of the convolution do The final output is obtained by arithmetic addition. The formula is H (x) = F (x) + x, x is the input, F (x) is the output of the convolution branch, and H (x) is the output of the entire structure. It can be shown that if all parameters in the F (x) branch are 0, H (x) is an identity mapping. The residual structure artificially creates an identity map, which can make the entire structure converge in the direction of the identity map, ensuring that the final error rate will not become worse because the depth becomes larger. If a network can achieve the desired result by simply setting the parameter values by hand, then this structure can easily converge to the result through training. This is a rule that is unsuccessful when designing complex networks. Recall that in order to restore the original distribution after BN processing, the formula y = rx + delta is used. When r is manually set to standard deviation and delta is the mean, y is the distribution before BN processing. This is the use of this Rules.



What does residual learning mean?

The idea of residual learning is the above picture, which can be understood as a block, defined as follows:

$$y = F(x, \{W_i\}) + x$$

The residual learning block contains two branches or two mappings:

1. Identity mapping refers to the curved curve on the right side of the figure above. As its name implies, identity mapping refers to its own mapping, which is x itself;

1. *F*(x) Residual mapping refers to another branch, that is, part. This part is called residual mapping (y -x) .

What role does the residual module play in back propagation?

- The residual module will significantly reduce the parameter value in the module, so that the parameters in the network have a more sensitive response ability to the loss of reverse conduction, although the fundamental It does not solve the problem that the loss of backhaul is too small, but it reduces the parameters. Relatively speaking, it increases the effect of backhaul loss and also generates a certain regularization effect.
- Secondly, because there are branches of the identity mapping in the forward process, the gradient conduction in the back-propagation process also has more simple paths , and the gradient can be transmitted to the previous module after only one relu.
- The so-called backpropagation is that the network outputs a value, and then compares it with the real value to an error loss. At the same time, the loss is changed to change the parameter. The returned loss depends on the original loss and gradient. Since the purpose is to change the parameter, The problem is that if the intensity of changing the parameter is too small, the value of the parameter can be reduced, so that the loss of the intensity of changing the parameter is relatively greater.
- Therefore, the most important role of the residual module is to change the way of forward and backward information transmission, thereby greatly promoting the optimization of the network.
- Using the four criteria proposed by Inceptionv3, we will use them again to improve the residual module. Using criterion 3, the dimensionality reduction before spatial aggregation will not cause information loss, so the same method is also used here, adding 1 * 1 convolution The kernel is used to increase the non-linearity and reduce the depth of the output to reduce the computational cost. You get the form of a residual module that becomes a bottleneck. The figure above shows the basic form on the left and the bottleneck form on the right.

- To sum up, the shortcut module will help the features in the network perform identity mapping in the forward process, and help conduct gradients in the reverse process, so that deeper models can be successfully trained.

Why can the residual learning solve the problem of "the accuracy of the network deepening declines"?

For a neural network model, if the model is optimal, then training can easily optimize the residual mapping to 0, and only identity mapping is left at this time. No matter how you increase the depth, the network will always be in an optimal state in theory. Because it is equivalent to all the subsequent added networks to carry information transmission along the identity mapping (self), it can be understood that the number of layers behind the optimal network is discarded (without the ability to extract features), and it does not actually play a role. . In this way, the performance of the network will not decrease with increasing depth.

The author used two types of data, **ImageNet** and **CIFAR**, to prove the effectiveness of ResNet:

The first is ImageNet. The authors compared the training effect of ResNet structure and traditional structure with the same number of layers. The left side of Figure is a VGG-19 network with a traditional structure (each followed by BN), the middle is a 34-layer network with a traditional structure (each followed by BN), and the right side is 34 layers ResNet (the solid line indicates a direct connection, and the dashed line indicates a dimensional change using 1x1 convolution to match the number of features of the input and output). Figure 3 shows the results after training these types of networks.

The data on the left shows that the 34-layer network (red line) with the traditional structure has a higher error rate than the VGG-19 (blue-green line). Because the BN structure is added to each layer Therefore, the high error is not caused by the gradient disappearing after the level is increased, but by the degradation problem; the ResNet structure on the right side of Figure 3 shows that the 34-layer network (red line) has a higher error rate than the 18-layer network (blue-green line). Low, this is because the ResNet structure has overcome the degradation problem. In addition, the final error rate of the ResNet 18-layer network on the right is similar to the error rate of the traditional 18-layer network on the left. This is because the 18-layer network is simpler and can converge to a more ideal result even without the ResNet structure.

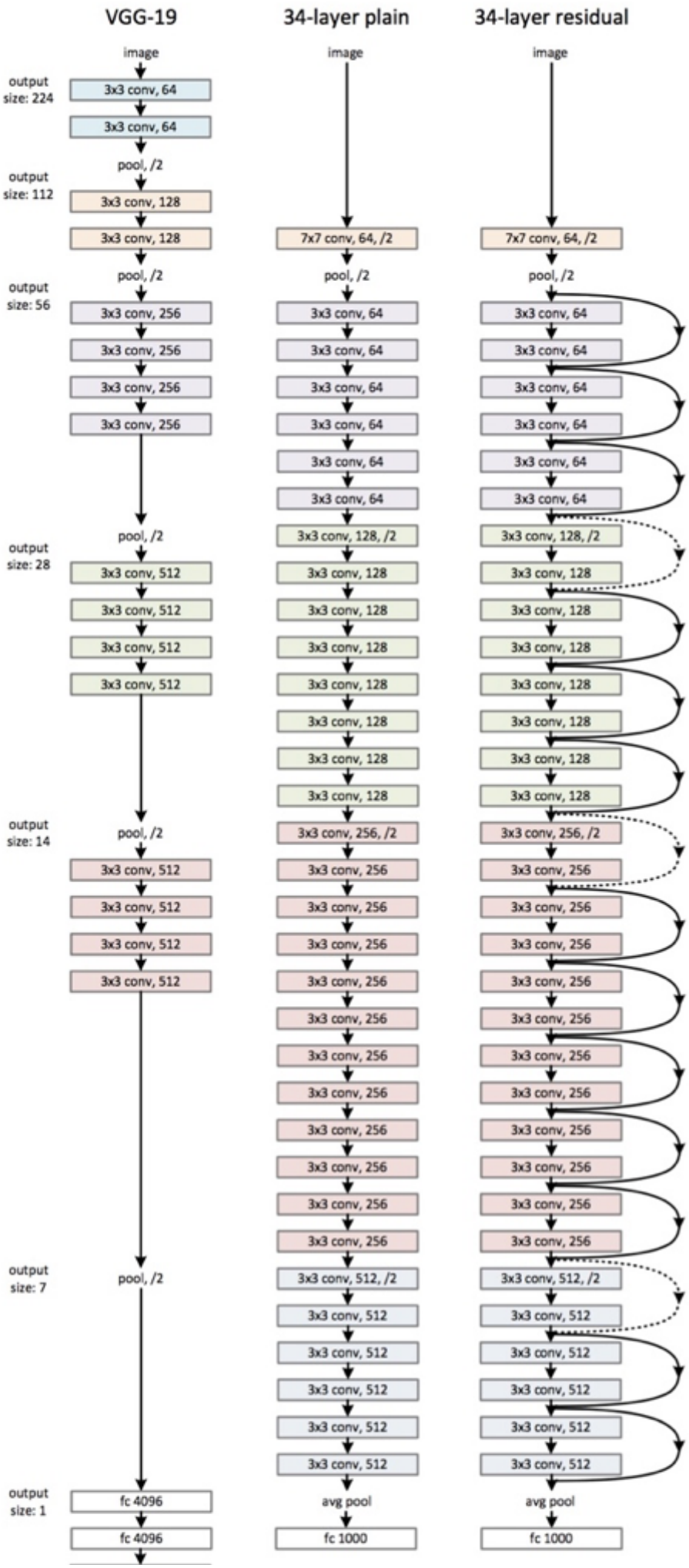

VGG-19
34-layer plain
34-layer residual
image
output size: 224
3x3 conv, 64
3x3 conv, 64
pool, /2
output size: 112
3x3 conv, 128
3x3 conv, 128
pool, /2
output size: 56
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
pool, /2
output size: 28
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
pool, /2
output size: 14
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
pool, /2
output size: 7
fc 4096
fc 4096
output size: 1
7x7 conv, 64, /2
pool, /2
3x3 conv, 64
3x3 conv, 128, /2
3x3 conv, 128
3x3 conv, 256, /2
3x3 conv, 256
3x3 conv, 512, /2
3x3 conv, 512
avg pool
fc 1000

Figure 3. Example network architectures for ImageNet. **Left**: the VGG-19 model [41] (19.6 billion FLOPs) as a reference. **Middle**: a plain network with 34 parameter layers (3.6 billion FLOPs). **Right**: a residual network with 34 parameter layers (3.6 billion FLOPs). The dotted shortcuts increase dimensions. **Table 1** shows more details and other variants.

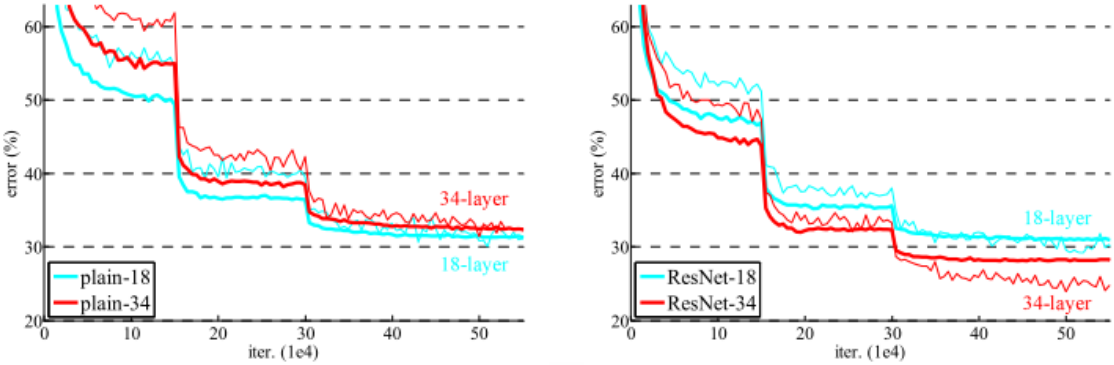
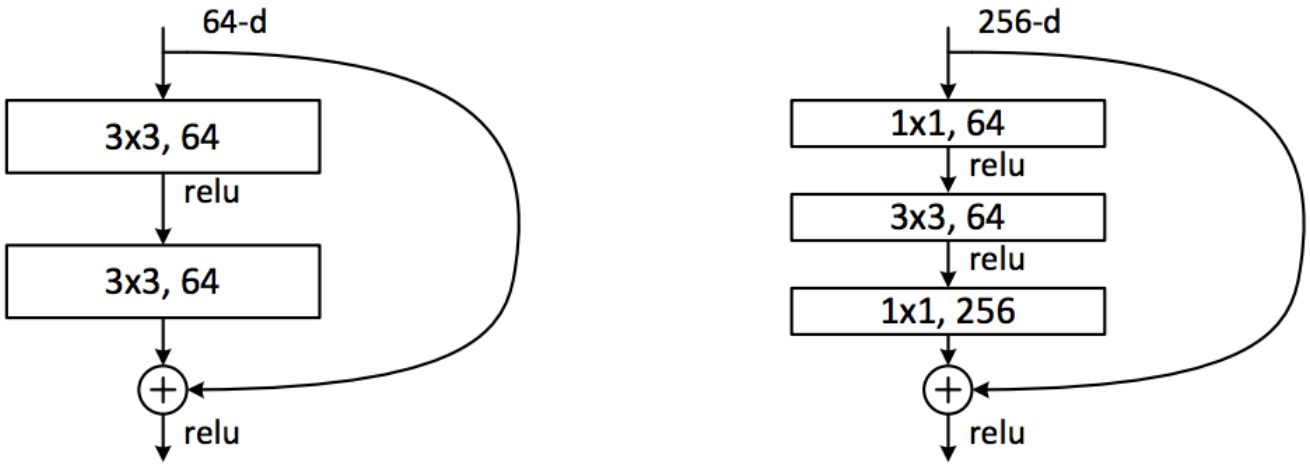


Figure 4. Training on **ImageNet**. Thin curves denote training error, and bold curves denote validation error of the center crops. Left: plain networks of 18 and 34 layers. Right: ResNets of 18 and 34 layers. In this plot, the residual networks have no extra parameter compared to their plain counterparts.

The ResNet structure like the left side of Fig. 4 is only used for shallow ResNet networks. If there are many network layers, the dimensions near the output end of the network will be very large. Still using the structure on the left side of Fig. 4 will cause a huge amount of calculation. For deeper networks, we all use the bottleneck structure on the right side of Figure 4, first using a 1x1 convolution for dimensionality reduction, then 3x3 convolution, and finally using 1x1 dimensionality to restore the original dimension.

In practice, considering the cost of the calculation, the residual block is calculated and optimized, that is, the two 3x3 convolution layers are replaced with 1x1 + 3x3 + 1x1 , as shown below. The middle 3x3 convolutional layer in the new structure first reduces the calculation under one dimensionality-reduced 1x1 convolutional layer , and then restores it under another 1x1 convolutional layer , both maintaining accuracy and reducing the amount of calculation .



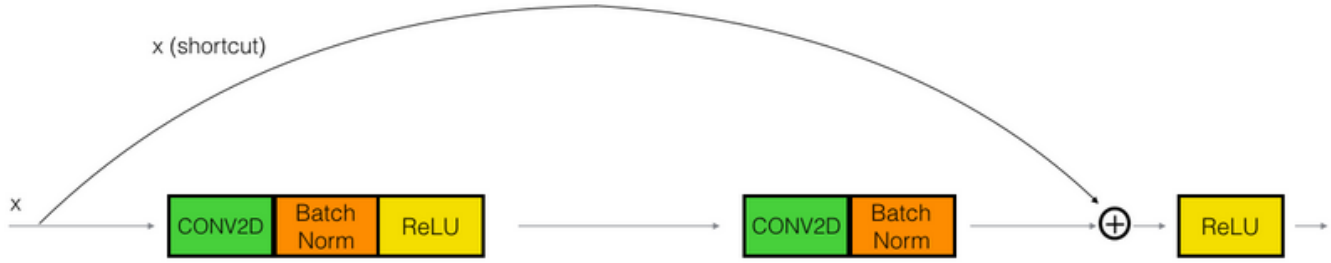
This is equivalent to reducing the amount of parameters for the same number of layers , so it can be extended to deeper models. So the author proposed ResNet with 50, 101 , and 152 layers , and not only did not have degradation problems, the error rate was greatly reduced, and the computational complexity was also kept at a very low level .

At this time, the error rate of ResNet has already dropped other networks a few streets, but it does not seem to be satisfied. Therefore, a more abnormal 1202 layer network has been built. For such a deep network, optimization is still not difficult, but it appears The problem of overfitting is quite normal. The author also said that the 1202 layer model will be further improved in the future.

There are two main types of blocks are used in a ResNet, depending mainly on whether the input/output dimensions are the same or different.

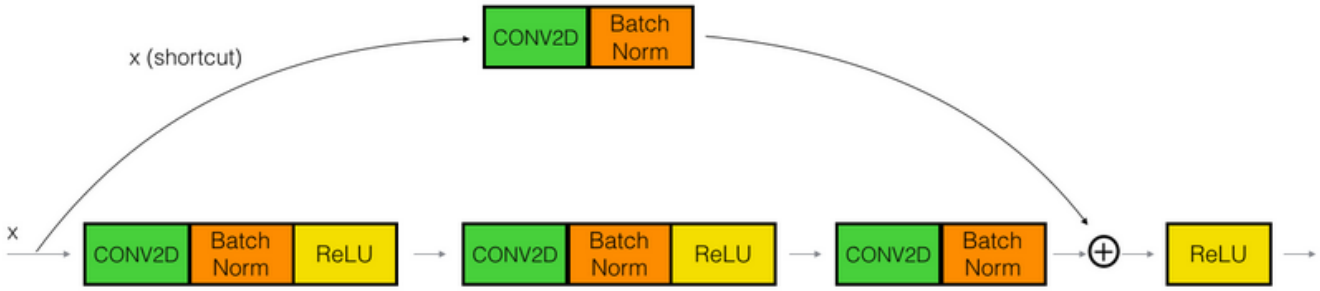
1. Identity Block

The identity block is the standard block used in ResNets and corresponds to the case where the input activation has the same dimension as the output activation.



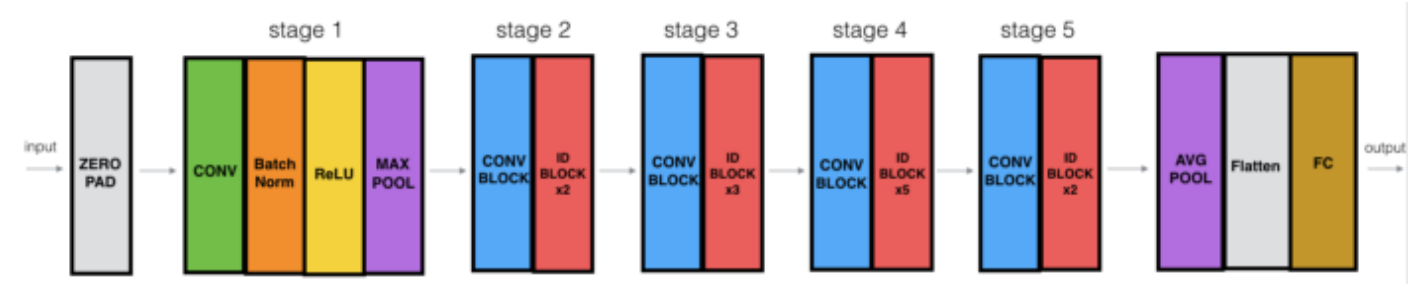
2. Convolutional Block

We can use this type of block when the input and output dimensions don't match up. The difference with the identity block is that there is a CONV2D layer in the shortcut path.



ResNet-50

The ResNet-50 model consists of 5 stages each with a convolution and Identity block. Each convolution block has 3 convolution layers and each identity block also has 3 convolution layers. The ResNet-50 has over 23 million trainable parameters.



Diffrent Variants : -

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Below is the transcript of resnet, winning the championship at ImageNet2015

method	top-5 err. (**test**)
VGG [40] (ILSVRC'14)	7.32
GoogLeNet [43] (ILSVRC'14)	6.66
VGG [40] (v5)	6.8
PReLU-net [12]	4.94
BN-inception [16]	4.82
ResNet (ILSVRC'15)	**3.57**

Code Implement

In [1]:

```
import cv2
import numpy as np
import os
from keras.preprocessing.image import ImageDataGenerator
from keras import backend as K
import keras
from keras.models import Sequential, Model,load_model
from keras.optimizers import SGD
from keras.callbacks import EarlyStopping,ModelCheckpoint
from keras.layers import Input, Add, Dense, Activation, ZeroPadding2D, BatchNormalization,
from keras.preprocessing import image
from keras.initializers import glorot_uniform
```

In [2]:

```
def identity_block(X, f, filters, stage, block):

    conv_name_base = 'res' + str(stage) + block + '_branch'
    bn_name_base = 'bn' + str(stage) + block + '_branch'
    F1, F2, F3 = filters

    X_shortcut = X

    X = Conv2D(filters=F1, kernel_size=(1, 1), strides=(1, 1), padding='valid', name=conv_r
    X = BatchNormalization(axis=3, name=bn_name_base + '2a')(X)
    X = Activation('relu')(X)

    X = Conv2D(filters=F2, kernel_size=(f, f), strides=(1, 1), padding='same', name=conv_na
    X = BatchNormalization(axis=3, name=bn_name_base + '2b')(X)
    X = Activation('relu')(X)

    X = Conv2D(filters=F3, kernel_size=(1, 1), strides=(1, 1), padding='valid', name=conv_r
    X = BatchNormalization(axis=3, name=bn_name_base + '2c')(X)

    X = Add()([X, X_shortcut])# SKIP Connection
    X = Activation('relu')(X)

    return X
```

In [3]:

```
def convolutional_block(X, f, filters, stage, block, s=2):

    conv_name_base = 'res' + str(stage) + block + '_branch'
    bn_name_base = 'bn' + str(stage) + block + '_branch'

    F1, F2, F3 = filters

    X_shortcut = X

    X = Conv2D(filters=F1, kernel_size=(1, 1), strides=(s, s), padding='valid', name=conv_r
    X = BatchNormalization(axis=3, name=bn_name_base + '2a')(X)
    X = Activation('relu')(X)

    X = Conv2D(filters=F2, kernel_size=(f, f), strides=(1, 1), padding='same', name=conv_na
    X = BatchNormalization(axis=3, name=bn_name_base + '2b')(X)
    X = Activation('relu')(X)

    X = Conv2D(filters=F3, kernel_size=(1, 1), strides=(1, 1), padding='valid', name=conv_r
    X = BatchNormalization(axis=3, name=bn_name_base + '2c')(X)

    X_shortcut = Conv2D(filters=F3, kernel_size=(1, 1), strides=(s, s), padding='valid', na
    X_shortcut = BatchNormalization(axis=3, name=bn_name_base + '1')(X_shortcut)

    X = Add()([X, X_shortcut])
    X = Activation('relu')(X)

    return X
```

In [4]:

```
def ResNet50(input_shape=(224, 224, 3)):

    X_input = Input(input_shape)

    X = ZeroPadding2D((3, 3))(X_input)

    X = Conv2D(64, (7, 7), strides=(2, 2), name='conv1', kernel_initializer=glorot_uniform(
    X = BatchNormalization(axis=3, name='bn_conv1')(X)
    X = Activation('relu')(X)
    X = MaxPooling2D((3, 3), strides=(2, 2))(X)

    X = convolutional_block(X, f=3, filters=[64, 64, 256], stage=2, block='a', s=1)
    X = identity_block(X, 3, [64, 64, 256], stage=2, block='b')
    X = identity_block(X, 3, [64, 64, 256], stage=2, block='c')


    X = convolutional_block(X, f=3, filters=[128, 128, 512], stage=3, block='a', s=2)
    X = identity_block(X, 3, [128, 128, 512], stage=3, block='b')
    X = identity_block(X, 3, [128, 128, 512], stage=3, block='c')
    X = identity_block(X, 3, [128, 128, 512], stage=3, block='d')

    X = convolutional_block(X, f=3, filters=[256, 256, 1024], stage=4, block='a', s=2)
    X = identity_block(X, 3, [256, 256, 1024], stage=4, block='b')
    X = identity_block(X, 3, [256, 256, 1024], stage=4, block='c')
    X = identity_block(X, 3, [256, 256, 1024], stage=4, block='d')
    X = identity_block(X, 3, [256, 256, 1024], stage=4, block='e')
    X = identity_block(X, 3, [256, 256, 1024], stage=4, block='f')

    X = X = convolutional_block(X, f=3, filters=[512, 512, 2048], stage=5, block='a', s=2)
    X = identity_block(X, 3, [512, 512, 2048], stage=5, block='b')
    X = identity_block(X, 3, [512, 512, 2048], stage=5, block='c')

    X = AveragePooling2D(pool_size=(2, 2), padding='same')(X)

    model = Model(inputs=X_input, outputs=X, name='ResNet50')

    return model
```

In [6]:

```
base_model = ResNet50(input_shape=(224, 224, 3))
```

In [7]:

```
headModel = base_model.output
headModel = Flatten()(headModel)
headModel=Dense(256, activation='relu', name='fc1',kernel_initializer=glorot_uniform(seed=0
headModel=Dense(128, activation='relu', name='fc2',kernel_initializer=glorot_uniform(seed=0
headModel = Dense( 1,activation='sigmoid', name='fc3',kernel_initializer=glorot_uniform(see
```

In [8]:

```
model = Model(inputs=base_model.input, outputs=headModel)
```

In [9]:

```
model.summary()
```

```
Model: "model"
__________________________________________________________________________________
__________________________
Layer (type)                    Output Shape         Param #     Connected
to
==================================================================================
==========================
input_1 (InputLayer)            [(None, 224, 224, 3) 0
__________________________________________________________________________________
__________________________
zero_padding2d (ZeroPadding2D)  (None, 230, 230, 3)  0           input_1
[0][0]
__________________________________________________________________________________
__________________________
conv1 (Conv2D)                  (None, 112, 112, 64) 9472        zero_padd
ing2d[0][0]
__________________________________________________________________________________
__________________________
bn_conv1 (BatchNormalization)   (None, 112, 112, 64) 256         conv1[0]
[0]
```

In [ ]:

```

```

Introduction

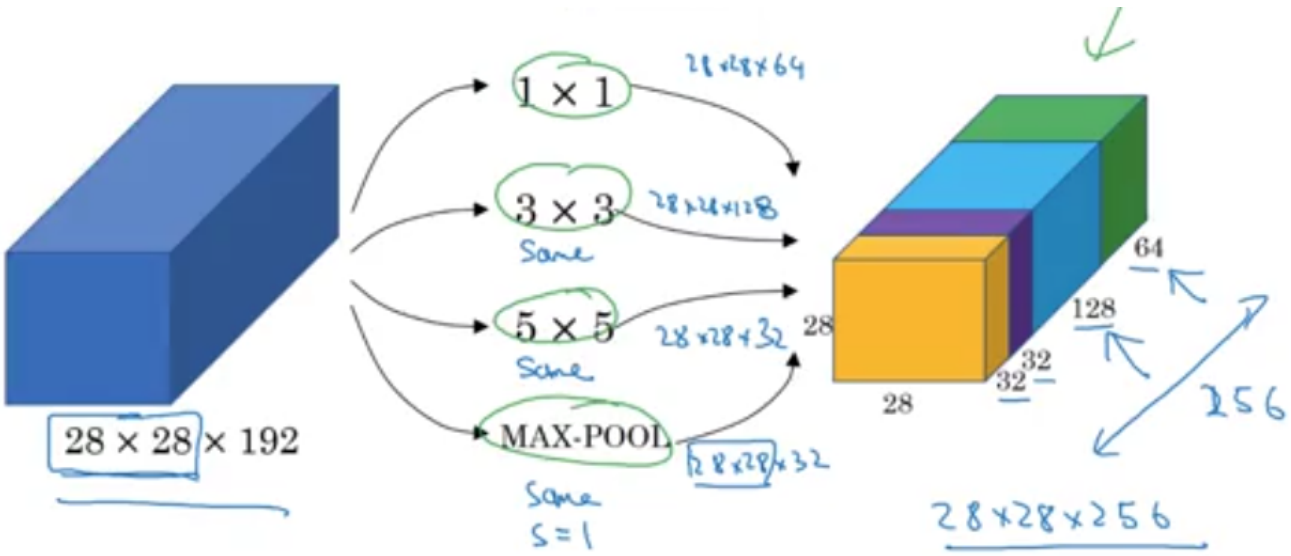
- Inception net achieved a milestone in CNN classifiers when previous models were just going deeper to improve the performance and accuracy but compromising the computational cost. The Inception network, on the other hand, is heavily engineered.
- It uses a lot of tricks to push performance, both in terms of speed and accuracy. It is the winner of the ImageNet Large Scale Visual Recognition Competition in 2014, an image classification competition, which has a significant improvement over ZFNet (The winner in 2013), AlexNet (The winner in 2012) and has relatively lower error rate compared with the VGGNet (1st runner-up in 2014).

The major issues faced by deeper CNN models such as VGGNet were:

1. Although, previous networks such as VGG achieved a remarkable accuracy on the ImageNet dataset, deploying these kinds of models is highly computationally expensive because of the deep architecture.
2. Very deep networks are susceptible to overfitting. It is also hard to pass gradient updates through the entire network.

Motivation for Inception Network

Instead of deciding whether to use a 1x1 convolution, or a 3x3 or a 5x5 Convolution, or whether to use a Pooling layer - Why not use all of them?

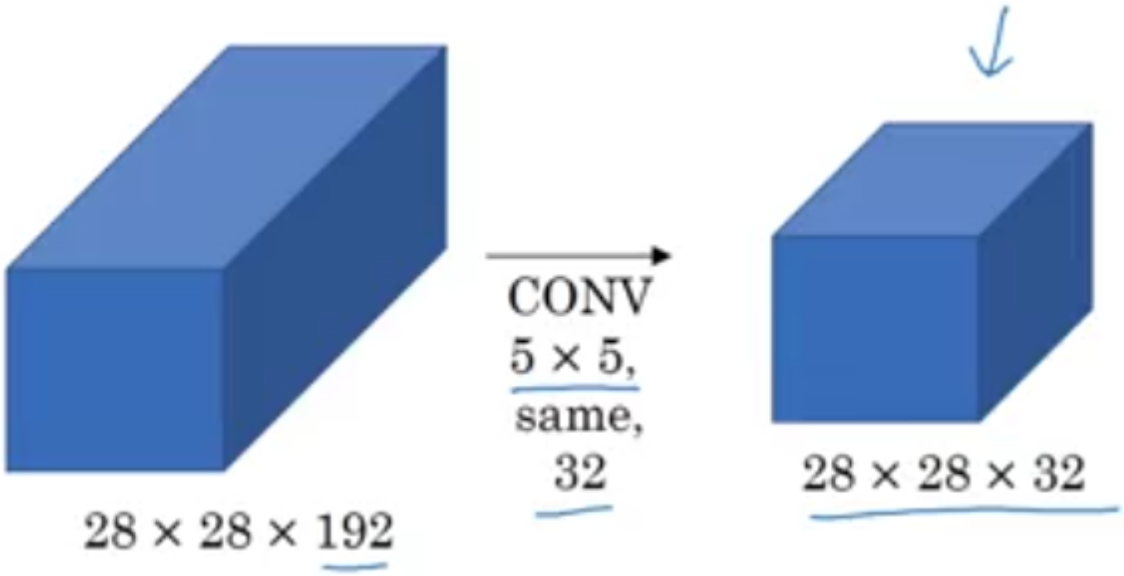


In the example above, all the filters are applied to the input to generate a stacked outpu, whioch contains the output of each filter stacked on top of each other. The Padding is kept at ‘same’ to ensure that the output from all the filters are of the same size.

Disadvantage: Huge memory cost

Solving the problem of memory cost

For example, the computational cost of th 5x5 filer in the above diagram:



Input: 28x28x192

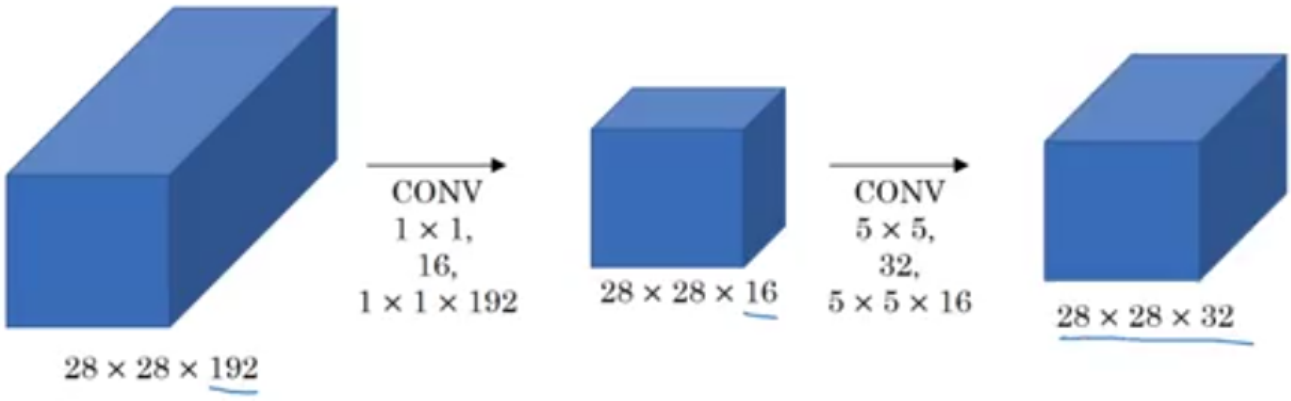
Filter: Conv 5x5x192, same, 32

Output: 28x28x32

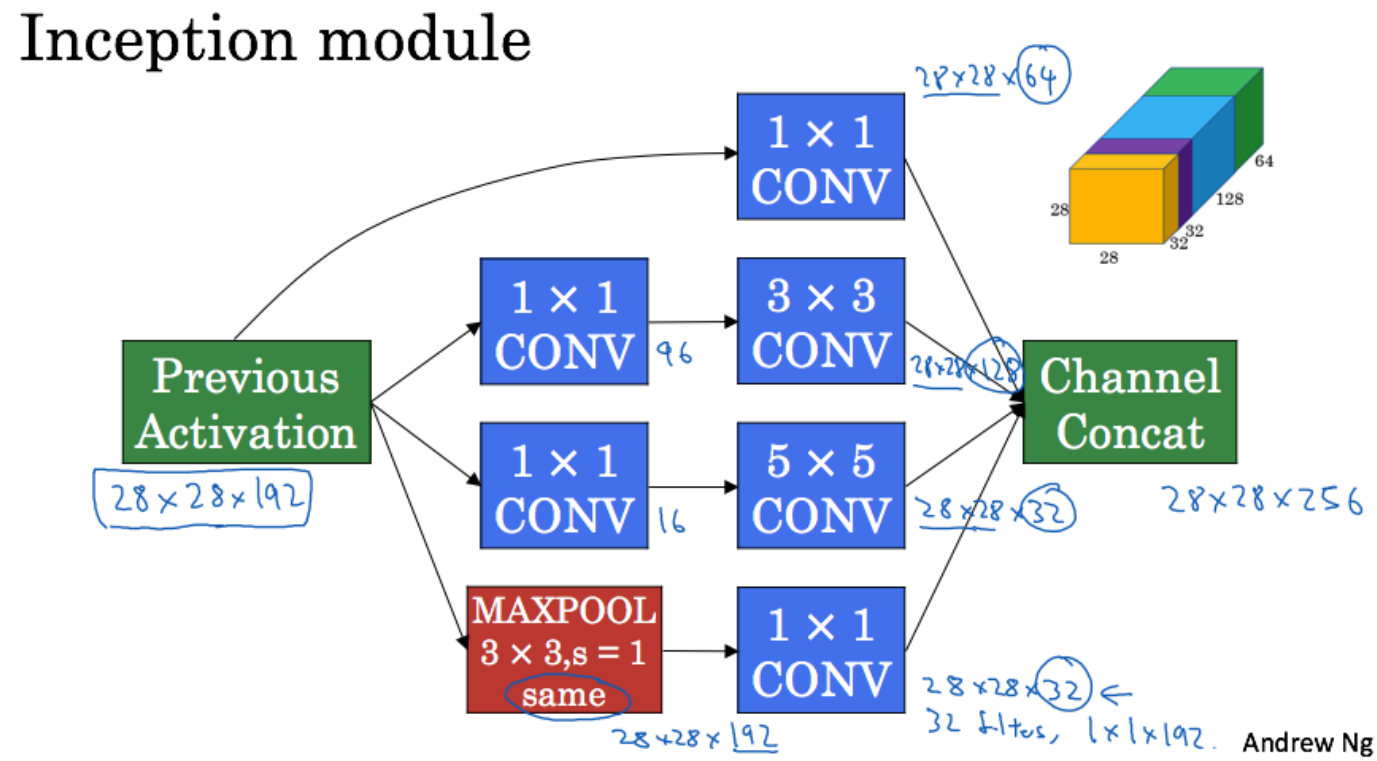
Total number of calculations = (28 *28* 32) *(5* 5 * 192) = 120 Million !!

Using 1x1 Convolution to reduce computation cost

A 1x1 convolution is added before the 5x5 cvonvolution -= Also called a bottleneck layer



Total number of calculations = [(28 *28* 16) *(1* 1 *192)]* + *[(28* 28 *32)* (5 *5* 16)] = 12.4 Million !! (earlier the cost was 120 Million)



The popular versions are as follows:

Inception v1.

Inception v2 and Inception v3.

Inception v4 and Inception-ResNet.

Inception V1

- This architecture has 22 layers in total! Using the dimension-reduced inception module, a neural network architecture is constructed. This is popularly known as GoogLeNet (Inception v1).
- GoogLeNet has 9 such inception modules fitted linearly. It is 22 layers deep (27, including the pooling layers). At the end of the architecture, fully connected layers were replaced by a global average pooling which calculates the average of every feature map. This indeed dramatically declines the total number of parameters.
- The above are the explainof V1

Problems of Inception V1 architecture:

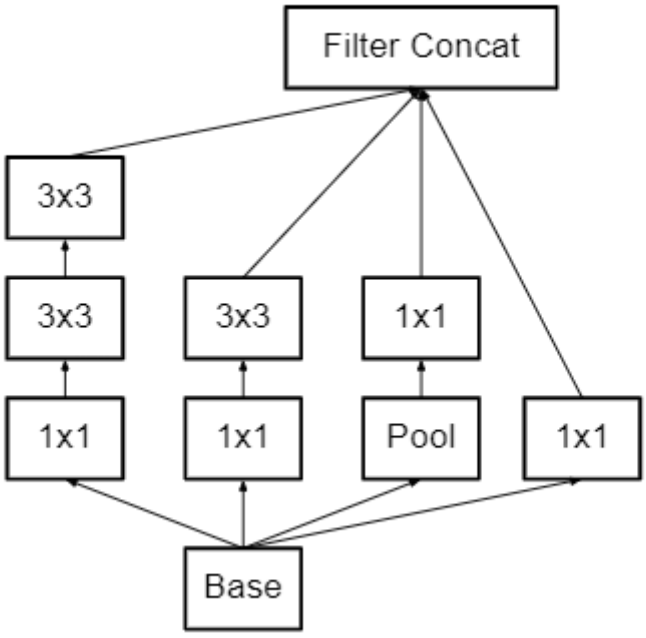
- Inception V1 have sometimes use convolutions such as 5*5 that causes the input dimensions to decrease by a large margin. This causes the neural network some accuracy decrease. The reason behind that the neural network is susceptible to information loss if the input dimension decreases too drastically.
- Furthermore, there is also complexity decrease when we use bigger convolutions like 5×5 as compared to 3×3.We can go further in terms of factorization i.e. that we can divide a 3×3 convolution into an asymmetric convolution of 1×3 then followed by 3×1 convolution. This is equivalent to sliding a two-layer network with the same receptive field as in a 3×3 convolution but 33% more cheaper than 3×3. This factorization does not work well for early layers when input dimensions are big but only when the input size mxm (m is between 12 and 20). According to the Inception V1 architecture, the auxiliary classifier improves the convergence of the network. They argue that it can help reduce the effect of the vanishing gradient problem in deep network by pushing the useful gradient to earlier layers (to reduce the loss). But, the authors of this paper found that this classifier didn't improve the convergence very much early in the training.

Inception V2 And Inception V3

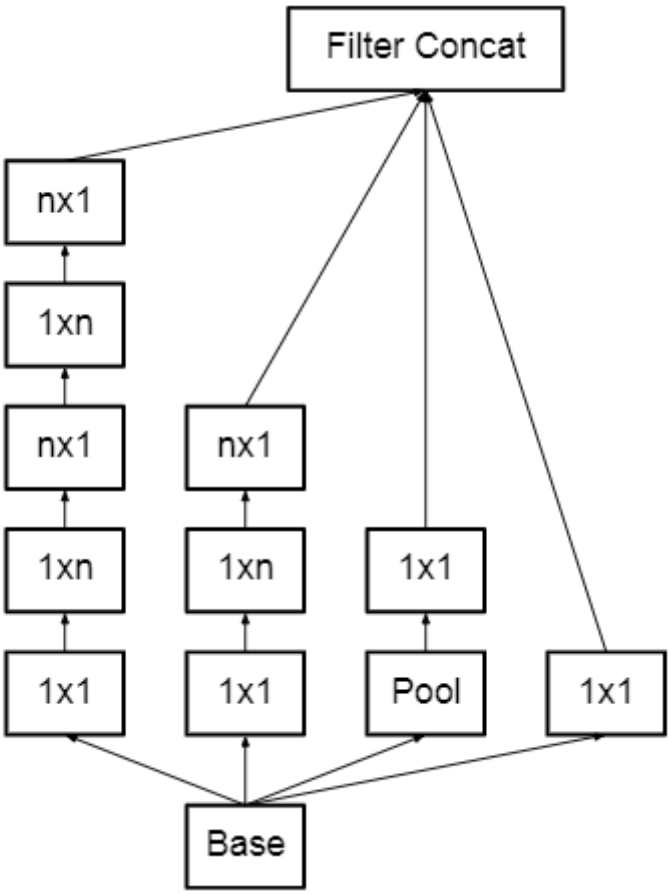
Inception-v2(2015)

Architectural Changes in Inception V2:

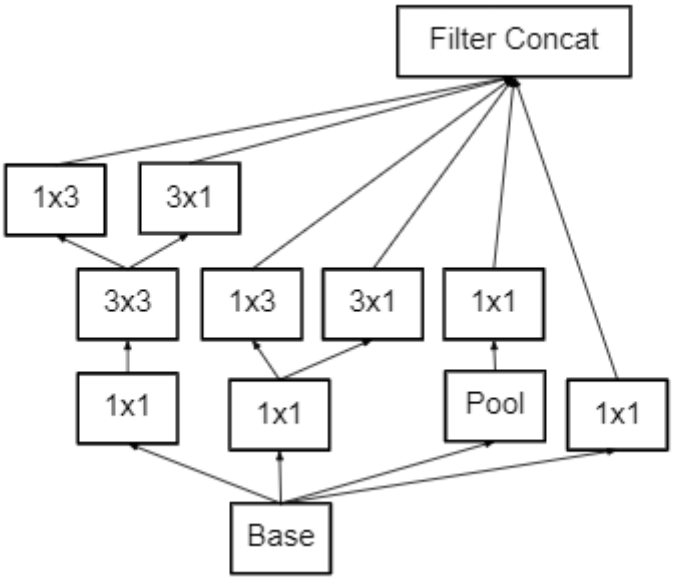
In the Inception V2 architecture. The 5×5 convolution is replaced by the two 3×3 convolutions. This also decreases computational time and thus increase computational speed because a 5×5 convolution is 2.78 more expensive than 3×3 convolution. So, Using two 3×3 layers instead of 5×5 increases the performance of architecture.



This architecture also converts nXn factorization into 1xn and nx1 factorization. As we discuss above that a 3×3 convolution can be converted into 1×3 then followed by 3×1 convolution which is 33% cheaper in terms of computational complexity as compared to 3×3.



To deal with the problem of the representational bottleneck, the feature banks of the module were expanded instead of making it deeper. This would prevent the loss of information that causes when we make it deeper.



The above three principles were used to build three different types of inception modules (Let's call them modules A,B and C in the order they were introduced. These names are introduced for clarity, and not the official names). The architecture is as follows:

type	patch size/stride or remarks	input size
conv	$3\times3/2$	$299\times299\times3$
conv	$3\times3/1$	$149\times149\times32$
conv padded	$3\times3/1$	$147\times147\times32$
pool	$3\times3/2$	$147\times147\times64$
conv	$3\times3/1$	$73\times73\times64$
conv	$3\times3/2$	$71\times71\times80$
conv	$3\times3/1$	$35\times35\times192$
$3\times$Inception	As in figure 5	$35\times35\times288$
$5\times$Inception	As in figure 6	$17\times17\times768$
$2\times$Inception	As in figure 7	$8\times8\times1280$
pool	8×8	$8\times8\times2048$
linear	logits	$1\times1\times2048$
softmax	classifier	$1\times1\times1000$

Algorithm advantages:

1. **Improved learning rate** : In the BN model, a higher learning rate is used to accelerate training convergence, but it will not cause other effects. Because if the scale of each layer is different, then the learning rate required by each layer is different. The scale of the same layer dimension often also needs different learning rates. Usually, the minimum learning is required to ensure the loss function to decrease, but The BN layer keeps the scale of each layer and dimension consistent, so you can directly use a higher learning rate for optimization.

1. **Remove the dropout layer** : The BN layer makes full use of the goals of the dropout layer. Remove the dropout layer from the BN-Inception model, but no overfitting will occur.

1. **Decrease the attenuation coefficient of L2 weight** : Although the L2 loss controls the overfitting of the Inception model, the loss of weight has been reduced by five times in the BN-Inception model.

1. **Accelerate the decay of the learning rate** : When training the Inception model, we let the learning rate decrease exponentially. Because our network is faster than Inception, we will increase the speed of reducing the learning rate by 6 times.

1. **Remove the local response layer** : Although this layer has a certain role, but after the BN layer is added, this layer is not necessary.

1. **Scramble training samples more thoroughly** : We scramble training samples, which can prevent the same samples from appearing in a mini-batch. This can improve the accuracy of the validation set by 1%, which is the advantage of the BN layer as a regular term. In our method, random selection is more effective when the model sees different samples each time.

1. **To reduce image distortion**: Because BN network training is faster and observes each training sample less often, we want the model to see a more realistic image instead of a distorted image.

Inception-v3-2015

This architecture focuses, how to use the convolution kernel two or more smaller size of the convolution kernel to replace, but also the introduction of **asymmetrical layers i.e. a convolution dimensional convolution** has also been proposed for pooling layer Some remedies that can cause loss of spatial information; there are ideas such as **label-smoothing , BN-ahxiliary** .

Experiments were performed on inputs with different resolutions . The results show that although low-resolution inputs require more time to train, the accuracy and high-resolution achieved are not much different.

The computational cost is reduced while improving the accuracy of the network.

General Design Principles

We will describe some design principles that have been proposed through extensive experiments with different architectural designs for convolutional networks. At this point, full use of the following principles can be guessed, and some additional experiments in the future will be necessary to estimate their accuracy and effectiveness.

1. **Prevent bottlenecks in characterization** . The so-called bottleneck of feature description is that a large proportion of features are compressed in the middle layer (such as using a pooling operation). This operation will cause the loss of feature space information and the loss of features. Although the operation of pooling in CNN is important, there are some methods that can be used to avoid this loss as much as possible (I note: later hole convolution operations).
2. **The higher the dimensionality of the feature, the faster the training converges** . That is, the independence of features has a great relationship with the speed of model convergence. The more independent features, the more thoroughly the input feature information is decomposed. It is easier to converge if the correlation is strong. Hebbin principle : fire together, wire together.
3. **Reduce the amount of calculation through dimensionality reduction** . In v1, the feature is first reduced by 1x1 convolutional dimensionality reduction. There is a certain correlation between different dimensions. Dimension reduction can be understood as a lossless or low-loss compression. Even if the dimensions are reduced, the correlation can still be used to restore its original information.
4. **Balance the depth and width of the network** . Only by increasing the depth and width of the network in the same proportion can the performance of the model be maximized.

Inception V3 is similar to and contains all the features of Inception V2 with following changes/additions:

```
    1. Use of RMSprop optimizer.
    2. Batch Normalization in the fully connected layer of Auxiliary classifier.
    3. Use of 7×7 factorized Convolution
    4. Label Smoothing Regularization: It is a method to regularize the classifier
by estimating the effect of label-dropout during training. It prevents the classif
ier to predict too confidently a class. The addition of label smoothing gives 0.2%
improvement from the error rate.
```

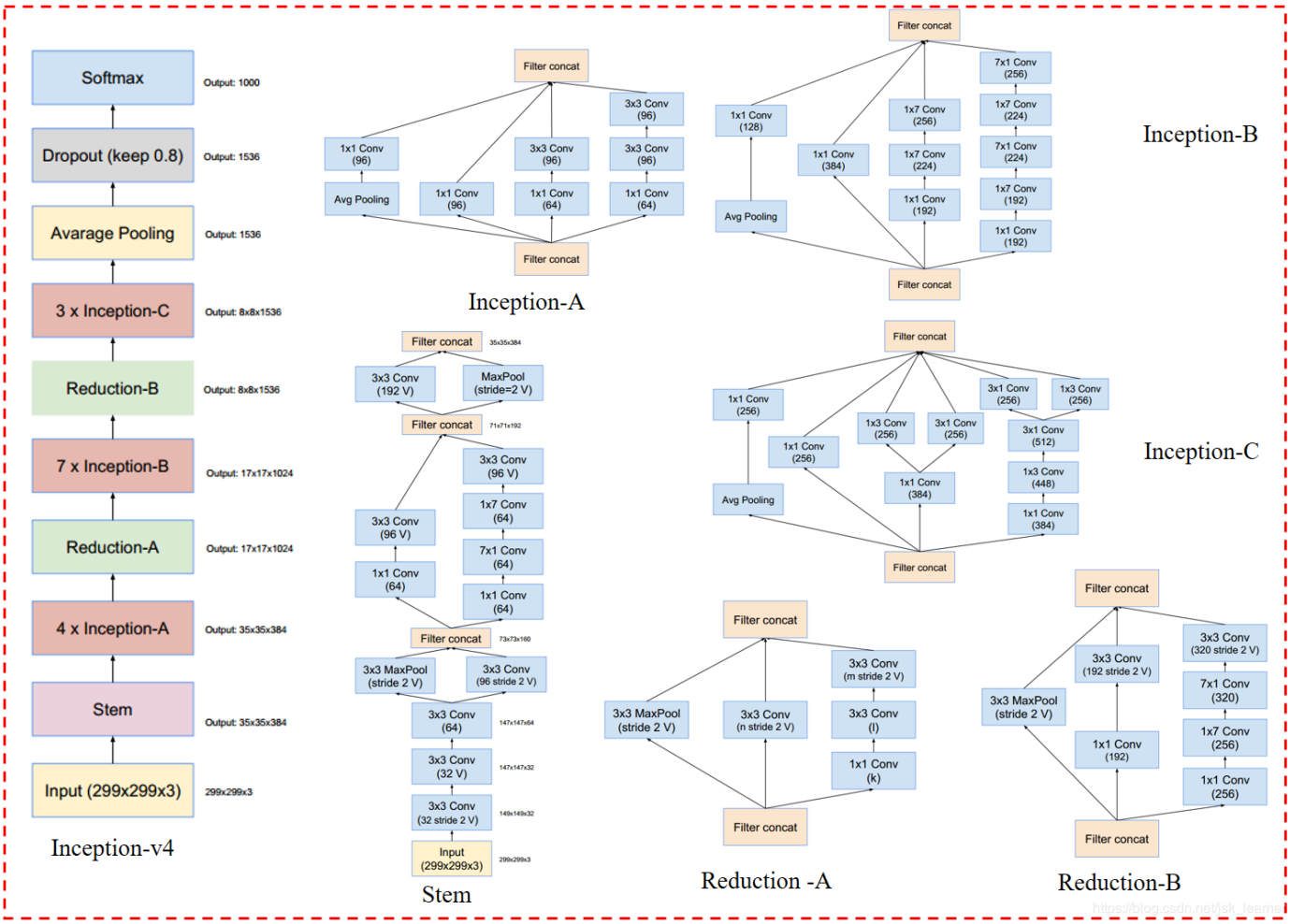
Inception-v4-2016

- Inception V4 was introduced in combination with Inception-ResNet by thee researchers a Google in 2016. The main aim of the paper was to reduce the complexity of Inception V3 model which give the state-of-the-art accuracy on ILSVRC 2015 challenge. This paper also explores the possibility of using residual networks on Inception model.

Introduction

Residual conn works well when training very deep networks. Because the Inception network architecture can be very deep, it is reasonable to use residual conn instead of concat.

Compared with v3, Inception-v4 has more unified simplified structure and more inception modules.



The big picture of Inception-v4:

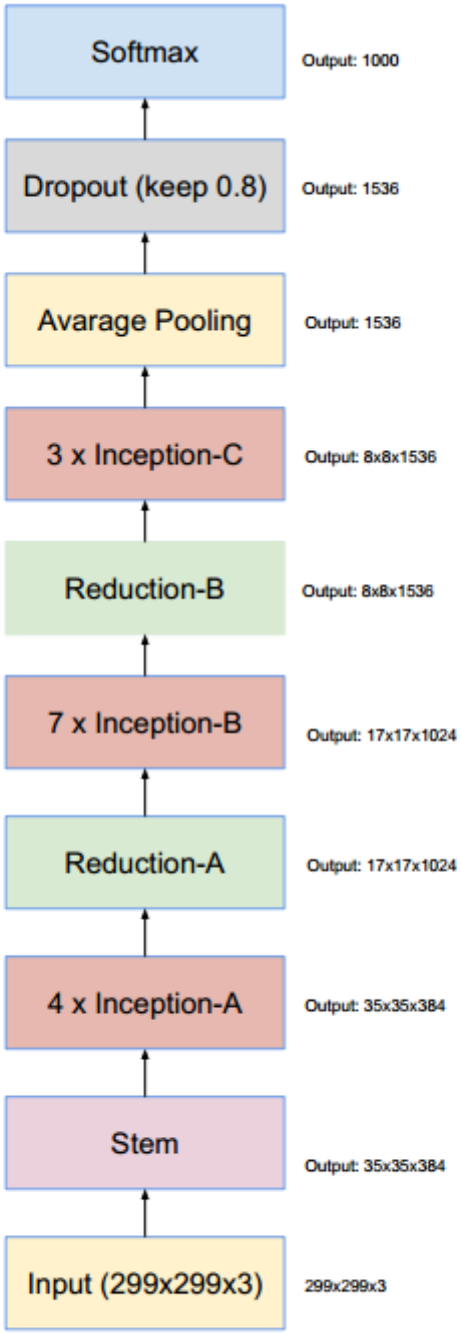


Figure 9. The overall schema of the Inception-v4 network. For the detailed modules, please refer to Figures 3, 4, 5, 6, 7 and 8 for the detailed structure of the various components.

Fig9 is an overall picture, and Fig3,4,5,6,7,8 are all local structures. For the specific structure of each module, see the end of the article.

Residual Inception Blocks

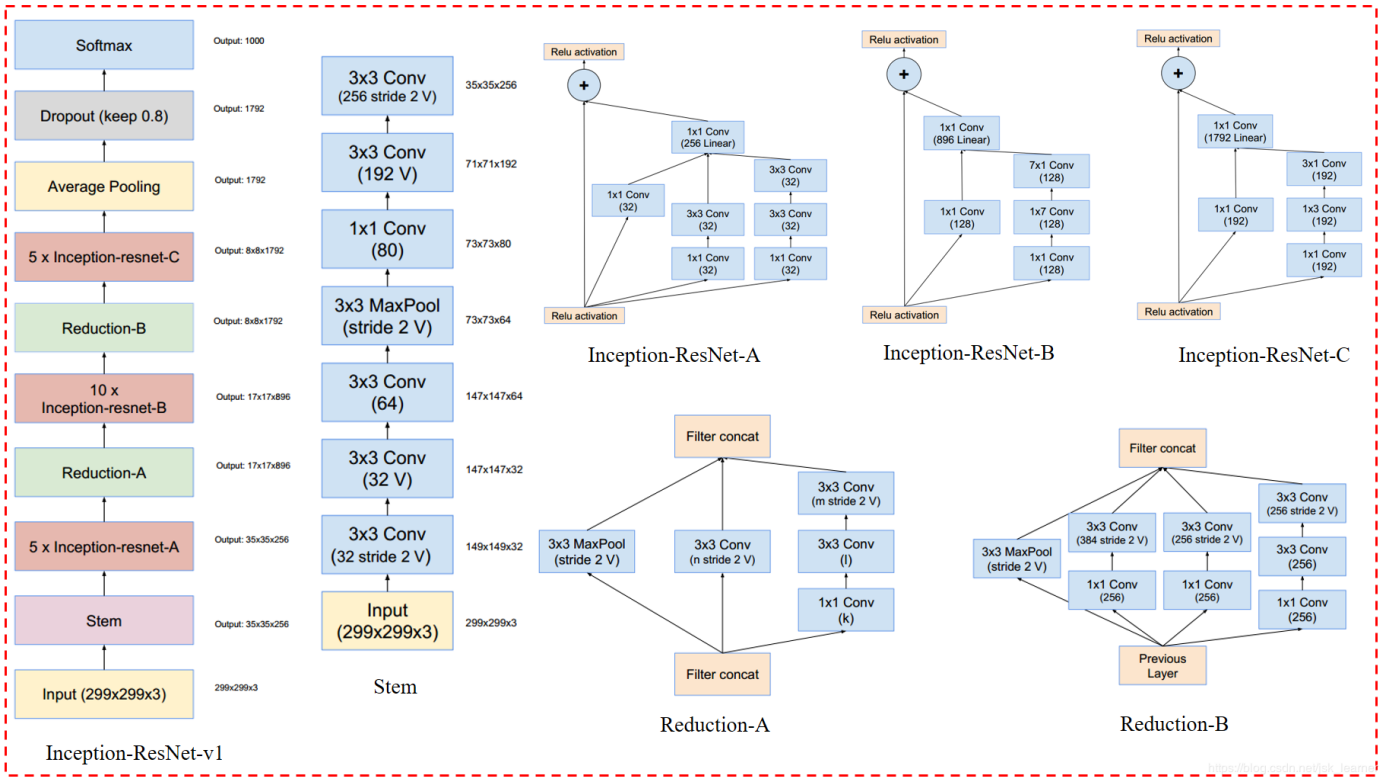
For the residual version in the Inception network, we use an Inception module that consumes less than the original Inception. The convolution kernel (followed by 1x1) of each Inception module is used to modify the dimension, which can compensate the reduction of the Inception dimension to some extent.

One is named **Inception-ResNet-v1**, which is consistent with the calculation cost of Inception-v3. One is named **Inception-ResNet-v2**, which is consistent with the calculation cost of Inception-v4.

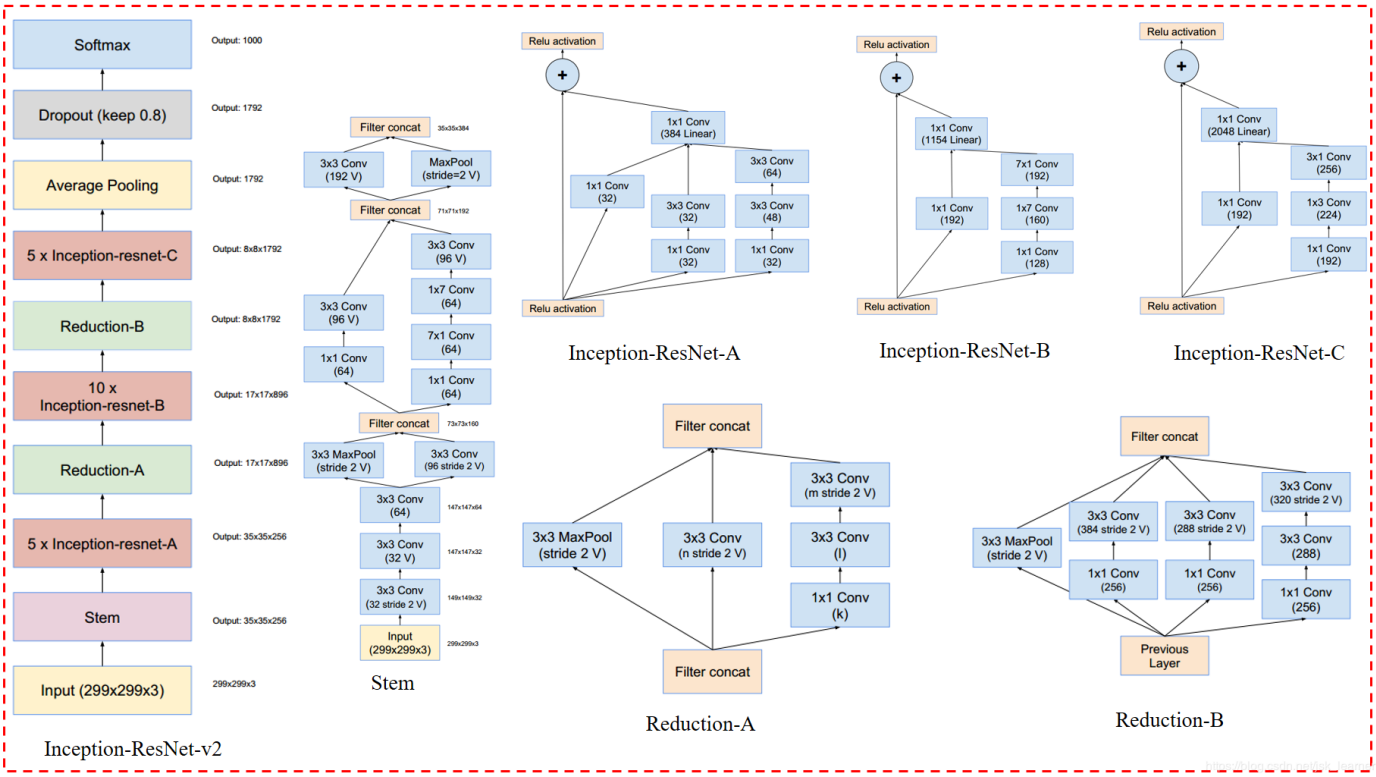
Figure 15 shows the structure of both. However, Inception-v4 is actually slower in practice, probably because it has more layers.

Another small technique is that we use the BN layer in the header of the traditional layer in the Inception-ResNet module, but not in the header of the summations. ** There is reason to believe that the BN layer is effective. But in order to add more Inception modules, we made a compromise between the two.

Inception-ResNet-v1



Inception-ResNet-v2



Scaling of the Residuals

This paper finds that when the number of convolution kernels exceeds 1,000 , the residual variants will start to show instability , and the network will die in the early stages of training, which means that the last layer before the average pooling layer is in the Very few iterations start with just a zero value . This situation cannot be

prevented by reducing the learning rate or by adding a BN layer . Hekaiming's ResNet article also mentions this phenomenon.

This article finds that scale can stabilize the training process before adding the residual module to the activation layer . This article sets the scale coefficient between 0.1 and 0.3.

In order to prevent the occurrence of unstable training of deep residual networks, He suggested in the article that it is divided into two stages of training. The first stage is called warm-up (preheating) , that is, training the model with a very low learning first. In the second stage, a higher learning rate is used. And this article finds that if the convolution sum is very high, even a learning rate of 0.00001 cannot solve this training instability problem, and the high learning rate will also destroy the effect. But this article considers scale residuals to be more reliable than warm-up.

Even if scal is not strictly necessary, it has no effect on the final accuracy, but it can stabilize the training process.

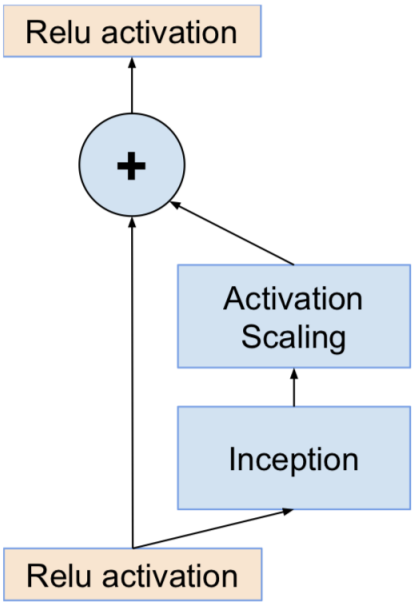


Figure 20. The general schema for scaling combined Inception-resnet moduels. We expect that the same idea is useful in the general resnet case, where instead of the Inception block an arbitrary subnetwork is used. The scaling block just scales the last linear activations by a suitable constant, typically around 0.1.

Conclusion

Inception-ResNet-v1 : a network architecture combining inception module and resnet module with similar calculation cost to Inception-v3;

Inception-ResNet-v2 : A more expensive but better performing network architecture.

Inception-v4 : A pure inception module, without residual connections, but with performance similar to Inception-ResNet-v2.

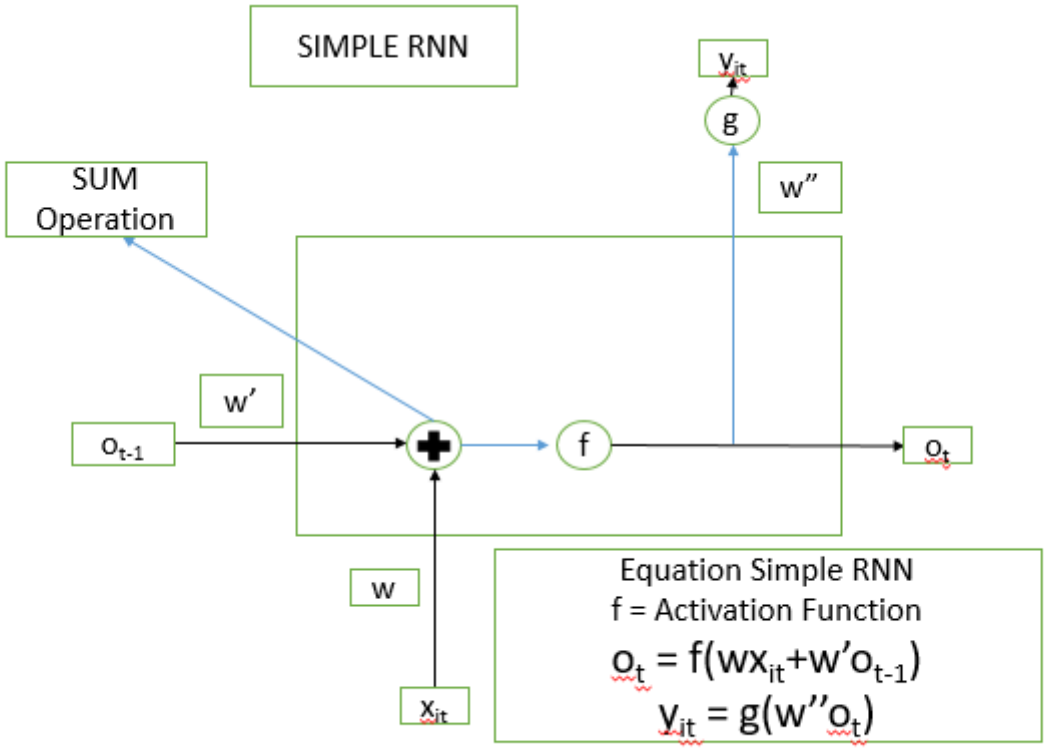
Q. Why do we use an RNN instead of a simple neural network?

We use Recurrent Neural Networks mostly in sequential data.We use RNN over standard neural networks due to the following reasons :

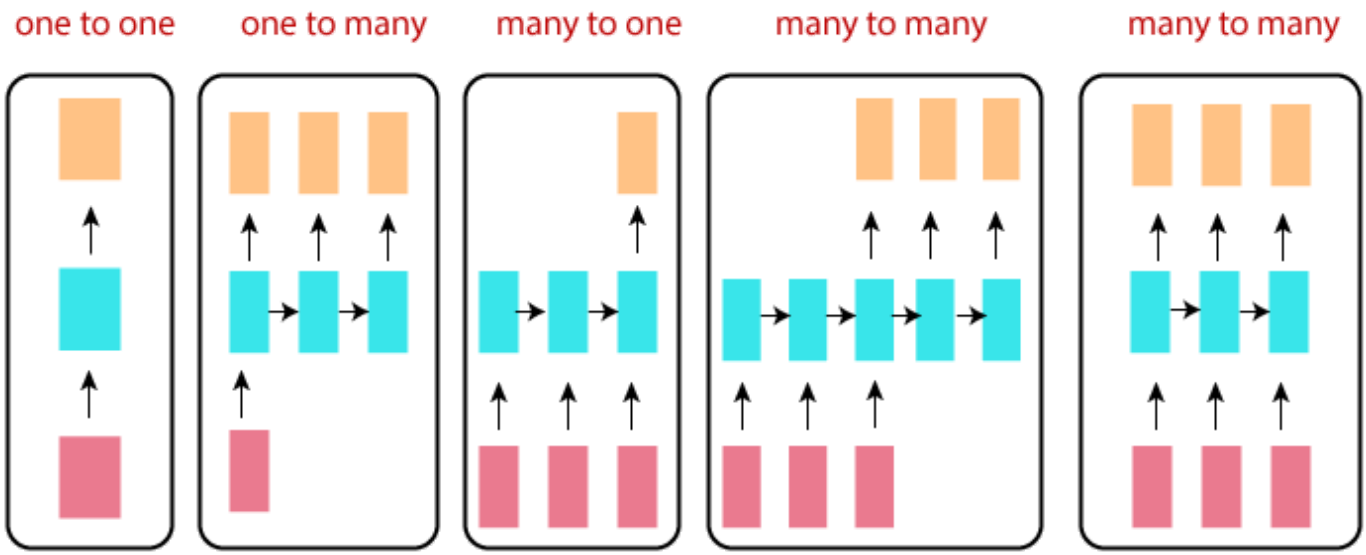
- In case of sequential data, the inputs and outputs can be of different lengths. For e.g. , in sentiment analysis, we map the input sentences to one number describing the sentiment of the text.
- Standard neural network does not share features learnt across different positions of text. For e.g. , in named entity recognition(identifying names of person in sentences), suppose we identify Henry occurs in first position as name, we would want the algorithm to use this information in case Henry occurs later again in the sentence. We want things learnt in one part to generalize in others parts in sequence data.
- The parameters required for handling text will be very large in case of Standard neural networks. RNN requires much less parameters to learn.

Recurrent Neural Network

- Recurrent Neural Network(RNN) are a type of Neural Network where the output from previous step are fed as input to the current step.
- In traditional neural networks, all the inputs and outputs are independent of each other, but in cases like when it is required to predict the next word of a sentence, the previous words are required and hence there is a need to remember the previous words.
- Thus RNN came into existence, which solved this issue with the help of a Hidden Layer. The main and most important feature of RNN is Hidden state, which remembers some information about a sequence.
- RNN have a “memory” which remembers all information about what has been calculated. It uses the same parameters for each input as it performs the same task on all the inputs or hidden layers to produce the output.
- This reduces the complexity of parameters, unlike other neural networks.



Types of RNN



One-to-one:

This is also called Plain Neural networks. It deals with a fixed size of the input to the fixed size of output, where they are independent of previous information/output.

Example: Image classification.

One-to-Many:

It deals with a fixed size of information as input that gives a sequence of data as output.

Example: Image Captioning takes the image as input and outputs a sentence of words.

Many-to-One:

It takes a sequence of information as input and outputs a fixed size of the output.

Example: sentiment analysis where any sentence is classified as expressing the positive or negative sentiment.

Many-to-Many:

It takes a Sequence of information as input and processes the recurrently outputs as a Sequence of data.

Example: Machine Translation, where the RNN reads any sentence in English and then outputs the sentence in French.

Bidirectional Many-to-Many:

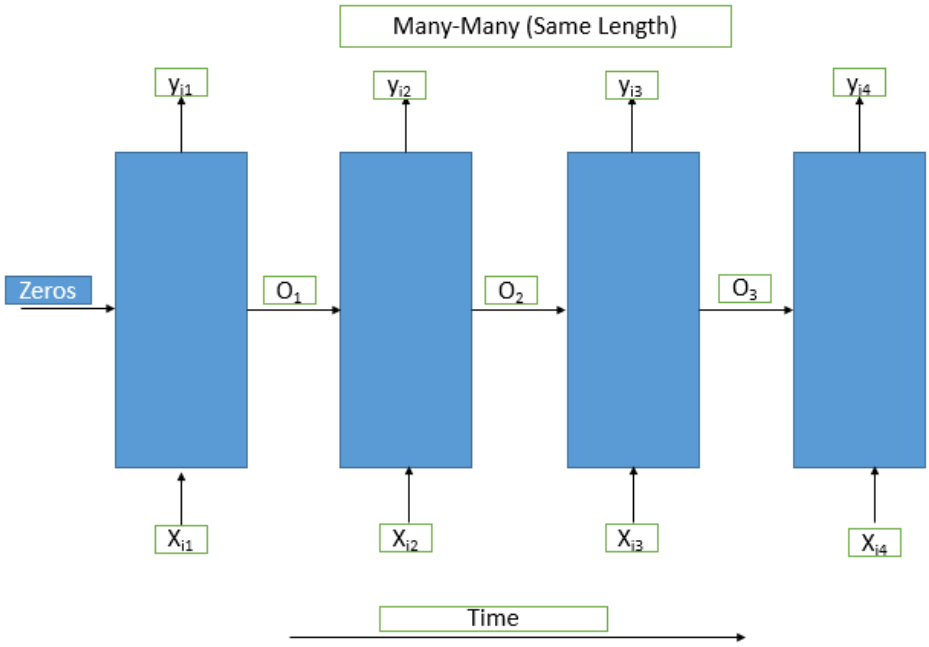
Synced sequence input and output. Notice that in every case are no pre-specified constraints on the lengths sequences because the recurrent transformation (green) is fixed and can be applied as many times as we like.

Example: Video classification where we wish to label every frame of the video.

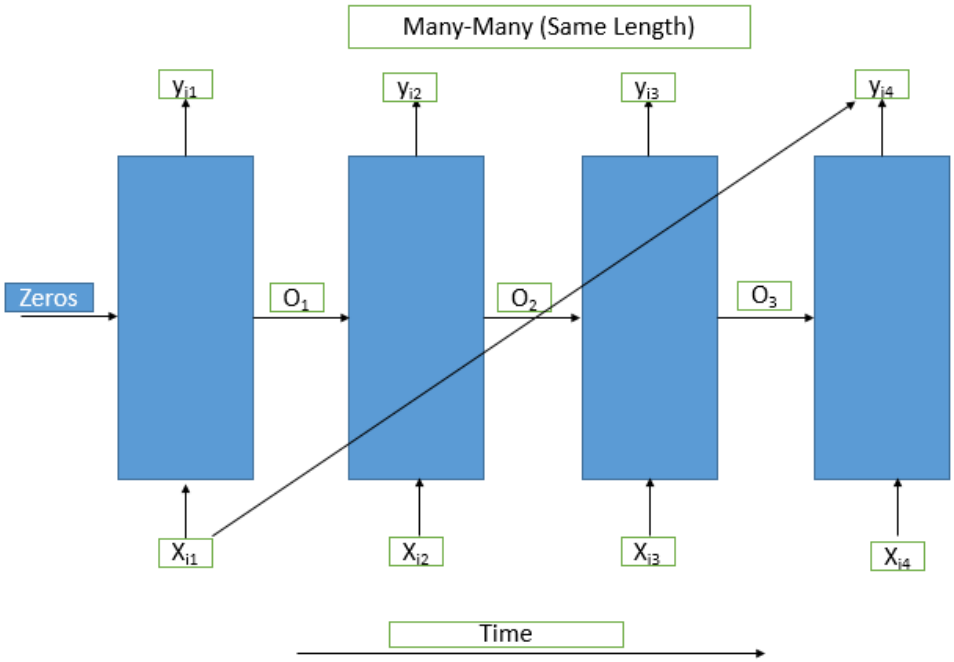
S.no	CNN	RNN
1	**CNN** stands for **Convolutional Neural Network**.	**RNN** stands for **Recurrent Neural Network**.
2	CNN is considered to be more potent than RNN.	RNN includes less feature compatibility when compared to CNN.
3	CNN is ideal for images and video processing.	RNN is ideal for text and speech Analysis.
4	It is suitable for spatial data like images.	RNN is used for temporal data, also called sequential data.
5	The network takes fixed-size inputs and generates fixed size outputs.	RNN can handle arbitrary input/ output lengths.
6	CNN is a type of feed-forward artificial neural network with variations of multilayer perceptron's designed to use minimal amounts of preprocessing.	RNN, unlike feed-forward neural networks- can use their internal memory to process arbitrary sequences of inputs.
7	CNN's use of connectivity patterns between the neurons. CNN is affected by the organization of the animal **visual cortex**, whose individual neurons are arranged in such a way that they can respond to overlapping regions in the visual field.	Recurrent neural networks use time-series information- what a user spoke last would impact what he will speak next.

Q. What is the Problem With Simple RNN ?

- Ans:- Lets take Many-Many (Same Length)



- Here yi4 value depends a lot on xi4 and 03 and less depends on xi1 and 01 which is a limitation of this simple RNN because in real world application yi4 may depends more on xi1 and 01 and less xi4 and 03. this is called long term dependency.
- Simple RNN Cant handle long term dependency.
- long term dependency - Later output depends a lot on earlier input i.e. (yi4 -> xi1)



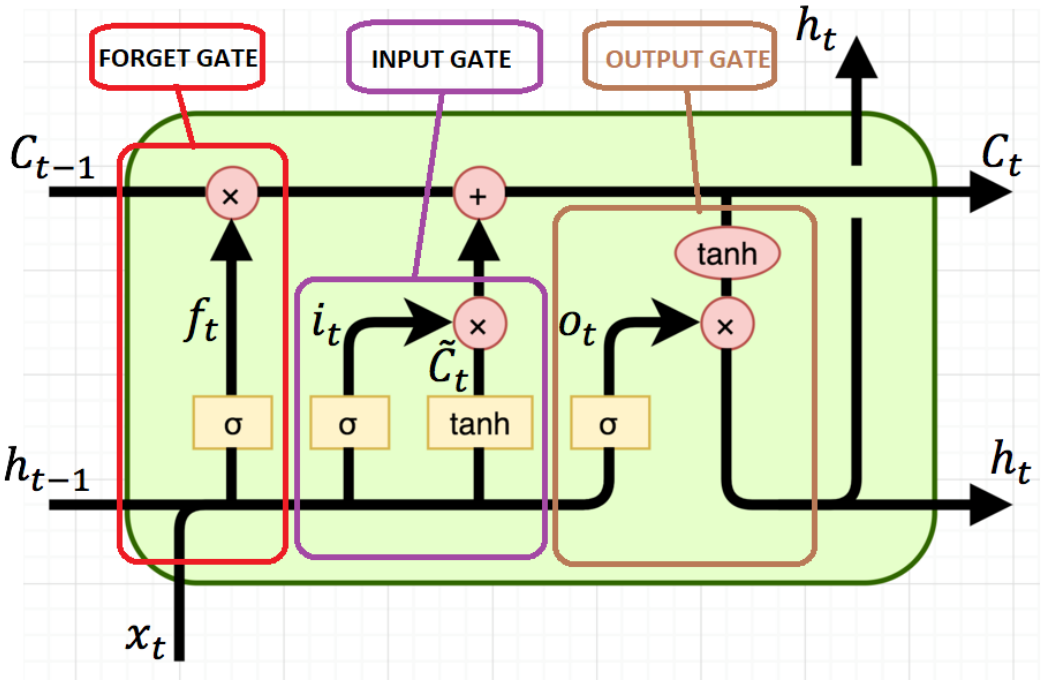
- So that small change in yi4 should create similar change in weight of xi1 result vanish the gradient. in simple RNN with Sigmoid and tanh later output nodes of network are less sensitive to the input this is happen due to vanishing gradient problem.

- If a sequence is long enough, they'll have a hard time carrying information from earlier time steps to later ones. So if you are trying to process a paragraph of text to do predictions, RNN's may leave out important information from the beginning.
- During back propagation, recurrent neural networks suffer from the vanishing gradient problem. Gradients are values used to update a neural networks weights. The vanishing gradient problem is when the gradient shrinks as it back propagates through time. If a gradient value becomes extremely small, it doesn't contribute too much learning.
- So in recurrent neural networks, layers that get a small gradient update stops learning. Those are usually the earlier layers. So because these layers don't learn, RNN's can forget what it seen in longer sequences, thus having a short-term memory. If you want to know more about the mechanics of recurrent neural networks in general, you can read my previous post here.

- To overcome this we use LSTM (Long Short Term Memory) And GRU (Gated Recurrent Unit)

LSTM (Long Short Term Memory)

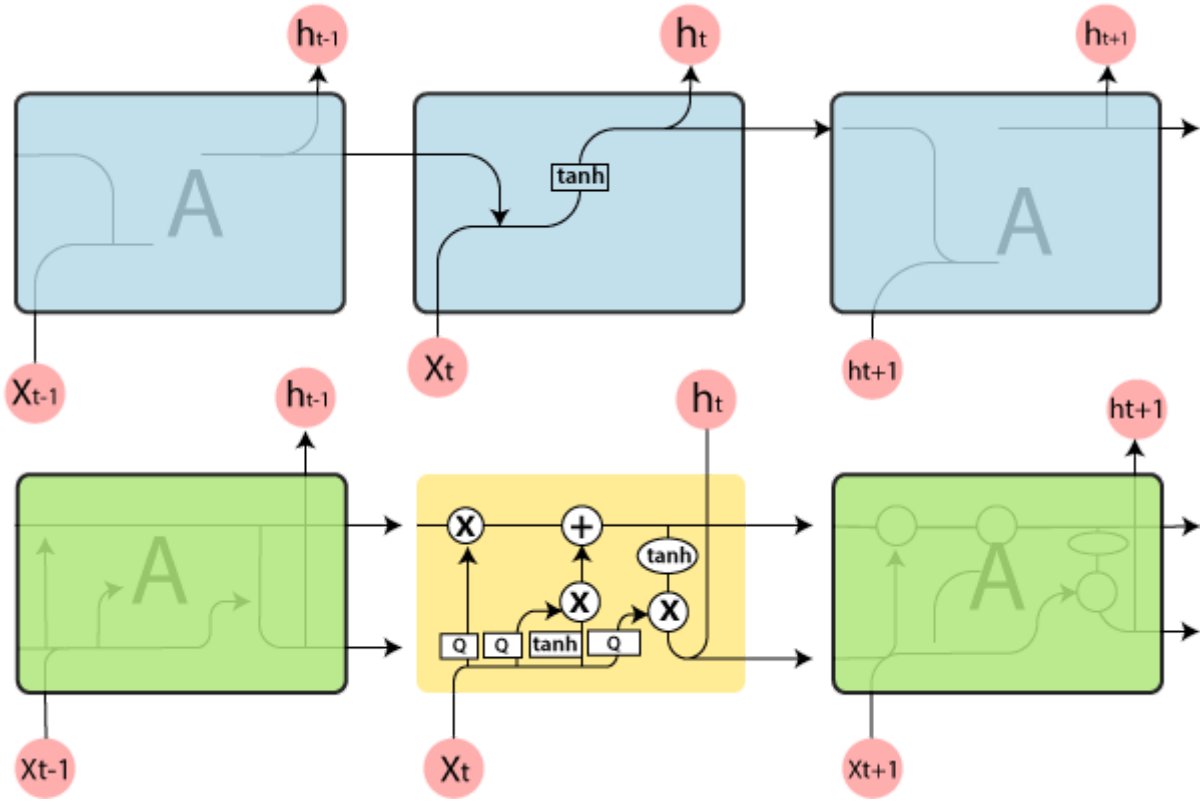
- It takes care both long and short term dependency.



1. Input gate- It discover which value from input should be used to modify the memory. Sigmoid function decides which values to let through 0 or 1. And tanh function gives weightage to the values which are passed, deciding their level of importance ranging from -1 to 1.

2. Forget gate- It discover the details to be discarded from the block. A sigmoid function decides it. It looks at the previous state (ht-1) and the content input (Xt) and outputs a number between 0(omit this) and 1(keep this) for each number in the cell state Ct-1.

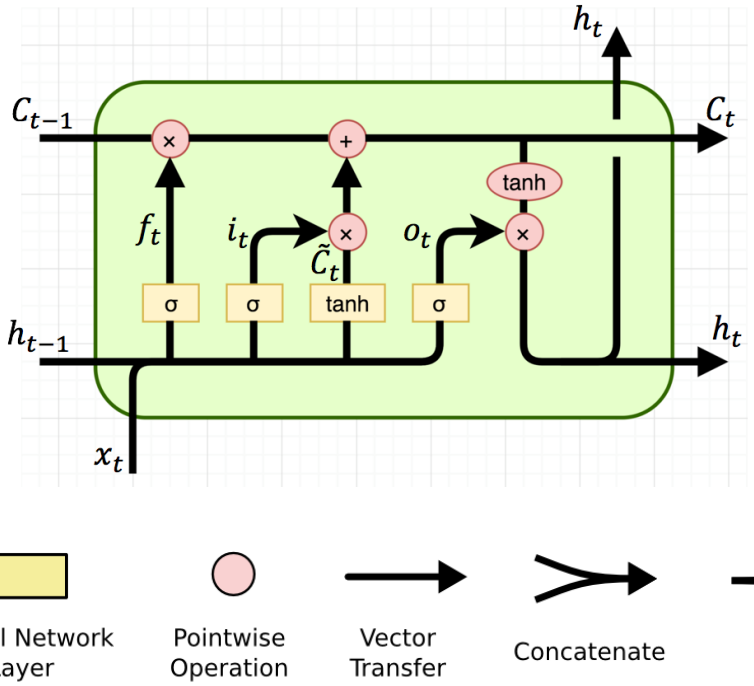
3. Output gate- The input and the memory of the block are used to decide the output. Sigmoid function decides which values to let through 0 or 1. And tanh function decides which values to let through 0, 1. And tanh function gives weightage to the values which are passed, deciding their level of importance ranging from -1 to 1 and multiplied with an output of sigmoid.



- It represents a full RNN cell that takes the current input of the sequence xi, and outputs the current hidden state, hi, passing this to the next RNN cell for our input sequence. The inside of an LSTM cell is a lot more complicated than a traditional RNN cell, while the conventional RNN cell has a single "internal layer" acting on the current state (ht-1) and input (xt).

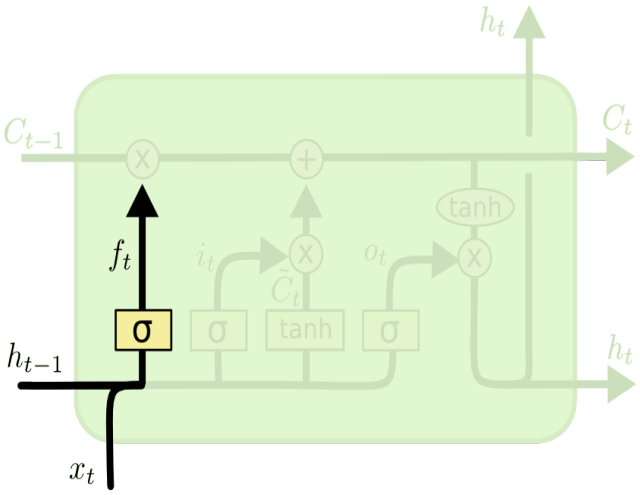
Step-by-Step LSTM Walk Through

- It takes care both long and short term dependency.



Step-1 (forget gate layer)

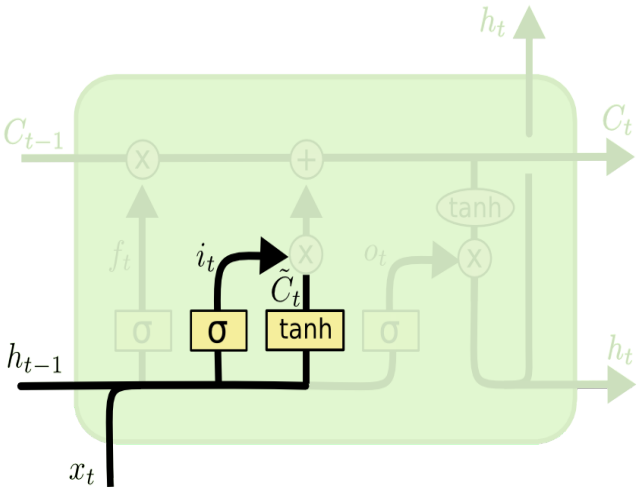
- The first step in our LSTM is to decide what information we're going to throw away from the cell state. This decision is made by a sigmoid layer called the "forget gate layer."
- It looks at ht−1 and xt, and outputs a number between 0 and 1 for each number in the cell state Ct−1. A 1 represents "completely keep this" while a 0 represents "completely get rid of this."



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

Step-2 (input gate layer)

- In this step is to decide what new information we're going to store in the cell state.
- This has two parts. First, a sigmoid layer called the "input gate layer" decides which values we'll update. Next, a tanh layer creates a vector of new candidate values, C~t, that could be added to the state. In the next step, we'll combine these two to create an update to the state.

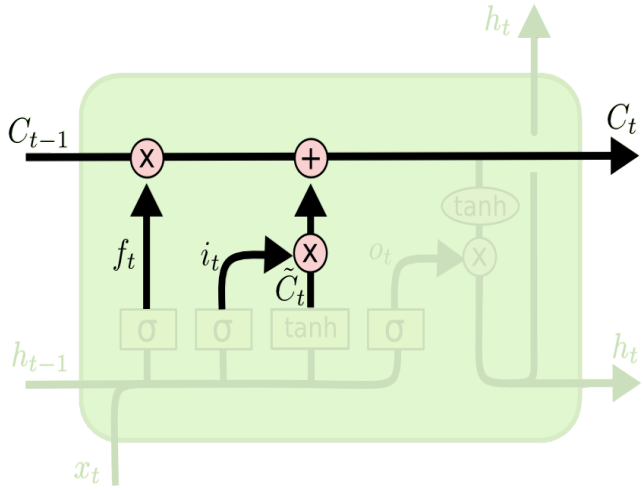


$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

Step-3 (input gate layer)

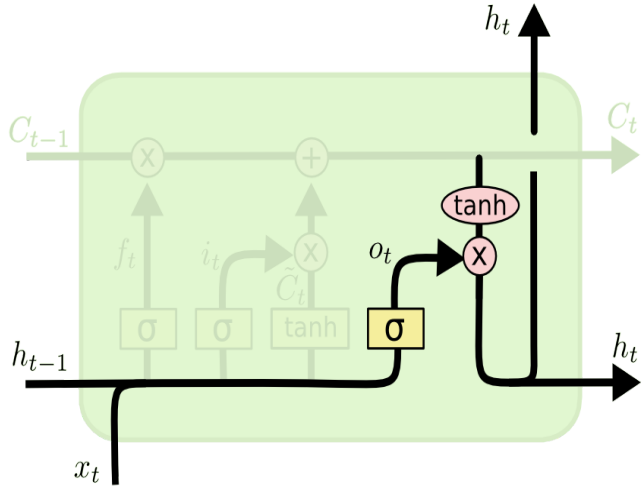
- It's now time to update the old cell state, Ct−1, into the new cell state Ct. The previous steps already decided what to do, we just need to actually do it.
- We multiply the old state by ft, forgetting the things we decided to forget earlier. Then we add it∗C~t. This is the new candidate values, scaled by how much we decided to update each state value.



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Step-4 (output gate layer)

- Finally, we need to decide what we're going to output.
- This output will be based on our cell state, but will be a filtered version. First, we run a sigmoid layer which decides what parts of the cell state we're going to output. Then, we put the cell state through tanh (to push the values to be between −1 and 1) and multiply it by the output of the sigmoid gate, so that we only output the parts we decided to.

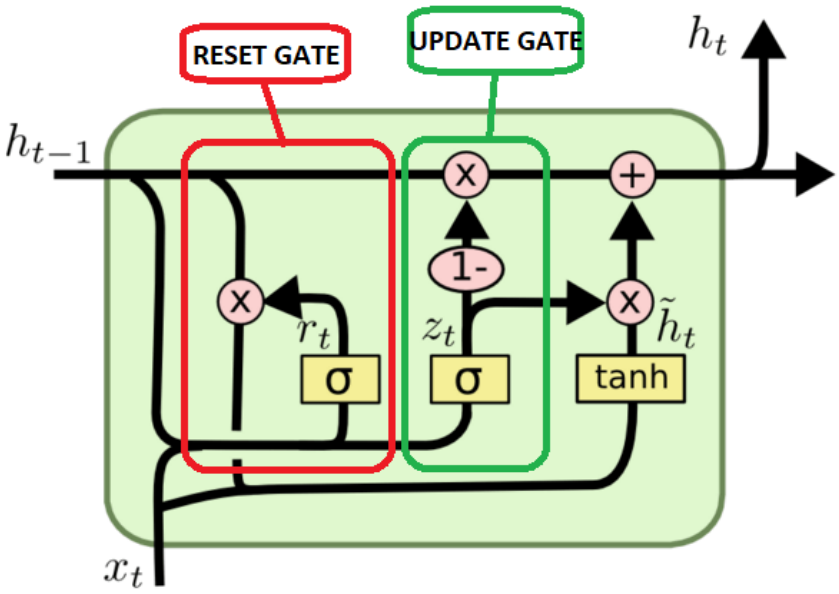


$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$

$$h_t = o_t * \tanh\left(C_t\right)$$

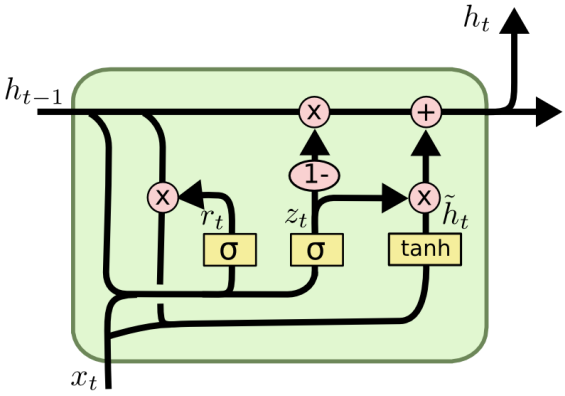
GRU (Gated Recurrent Units)

- So now we know how an LSTM work, let's briefly look at the GRU. The GRU is the newer generation of Recurrent Neural networks and is pretty similar to an LSTM. GRU's got rid of the cell state and used the hidden state to transfer information. It also only has two gates, a reset gate and update gate.



1. Update Gate - The update gate acts similar to the forget and input gate of an LSTM. It decides what information to throw away and what new information to add.
2. Reset Gate- The reset gate is another gate is used to decide how much past information to forget.

- GRU's has fewer tensor operations; therefore, they are a little speedier to train then LSTM's. There isn't a clear winner which one is better. Researchers and engineers usually try both to determine which one works better for their use case.



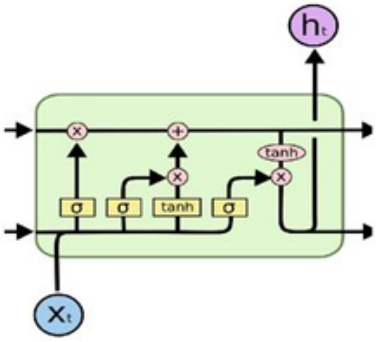
$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

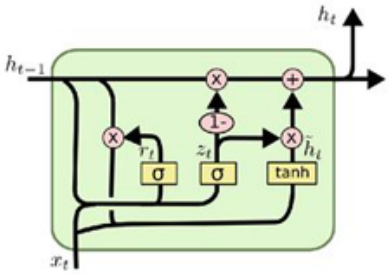
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

LSTM vs. GRU



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$
$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$
$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$
$$o_t = \sigma\left(W_o \; [h_{t-1}, x_t] \; + \; b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$