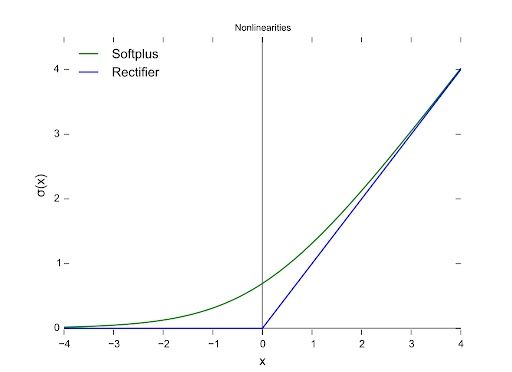
## ReLU Activation Function

In artificial neural networks, the activation function of a node defines the output of that node given an input or set of inputs. A standard integrated circuit can be seen as a digital network of activation functions that can be “ON” or “OFF,” depending on the input.

A rectified linear unit (ReLU) is an activation function that introduces the property of non-linearity to a deep learning model and solves the vanishing gradients issue. "It interprets the positive part of its argument. It is one of the most popular activation functions in deep learning.

Sigmoid and tanh were monotonous, differentiable and previously more popular activation functions. However, these functions suffer saturation over time, and this leads to problems occurring with vanishing gradients. An alternative and the most popular activation function to overcome this issue is theRectified Linear Unit (ReLU).

The diagram below with the blue line is the representation of the Rectified Linear Unit (ReLU), whereas the green line is a variant of ReLU called Softplus. The other variants of ReLU include leaky ReLU, exponential linear unit (ELU) and Sigmoid linear unit (SiLU), etc., which are used to improve performances in some tasks.



we’ll only look at the rectified linear unit (ReLU) because it’s still the most used activation function by default for performing a majority of deep learning tasks. Its variants are typically used for specific purposes in which they might have a slight edge over the ReLU.

This activation function was first introduced to a dynamical network by Hahnloser et al. in 2000, with strong biological motivations and mathematical justifications. It was demonstrated for the first time in 2011 as a way to enable better training of deeper networks compared to other widely used activation functions including the logistic sigmoid (which is inspired by probability theory and logistic regression) and the hyperbolic tangent.

The rectifier is, as of 2017, the most popular activation function for deep neural networks. A unit employing the rectifier is also called a rectified linear unit (ReLU).

## Why Is ReLU a Good Activation Function?

The main reason ReLU  wasn’t used until more recently is because it was not differentiable at the point zero. Researchers tended to use differentiable functions like sigmoid and tanh. However, it’s now determined that ReLU is the best activation function for deep learning.

The ReLU activation function is differentiable at all points except at zero. For values greater than zero, we just consider the max of the function. This can be written as:

f(x) = max{0, z}

In simple terms, this can also be written as follows:

if input > 0:

   return input

else:

   return 0

All the negative values default to zero, and the maximum for the positive number is taken into consideration.

For the computation of the backpropagation of neural networks, the differentiation for the ReLU is relatively easy. The only assumption we will make is the derivative at the point zero, which will also be considered as zero. This is usually not such a big concern, and it works well for the most part. The derivative of the function is the value of the slope. The slope for negative values is 0.0, and the slope for positive values is 1.0.

## Disadvantages of the ReLU Activation Function

The main issue with ReLU is that all the negative values become zero immediately, which decreases the ability of the model to fit or train from the data properly.

That means any negative input given to the ReLU activation function turns the value into zero immediately in the graph, which in turn affects the resulting graph by not mapping the negative values appropriately. This can however be easily fixed by using the different variants of the ReLU activation function, like the leaky ReLU and other functions discussed earlier in the article.

This is just a short introduction to the rectified linear unit and its importance in deep learning technology today. It’s more popular than all other activation functions, and for good reason.

# **Adam Optimization Algorithm**

The choice of optimization algorithm for your deep learning model can mean the difference between good results in minutes, hours, and days.

The Adam optimization algorithm is an extension to stochastic gradient descent that has recently seen broader adoption for deep learning applications in computer vision and natural language processing.

In this post, you will get a gentle introduction to the Adam optimization algorithm for use in deep learning.

## What is the Adam optimization algorithm?

Adam is an optimization algorithm that can be used instead of the classical stochastic gradient descent procedure to update network weights iterative based in training data.

Adam was presented by Diederik Kingma from OpenAI and Jimmy Ba from the University of Toronto in their 2015 ICLR paper (poster) titled “Adam: A Method for Stochastic Optimization“. I will quote liberally from their paper in this post, unless stated otherwise.

The algorithm is called Adam. It is not an acronym and is not written as “ADAM”.

When introducing the algorithm, the authors list the attractive benefits of using Adam on non-convex optimization problems, as follows:

* Straightforward to implement.
* Computationally efficient.
* Little memory requirements.
* Invariant to diagonal rescale of the gradients.
* Well suited for problems that are large in terms of data and/or parameters.
* Appropriate for non-stationary objectives.
* Appropriate for problems with very noisy/or sparse gradients.
* Hyper-parameters have intuitive interpretation and typically require little tuning.

## How Does Adam Work?

Adam is different to classical stochastic gradient descent.

Stochastic gradient descent maintains a single learning rate (termed alpha) for all weight updates and the learning rate does not change during training.

A learning rate is maintained for each network weight (parameter) and separately adapted as learning unfolds.

The authors describe Adam as combining the advantages of two other extensions of stochastic gradient descent. Specifically:

* **Adaptive Gradient Algorithm** (AdaGrad) that maintains a per-parameter learning rate that improves performance on problems with sparse gradients (e.g. natural language and computer vision problems).
* **Root Mean Square Propagation** (RMSProp) that also maintains per-parameter learning rates that are adapted based on the average of recent magnitudes of the gradients for the weight (e.g. how quickly it is changing). This means the algorithm does well on online and non-stationary problems (e.g. noisy).

Adam realizes the benefits of both AdaGrad and RMSProp.

Instead of adapting the parameter learning rates based on the average first moment (the mean) as in RMSProp, Adam also makes use of the average of the second moments of the gradients (the uncentered variance).

Specifically, the algorithm calculates an exponential moving average of the gradient and the squared gradient, and the parameters beta1 and beta2 control the decay rates of these moving averages.

**CNN**

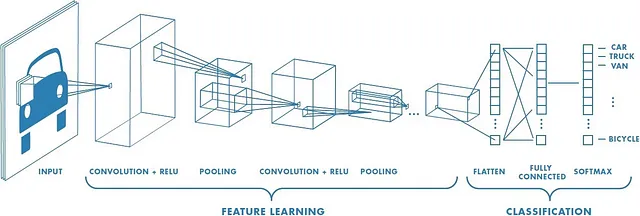
CNN or the convolutional neural network (CNN) is a class of **deep learning neural networks**. In short think of CNN as a machine learning algorithm that can take in an input image, assign importance (learnable weights and biases) to various aspects/objects in the image, and be able to differentiate one from the other.

CNN works by extracting features from the images. Any CNN consists of the following:

1. The input layer which is a grayscale image
2. The Output layer which is a binary or multi-class labels
3. Hidden layers consisting of convolution layers, ReLU (rectified linear unit) layers, the pooling layers, and a fully connected Neural Network

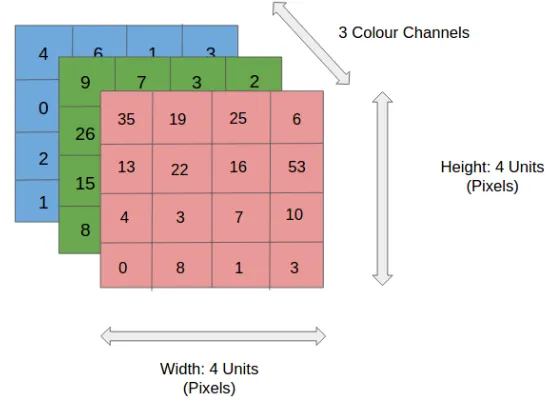
It is very important to understand that ANN or Artificial Neural Networks, made up of multiple neurons is not capable of extracting features from the image. This is where a combination of convolution and pooling layers comes into the picture. Similarly, the convolution and pooling layers can’t perform classification hence we need a fully connected Neural Network.

Before we jump into the concepts further let’s try and understand these individual segments separately.



Let’s consider that we have access to multiple images of different vehicles, each labeled into a truck, car, van, bicycle, etc. Now the idea is to take these pre-label/classified images and develop a machine learning algorithm that is capable of accepting a new vehicle image and classify it into its correct category or label. Now before we start building a neural network we need to understand that most of the images are converted into a grayscale form before they are processed.

## Why grayscale and not RGB/Color Images?



We discussed earlier that any color image has three channels, i.e. red, green, and blue as shown in other figure . There are several such color spaces like the grayscale, CMYK, HSV in which an image can exist.

The challenge with images having multiple color channels is that we have huge volumes of data to work with which makes the process computationally intensive. In other worlds think of it like a complicated process where the Neural Network or any machine learning algorithm has to work with three different data (R-G-B values in this case) to extract features of the images and classify them into their appropriate categories.

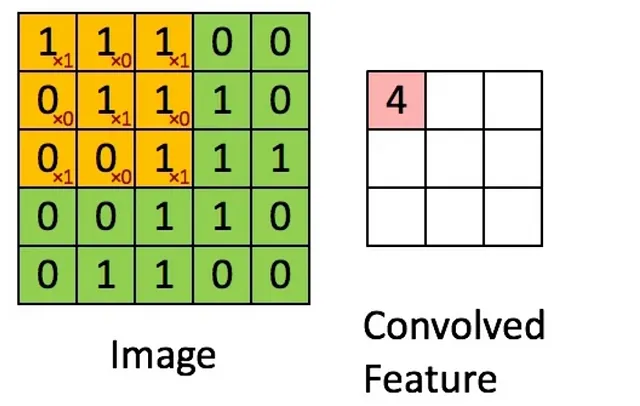
The role of CNN is to reduce the images into a form that is easier to process, without losing features critical towards a good prediction. This is important when we need to make the algorithm scalable to massive datasets.

## What are convolutions?

We understand that the training data consists of grayscale images which will be an input to the convolution layer to extract features. The convolution layer consists of one or more Kernels with different weights that are used to extract features from the input image. Say in the example above we are working with a Kernel (K) of size 3 x 3 x 1 (x 1 because we have one color channel in the input image), having weights outlined below.

Kernel/Filter, K =   
1 0 1  
0 1 0  
1 0 1

When we slide the Kernel over the input image (say the values in the input image are grayscale intensities) based on the weights of the Kernel we end up calculating features for different pixels based on their surrounding/neighboring pixel values. E.g. when the Kernel is applied on the image for the first time as illustrated in Figure 5 below we get a feature value equal to 4 in the convolved feature matrix as shown below.



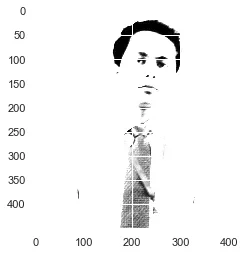
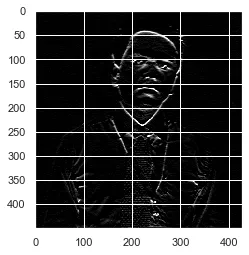
Now we will see that the kernel shifts 9 times across image. This process is called **Stride.**When we use a stride value of 1 (**Non-Strided**) operation we need 9 iterations to cover the entire image. **The CNN learns the weights of these Kernels on its own.**The result of this operation is a feature map that basically detects features from the images rather than looking into every single pixel value.

Image features, such as edges and interest points, provide rich information on the image content. They correspond to local regions in the image and are fundamental in many applications in image analysis: recognition, matching, reconstruction, etc. Image features yield two different types of problem: the detection of the area of interest in the image, typically contours, and the description of local regions in the image, typically for matching in different images, (Image features. (n.d.))

## Let’s take a deeper look into what we are talking about.

Extracting features from an image is similar to detecting edges in the image. We can use the openCV package to perform the same. We will declare a few matrices, apply them on a grayscale image, and try and look for edges.

# 3x3 array for edge detection  
mat\_y = np.array([[ -1, -2, -1],   
 [ 0, 0, 0],   
 [ 1, 2, 1]])  
mat\_x = np.array([[ -1, 0, 1],   
 [ 0, 0, 0],   
 [ 1, 2, 1]])  
   
filtered\_image = cv2.filter2D(gray, -1, mat\_y)  
plt.imshow(filtered\_image, cmap='gray')filtered\_image = cv2.filter2D(gray, -1, mat\_x)  
plt.imshow(filtered\_image, cmap='gray')



Illustrates images with edges when a filter2D transformation is applied to the data. Note that the two images are significantly different. When we talk about convolution layers and Kernels we basically want to identify the edges in an image. When working with CNN the matrix\_x and matrix\_y values are determined automatically by the network.

Depending on the weights associated with a filter, the features are detected from the image. Notice when an image is passed through a convolution layer, it and tries and identify the features by analyzing the change in neighboring pixel intensities. E.g. the top right of the image has similar pixel intensity throughout, hence no edges are detected. It is only when the pixels change intensity the edges are visible.

**ANN**

Artificial Neural Networks (ANNs) make up an integral part of the Deep Learning process. They are inspired by the neurological structure of the human brain. According to AILabPage, ANNs are “complex computer code written with the number of simple, highly interconnected processing elements which is inspired by human biological brain structure for simulating human brain working & processing data (Information) models.”

 ANN architecture in Neural Network is a part of Machine Learning and also very crucial because its structure is similar to the human brain. It also functions like a brain by sending neural signals from one end to the other.

Deep Learning focuses on five core Neural Networks, including:

* Multi-Layer Perceptron
* Radial Basis Network
* Recurrent Neural Networks
* Generative Adversarial Networks
* Convolutional Neural Networks.

## Neural Network: Architecture

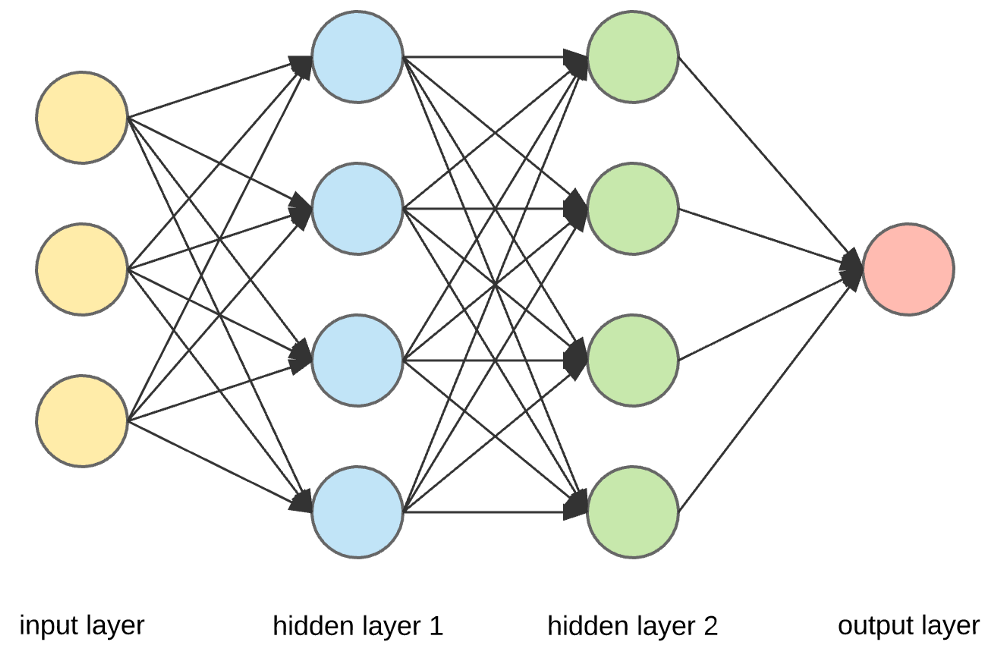
Neural Networks are complex structures made of artificial neurons that can take in multiple inputs to produce a single output. This is the primary job of a Neural Network – to transform input into a meaningful output. Usually, a Neural Network consists of an input and output layer with one or multiple hidden layers within. It is also known as Artificial Neural Network or ANN. ANN architecture in Neural Network functions just like a human brain and is very important.

In a Neural Network, all the neurons influence each other, and hence, they are all connected. The network can acknowledge and observe every aspect of the dataset at hand and how the different parts of data may or may not relate to each other. This is how Neural Networks are capable of finding extremely complex patterns in vast volumes of data.

In a Neural Network, the flow of information occurs in two ways

* **Feedforward Networks:** In this model, the signals only travel in one direction, towards the output layer. Feedforward Networks have an input layer and a single output layer with zero or multiple hidden layers. They are widely used in pattern recognition.
* **Feedback Networks:** In this model, the recurrent or interactive networks use their internal state (memory) to process the sequence of inputs. In them, signals can travel in both directions through the loops (hidden layer/s) in the network. They are typically used in time-series and sequential tasks.

## Neural Network: Components



**Input Layers, Neurons, and Weights** –

In the picture given above, the outermost yellow layer is the input layer. A neuron is the basic unit of a neural network. They receive input from an external source or other nodes. Each node is connected with another node from the next layer, and each such connection has a particular weight. Weights are assigned to a neuron based on its relative importance against other inputs.

When all the node values from the yellow layer are multiplied (along with their weight) and summarized, it generates a value for the first hidden layer. Based on the summarized value, the blue layer has a predefined “activation” function that determines whether or not this node will be “activated” and how “active” it will be.

Let’s understand this using a simple everyday task – making tea. In the tea making process, the ingredients used to make tea (water, tea leaves, milk, sugar, and spices) are the “neurons” since they make up the starting points of the process. The amount of each ingredient represents the “weight.” Once you put in the tea leaves in the water and add the sugar, spices, and milk in the pan, all the ingredients will mix and transform into another state. This transformation process represents the “activation function.”

Hidden Layers and Output Layer

The layer or layers hidden between the input and output layer is known as the hidden layer. It is called the hidden layer since it is always hidden from the external world. The main computation of a Neural Network takes place in the hidden layers. So, the hidden layer takes all the inputs from the input layer and performs the necessary calculation to generate a result. This result is then forwarded to the output layer so that the user can view the result of the computation.

In our tea-making example, when we mix all the ingredients, the formulation changes its state and color on heating. The ingredients represent the hidden layers. Here heating represents the activation process that finally delivers the result – tea.

**Neural Network: Algorithms**

In a Neural Network, the learning (or training) process is initiated by dividing the data into three different sets:

* **Training dataset –** This dataset allows the Neural Network to understand the weights between nodes.
* **Validation dataset –** This dataset is used for fine-tuning the performance of the Neural Network.
* **Test dataset –** This dataset is used to determine the accuracy and margin of error of the Neural Network.

Once the data is segmented into these three parts, Neural Network algorithms are applied to them for training the Neural Network. The procedure used for facilitating the training process in a Neural Network is known as the optimization, and the algorithm used is called the optimizer. There are different types of optimization algorithms, each with their unique characteristics and aspects such as memory requirements, numerical precision, and processing speed.

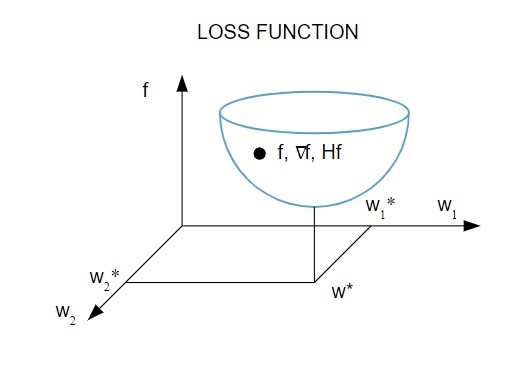
Before we dive into the discussion of the different **Neural Network algorithms**, let’s understand the learning problem first.

## What is the Learning Problem?

We represent the learning problem in terms of the minimization of a loss index (*f*). Here, “*f*” is the function that measures the performance of a Neural Network on a given dataset. Generally, the loss index consists of an error term and a regularization term. While the error term evaluates how a Neural Network fits a dataset, the regularization term helps prevent the overfitting issue by controlling the effective complexity of the Neural Network.

The loss function [*f(w*] depends on the adaptative parameters – weights and biases – of the Neural Network. These parameters can be grouped into a single n-dimensional weight vector (*w*).

Here’s a pictorial representation of the loss function:



According to this diagram, the minimum of the loss function occurs at the point (*w\**). At any point, you can calculate the first and second derivatives of the loss function. The first derivatives are grouped in the gradient vector, and its components are depicted as:

Here, *i = 1,…..,n*.

The second derivatives of the loss function are grouped in the **Hessian matrix**, like so:

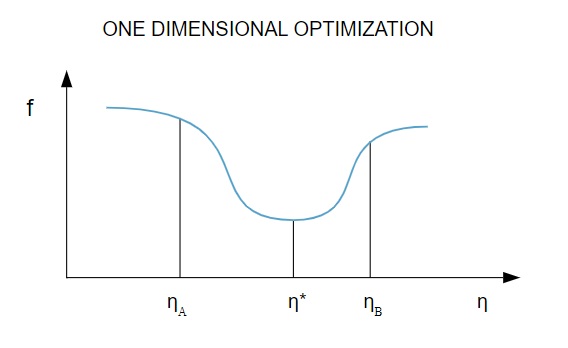
Here, *i,j = 0,1,…*

Now that we know what the learning problem is, we can discuss the five main

**Neural Network algorithms**.

**1. One-dimensional optimization**

Since the loss function depends on multiple parameters, one-dimensional optimization methods are instrumental in training Neural Network. Training algorithms first compute a training direction (*d*) and then calculate the training rate (*η*) that helps minimize the loss in the training direction [*f(η)*].



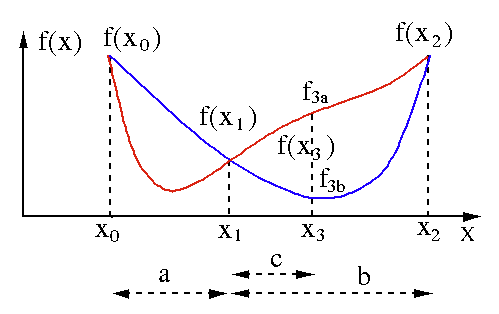
n the diagram, the points η1 and η2 define the interval containing the minimum of *f, η\**.

Thus, one-dimensional optimization methods aim to find the minimum of a given one-dimensional function. Two of the most commonly used one-dimensional algorithms are the Golden Section Method and Brent’s Method.

**Golden Section Method**

The golden section search algorithm is used to find the minimum or maximum of a single-variable function [*f(x)*]. If we already know that a function has a minimum between two points, then we can perform an iterative search just like we would in the bisection search for the root of an equation *f(x) = 0*. Also, if we can find three points (*x0 < x1 < x2*) corresponding to *f(x0) > f(x1) > f(X2)* in the neighborhood of the minimum, then we can deduce that a minimum exists between *x0* and *x2*. To find out this minimum, we can consider another point *x3* between *x1* and *x2*, which will give us the following outcomes:

* If *f(x3) = f3a > f(x1),* the minimum is inside the interval*x3 – x0 = a + c* that is related with three new points *x0 < x1 < x3* (here *x2* is replaced by *x3*).
* If *f(x3) = f3b > f(x1*), the minimum is inside the interval *x2 – x1 = b* related with three new points *x1 < x3 < x2* (here *x0* is replaced by *x1*).



**Brent’s Method**

Brent’s method is a root-finding algorithm that combines root bracketing, bisection, secant, and inverse quadratic interpolation. Although this algorithm tries to use the fast-converging secant method or inverse quadratic interpolation whenever possible, it usually reverts to the bisection method. Implemented in the Wolfram Language, Brent’s method is expressed as:

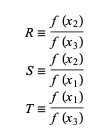
*Method -> Brent in FindRoot[eqn, x, x0, x1].*

In Brent’s method, we use a Lagrange interpolating polynomial of degree 2. In 1973, Brent claimed that this method will always converge, provided the values of the function are computable within a specific region, including a root. If there are three points *x1, x2,* and *x3*, Brent’s method fits *x* as a quadratic function of *y*, using the interpolation formula:

The subsequent root estimates are achieved by considering, thereby producing the following equation:

Here, *P = S [ T(R – T) (x3 – x2) – (1 – R) (x2 -x1) ]*and Q = (T – 1) (R – 1) (S – 1)

and,



**2. Multidimensional optimization**

By now, we already know that the learning problem for Neural Networks aims to find the parameter vector (*w\**) for which the loss function (*f*) takes a minimum value. According to the mandates of the standard condition, if the Neural Network is at a minimum of the loss function, the gradient is the zero vector.

Since the loss function is a non-linear function of the parameters, it is impossible to find the closed training algorithms for the minimum. However, if we consider searching through the parameter space that includes a series of steps, at each step, the loss will reduce by adjusting the parameters of the Neural Network.

In multidimensional optimization, a Neural Network is trained by choosing a random we parameter vector and then generating a sequence of parameters to ensure that the loss function decreases with each iteration of the algorithm. This variation of loss between two subsequent steps is known as “loss decrement.” The process of loss decrement continues until the training algorithm reaches or satisfies the specified condition.

Here are three examples of multidimensional optimization algorithms:

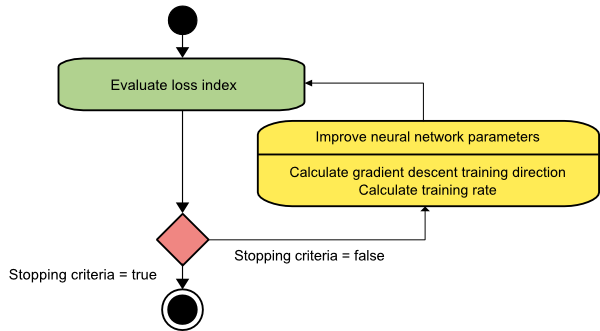
**Gradient descent**

The gradient descent algorithm is probably the simplest of all training algorithms. As it relies on the information provided `from the gradient vector, it is a first-order method. In this method, we’ll take *f[w(i)] = f(i)* and *∇f[w(i)] = g(i)*. The starting point of this training algorithm is w(0) that keeps progressing until the specified criterion is satisfied – it moves from w(i) to w(i+1) in the training direction *d(i) = −g(i)*. Hence, the gradient descent iterates as follows:

*w(i+1) = w(i)−g(i)η(i),*

Here, *i = 0,1,…*

The parameter *η* represents the training rate. You can set a fixed value for *η* or set it to the value found by one-dimensional optimization along the training direction at every step. However, it is preferred to set the optimal value for the training rate achieved by line minimization at each step.



This algorithm has many limitations since it requires numerous iterations for functions that have long and narrow valley structures. While the loss function decreases most rapidly in the direction of the downhill gradient, it does not always ensure the fastest convergence.

**Newton’s method**

This is a second-order algorithm as it leverages the Hessian matrix. Newton’s method aims to find better training directions by making use of the second derivatives of the loss function. Here, we’ll denote *f[w(i)] = f(i), ∇f[w(i)]=g(i)*, and *Hf[w(i)] = H(i)*. Now, we’ll consider the quadratic approximation of *f* at *w(0)* using Taylor’s series expansion, like so:

*f = f(0)+g(0)⋅[w−w(0)] + 0.5⋅[w−w(0)]2⋅H(0)*

Here, *H(0)* is the Hessian matrix of *f* calculated at the point *w(0)*. By considering *g = 0* for the minimum of *f(w)*, we get the following equation:

*g = g(0)+H(0)⋅(w−w(0))=0*

As a result, we can see that starting from the parameter vector w(0), Newton’s method iterates as follows:

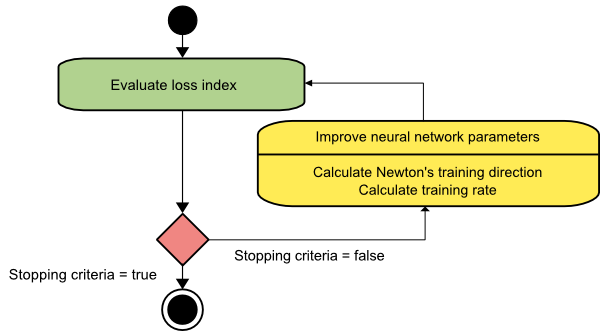
*w(i+1) = w(i)−H(i)−1⋅g(i)*

Here, *i = 0,1*,… and the vector *H(i)−1⋅g(i)* is referred to as “Newton’s Step.” You must remember that the parameter change may move towards a maximum instead of going in the direction of a minimum. Usually, this happens if the Hessian matrix is not positive definite, thereby causing the function evaluation to be reduced at each iteration. However, to avoid this issue, we usually modify the method equation as follows:

*w(i+1) = w(i)−(H(i)−1⋅g(i))η*

Here, *i = 0,1*,….

You can either set the training rate η to a fixed value or the value obtained via line minimization. So, the vector *d(i)=H(i)−1⋅g(i)* becomes the training direction for Newton’s method.



The major drawback of Newton’s method is that the exact evaluation of the Hessian and its inverse are pretty expensive computations.

**Conjugate gradient**

The conjugate gradient method falls between the gradient descent and Newton’s method. It is an intermediate algorithm – while it aims to accelerate the slow convergence factor of the gradient descent method, it also eliminates the need for the information requirements concerning the evaluation, storage, and inversion of the Hessian matrix usually required in Newton’s method.

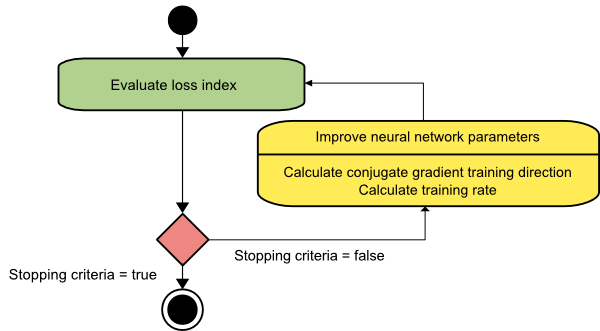
The conjugate gradient training algorithm performs the search in the conjugate directions that delivers faster convergence than gradient descent directions. These training directions are conjugated in accordance to the Hessian matrix. Here, d denotes the training direction vector. If we start with an initial parameter vector [w(0)] and an initial training direction vector [d(0)=−g(0)], the conjugate gradient method generates a sequence of training directions represented as:

d(i+1) = g(i+1)+d(i)⋅γ(i),

Here, i = 0,1,… and γ is the conjugate parameter. The training direction for all the conjugate gradient algorithms is periodically reset to the negative of the gradient. The parameters are improved, and the training rate (η) is achieved via line minimization, according to the expression shown below:

w(i+1) = w(i)+d(i)⋅η(i)

Here, i = 0,1,…



**ResNet**

**What is ResNet?**

According to the universal approximation theorem, given enough capacity, we know that a feedforward network with a single layer is sufficient to represent any function. However, the layer might be massive, and the network is prone to overfitting the data. Therefore, there is a common trend in the research community that our network architecture needs to go deeper.

Since AlexNet, the state-of-the-art convolutional neural network (CNN) architecture is going deeper and deeper. While AlexNet had only five convolutional layers, the [VGG network](https://arxiv.org/abs/1409.1556) and [GoogleNet](https://ieeexplore.ieee.org/document/7298594" \t "_blank) (also codenamed Inception\_v1) had 19 and 22 layers respectively.

However, you can’t simply stack layers together to increase network depth. Deep networks are hard to train because of the notorious vanishing gradient problem. As the gradient is backpropagated to earlier layers, repeated multiplication may make the gradient infinitely small. As a result, the deeper the network goes, the more its performance becomes saturated or even starts rapidly degrading.

Before ResNet, there had been several ways to deal with the vanishing gradient issue. For instance, GoogleNet adds an auxiliary loss in a middle layer for extra supervision, but none of those solutions seemed to really tackle the problem once and for all.

The core idea of ResNet is that it introduced a so-called “identity shortcut connection” that skips one or more layers.

The authors of the study on deep residual learning for image recognition argue that stacking layers shouldn’t degrade the network performance because we could simply stack identity mappings — a layer that doesn’t do anything — on top of the current network, and the resulting architecture would perform the same. This indicates that the deeper model should not produce a training error higher than its shallower counterparts. They hypothesize that letting the stacked layers fit a residual mapping is easier than letting them directly fit the desired underlying mapping. And the residual block above explicitly allows it to do precisely that.

As a matter of fact, ResNet was not the first to make use of shortcut connections. The authors of a study on Highway Network also introduced gated shortcut connections. These parameterized gates control how much information is allowed to flow across the shortcut. A similar idea can be found in the report on long short-term memory (LSTM) cell, in which there is a parameterized forget gate that controls how much information will flow to the next time step. Therefore, ResNet can be thought of as a special case of highway network.

However, experiments show that the highway network performs no better than ResNet, which is unusual because the solution space of highway network contains ResNet. Therefore, it should perform at least as good as ResNet. This suggests that it is more important to keep these “gradient highways” clear than to go for a larger solution space.

Following this intuition, the authors of deep residual learning for image recognition refined the residual block and proposed in a study on identity mappings in deep ResNets a pre-activation variant of residual block, in which the gradients can flow unimpeded through the shortcut connections to any other earlier layer. In fact, using the original residual block in image recognition study, training a 1202-layer ResNet resulted in a worse performance than its 110-layer counterpart.

The authors of identity mappings in deep ResNets demonstrated with experiments that they can now train a 1001-layer deep ResNet to outperform its shallower counterparts. Because of its compelling results, ResNet quickly became one of the most popular architectures for various computer vision tasks.

Residual Network (ResNet) is one of the famous deep learning models that was introduced by Shaoqing Ren, Kaiming He, Jian Sun, and Xiangyu Zhang in their paper. The paper was named “Deep Residual Learning for Image Recognition” in 2015. The ResNet model is one of the popular and most successful deep learning models so far.

**Residual Blocks**

The problem of training very deep networks has been relieved with the introduction of these Residual blocks and the ResNet model is made up of these blocks.

The problem of training very deep networks has been relieved with the introduction of these Residual blocks and the ResNet model is made up of these blocks.

In the above figure, the very first thing we can notice is that there is a direct connection that skips some layers of the model. This connection is called ’skip connection’ and is the heart of residual blocks. The output is not the same due to this skip connection. Without the skip connection, input ‘X gets multiplied by the weights of the layer followed by adding a bias term.

Then comes the activation function, f() and we get the output as H(x).

**H(x)=f( wx + b ) or H(x)=f(x)**

Now with the introduction of a new skip connection technique, the output is H(x) is changed to

**H(x)=f(x)+x**

But the dimension of the input may be varying from that of the output which might happen with a convolutional layer or pooling layers. Hence, this problem can be handled with these two approaches:

· Zero is padded with the skip connection to increase its dimensions.

· 1×1 convolutional layers are added to the input to match the dimensions. In such a case, the output is:

**H(x)=f(x)+w1.x**

Here an additional parameter w1 is added whereas no additional parameter is added when using the first approach.

These skip connections technique in ResNet solves the problem of vanishing gradient in deep CNNs by allowing alternate shortcut path for the gradient to flow through. Also, the skip connection helps if any layer hurts the performance of architecture, then it will be skipped by regularization.

## Architecture of ResNet

There is a 34-layer plain network in the architecture that is inspired by VGG-19 in which the shortcut connection or the skip connections are added. These skip connections or the residual blocks then convert the architecture into the residual network as shown in the figure below.

## 

## ResNet with Keras:

Keras is an open-source deep-learning library capable of running on top of TensorFlow.

Keras Applications provides the following ResNet versions.

– ResNet50

– ResNet50V2

– ResNet101

– ResNet101V2

– ResNet152

– ResNet152V2

### Build ResNet from scratch:

### 

### ResNet architecture uses the CNN blocks multiple times, so let us create a class for CNN block, which takes input channels and output channels. There is a batchnorm2d after each conv layer.

import torch

import torch.nn as nn

class block(nn.Module):

def \_\_init\_\_(

self, in\_channels, intermediate\_channels, identity\_downsample=None, stride=1

):

super(block, self).\_\_init\_\_()

self.expansion = 4

self.conv1 = nn.Conv2d(

in\_channels, intermediate\_channels, kernel\_size=1, stride=1, padding=0, bias=False

)

self.bn1 = nn.BatchNorm2d(intermediate\_channels)

self.conv2 = nn.Conv2d(

intermediate\_channels,

intermediate\_channels,

kernel\_size=3,

stride=stride,

padding=1,

bias=False

)

self.bn2 = nn.BatchNorm2d(intermediate\_channels)

self.conv3 = nn.Conv2d(

intermediate\_channels,

intermediate\_channels \* self.expansion,

kernel\_size=1,

stride=1,

padding=0,

bias=False

)

self.bn3 = nn.BatchNorm2d(intermediate\_channels \* self.expansion)

self.relu = nn.ReLU()

self.identity\_downsample = identity\_downsample

self.stride = stride

def forward(self, x):

identity = x.clone()

x = self.conv1(x)

x = self.bn1(x)

x = self.relu(x)

x = self.conv2(x)

x = self.bn2(x)

x = self.relu(x)

x = self.conv3(x)

x = self.bn3(x)

if self.identity\_downsample is not None:

identity = self.identity\_downsample(identity)

x += identity

x = self.relu(x)

return x

Then create a ResNet class that takes the input of a number of blocks, layers, image channels, and the number of classes.

In the below code the function ‘\_make\_layer’  
creates the ResNet layers, which takes the input of blocks, number of residual  
blocks, out channel, and strides.

class ResNet(nn.Module):

def \_\_init\_\_(self, block, layers, image\_channels, num\_classes):

super(ResNet, self).\_\_init\_\_()

self.in\_channels = 64

self.conv1 = nn.Conv2d(image\_channels, 64, kernel\_size=7, stride=2, padding=3, bias=False)

self.bn1 = nn.BatchNorm2d(64)

self.relu = nn.ReLU()

self.maxpool = nn.MaxPool2d(kernel\_size=3, stride=2, padding=1)

# Essentially the entire ResNet architecture are in these 4 lines below

self.layer1 = self.\_make\_layer(

block, layers[0], intermediate\_channels=64, stride=1

)

self.layer2 = self.\_make\_layer(

block, layers[1], intermediate\_channels=128, stride=2

)

self.layer3 = self.\_make\_layer(

block, layers[2], intermediate\_channels=256, stride=2

)

self.layer4 = self.\_make\_layer(

block, layers[3], intermediate\_channels=512, stride=2

)

self.avgpool = nn.AdaptiveAvgPool2d((1, 1))

self.fc = nn.Linear(512 \* 4, num\_classes)

def forward(self, x):

x = self.conv1(x)

x = self.bn1(x)

x = self.relu(x)

x = self.maxpool(x)

x = self.layer1(x)

x = self.layer2(x)

x = self.layer3(x)

x = self.layer4(x)

x = self.avgpool(x)

x = x.reshape(x.shape[0], -1)

x = self.fc(x)

return x

def \_make\_layer(self, block, num\_residual\_blocks, intermediate\_channels, stride):

identity\_downsample = None

layers = []

# Either if we half the input space for ex, 56x56 -> 28x28 (stride=2), or channels changes

# we need to adapt the Identity (skip connection) so it will be able to be added

# to the layer that's ahead

if stride != 1 or self.in\_channels != intermediate\_channels \* 4:

identity\_downsample = nn.Sequential(

nn.Conv2d(

self.in\_channels,

intermediate\_channels \* 4,

kernel\_size=1,

stride=stride,

bias=False

),

nn.BatchNorm2d(intermediate\_channels \* 4),

)

layers.append(

block(self.in\_channels, intermediate\_channels, identity\_downsample, stride)

)

# The expansion size is always 4 for ResNet 50,101,152

self.in\_channels = intermediate\_channels \* 4

# For example for first resnet layer: 256 will be mapped to 64 as intermediate layer,

# then finally back to 256. Hence no identity downsample is needed, since stride = 1,

# and also same amount of channels.

for i in range(num\_residual\_blocks - 1):

layers.append(block(self.in\_channels, intermediate\_channels))

return nn.Sequential(\*layers)

Then define different versions of ResNet

– For ResNet50 the layer sequence is [3, 4, 6, 3].

– For ResNet101 the layer sequence is [3, 4, 23, 3].

– For ResNet152 the layer sequence is [3, 8, 36, 3].

def ResNet50(img\_channel=3, num\_classes=1000):

return ResNet(block, [3, 4, 6, 3], img\_channel, num\_classes)

def ResNet101(img\_channel=3, num\_classes=1000):

return ResNet(block, [3, 4, 23, 3], img\_channel, num\_classes)

def ResNet152(img\_channel=3, num\_classes=1000):

return ResNet(block, [3, 8, 36, 3], img\_channel, num\_classes)

Then write a small test code to check whether the model is working fine.

def test():

net = ResNet101(img\_channel=3, num\_classes=1000)

device = "cuda" if torch.cuda.is\_available() else "cpu"

y = net(torch.randn(4, 3, 224, 224)).to(device)

print(y.size())

test()

### DEEP NETWORK WITH STOCHASTIC DEPTH

Although ResNet has proven powerful in many applications, one major drawback is that a deeper network usually requires weeks for training, making it practically infeasible in real-world applications. To tackle this issue, the researchers for a study on “Deep Networks with Stochastic Depth” introduced a counter-intuitive method of randomly dropping layers during training and using the full network in testing.

The authors used the residual block as their network’s building block. Therefore, during training, when a particular residual block is enabled, its input flows through both the identity shortcut and the weight layers, otherwise the input only flows through the identity shortcut. In training time, each layer has a “survival probability” and is randomly dropped. In testing time, all blocks are kept active and re-calibrated according to its survival probability during training.

Formally, let H\_l be the output of the l\_th residual block, f\_l be the mapping defined by the l\_th block’s weighted mapping, b\_l  be a Bernoulli random variable that can only be a one or zero (indicating whether a block is active), during training:

resnet architecture equation

When b\_l = 1, this block becomes a normal residual block. And when b\_l = 0, the above formula becomes:

resnet architecture updated equation

Since we know that H\_(l-1) is the output of a ReLU, which is already non-negative, the above equation reduces to an identity layer that only passes the input through to the next layer:

resnet architecture next layer equation

Let p\_l be the survival probability of layer l during training, during test time, we have:

resnet architecture updated equation

The authors applied a linear decay rule to the survival probability of each layer. They argue that since earlier layers extract low-level features that will be used by later ones, they should not be dropped too frequently. The resulting rule therefore becomes:

resnet architecture linear decay rule

Where L denotes the total number of blocks, thus p\_L is the survival probability of the last residual block and is fixed to 0.5 throughout experiments. Also note that in this setting, the input is treated as the first layer (l = 0) and thus, is never dropped. The overall framework over stochastic depth training is demonstrated in the figure below.

Similar to Dropout, training a deep network with stochastic depth can be viewed as training an ensemble of many smaller ResNets. The difference is that this method randomly drops an entire layer while Dropout only drops part of the hidden units in one layer during training.

Experiments show that training a 110-layer ResNet with stochastic depth results in better performance than training a constant-depth 110-layer ResNet, while also dramatically reducing the training time. This suggests that some of the layers (paths) in ResNet might be redundant.

## ResNet as an Ensemble of Smaller Networks

In the study on deep networks with stochastic depth, the researchers proposed a counter-intuitive way of training a very deep network that involved randomly dropping its layers during training and using the full network in testing time. The researchers of the study, “Residual Networks Behave Like Ensembles of Relatively Shallow Networks” had an even more counter-intuitive finding. We can actually drop some of the layers of a trained ResNet and still have comparable performance. This makes the ResNet architecture even more interesting, as the study authors also dropped layers of a VGG network and degraded its performance dramatically.

This study first provides an unraveled view of ResNet to make things clearer. After we unroll the network architecture, it is quite clear that a ResNet architecture with i residual blocks has 2^i different paths (because each residual block provides two independent paths).

Given that finding, it is quite clear why removing a couple of layers in a ResNet architecture doesn’t compromise its performance too much. The architecture has many independent effective paths and the majority of them remain intact after we remove a couple of layers. On the contrary, the VGG network has only one effective path, so removing a single layer compromises this one. As shown in the study’s extensive experiments.

The authors also conducted experiments to show that the collection of paths in ResNet have ensemble-like behavior. They did so by deleting different numbers of layers at test time, and checked to see if the performance of the network smoothly correlated with the number of deleted layers. The results suggested that the network indeed behaves like an ensemble.

Finally, the authors looked into the characteristics of the paths in ResNet.

It is apparent that the distribution of all possible path lengths follows a binomial distribution. The majority of paths go through 19 to 35 residual blocks.

The authors also conducted experiments to investigate the relationship between path length and the magnitude of the gradients flowing through it. To get the magnitude of gradients in the path of length k, the authors first fed a batch of data to the network and randomly sampled k residual blocks. When backpropagating the gradients, they propagated through the weight layer only for the sampled residual blocks. Their graphs show that the magnitude of gradients decreases rapidly as the path becomes longer.

We can now multiply the frequency of each path length with its expected magnitude of gradients to get a feel for how many paths of each length contribute to training. Surprisingly, most contributions come from paths of length nine to 18, but they constitute only a tiny portion of the total paths. This is a very interesting finding, as it suggests that ResNet did not solve the vanishing gradients problem for very long paths, and that ResNet actually enables training very deep networks by shortening its effective paths.

In this article, I revisited the compelling ResNet architecture and briefly explained the intuitions behind its recent success. I hope it helps strengthen your understanding of this groundbreaking work.

**AlexNet**

Alexnet won the Imagenet large-scale visual recognition challenge in 2012. The model was proposed in 2012 in the research paper named Imagenet Classification with Deep Convolution Neural Network by Alex Krizhevsky and his colleagues.

*In this model, the depth of the network was increased in comparison to Lenet-5. In case you want to know more about Lenet-5, I will recommend you to check the following article-*

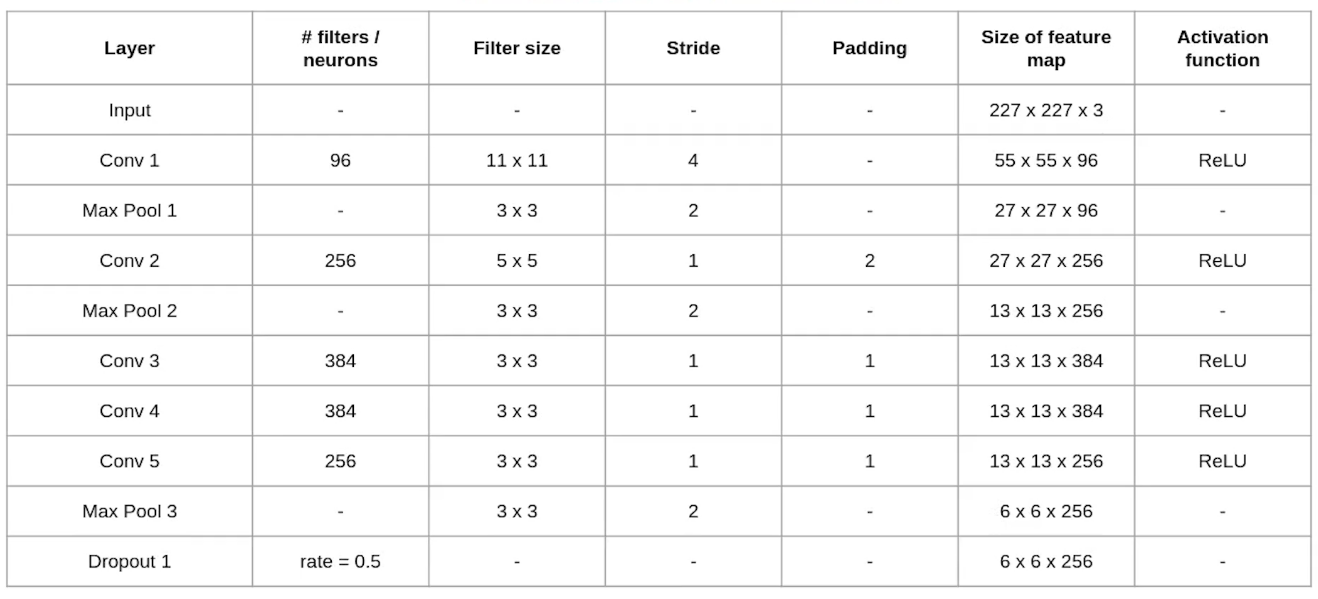
* The Architecture of Lenet-5

The Alexnet has eight layers with learnable parameters. The model consists of five layers with a combination of max pooling followed by 3 fully connected layers and they use Relu activation in each of these layers except the output layer.

They found out that using the relu as an activation function accelerated the speed of the training process by almost six times. They also used the dropout layers, that prevented their model from overfitting. Further, the model is trained on the Imagenet dataset. The Imagenet dataset has almost 14 million images across a thousand classes.

## Alexnet Architecture

One thing to note here, since Alexnet is a deep architecture, the authors introduced padding to prevent the size of the feature maps from reducing drastically. The input to this model is the images of size 227X227X3.



### Convolution and Maxpooling Layers

Then we apply the first convolution layer with 96 filters of size 11X11 with stride 4. The activation function used in this layer is relu. The output feature map is 55X55X96.

In case, you are unaware of how to calculate the output size of a convolution layer

output= ((Input-filter size)/ stride)+1

Also, the number of filters becomes the channel in the output feature map.

Next, we have the first Maxpooling layer, of size 3X3 and stride 2. Then we get the resulting feature map with the size 27X27X96.

After this, we apply the second convolution operation. This time the filter size is reduced to 5X5 and we have 256 such filters. The stride is 1 and padding 2. The activation function used is again relu. Now the output size we get is 27X27X256.

Again we applied a max-pooling layer of size 3X3 with stride 2. The resulting feature map is of shape 13X13X256.

Now we apply the third convolution operation with 384 filters of size 3X3 stride 1 and also padding 1. Again the activation function used is relu. The output feature map is of shape 13X13X384.

Then we have the fourth convolution operation with 384 filters of size 3X3. The stride along with the padding is 1. On top of that activation function used is relu. Now the output size remains unchanged i.e 13X13X384.

After this, we have the final convolution layer of size  3X3 with 256 such filters. The stride and padding are set to one also the activation function is relu. The resulting feature map is of shape 13X13X256.

So if you look at the architecture till now, the number of filters is increasing as we are going deeper. Hence it is extracting more features as we move deeper into the architecture. Also, the filter size is reducing, which means the initial filter was larger and as we go ahead the filter size is decreasing, resulting in a decrease in the feature map shape.

Next, we apply the third max-pooling layer of size 3X3 and stride 2. Resulting in the feature map of the shape 6X6X256.

### Fully Connected and Dropout Layers:



After this, we have our first dropout layer. The drop-out rate is set to be 0.5.Then we have the first fully connected layer with a relu activation function. The size of the output is 4096. Next comes another dropout layer with the dropout rate fixed at 0.5.This followed by a second fully connected layer with 4096 neurons and relu activation.

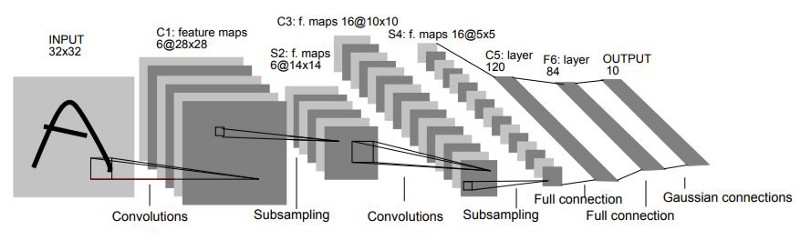
To quickly summarize the architecture that we have seen :

* It has 8 layers with learnable parameters.
* The input to the Model is RGB images.
* It has 5 convolution layers with a combination of max-pooling layers.
* Then it has 3 fully connected layers.
* The activation function used in all layers is Relu.
* It used two Dropout layers.
* The activation function used in the output layer is Softmax.
* The total number of parameters in this architecture is 62.3 million.

**LeNet**

Yann LeCun, Leon Bottou, Yosuha Bengio and Patrick Haffner proposed a neural network architecture for handwritten and machine-printed character recognition in 1990’s which they called LeNet-5. The architecture is straightforward and simple to understand that’s why it is mostly used as a first step for teaching Convolutional Neural Network.

### ****LeNet-5 Architecture****

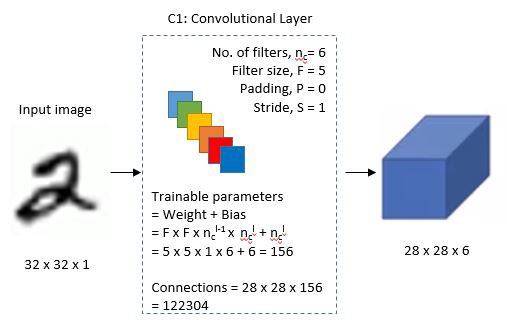


Original image published in [LeCun et al., 1998]

The LeNet-5 architecture consists of two sets of convolutional and average pooling layers, followed by a flattening convolutional layer, then two fully-connected layers and finally a softmax classifier.

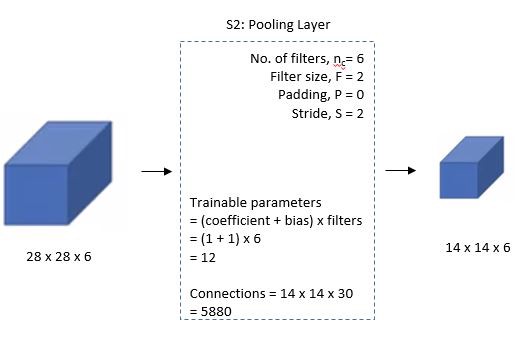
**First Layer:**

The input for LeNet-5 is a 32×32 grayscale image which passes through the first convolutional layer with 6 feature maps or filters having size 5×5 and a stride of one. The image dimensions changes from 32x32x1 to 28x28x6.



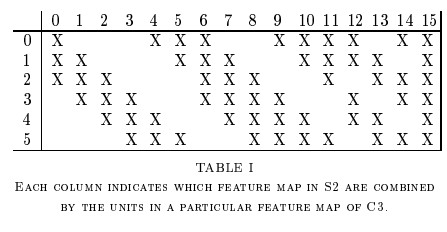
**Second Layer:**

Then the LeNet-5 applies average pooling layer or sub-sampling layer with a filter size 2×2 and a stride of two. The resulting image dimensions will be reduced to 14x14x6.

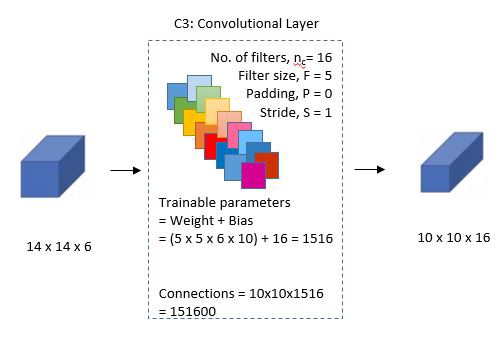


**Third Layer:**

Next, there is a second convolutional layer with 16 feature maps having size 5×5 and a stride of 1. In this layer, only 10 out of 16 feature maps are connected to 6 feature maps of the previous layer as shown below.

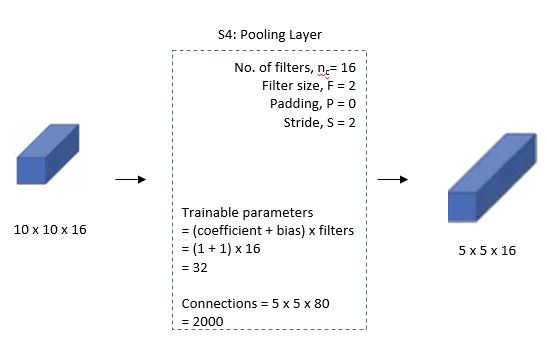


The main reason is to break the symmetry in the network and keeps the number of connections within reasonable bounds. That’s why the number of training parameters in this layers are 1516 instead of 2400 and similarly, the number of connections are 151600 instead of 240000.



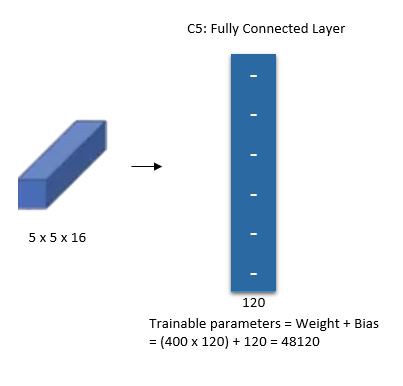
**Fourth Layer:**

The fourth layer (S4) is again an average pooling layer with filter size 2×2 and a stride of 2. This layer is the same as the second layer (S2) except it has 16 feature maps so the output will be reduced to 5x5x16.



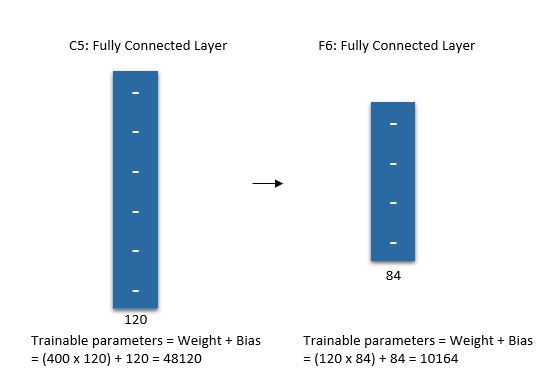
**Fifth Layer:**

The fifth layer (C5) is a fully connected convolutional layer with 120 feature maps each of size 1×1. Each of the 120 units in C5 is connected to all the 400 nodes (5x5x16) in the fourth layer S4.



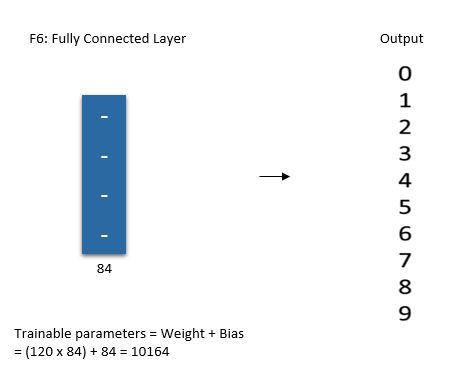
**Sixth Layer:**

The sixth layer is a fully connected layer (F6) with 84 units.



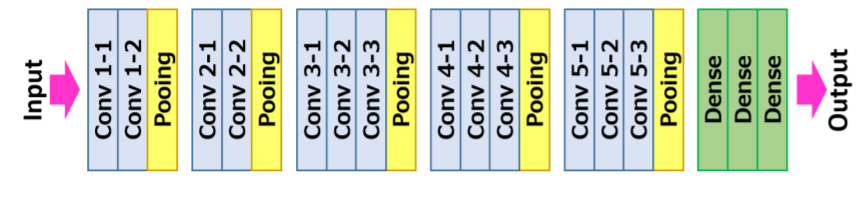
**Output Layer:**

Finally, there is a fully connected softmax output layer ŷ with 10 possible values corresponding to the digits from 0 to 9.



**VGG**

VGG- Network is a convolutional neural network model proposed by K. Simonyan and A. Zisserman in the paper “Very Deep Convolutional Networks for Large-Scale Image Recognition” [1]. This architecture achieved top-5 test accuracy of 92.7% in ImageNet, which has over 14 million images belonging to 1000 classes.It is one of the famous architectures in the deep learning field. Replacing large kernel-sized filters with 11 and 5 in the first and second layer respectively showed the improvement over AlexNet architecture, with multiple 3×3 kernel-sized filters one after another. It was trained for weeks and was using NVIDIA Titan Black GPU’s.



AlexNet came out in 2012 and was a revolutionary advancement; it improved on traditional Convolutional Neural Networks (CNNs) and became one of the best models for image classification… until VGG came out.

When AlexNet was published, it easily won the ImageNet Large-Scale Visual Recognition Challenge (ILSVRC) and proved itself to be one of the most capable models for object-detection out there. Its key features include using ReLU instead of the tanh function, optimization for multiple GPUs, and overlapping pooling. It addressed overfitting by using data augmentation and dropout. So what was wrong with AlexNet? Well nothing was, say, particularly “wrong” with it. People just wanted even more accurate models.

**The Dataset.** The general baseline for image recognition is ImageNet, a dataset that consists of more than 15 million images labeled with more than 22 thousand classes. Made through web-scraping images and crowd-sourcing human labelers, ImageNet even hosts its own competition: the previously mentioned ImageNet Large-Scale Visual Recognition Challenge (ILSVRC). Researchers from around the world are challenged to innovate methodology that yields the lowest top-1 and top-5 error rates (top-5 error rate would be the percent of images where the correct label is not one of the model’s five most likely labels). The competition gives out a 1,000 class training set of 1.2 million images, a validation set of 50 thousand images, and a test set of 150 thousand images; data is plentiful. AlexNet won this competition in 2012, and models based off of its design won the competition in 2013.

While previous derivatives of AlexNet focused on smaller window sizes and strides in the first convolutional layer, VGG addresses another very important aspect of CNNs: depth. Let’s go over the architecture of VGG:

* **Input.** VGG takes in a 224x224 pixel RGB image. For the ImageNet competition, the authors cropped out the center 224x224 patch in each image to keep the input image size consistent.
* **Convolutional Layers.**The convolutional layers in VGG use a very small receptive field (3x3, the smallest possible size that still captures left/right and up/down). There are also 1x1 convolution filters which act as a linear transformation of the input, which is followed by a ReLU unit. The convolution stride is fixed to 1 pixel so that the spatial resolution is preserved after convolution.
* **Fully-Connected Layers.** VGG has three fully-connected layers: the first two have 4096 channels each and the third has 1000 channels, 1 for each class.
* **Hidden Layers.**All of VGG’s hidden layers use ReLU (a huge innovation from AlexNet that cut training time). VGG does not generally use Local Response Normalization (LRN), as LRN increases memory consumption and training time with no particular increase in accuracy.

**The Difference.** VGG, while based off of AlexNet, has several differences that separates it from other competing models:

* Instead of using large receptive fields like AlexNet (11x11 with a stride of 4), VGG uses very small receptive fields (3x3 with a stride of 1). Because there are now three ReLU units instead of just one, the decision function is more discriminative. There are also fewer parameters (27 times the number of channels instead of AlexNet’s 49 times the number of channels).
* VGG incorporates 1x1 convolutional layers to make the decision function more non-linear without changing the receptive fields.

The small-size convolution filters allows VGG to have a large number of weight layers; of course, more layers leads to improved performance. This isn’t an uncommon feature, though. GoogLeNet, another model that uses deep CNNs and small convolution filters, was also showed up in the 2014 ImageNet competition.

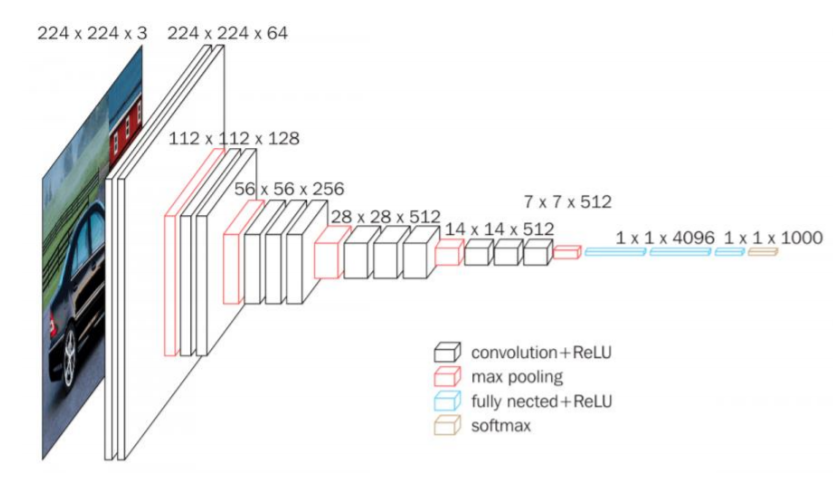
## VGG16 Architecture

The input to the convolution neural network is a fixed-size 224 × 224 RGB image. The only preprocessing it does is subtracting the mean RGB values, which are computed on the training dataset, from each pixel.

Then the image is running through a stack of convolutional (Conv.) layers, where there are filters with a very small receptive field that is 3 × 3, which is the smallest size to capture the notion of left/right, up/down, and center part.

In one of the configurations, it also utilizes 1 × 1 convolution filters, which can be observed as a linear transformation of the input channels followed by non-linearity. The convolutional strides are fixed to 1 pixel; the spatial padding of convolutional layer input is such that the spatial resolution is maintained after convolution, that is the padding is 1 pixel for 3 × 3 Conv. layers.

Then the Spatial pooling is carried out by five max-pooling layers, 16 which follow some of the Conv. layers but not all the Conv. layers are followed by max-pooling. This Max-pooling is performed over a 2 × 2-pixel window, with stride 2.



The architecture contains a stack of convolutional layers which have a different depth in different architectures which are followed by three Fully-Connected (FC) layers: the first two FC have 4096 channels each and the third FC performs 1000-way classification and thus contains 1000 channels that is one for each class.

The final layer is the soft-max layer. The configuration of the fully connected layers is similar in all networks.

All of the hidden layers are equipped with rectification (ReLU) non-linearity. Also, here one of the networks contains Local Response Normalization (LRN), such normalization does not improve the performance on the trained dataset, but usage of that leads to increased memory consumption and computation time.

Architecture Summary:

• Input to the model is a fixed size 224×224224×224 RGB image

• Pre-processing is subtracting the training set RGB value mean from each pixel

• Convolutional layers 17

           – Stride fixed to 1 pixel

           – padding is 1 pixel for 3×33×3

• Spatial pooling layers

           – This layer doesn’t count to the depth of the network by convention

           – Spatial pooling is done using max-pooling layers

           – window size is 2×22×2

           – Stride fixed to 2

           – Convnets used 5 max-pooling layers

• Fully-connected layers:

           • 1st: 4096 (ReLU).

            2nd: 4096 (ReLU).

            3rd: 1000 (Softmax).

### Architecture Configuration

The below figure contains the Convolution Neural Network configuration of the VGG net with the

following layers:

     • VGG-11

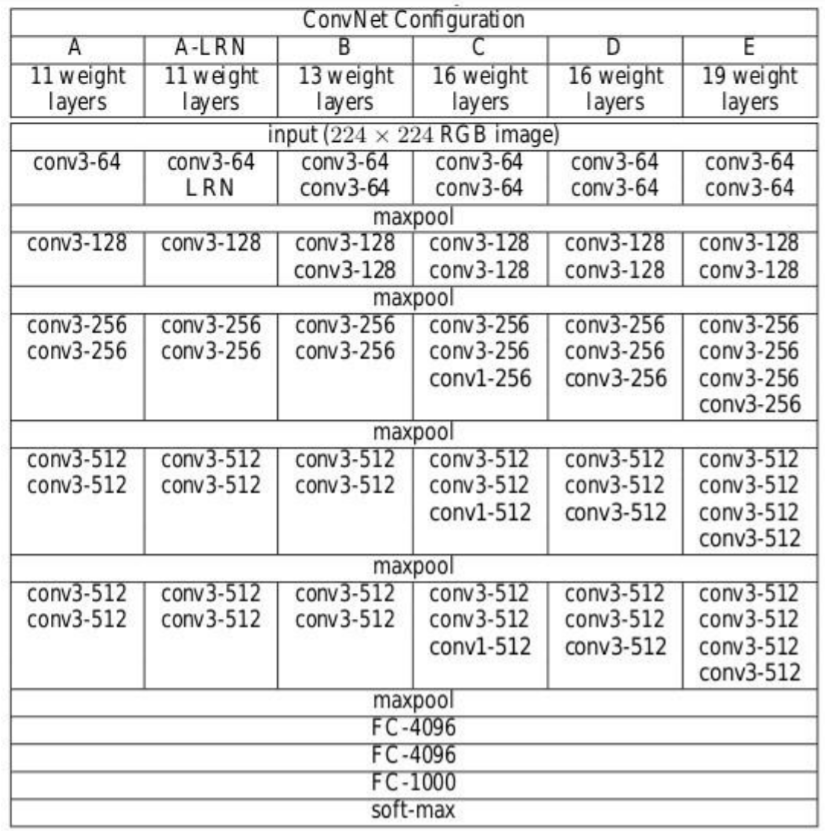
     • VGG-11 (LRN)

     • VGG-13

     • VGG-16 (Conv1)

     • VGG-16

     • VGG-19



The convolutional Neural Network configurations are mentioned above one per column.

In the following, the networks are referred by their names (A–E). All configurations follow the traditional design and differ only in the depth: from 11 weight layers in network A that is 8 Conv. and 3 FC layers to 19 weight layers in network E that is 16 Conv. and 3 FC layers. The width of each Conv. layer is the number of channels is rather small, which is starting from 64 in the first layer and then goes on increasing by a factor of 2 after each max-pooling layer until it reaches 512.

The number of parameters for each configuration is described below. Although it has a large depth, the number of weights in the networks is not greater than the number of weights in a shallower net with larger Conv. layer widths and receptive fields

## Training

• Loss function is multinomial logistic regression

• Learning algorithm is mini-batch stochastic gradient descent (SGD) based on back-propagation with momentum

    · Batch size was 256

    · Momentum was 0.9

• Regularization

    · L2 Weight decay (penalty multiplier was 0.0005)

    · Dropout for first two fully connected layers is set to 0.5

• Learning rate

   · Initial: 0.01

   · When validation set accuracy stopped improving it is decreased to 10.

• Though it has a greater number of parameters and also depth compared to Alexnet, the CNN’s required less epochs for loss function to converge due to

   · Small convolutional kernels and more regularization by large depth.

   · Pre-initialization of certain layers.

• Training image size

   · S is the smallest side of the isotopically-rescaled image

   · Two approaches for setting S

 Fix S, known as single scale training

 Here S = 256 and S = 384

 Vary S, known as multi-scale training

 S from [Smin, Smax] where Smin = 256, Smax = 512

– Then 224×224224×224  
the image was randomly cropped from rescaled image per SGD iteration.

## Key Features

• VGG16 has a total of 16 layers that has some weights.

• Only Convolution and pooling layers are used.

• Always uses a 3 x 3 Kernel for convolution. 20

• 2×2 size of the max pool.

• 138 million parameters.

• Trained on ImageNet data.

• It has an accuracy of 92.7%.

• Another version that is VGG 19, has a total of 19 layers with weights.

• It is a very good Deep learning architecture for benchmarking on any particular task.

• The pre-trained networks for VGG is made open-source, so it can be commonly used out of the box for various types of applications.

## Implement VGG Net

First Let’s create the filter mapping for each version of the VGG net. Refer to the above configuration image to know about the number of filters. That is create a dictionary for the version with a key named VGG11, VGG13, VGG16, VGG19 and create a list according to the number of filters in each version respectively. Here “M” in the list is known as Maxpool operation.

import torch

import torch.nn as nn

VGG\_types = {

"VGG11": [64, "M", 128, "M", 256, 256, "M", 512, 512, "M", 512, 512, "M"],

"VGG13": [64, 64, "M", 128, 128, "M", 256, 256, "M", 512, 512, "M", 512, 512, "M"],

"VGG16": [64,64,"M",128,128,"M",256,256,256,"M",512,512,512,"M",512,512,512,"M",],

"VGG19": [64,64,"M",128,128,"M",256,256,256,256,"M",512,512,512,512,

"M",512,512,512,512,"M",],}

Create a global variable to mention the version of the architecture. Then create a class called VGG\_net with inputs as in\_channels and num\_classes, It takes inputs like a number of Image channels and the Number of output classes.

Initialize the Sequential layers, that is in the sequence, Linear layer–>ReLU–>Dropout.

Then create a function called create\_conv\_layers which takes VGGnet architecture configuration as input that is the list that we created above for different versions. When it comes across the letter “M” from the above list, it performs the MaxPool2d operation.

VGGType = "VGG16"

class VGGnet(nn.Module):

def \_\_init\_\_(self, in\_channels=3, num\_classes=1000):

super(VGGnet, self).\_\_init\_\_()

self.in\_channels = in\_channels

self.conv\_layers = self.create\_conv\_layers(VGG\_types[VGGType])

self.fcs = nn.Sequential(

nn.Linear(512 \* 7 \* 7, 4096),

nn.ReLU(),

nn.Dropout(p=0.5),

nn.Linear(4096, 4096),

nn.ReLU(),

nn.Dropout(p=0.5),

nn.Linear(4096, num\_classes),

)

def forward(self, x):

x = self.conv\_layers(x)

x = x.reshape(x.shape[0], -1)

x = self.fcs(x)

return x

def create\_conv\_layers(self, architecture):

layers = []

in\_channels = self.in\_channels

for x in architecture:

if type(x) == int:

out\_channels = x

layers += [

nn.Conv2d(

in\_channels=in\_channels,

out\_channels=out\_channels,

kernel\_size=(3, 3),

stride=(1, 1),

padding=(1, 1),

),

nn.BatchNorm2d(x),

nn.ReLU(),

]

in\_channels = x

elif x == "M":

layers += [nn.MaxPool2d(kernel\_size=(2, 2), stride=(2, 2))]

return nn.Sequential(\*layers)

Once this is done write a small test code to check whether our implementation is working well.

In the below test code the number of classes given is 500.

if \_\_name\_\_ == "\_\_main\_\_":

device = "cuda" if torch.cuda.is\_available() else "cpu"

model = VGGnet(in\_channels=3, num\_classes=500).to(device)

# print(model)

x = torch.randn(1, 3, 224, 224).to(device)

print(model(x).shape)

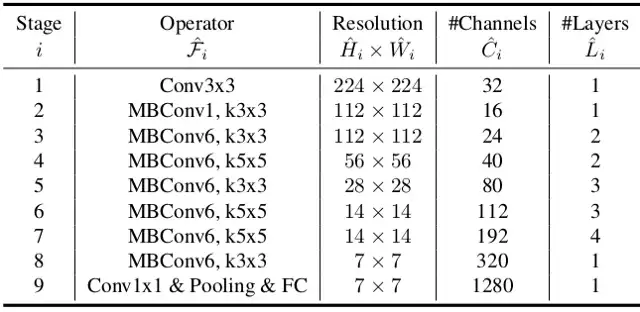
# **EfficientNet**

Ever since Alex net won the 2012 ImageNet Challenge, Convolutional Neural Networks have become ubiquitous in the world of Computer Vision. They have even found their applications in natural language processing, with state of the art models using convolution operations to retain context and provide better predictions. However, one of the key issues in designing CNNs, as with all other neural networks, is model scaling i.e deciding how to increase the model size so as to provide better accuracy.

This is a tedious process, requiring manual hit and trial until a sufficiently accurate model is produced that satisfies the resource constraints. The process is resource and time consuming and often yields models with sub-optimal accuracy and efficiency.

Taking this issue in consideration, Google released a paper in 2019 that dealt with a new family of CNNs i.e EfficientNet . These CNNs not only provide better accuracy but also improve the efficiency of the models by reducing the parameters and FLOPS (Floating Point Operations Per Second) manifold in comparison to the state of art models such as GPipe. The main contributions of this paper are:

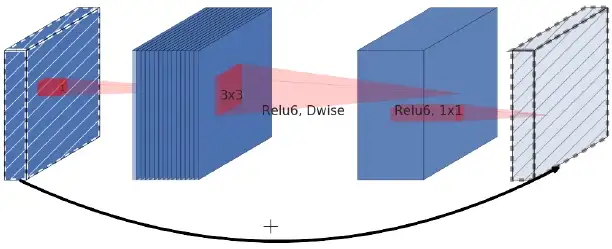
* Designing a simple mobile-size baseline architecture: **EfficientNet-B0**
* Providing an effective **compound scaling** method for increasing the model size to achieve maximum accuracy gains.



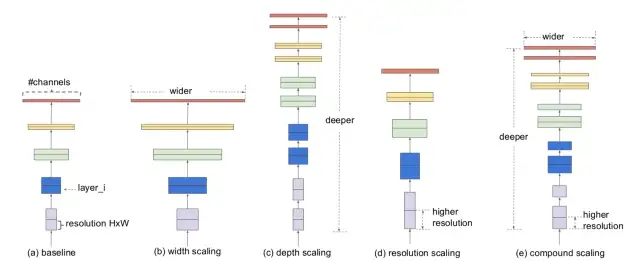
The compound scaling method can be generalized to existing CNN architectures such as Mobile Net and ResNet. However, choosing a good baseline network is critical for achieving the best results since the compound scaling method only **enhances** the predictive capacity of the networks by replicating base network’s underlying convolutional operations and network structure.

To this end, the authors use Neural Architecture Search to build an efficient network architecture, **EfficientNet-B0**. It achieves **77.3% accuracy** on ImageNet with only **5.3M**parameters and **0.39B** FLOPS. (Resnet-50 provides 76% accuracy with 26M parameters and 4.1B FLOPS).

The main building block of this network consists of MBConv to which squeeze-and-excitation optimization is added. MBConv is similar to the inverted residual blocks used in MobileNet v2. These form a shortcut connection between the beginning and end of a convolutional block. The input activation maps are first expanded using 1x1 convolutions to increase the depth of the feature maps. This is followed by 3x3 Depth-wise convolutions and Point-wise convolutions that reduce the number of channels in the output feature map. The shortcut connections connect the narrow layers whilst the wider layers are present between the skip connections. This structure helps in decreasing the overall number of operations required as well as the model size.



# **Compound Scaling**



Model Scaling. (a) is a baseline network example; (b)-(d) are conventional scaling that only increases one dimension of network width, depth, or resolution. (e) is our proposed compound scaling method that uniformly scales all three dimensions with a fixed ratio.

A convolutional neural network can be scaled in three dimensions: depth, width, resolution. The **depth** of the network corresponds to the number of layers in a network. The **width** is associated with the number of neurons in a layer or more pertinently, the number of filters in a convolutional layer. The **resolution** is simply the height and width of the input image. Figure 2 above, gives a clearer picture of scaling across these 3 dimensions.

Increasing the depth, by stacking more convolutional layers, allows the network to learn more complex features. However deeper networks tend to suffer from vanishing gradients and become difficult to train. Although new techniques such as batch normalization and skip connections are effective in resolving this problem, empirical studies suggest that the actual accuracy gains by only increasing the depth of the network quickly saturate. For instance Resnet-1000 provides the same accuracy as Resnet-100 despite all the extra layers.

Scaling the width of the networks allows layers to learn more fine grained features. This concept has been used extensively in numerous works such as Wide ResNet and Mobile Net. However, as is the case of increasing depth, increasing width prevents the network from learning complex features , resulting in diminishing accuracy gains.

Higher input resolution provides a greater detail about the image and hence enhances the model’s ability to reason about smaller objects and extract finer patterns. But like the other scaling dimensions, this too provides limited accuracy gains on its own.

This implies that the scaling of network for increase in accuracy should be contributed in part by a combination of the three dimensions. This is corroborated by empirical evidence, where the networks’s accuracy is modeled with an increasing width for various depth and resolution settings.

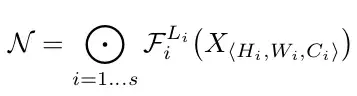
The results depict that scaling only one dimension (width) quickly stagnates the accuracy gains. However, coupling this with an increase in number of layers (depth) or input resolution enhances the models predictive capabilities.

These observations are somewhat expected and can be explained by intuition. For instance, if the spatial resolution of the input image is increased , the number of convolutional layers should also be increased so that the receptive field is large enough to span the entire image that now contains more pixels. This leads to the second observation :

**Observation 2:**In order to pursue better accuracy and efficiency, it is critical to balance all dimensions of network width, depth, and resolution during ConvNet scaling.

## ****The proposed scaling method****

A convolutional neural network can be thought of as stacking or composition of various convolutional layers. Furthermore these layers can be partitioned into different stages e.g ResNet has five stages, and all layers in each stage have the same convolutional type. Therefore, a CNN can be represented mathematically as:

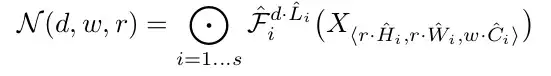


where N depicts the network, i represents the stage number, F ᵢ represents the convolution operation for the i-th stage, and L ᵢ represents the number of times F ᵢ is repeated in stage i. H ᵢ , W ᵢ and C ᵢ simply denote the input tensor shape for stage i.

As can be deduced from the equation 1, L ᵢ controls the depth of the network, C ᵢ is responsible for the width of the network whereas H ᵢ and W ᵢ affect the input resolution. Finding a set of good coefficients to scale these dimensions for each layer is impossible, since the search space is huge. So, in order to restrict the search space, the authors lay down a set of ground rules.

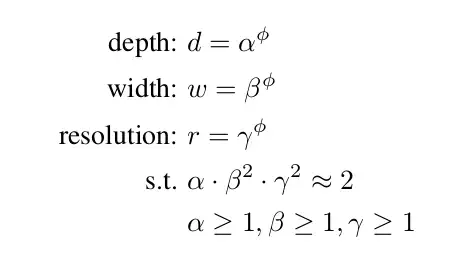
* All the layers/stages in the scaled models will use the same convolution operations as the baseline network
* All layers must be scaled uniformly with constant ratio

With these rules established , equation 1 can be parameterized as:



where w, d, r are coefficients for scaling network width,depth, and resolution; F̂ ᵢ , L̂ ᵢ , Ĥ ᵢ , Ŵ ᵢ , Ĉ ᵢ are predefined parameters in baseline network.

The authors propose a simple, albeit effective scaling technique that uses a **compound coefficient ɸ**to uniformly scale network width, depth, and resolution in a principled way:



ɸ is a user-defined, global scaling factor (integer) that controls how many resources are available whereas **α**,**β**, and **γ** determine how to assign these resources to network depth, width, and resolution respectively. The FLOPS of a convolutional operation are proportional to **d, w², r²,**sincedoubling the depth will double the FLOPS while doubling width or resolution increases FLOPS almost by four times. So ,scaling the network using equation 3 will increase the total FLOPS by (α \* β² \* γ²) ^ɸ . Hence, in order to make sure that the total FLOPS don’t exceed 2^ϕ, the constraint (α \* β² \* γ²) ≈ 2 is applied. What this means, is that if we have twice the resources available we can simply use compound coefficient of 1 to scale the FLOPS by 2¹.

The parameters - **α**,**β**, and **γ-**can be determined using grid search by setting **ɸ=1**and finding parameters that result in the best accuracy. Once found, these parameters can then be fixed , and the compound coefficient **ɸ**can be increased to get larger but more accurate models. This was how EfficientNet-B1 to EfficientNet-B7 are constructed , with the integer in the end of the name indicating the value of compound coefficient.

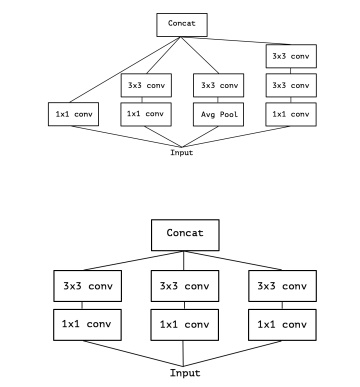
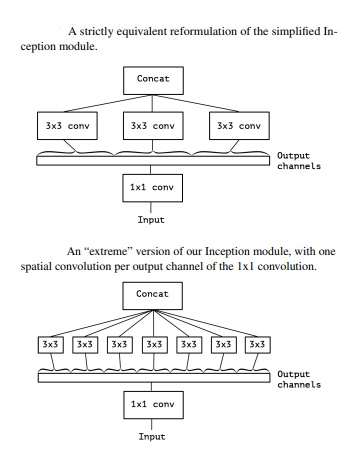
**Xception**

Convolutional neural networks have emerged as the master algorithm in computer vision in recent years, and developing recipes for designing them has been a subject of considerable attention. The history of convolutional neural network design started with LeNet-style models , which were simple stacks of convolutions for feature extraction and max-pooling operations for spatial sub-sampling. In 2012, these ideas were refined into the AlexNet architecture , where convolution operations were being repeated multiple times in-between max-pooling operations, allowing the network to learn richer features at every spatial scale. What followed was a trend to make this style of network increasingly deeper, mostly driven by the yearly ILSVRC competition; first with Zeiler and Fergus in 2013 and then with the VGG architecture in 2014 .At this point a new style of network emerged, the Inception architecture, introduced by Szegedy et al. in 2014 as GoogLeNet (Inception V1), later refined as Inception V2, Inception V3 , and most recently Inception-ResNet.Inception itself was inspired by the earlier Network In-Network architecture . Since its first introduction, Inception has been one of the best performing family of models on the ImageNet dataset , as well as internal datasets in use at Google, in particular JFT .The fundamental building block of Inception-style models is the Inception module, of which several different versions exist., as found in the Inception V3 architecture. An Inception model can be understood as a stack of such modules. This is a departure from earlier VGG-style networks which were stacks of simple convolution layers. While Inception modules are conceptually similar to convolutions (they are convolutional feature extractors)

**The Inception hypothesis**

A convolution layer attempts to learn filters in a 3D space, with 2 spatial dimensions (width and height) and a channel dimension; thus a single convolution kernel is tasked with simultaneously mapping cross-channel correlations and spatial correlations. This idea behind the Inception module is to make this process easier and more efficient by explicitly factoring it into a series of operations that would independently look at cross-channel correlations and at spatial correlations. More precisely, the typical Inception module first looks at crosschannel correlations via a set of 1x1 convolutions, mapping the input data into 3 or 4 separate spaces that are smaller than the original input space, and then maps all correlations in these smaller 3D spaces, via regular 3x3 or 5x5 convolutions. In effect, the fundamental hypothesis behind Inception is that cross-channel correlations and spatial correlations are sufficiently decoupled that it is preferable not to map them jointly 1.

Consider a simplified version of an Inception module that only uses one size of convolution (e.g. 3x3) and does not include an average pooling tower (figure 2). This Inception module can be reformulated as a large 1x1 convolution followed by spatial convolutions that would operate on nonoverlapping segments of the output channels. This observation naturally raises the question: what is the effect of the number of segments in the partition (and their size)? Would it be reasonable to make a much stronger hypothesis than the Inception hypothesis, and assume that cross-channel correlations and spatial correlations can be mapped completely separately?



network design as early as 2014 and has become more popular since its inclusion in the TensorFlow framework in 2016. A depthwise separable convolution, commonly called “separable convolution” in deep learning frameworks such as TensorFlow and Keras, consists in a depthwise convolution, i.e. a spatial convolution performed independently over each channel of an input, followed by a pointwise convolution, i.e. a 1x1 convolution, projecting the channels output by the depthwise convolution onto a new channel space. This is not to be confused with a spatially separable convolution, which is also commonly called “separable convolution” in the image processing community.Two minor differences between and “extreme” version of an Inception module and a depthwise separable convolution would be:

• The order of the operations: depthwise separable convolutions as usually implemented (e.g. in TensorFlow) perform first channel-wise spatial convolution and then perform 1x1 convolution, whereas Inception performs the 1x1 convolution first.

• The presence or absence of a non-linearity after the first operation. In Inception, both operations are followed by a ReLU non-linearity, however depthwise separable convolutions are usually implemented without non-linearities.

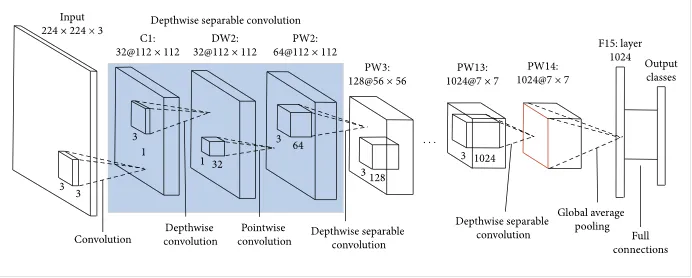
**The Xception architecture**

Xception based entirely on depthwise separable convolution layers. In effect, we make the following hypothesis: that the mapping of cross-channels correlations and spatial correlations in the feature maps of convolutional neural networks can be entirely decoupled. Because this hypothesis is a stronger version of the hypothesis underlying the Inception architecture, we name our proposed architecture Xception, which stands for “Extreme Inception”.

The Xception architecture has 36 convolutional layers forming the feature extraction base of the network. In our experimental evaluation we will exclusively investigate image classification and therefore our convolutional base will be followed by a logistic regression layer. Optionally one may insert fully-connected layers before the logistic regression layer, which is explored in the experimental evaluation section. The 36 convolutional layers are structured into 14 modules, all of which have linear residual connections around them, except for the first and last modules. In short, the Xception architecture is a linear stack of depthwise separable convolution layers with residual connections. This makes the architecture very easy to define and modify; it takes only 30 to 40 lines of code using a highlevel library such as Keras or TensorFlow-Slim, not unlike an architecture such as VGG-16, but rather unlike architectures such as Inception V2 or V3 which are far more complex to define. An open-source implementation of Xception using Keras and TensorFlow is provided as part of the Keras Applications module2 , under the MIT license.

**MobileNet**

As the name applied, the MobileNet model is designed to be used in mobile applications, and it is TensorFlow’s first mobile computer vision model.



MobileNet uses **depthwise separable** **convolutions.**It significantly **reduces the number of parameters** when compared to the network with regular convolutions with the same depth in the nets. This results in lightweight deep neural networks.

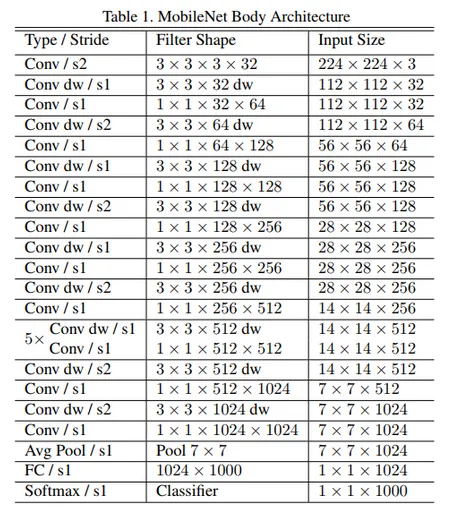
A depthwise separable convolution is made from two operations.

1. **Depthwise convolution.**
2. **Pointwise convolution**.

MobileNet is a class of CNN that was open-sourced by Google, and therefore, this gives us an excellent starting point for training our classifiers that are insanely small and insanely fast.

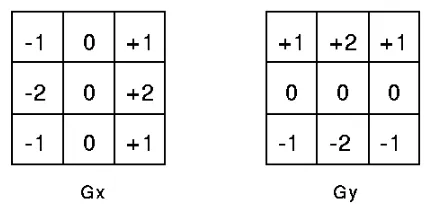
The speed and power consumption of the network is proportional to the number of MACs (Multiply-Accumulates) which is a measure of the number of fused Multiplication and Addition operations.

# **The Architecture of MobileNet**



# **Depthwise Separable Convolution**

This convolution originated from the idea that a filter’s depth and spatial dimension can be separated- thus, the name separable. Let us take the example of Sobel filter, used in image processing to detect edges.



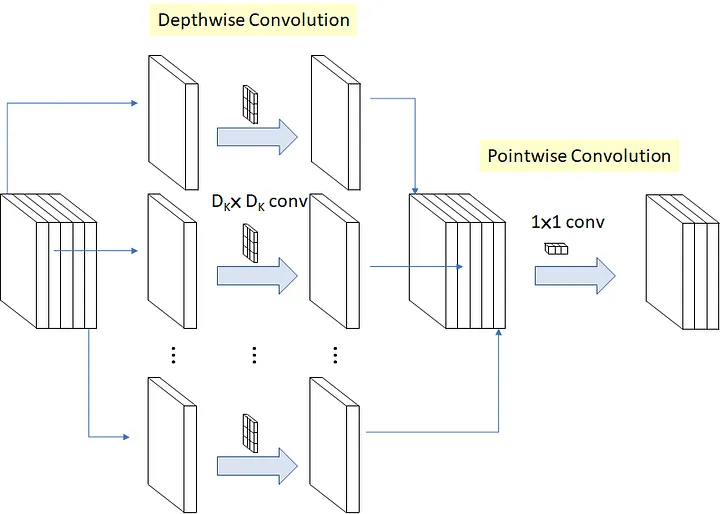
You can separate the height and width dimensions of these filters. Gx filter can be viewed as a matrix product of [1 2 1] transpose with [-1 0 1].We notice that the filter had disguised itself. It shows it had nine parameters, but it has 6. This has been possible because of the separation of its height and width dimensions.The same idea applied to separate depth dimension from horizontal (width\*height) gives us depth-wise separable convolution whare we perform depth-wise convolution. After that, we use a 1\*1 filter to cover the depth dimension.

One thing to notice is how much parameters are reduced by this convolution to output the same no. of channels. To produce one channel, we need 3\*3\*3 parameters to perform depth-wise convolution and 1\*3 parameters to perform further convolution in-depth dimension.

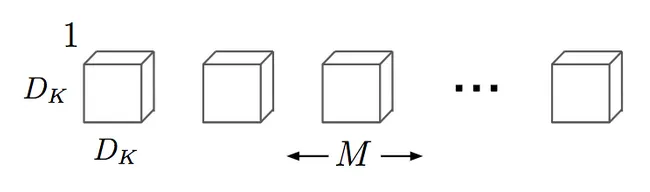
But If we need three output channels, we only need 31\*3 depth filter, giving us a total of 36

( = 27 +9) parameters while for the same no. of output channels in regular convolution, we need 33\*3\*3 filters giving us a total of 81 parameters.

Depthwise separable convolution is **a depthwise convolution followed by a pointwise convolution as follows:**



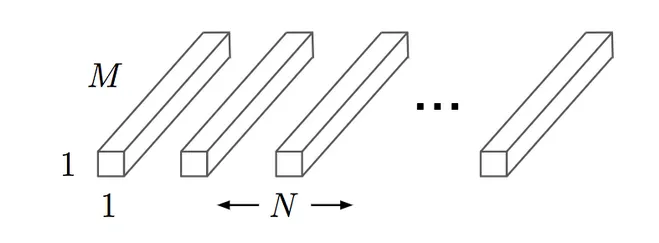
1. **Depthwise convolution** is the **channel-wise DK×DK spatial convolution**. Suppose in the figure above, and we have five channels; then, we will have 5 DK×DK spatial convolutions.
2. **Pointwise convolution** is the **1×1 convolution** to change the dimension.
3. **Depthwise convolution.**



It is a map of a single convolution on each input channel separately. Therefore its number of output channels is the same as the number of the input channels. Its computational cost is

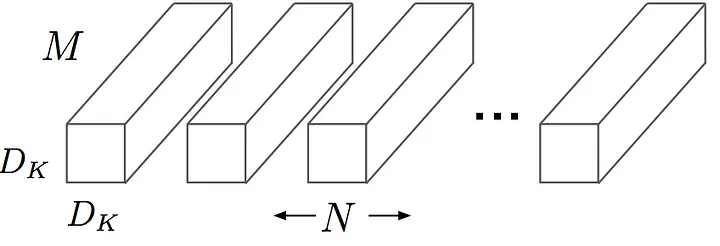
**Df² \* M \* Dk².**

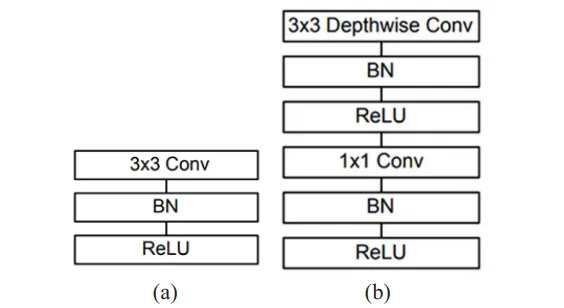
**2. pointwise convolution.**

 Convolution with a kernel size of 1x1 that simply combines the features created by the depthwise convolution. Its computational cost is

**M \* N \* Df²**

# Difference between Standard Convolution and Depthwise separable convolution

 The main difference between MobileNet architecture and a traditional CNN instead of a single 3x3 convolution layer followed by the batch norm and ReLU. Mobile Nets split the convolution into a 3x3 depth-wise conv and a 1x1 pointwise conv, as shown in the figure.

 (a) Standard convolutional layer with batch normalization and ReLU

(b) Depth-wise separable convolution with depth-wise and pointwise layers followed by batch normalization and ReLU.

**Approximate a Target Function in Machine Learning**

Supervised machine learning is best understood as approximating a target function (f) that maps input variables (X) to an output variable (Y).

Y = f(X)

This characterization describes the range of classification and prediction problems and the machine algorithms that can be used to address them.

An important consideration in learning the target function from the training data is how well the model generalizes to new data. Generalization is important because the data we collect is only a sample, it is incomplete and noisy.

**Generalization in Machine Learning**

In machine learning we describe the learning of the target function from training data as inductive learning.

Induction refers to learning general concepts from specific examples which is exactly the problem that supervised machine learning problems aim to solve. This is different from deduction that is the other way around and seeks to learn specific concepts from general rules.

Generalization refers to how well the concepts learned by a machine learning model apply to specific examples not seen by the model when it was learning.

The goal of a good machine learning model is to generalize well from the training data to any data from the problem domain. This allows us to make predictions in the future on data the model has never seen.

There is a terminology used in machine learning when we talk about how well a machine learning model learns and generalizes to new data, namely overfitting and underfitting.

Overfitting and underfitting are the two biggest causes for poor performance of machine learning algorithms.

**Statistical Fit**

In statistics, a fit refers to how well you approximate a target function.

This is good terminology to use in machine learning, because supervised machine learning algorithms seek to approximate the unknown underlying mapping function for the output variables given the input variables.

Statistics often describe the goodness of fit which refers to measures used to estimate how well the approximation of the function matches the target function.

Some of these methods are useful in machine learning (e.g. calculating the residual errors), but some of these techniques assume we know the form of the target function we are approximating, which is not the case in machine learning.

If we knew the form of the target function, we would use it directly to make predictions, rather than trying to learn an approximation from samples of noisy training data.

**Overfitting in Machine Learning**

Overfitting refers to a model that models the training data too well.

Overfitting happens when a model learns the detail and noise in the training data to the extent that it negatively impacts the performance of the model on new data. This means that the noise or random fluctuations in the training data is picked up and learned as concepts by the model. The problem is that these concepts do not apply to new data and negatively impact the models ability to generalize.

Overfitting is more likely with nonparametric and nonlinear models that have more flexibility when learning a target function. As such, many nonparametric machine learning algorithms also include parameters or techniques to limit and constrain how much detail the model learns.

For example, decision trees are a nonparametric machine learning algorithm that is very flexible and is subject to overfitting training data. This problem can be addressed by pruning a tree after it has learned in order to remove some of the detail it has picked up.

**Underfitting in Machine Learning**

Underfitting refers to a model that can neither model the training data nor generalize to new data.

An underfit machine learning model is not a suitable model and will be obvious as it will have poor performance on the training data.

Underfitting is often not discussed as it is easy to detect given a good performance metric. The remedy is to move on and try alternate machine learning algorithms. Nevertheless, it does provide a good contrast to the problem of overfitting.

**A Good Fit in Machine Learning**

Ideally, you want to select a model at the sweet spot between underfitting and overfitting.

This is the goal, but is very difficult to do in practice.

To understand this goal, we can look at the performance of a machine learning algorithm over time as it is learning a training data. We can plot both the skill on the training data and the skill on a test dataset we have held back from the training process.

Over time, as the algorithm learns, the error for the model on the training data goes down and so does the error on the test dataset. If we train for too long, the performance on the training dataset may continue to decrease because the model is overfitting and learning the irrelevant detail and noise in the training dataset. At the same time the error for the test set starts to rise again as the model’s ability to generalize decreases.

The sweet spot is the point just before the error on the test dataset starts to increase where the model has good skill on both the training dataset and the unseen test dataset.

You can perform this experiment with your favorite machine learning algorithms. This is often not useful technique in practice, because by choosing the stopping point for training using the skill on the test dataset it means that the testset is no longer “unseen” or a standalone objective measure. Some knowledge (a lot of useful knowledge) about that data has leaked into the training procedure.

There are two additional techniques you can use to help find the sweet spot in practice: resampling methods and a validation dataset.

**How To Limit Overfitting**

Both overfitting and underfitting can lead to poor model performance. But by far the most common problem in applied machine learning is overfitting.

Overfitting is such a problem because the evaluation of machine learning algorithms on training data is different from the evaluation we actually care the most about, namely how well the algorithm performs on unseen data.

There are two important techniques that you can use when evaluating machine learning algorithms to limit overfitting:

1. Use a resampling technique to estimate model accuracy.
2. Hold back a validation dataset.

The most popular resampling technique is k-fold cross validation. It allows you to train and test your model k-times on different subsets of training data and build up an estimate of the performance of a machine learning model on unseen data.

A validation dataset is simply a subset of your training data that you hold back from your machine learning algorithms until the very end of your project. After you have selected and tuned your machine learning algorithms on your training dataset you can evaluate the learned models on the validation dataset to get a final objective idea of how the models might perform on unseen data.

Using cross validation is a gold standard in applied machine learning for estimating model accuracy on unseen data. If you have the data, using a validation dataset is also an excellent practice.