7-22-2022

Artificial Neural Network Concepts and Examples

Harcharan Kabbay
University of Missouri-St. Louis, hkqfc@umsl.edu

Artificial Neural Network Concepts and Examples

Harcharan Singh Kabbay
M.A. Mathematics, University of Missouri-St. Louis, 2022

A Thesis Submitted to The Graduate School at the University of Missouri-St. Louis
in partial fulfillment of the requirements for the degree
Masters of Arts in Mathematics
with an emphasis in Data Science

August, 2022

Advisory Committee

Dr. Adrian Clingher, Ph.D
Chairperson
Dr. Qingtang Jiang, Ph.D
Dr. Haiyan Cai, Ph.D

Contents

Abstract

Artificial Neural Networks have gained much media attention in the last few years. Every day, numerous articles on Artificial Intelligence, Machine Learning, and Deep Learning exist. Both academics and business are becoming increasingly interested in deep learning. Deep learning has innumerable uses, including autonomous driving, computer vision, robotics, security and surveillance, and natural language processing. The recent development and focus have primarily been made possible by the convergence of related research efforts and the introduction of APIs like Keras. The availability of high-speed compute resources such as GPUs and TPUs has also been instrumental in developing deep learning models.

While the development of the APIs like Keras offers a layer of abstraction and makes the model development convenient, the Mathematical logic behind the working of the Neural Networks is often misunderstood. The thesis focuses on the building blocks of a Neural Network in terms of Mathematical terms and formulas. The research article also includes the details on the core parts of the Deep Learning algorithms like Forwardpropagation, Gradient Descent, and Backpropagation.

The research briefly covers the basic operations in Convolution Neural Networks, and a working example of multi-class classification problem using Keras library in R. CNN is a vast area of research in itself, and covering all the aspects of the ConvNets is out of scope of this paper. However, it provides an excellent foundation for understanding how Neural Networks work and how a CNN uses the concepts of the building blocks of a primary Neural Network in an image classification problem.

Contents

I Math Foundations of Artificial Neural Networks

I.I Differences between Artificial Intelligence, Machine Learning and Deep-Learning

Before starting the research on Artificial Neural Networks, it is essential to understand the history and evolution of various technologies considered or related to Data Science. The field of **Artificial Intelligence** goes back to the 1950s when John McCarthy, Assistant Professor of Mathematics at Dartmouth College, organized a summer workshop with fellow research scientists with a proposal that the Machines could be made to think to solve human intellectual tasks. Artificial Intelligence, or AI, is the attempt to automate intellectual work that people would otherwise perform.

Machine Learning (ML) is an artificial intelligence (AI) subdomain that allows a machine to automatically find (learn) the statistical structure of data and transform such representations (patterns) to come closer to the intended output. The learning process is improved via a feedback channel to compare the expected and computed results.

Statistical modeling is a mathematical description of the relationship involving multiple variables, whereas a machine learning model is an engine that can learn from data without being directly coded. Statistics is a sub-domain of mathematics, whereas machine learning is a sub-domain of AI.

We talked about the learning process in Machine Learning which helps find biases and weights to understand the transformation from the input data to the expected output. However, this technique may not be possible to analyze advanced use cases like computer vision and speech recognition, which require enhanced feature engineering. **Deep Learning** attempts to resolve this problem by providing multiple learning layers and getting a representation of each layer with its own biases and weights through a process called forward propagation.

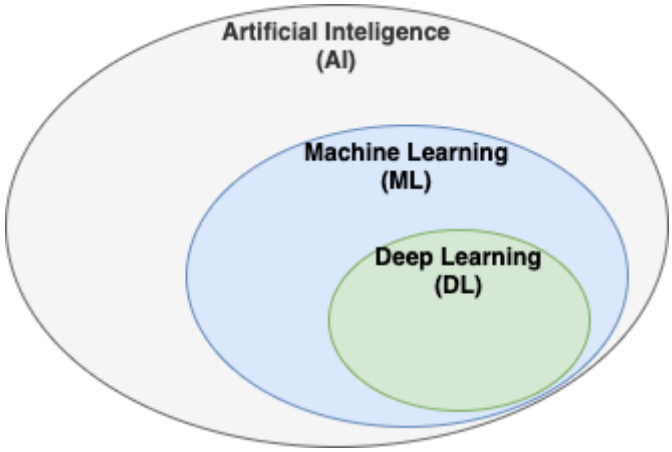


Figure 1: AI, ML, and DL

I.II What is Artificial Neural Networks?

Artificial neural networks (ANNs), also known as neural networks (NNs), are computer systems modeled after the biological neural networks that make up animal brains. Artificial neurons are linked units or

nodes in an ANN that loosely replicate the neurons in a biological brain. Like synapses in the brain, each link may send a signal to other neurons.

A primary neural network has an input layer, a hidden layer, and an output layer. The input layer feeds the input features, and the hidden layer computes the activation function of the weighted input to predict the output. The number of hidden layers can be expanded based on the requirements of the model. This number of hidden layers is also referred to as the depth of the model.

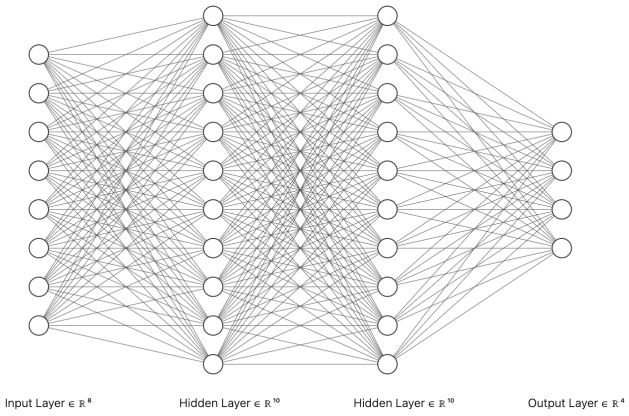


Figure 2: Deep Learning model with 2 hidden layers

I.III Supervised Learning

Supervised Learning is generally referred to process of machine learning where the **training data** set is given with the Input variables and their associated target labels (1). To set some context for discussion, we can think of the training data set as: -

$$(V_i, y_i) \quad , i = (1, 2, 3, ..., m) \longleftarrow m\ is\ the\ size\ of\ dataset$$

Each input V_i is a numerical vector with n dimensions or number of features.

$$Vi \in \mathbf{R}^n$$

$$V_i = (X_1^i, X_2^i, X_3^i, ..., X_n^i,) \longleftarrow Input\ features\ or\ variables$$

The output labels in form of Y_i is a single numerical variable.

$$y_i = (y_1, y_2, y_3, ..., y_m),\ y_i \in \mathbf{R}$$

$Input\ Var_1$	$Input\ Var_2$	...	$Input\ Var_n$	$Output\ label_i$
X_1^1	X_2^1	...	X_n^1	y_1
X_1^2	X_2^2	...	X_n^2	y_2
...	...	...	...	...
X_1^m	X_2^m	...	X_n^m	y_m

Table 1: Example-Training set for Supervised Learning

Supervised Learning aims to find an estimator or predictor function f for the output y_i, based on the given data set.

$$f : \mathbf{R}^n \longrightarrow \mathbf{R}, such\ that$$

$$f(V_i) \approx y_i\ for\ i = 1, 2, ..., m$$

The base idea is to find a formula or function that generates the output label y_i using the input features V_i. The process of finding the predictor function f is often referred to as fitting or training the solution/model.

Supervised Learning is classified into three categories:-

1. **Regression** - Output Y_i may take a continuous range of values. As an example estimating the Real-estate prices based on a given training set.
2. **Classification** - Output Y_i belongs to a finite set of values. A model to classify facial images from an image set into categories of mood like happy, sad, angry, etc.
3. **Binary Classification** - This is a special case of Classification where the output Y_i is classified into two categories, e.g., Yes/No, 0/1, ± 1. An example of Binary classification could be estimated if a person has diabetes or not based on historical training data for admitted patients.

The objectives of the Supervised Learning are as follows:-

- Prediction - Predict the output labels for unseen data with reasonable accuracy. The unseen data is not part of the training set.
- Inference - Understanding the effect of each input feature and which one affects the most.
- Accuracy - Measure the quality of predictions and inferences.

Mathematical Explanation of Supervised Learning

So far, we have discussed the essential requirement to find the estimator or predictor function $f(v)$, which is close to the output y. To put in a Math equation, it looks as follows:-

$$f(v) = y + \epsilon$$

$f(v)$ is to be estimate, v is the input vector and y is the output class. ϵ is the noise, a random numerical variable from distribution $\mathcal{N}(0, \sigma^2)$

.

$$\sigma^2 = Var(\epsilon) \Longleftarrow Irreducible\ error$$

The goal is to find an estimator function $\hat{f}(v)$ close to $f(v)$ such that

$$\hat{f}(v) \approx f(v)$$

Here is a plot (3) of some data points (V_i, y_i) with $i = 1, 2, ..., 30$. The true predictor in this example is

$$f(x) = sin(x^3 + 1)$$

And noise $\epsilon \approx \mathcal{N}(0, 0.01)$

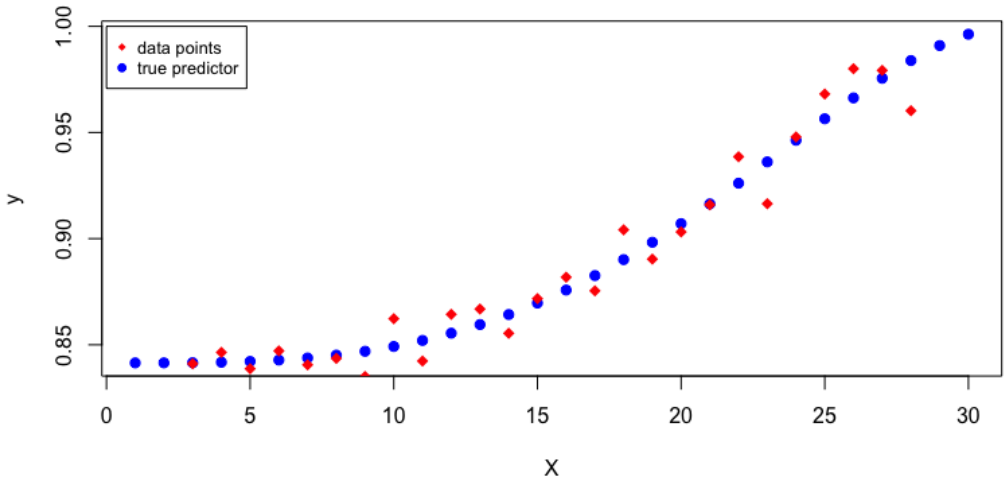


Figure 3: Data points and True predictor

In real case scenarios, the true predictor is unknown, and we have to find the best estimator function $\hat{f}(v)$. There could be different candidates for $\hat{f}(v)$. Let us represent those candidate estimators

$$g : \mathbf{R}^n \longrightarrow \mathbf{R}$$

$Input\ Var_1$	$Input\ Var_2$	...	$Input\ Var_n$	$Output\ label_i$	$g(V_i)$	$[g(V_i) - y_i]^2$
X_1^1	X_2^1	...	X_n^1	y_1	$g(V_1)$	$[g(V_1) - y_1]^2$
X_1^2	X_2^2	...	X_n^2	y_2	$g(V_2)$	$[g(V_2) - y_2]^2$
...	...	...	...	...		
X_1^m	X_2^m	...	X_n^m	y_m	$g(V_m)$	$[g(V_m) - y_m]^2$

Table 2: Square Errors in Predictors

The average of all the square errors or Mean Square Error (MSE) can be found as

$$MSE = \frac{1}{m}\sum_{1}^{m}[g(V_i) - y_i]^2$$

We can keep the MSE as a baseline to compare the quality of the estimator function. We try to find $\hat{f}(v)$, which minimizes the value of the MSE, also called a **Cost function**. Some other versions of the MSE are Root Mean Square Error (RMSE) $\sqrt{\frac{1}{m}\sum_1^m[g(V_i) - y_i]^2}$, Mean Absolute Error (MAE) $\frac{1}{m}\sum_1^m |g(V_i) - y_i|$, and Mean Absolute Percentage Error (MAPE) $\frac{1}{m}\sum_1^m \left|\frac{g(V_i)-y_i}{y_i}\right|$.
Cost function and Loss function are sometimes used inter-changeably but the Loss function is calculated at each instance of the training data, and the Cost function is the average of the Loss functions over all the samples of the training data. Some other Loss functions for regression problems are as follow:-

- Root Mean Square Logarithmic Error - Also referred as RMSLE is an extention of RMSE. The formula of the RMSLE is as follow:-

$$RMSLE = \sqrt{\frac{1}{m}\sum_{1}^{m}[log(1+g(V_i)) - log(1+y_i)]^2}$$

- Cosine Similarity - This is a metric used in data analysis to compare two numerical sequences. Suppose, we have two non-zero vectors $\vec{a}$ and $\vec{b}$. Then the Cosine similarity between these two vectors is given by:

$$\cos\theta = \frac{\vec{a}.\vec{b}}{\|\vec{a}\|.\|\vec{b}\|}$$

  The closer the angle between the vectors, higher is the similarity.

- Log-Cosh error - Logarithmic of the hyperbolic cosine of the prediction error is another measure for the quality of the predictor. The Log-Cosh loss is measured as:

$$L(y, y^p) = \sum_{i=1}^{m} log(cosh(y_i^p - y_i))$$

The concept of MSE does not apply to the **Binary Classification**, where we are predicting the output as 0 or 1. So the square errors method does not make much sense here. Instead, we use the concept of Maximum Likelihood Estimator to help us.

Let us take an example of Bernoulli distribution with a data set (V_i, y_i), where $V_i \in \mathbf{R}^n$ and y_i can take a value $0\ or\ 1$. We can represent y as

$$y_i = \begin{cases} 1 & \text{with probability } p(v) \\ 0 & \text{with probability } 1 - p(v) \end{cases}$$

We have to find an optimal probability estimator $\hat{p}(v)$ to solve this problem. We can follow the following guidelines to estimate the $\hat{p}(v)$. Let us assume that there is some function $q \longrightarrow \mathbf{R}^n$ that could be a candidate for a good $\hat{p}(v)$. For a good probability estimator, the estimated value should be close to the true value.

If $y_i = 1, q(v_i)$ should be close to 1

If $y_i = 0, q(v_i)$ should be close to 0

And, it is desired to have $q(V_i)^{y_i}.[1 - q(V_i)]^{1-y_i}$ closer to 1. Since, we have m records in the data set we need to

Maximize: $\prod_{1}^{m} q(V_i)^{y_i}.[1 - q(V_i)]^{1-y_i} \longleftarrow$ **MLE**

We know that MLE in this case belongs to $(0, 1)$, so we apply the log to the above equation.

$$-log(MLE) = -log[\prod_{1}^{m} q(V_i)^{y_i}.[1 - q(V_i)]^{1-y_i}]$$

$$-log(MLE) = \sum_{i=1}^{m}[-y_i.log(q(V_i)) - (1 - y_i).log(1 - q(V_i))] \longleftarrow \textbf{Cross-Entropy function}$$

The cost function, in this case, is $-log(MLE)$, so we need to minimize this as part of the function/model training for binary classification. **Categorical Cross-Entropy** is a version of Binary Cross-Entropy, which is used for multi-class classification problems.
Other Loss functions for classification problems are as follow:-

- Hinge Loss - This loss function is mostly used in the classification problem in Support Vector Machines (SVMs). The formula for Hinge Loss is as follow:-

  $$\mathcal{L}(y) = max(0, 1 - t.y)$$

  Here, $t = \pm 1$ is the intended output for the classifier and $y = w.x + b$.

- Huber Loss - The Huber Loss function is derived from both the MSE and the MAE. This is less sensitive to outliers than the MSE and also differentiable over zero.

  $$L_\delta(y, f(x)) = \begin{cases} \frac{1}{2}(y - f(x))^2, & \text{for } |y - f(x)| \le \delta, \\ \delta|y - f(x)| - \frac{1}{2}\delta^2, & \text{otherwise} \end{cases}$$

- Kullback-Liebler Divergence - This is used to find the distance between two distributions i.e. existing distribution p and the predicted data distribution q. The discrimination information of probability p to q is given as

  $$D_{KL}(p||q) = \sum p(x) log\left(\frac{p(x)}{q(x)}\right)$$

Parametric Models

We discussed the guidelines for finding a regression predictor function and a binary classification problem. The predictor function is chosen from a specific set of functions that are indexed by parameters $\theta = (\theta_0, \theta_1, \dots)$ A predictor function θ has the biases and weights to find the output labels from the input vectors. Here are some examples: Suppose we have a Hypothesis set $\mathcal{H} = f(V)$, which contains the predictor functions. And suppose we have two variables in the input vector, i.e., x_1, x_2. And $\mathcal{H} = f(x_1, x_2)$ polynomial function of degree d.

$$d = 1, f(x_1, x_2) = a_0 + a_1 x_1 + a_2 x_2 \longleftarrow \theta = (a_0, a_1, a_2) \in \mathbf{R}^3$$

$$d = 2, f(x_1, x_2) = a_0 + a_1 x_1 + a_2 x_2 + a_3 x_1 x_2 + a_4 x_1^2 + a_5 x_2^2$$

$$\text{Here, } \theta = (a_0, a_1, a_2, a_3, a_4, a_5) \in \mathbf{R}^6$$

As part of the training, the estimator functions from the Hypothesis set are tried with the provided training set.

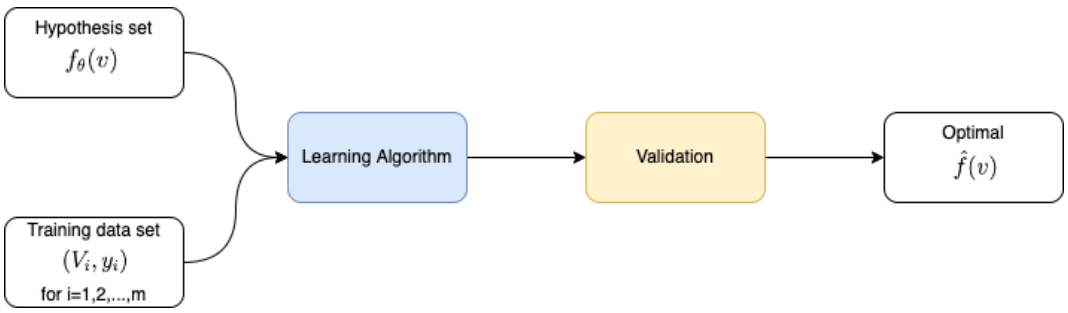


Figure 4: Overview of Parametric Model

The learning Algorithm uses the Cost function to find the closeness of the estimated value with the actual value. Some of the standard loss functions are Mean Square Error and Cross Entropy. Once the loss is calculated, the learning algorithm modifies the weights and biases(parameters) to find the optimal $\hat{\theta}$ through the optimization techniques like Gradient Descent. a high level learning workflow can be described by Figure 5.

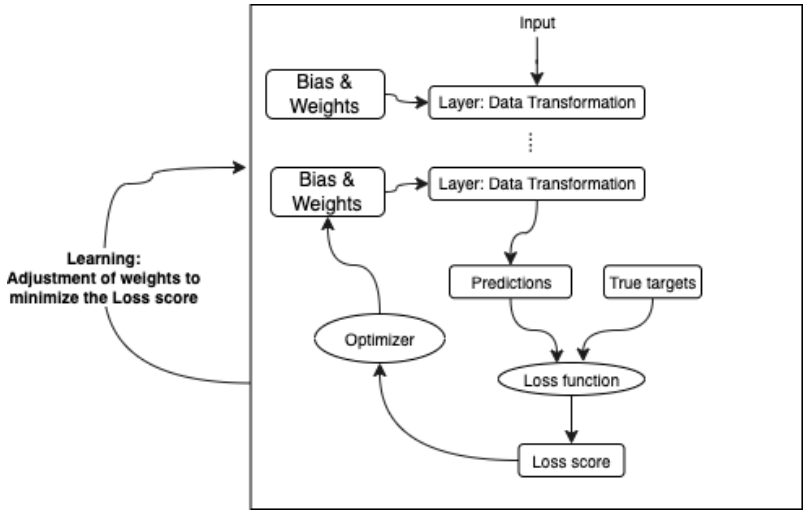


Figure 5: Overview of Training cycle

Besides the regular parameters, other sets of parameters called Hyper-parameters also impact the learning of a training algorithm. Hyper-parameters include but are not limited to no. of neurons per layer, learning rate, etc.

As part of the model learning process, we capture the validation metrics, i.e., the model's performance on the validation data set, which is separate from a training set. A loss function is used to calculate the validation error in regression problems. The loss function is evaluated using the validation data set on the optimal (trained) parameters $\hat{\theta}$.

In classification problems, Accuracy is commonly used as a validation metrics. Accuracy is calculated as follow:-

$$accuracy = \frac{1}{l} \sum I(\hat{f}(v) = y) \longrightarrow l \text{ - size of validation set}$$

$I(\hat{f}(v) = y)$ is the Indicator function where $I(true) = 1$ and $I(false) = 0$. Accuracy is a percentage of success of the algorithm. The validation error is computed as:

$$validation\ error = 1 - accuracy$$

There are two common problems in the life-cycle of model training:

- Over-fitting - Over-fitting happens when the model is complex to the extent that it learns the detail of data and noise. However, the model suffers from predicting the output of the validation data and any new data. To fix this problem, we need to generalize the model to ignore the noise and focus only on data patterns.
- Under-fitting - Under-fitting is the problem when the model cannot learn the training data or predict validation data. This means the model is unsuitable for learning and needs to be optimized.

II Optimizing the Loss function

We discussed the Loss function and how it measures the quality of a predictor or estimator. The important piece is how to find the optimal value of weights and biases to reduce this loss. This mathematical optimization problem has two components to minimize the function $\mathcal{J}(x_1, x_2, ..., x_n)$.

1. Determine if an actual minimum exists for this function.

2. If a minimum exists, find the inputs where this minimum is attained.

One common issue in the optimization problem is finding the difference between local minima and global minima. To illustrate, we plot (Figure 6) the values for $x^2 + 10 * sin(x)$ for x ranging from -10 to 10. In the given range of x values, this function has one global minimum, local minima, and another local minimum.

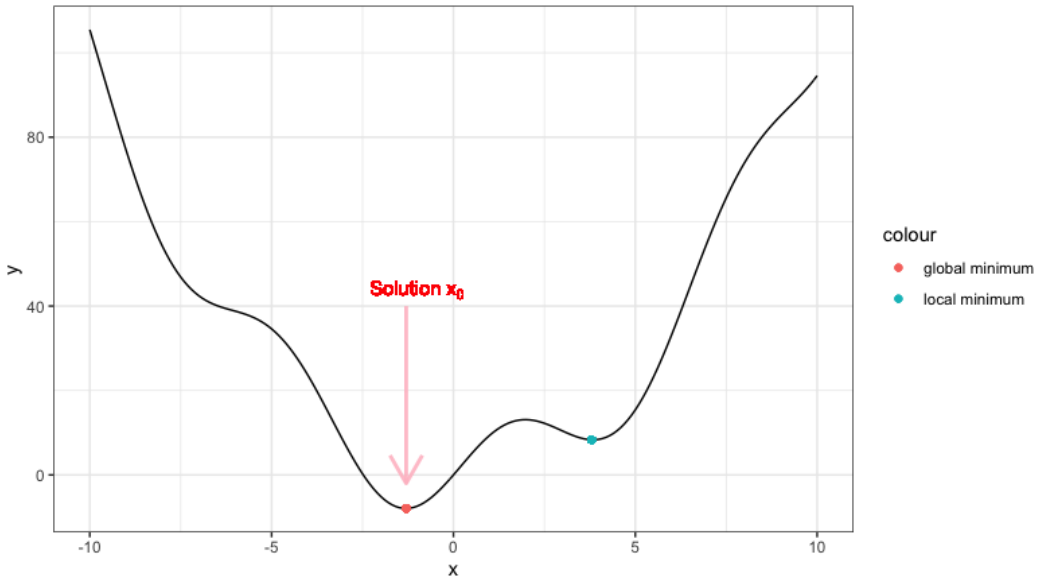


Figure 6: Local and Global minima

The minimization problem's solution is finding the global minimum, i.e., x_0. The minimum realization for function $\mathcal{J}$ is $\mathcal{J}(x_0)$.

The complexity of the minimization problem increases as the number of variables in the Loss function $\mathcal{J}$ increases. Here is an example plot for a two variable function $\mathcal{J}(x_1, x_2)$.

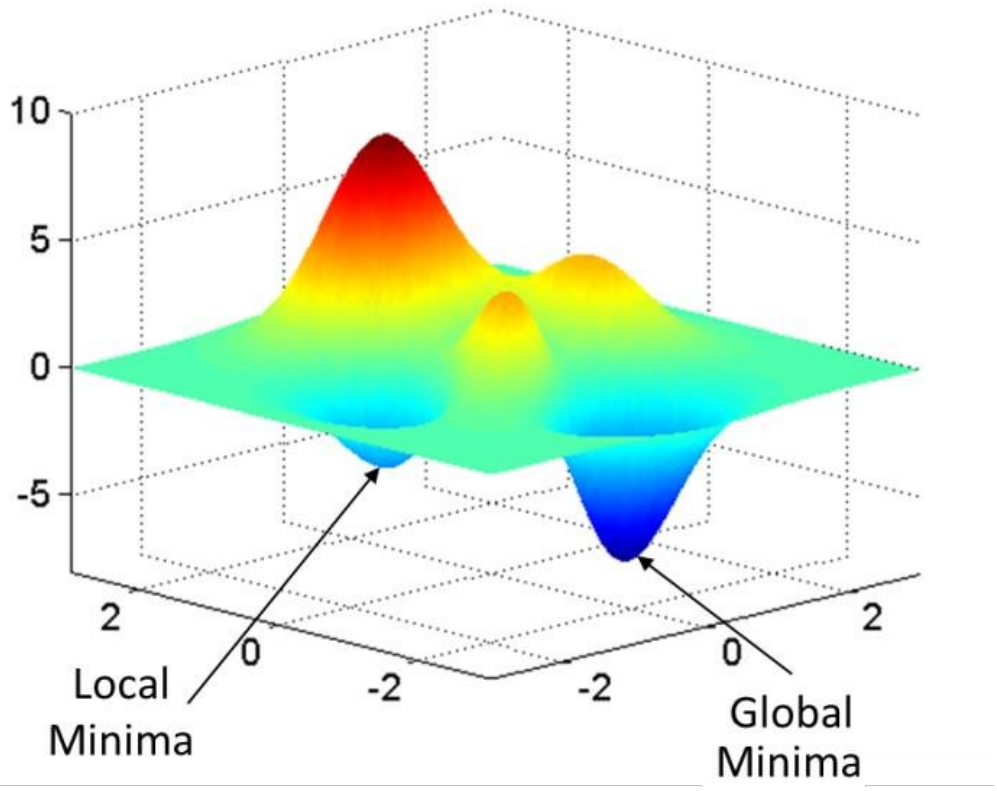


Figure 7: Local and Global minimum for $\mathcal{J}(x_1, x_2)$ - $References$[8]

A general theoretical procedure using Calculus to solve a multivariate loss function $\mathcal{J}(x_1, x_2, ..., x_n)$ is as follow:-

1. Compute all the partial derivatives

$$\frac{\partial \mathcal{J}}{\partial x_i} \longrightarrow \text{for i} = 1,2,...,n$$

2. Construct the gradient

$$\nabla \mathcal{J} = \left(\frac{\partial \mathcal{J}}{\partial x_1}, \frac{\partial \mathcal{J}}{\partial x_2}, \ldots, \frac{\partial \mathcal{J}}{\delta x_n} \right)$$

3. Solve the equation

$$\nabla \mathcal{J} = 0$$

   Here 0 on the r.h.s of the above equation represents a n-dimensional 0 vector. Solutions are critical points of $\mathcal{J}$ i.e. local minima, local maxima, and saddle points.

4. We can classify the critical points using the second derivative tests. This involves n^2 of such partial derivatives.

5. From the partial derivatives of second order, we can evaluate the local minima points to find the global minima.

In general Machine Learning problems, the equation $\nabla\mathcal{J} = 0$ is quite challenging to solve. Some numerical methods of minimization have been developed to approximate the global minimum of the function $\mathcal{J}$. We are going to discuss two commonly used algorithms, namely:

- Gradient Descent Method
- Newton-Raphson Method

II.I Gradient Descent

Gradient Descent is such an Optimization Algorithm that helps achieve the objective for a learning model to minimize the loss function. The word *descent* refers to the lowest point of the Loss function curve. Suppose we have the loss function represented by $\mathcal{J}(x_1, x_2, \ldots, x_n)$. Assume we start with a random point v_0, and we get $\partial\mathcal{J}(v_0) = 0$, which means by coincidence, we got a critical point.
In a normal case, where $\partial\mathcal{J}(v_0) \neq 0$, which direction shall we move from v_0 in order to decrease $\mathcal{J}$ and to get to a global minima.

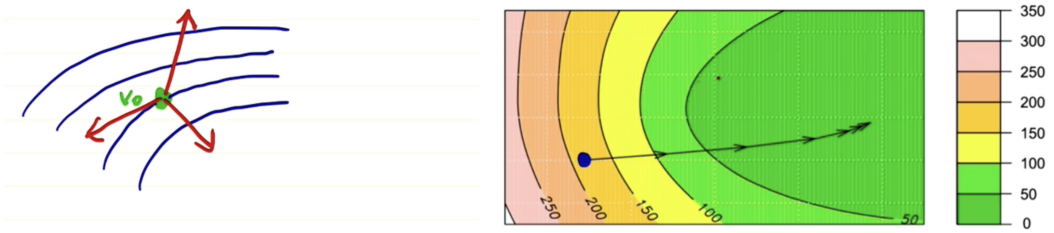


Figure 8: Gradient Descent direction - $References$[1]

Let us assume we choose a direction given by a unit vector direction $h \in \mathbf{R}^n$ and $t \geq 0, t \in \mathbf{R}$. The variation in vector v is given by $v = v_0 + th$. Now, we would like to know the impact of this variation on $\mathcal{J}(v_0 + th)$, which is a derivative of this one-variable function w.r.t t i.e. $\frac{d\mathcal{J}(v_0+th)}{dt}$. Using multi-variable chain rule.

$$\frac{d\mathcal{J}(v_0 + th)}{dt} = \sum_{i=1}^{n} \frac{d\mathcal{J}(v_0 + th)}{dx_i}.h_i$$

$$=< \nabla\mathcal{J}(v_0 + th), h_i >$$

Or the dot product of the gradient $\mathcal{J}$ and the direction vector. More specifically, we are interested in the instantaneous rate of change at v_0

$$\frac{d\mathcal{J}(v_0 + th)}{dt}(_0) =< \nabla\mathcal{J}(v_0), h >$$

The rate of change of this derivative, if $-ve$, means we move in the direction of minima. And, the direction that makes $< \nabla\mathcal{J}(v_0), h >$ negative and minimal is the direction of the fastest descent.
Treating this as a Linear Algebra problem, we have a fixed vector $\mathcal{J}(v_0)$ and a varying unit direction vector h. We need to find a direction h that minimizes the dot product $< \nabla\mathcal{J}(v_0), h >$. The cosine of the angle between these two vectors can be given by.

$$\cos(\theta) = \frac{< \nabla\mathcal{J}(v_0), h >}{\|\nabla\mathcal{J}(v_0)\|}$$

The denominator represents the length of the vector $\nabla\mathcal{J}(v_0)$, so if we minimize the value for the $\cos(\theta)$, we should be able to minimize the dot product $< \nabla\mathcal{J}(v_0), h >$. The $\cos(\theta)$ can take a value between $+1$ and -1. Going with the minimum value, we pick $\cos(\theta) = -1$, means

$$\theta = \pi = 180^o$$

. With this information, we decide to move to the direction opposite to the direction of the gradient.

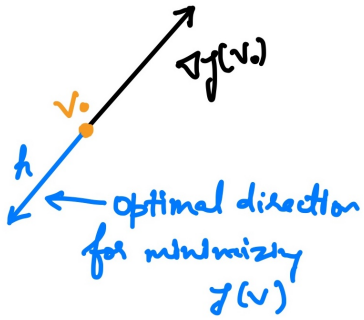


Figure 9: Direction of the movement

$$h = -\frac{1}{\|\nabla\mathcal{J}(v_0)\|}.\nabla\mathcal{J}(v_0)$$

Gradient Descent Algorithm

The Gradient Descent Algorithm follows the steps:

1. Initialize starting vector- Select a random vector $v_0 = (x_1^0, x_2^0, \ldots, x_n^0)$ in the domain of $\mathcal{J}$
2. Perform iterations to minimize the value of $\mathcal{J}$- Let us assume, from a prior iteration we have $v_k = (x_1^k, x_2^k, \ldots, x_n^k)$
   Find $\nabla\mathcal{J}(v_k)$
   Define $v_{k+1} = v_k - \alpha.\nabla\mathcal{J}(v_k)$

   Possibilities:

   - Fixed Learning rate α - Learning rate α is chosen at the beginning and it remains the same throughout all the iterations.
   - Exact line search (add details) - Learning rate α is chosen at each iteration by minimizing the 1-var function
     $$t \hookrightarrow \mathcal{J}(v_k - t.\nabla\mathcal{J}(v_k)$$
   - Backtracking line search - Uses advanced techniques of convex optimization.

   During this phase, we find the value of loss function $\mathcal{J}(v_i)$ which reduces in each iteration such that
   $$\mathcal{J}(v_0) > \mathcal{J}(v_1) > \mathcal{J}(v_2) \cdots > \mathcal{J}(v_k) > \mathcal{J}(v_{k+1}) > \ldots$$

This descent can be shown for a two-variable vector as follow:-

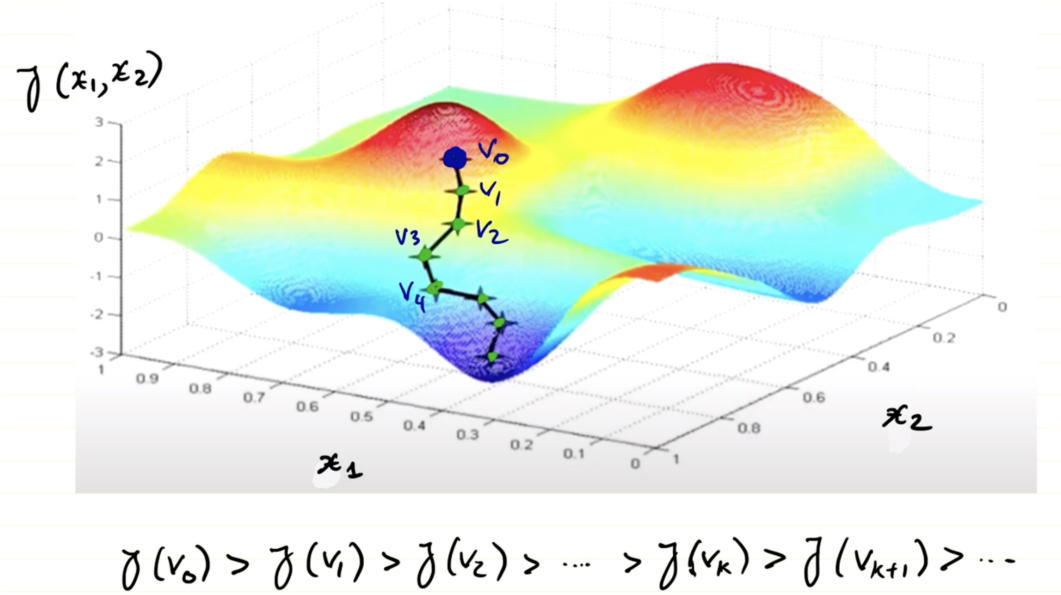


Figure 10: Loss decreases on each iteration to reach a global minima - $References$[6]

3. Termination - Per Mathematical Theorem (Convex Optimization, Boyd and Vendenberghe 2021), based on certain convergence conditions that depends upon the learning rate α:

   (a) The norm of gradient i.e. $\|\nabla\mathcal{J}(v_k)\|$ is decreasing and

   $$\lim_{k\to\infty} \|\nabla\mathcal{J}(v_k)\| = 0$$

   (b) $|\mathcal{J}(v_k) - \mathcal{J}(v^*)| < C.\|\nabla J(v_k)\|^2$, where v^* is a local minimum for $\mathcal{J}$, and C is a constant.

   We set a small variable $\epsilon > 0$, as acceptable error. The Gradient-Descent Algorithm stops when $\|\nabla\mathcal{J}(v_k\| < \epsilon$. Then, we have

   $$|\mathcal{J}(v_k) - \mathcal{J}(v^*)| < C.\epsilon^2$$

Observations

Some important observations around the characteristics and use of Gradient Descent are as follow:-

- The iteration methods like Exact line search and backtracking line search are computationally expensive compared to the use of a fixed learning rate α.
- The fixed learning rate α, if chosen too small, can make the algorithm very slow to reach a solution. And choosing α too high can make it hard to converge.

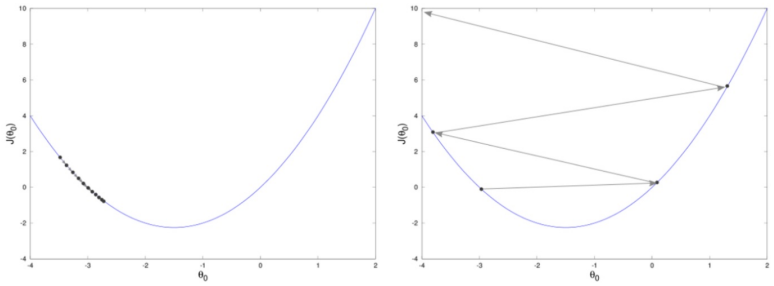


Figure 11: Impact of learning rate on convergence - $References^{6}$

- Gradient Descent Algorithm can get stuck in a local minimum. To address this, we should run the Gradient Descent multiple times, with a different initialization vector v_0. The solution is chosen as the value of v that produces the lowest value for $\mathcal{J}(v)$.

II.I.1 Gradient Descent in context of a SL Parametric Model

Let us consider a Supervised-Learning parametric model with s number of parameters such that $\theta = (\theta_0, \theta_1, \ldots, \theta_s) \in \mathbf{R}^s$. And, we have the Loss function denoted as $\mathcal{J}(\theta)$.
To calculate the Loss function, we sum it over all the pairs in the training set (v_t, y_t), where t is the total no. of records.

$$\mathcal{J}(\theta) = \frac{1}{m}\sum_{t=1}^{m} Q_\theta(v_t, y_t)$$

In case of Gradient Descent, at each iteration, we compute the gradient factor for the function $\mathcal{J}$, which will be like

$$\nabla\mathcal{J}(\theta) = \frac{1}{m}\sum_{t=1}^{m} \nabla Q_\theta(v_t, y_t)$$

Since this involves all the records in the training set and involves a large sum, doing this at each iteration would be computationally very expensive. To overcome this problem, there are some version of the Gradient Descent which are listed as follow:-

Stochastic Gradient Descent

Stochastic Gradient Descent is similar to how Gradient Descent except the slight changes to the iteration part. The Iteration in a Batch Gradient Descent is as follow:-

$$\theta^{k+1} = \theta^k - \alpha\nabla\mathcal{J}(\theta^k) = \theta^k - \frac{\alpha}{m}\sum_{t=1}^{m} \nabla Q_{\theta^k}(v_t, y_t)$$

An iteration in a Stochastic Gradient Descent looks like as below:-

1. Set $w := \theta^k$
2. Reshuffle the m pairs in the training data set
3. for t=1 to m do:
   $w := w - \frac{\alpha}{m}\sum_{t=1}^{m} \nabla Q_{\theta^w}(v_t, y_t)$
4. $\theta^{k+1} := w$

In Batch gradient Descent, we compute the entire gradient sum and then reset the parameter vector. In the case of the Stochastic Gradient Descent (SGD), we add the term to the gradient sum and reset the parameter vector.
From a computational point of view, SGD converges faster but could be hard to control due to the mathematical complexity involved in the algorithm.

Mini-Batch Gradient Descent

Min-Batch Gradient Descent is a compromise between the the Batch Gradient and the Stochastic Gradient Descent methods. In this, we randomly split the training data set into mini-batches of size b (usually $50 \leq b \leq 256$). Suppose,we perform the split of m training record into l batches as

$$\{1\ 2\ \ldots\ m\} = B_1 \cup B_2 \cup \cdots \cup B_l$$

$$|B_q| \leq b \text{ where } q = 1, 2, \ldots, l$$

The iteration for a Mini-Batch Gradient Descent is as follow:-

1. Set $w := \theta^k$
2. Reshuffle the m pairs in the training data set
3. for q=1 to l do:
   $w := w - \frac{\alpha}{m} \sum_{t \in B_q} \nabla Q_w(v_t, y_t)$
4. $\theta^{k+1} := w$

In this case, we work in mini-batches, i.e., for every mini-batch from 1 to l, we compute the gradient sum and update the parameter vector. This is more stable than the SGD and considered adequate for training Artificial Neural Networks on large training data sets.

II.II Newton-Raphson method

Newton-Raphson method is named after Issac Newton (1642-1726) and Joseph Raphson (1648-1715). This is another method for mathematical optimization with the goal to minimize a given differentiable function $\mathcal{J} : \mathbf{R}^n \to \mathbf{R}$. We have an n-dimensional function $\mathcal{J}(x_1, x_2, \ldots, x_n)$, and we want to approximate a global minimum $v^* = (x_1, x_2, \ldots, x_n) \in \mathbf{R}^n$ such that

$$\mathcal{J}(v^*) \leq \mathcal{J}(v) \text{ for } v \in \mathbf{R}^n$$

The basic strategy is to approximate the critical points or the solutions to the equation

$$\nabla \mathcal{J} = 0 \longleftarrow \text{zero vector in} \mathbf{R}^n$$

For mathematical convenience, we refer to this gradient as a function g

$$g = \nabla \mathcal{J} : \mathbf{R}^n \to \mathbf{R}^n$$

Then using the standard calculus method, we evaluate these critical points for function $\mathcal{J}$, to find out which one would be a global minimum. This method enables us to numerically solve a system of equations:

$$g(x_1, x_2, \ldots, x_n) = 0$$

$$g : \mathbf{R}^n \to \mathbf{R}^n$$

These are n non-linear equations i.e

$$g_1(x_1, x_2, \ldots, x_n) = 0$$

$$g_2(x_1, x_2, \ldots, x_n) = 0$$

$$\vdots$$

$$g_n(x_1, x_2, \ldots, x_n) = 0$$

Newton Raphson method - Case(n=1)

To understand this method, we start with the case of $n = 1$ or $g(x) = 0$. There is only one input to function g and one output. This method is commonly referred to as the Newton's method. The basic algorithm works as follow:-

1. Choose an initial value x_0 as an approximate for the solution x^*
2. Create a recursive sequence
$$x_k + 1 = x_k - \frac{g(x_k)}{g'(x_k)}$$
3. Per Mathematical Theorem (Convex Optimization, Boyd and Vendenberghe 2021) If the initial value x_0 is chosen properly, the sequence $(x_k)_{k\geq 0}$ will converge, with
$$\lim_{k\to\infty} x_k = x^*$$

Why does Newton's method work?

It might be difficult to solve $g(x) = 0$, but we can try the Taylor expansion of $g(x)$ i.e.

$$g(x) = g(x_0) + g'(x_0)(x - x_0) + \frac{g"(x_0)}{2}(x - x_0)^2 + \ldots$$

With the first-order approximation of $g(x)$, we have

$$g(x_0) + g'(x_0).(x - x_0) = 0$$

We solve the x from $x = x - \frac{g(x_0)}{g'(x_0)}$
On each iteration, we improve the approximation for x^* to find the exact root.

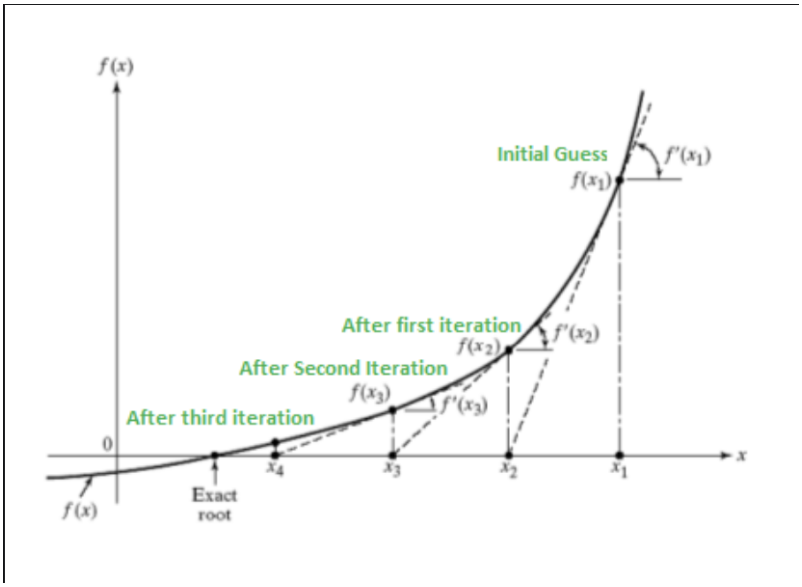


Figure 12: Convergence using Newton-Raphson method - $References$[7]

Some possible issues with the Newton-Raphson methods are as follow:-

1. We might get a derivation $g'(x_k) = 0$, and this term appears in the denominator when approximating the x_{k+1} i.e.

$$x_k + 1 = x_k - \frac{g(x_k)}{g'(x_k)}$$

   This would cause the algorithm to crash.

2. Slow convergence: $(x_k)_{k\geq 1}$ converges slowly or does not converge.

3. The equation $g(x) = 0$ could have multiple solutions. You may end up capturing another solution which is close to the initialization value.

All these issues can be resolved by re-setting the initialization value x_0 and re-running the algorithm.

Newton Raphson method - Case($n > 1$)

The Newton-Raphson method case $n = 1$ can be generalized to find a solution for a multi-variable case. We use the concept of Jacobian matrix of $g_i : \mathbf{R}^n \to \mathbf{R}$ with the rows representing the gradient vectors i.e. $\nabla g_1, \nabla g_2, \ldots, \nabla g_n$

$$\text{(Jacobian matrix of } g_i) = M_g = \begin{pmatrix} \frac{dg_1}{dx_1}, \frac{dg_1}{dx_2}, \ldots, \frac{dg_1}{dx_n} \\ \frac{dg_2}{dx_1}, \frac{dg_2}{dx_2}, \ldots, \frac{dg_2}{dx_n} \\ \vdots, \vdots, \vdots \\ \frac{dg_n}{dx_1}, \frac{dg_n}{dx_2}, \ldots, \frac{dg_n}{dx_n} \end{pmatrix} \tag{1}$$

It is worth noting that this is a Matrix function, because if we change the inputs $x_1, x_2, \ldots, x_n$, the partial derivatives would change and the whole Jacobian Matrix would change.
Our goal is to find a solution $v^* \in \mathbf{R}^n$ for the equation $g(v) = 0$. The algorithm follows the following steps:-

1. Choose an initial vector v_0 close to where we believe v^* should be.
2. Perform iterations to calculate

$$v_{k+1} = v_k - [M_g(v_k)]^{-1}.g(v_k)$$

$[M_g(v_k)]^{-1} \longrightarrow$ inverse of nxn matrix M_g We use this method to find the values for $v_1, v_2, v_3, \ldots$ The rationale for the Newton Raphson method states that there is a neighborhood of U of v^* such that, if $v_0 \in U$, we have:-

- The Jacobian matrix M_g is non-singular for any $k \geq 0$.
- The sequence $(v_k)_{k\geq 0}$ is convergent and $\lim_{k\to\infty} v_k = v^*$.

3. Terminate the algorithm estimating the permitted error $\|v_k - v^*\|$

Observations?

How does Newton-Raphson method apply to optimization?

In order to solve the optimization problem, we need to find the critical points or a solution for $\nabla\mathcal{J} = 0$. As mentioned earlier $\mathcal{J} : \mathbf{R}^n \to \mathbf{R}$. This means that $\nabla\mathcal{J} : \mathbf{R}^n \to \mathbf{R}^n$.
We have

$$g = \nabla\mathcal{J} = \left(\frac{d\mathcal{J}}{dx_1}, \frac{d\mathcal{J}}{dx_2}, \ldots, \frac{d\mathcal{J}}{dx_n}\right) \tag{2}$$

So, the Jacobian Matrix will consist of second-order partial derivatives of $\mathcal{J}$

$$M_g = \begin{pmatrix} \frac{d^2\mathcal{J}}{dx_1^2}, \frac{d^2\mathcal{J}}{dx_1 dx_2}, \ldots, \frac{d^2\mathcal{J}}{dx_1 dx_n} \\ \frac{d^2\mathcal{J}}{dx_2 dx_1}, \frac{d^2\mathcal{J}}{dx_2^2}, \ldots, \frac{d^2\mathcal{J}}{dx_2 dx_n} \\ \vdots, \vdots, \vdots \\ \frac{d^2\mathcal{J}}{dx_n dx_1}, \frac{d^2\mathcal{J}}{dx_n dx_2}, \ldots, \frac{d^2\mathcal{J}}{dx_n^2} \end{pmatrix} \tag{3}$$

The above matrix is also called as Hessian matrix of $\mathcal{J}$ and denoted by $\mathcal{H}_\mathcal{J}$.

Observations

Some important observations on Hessian matrix $\mathcal{H}_\mathcal{J}$ are as follows:-

- Hessian $\mathcal{H}_\mathcal{J}$ is a symmetric matrix, i.e., it is a square matrix that is equal to its transpose.
- If $\mathcal{H}_\mathcal{J}$ is convex, then $\mathcal{J}$ is a convex function, hence a good candidate for minimization. The convex functions have only one local minimum, a global minimum.

The algorithm works as follow to approximate the critical points:-

1. Choose an initial vector $v_0 = (x_1^0, x_2^0, \ldots, x_n^0)$.
2. Perform iterations to calculate

$$v_{k+1} = v_k - [\mathcal{H}_\mathcal{J}]^{-1}.\nabla\mathcal{J}(v_k)$$

$[M_g(v_k)]^{-1} \longrightarrow$ inverse of nxn matrix M_g On each iteration, we should get closer to v^*.

3. Terminate the algorithm estimating the permitted error $\|v_k - v^*\|$

The iteration process in the Newton-Raphson method is expensive, it requires calculating second-order partial derivatives as compared to first-order derivatives used in the Gradient Descent method.

III Basic Architecture of Neural Networks

Neural Networks are capable of dealing with both structured and unstructured data. Structured data is a highly organized format like tabular data stored in relational databases, CSV files, data frames, etc. Unstructured data, also categorized as qualitative data, is in different forms like videos, pictures, emails, social media posts, IoT sensor data, etc.
The basic architecture of a Neural Network consists of an input layer, a hidden layer, and an output layer.

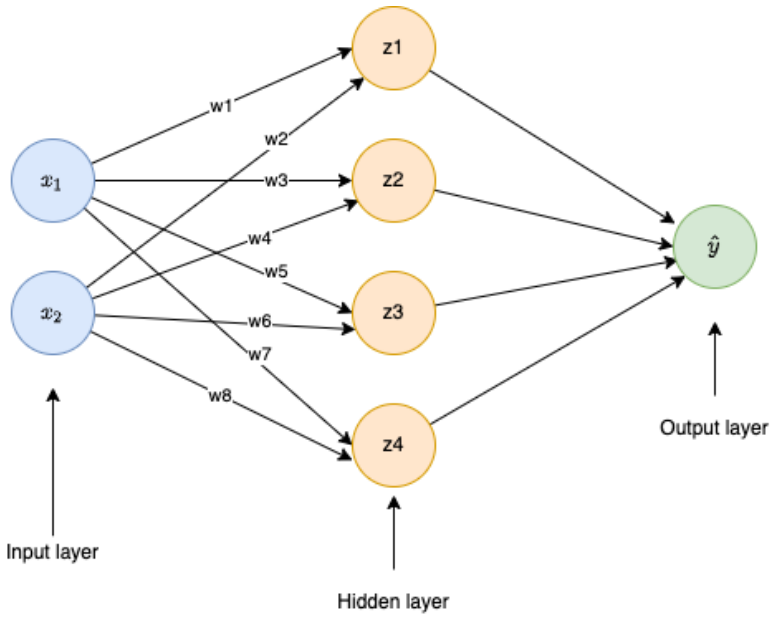


Figure 13: Basic Neural Network Architecture

In Figure 13, the Neural network has two input variables $X = [x_1, x_2]$ and a hidden layer with four neurons (nodes), which uses the inputs and weights $(w1, w2, \ldots, w8)$ to compute the activation function. The output from the hidden layer passes to the output layer to predict the output $\hat{y}$. The difference between the predicted and actual output is used to calculate the cost function. The cost function is minimized by adjusting the weights after each iteration. Gradients are used to update the weights, and the next iteration starts. This process repeats to get closer to the optimal parameter weights.

III.I Type of Neural Networks

Neural Networks can be classified into three main categories based on their architecture. The name and brief information on each group is as follows:-

Feed Forward Neural Networks (FFNNs)

Feed Forward Neural Networks are the most common type of neural network where information flows in one direction, starting from the input layer, and passing through the hidden layers to the output layer. The connection between the nodes do not form a cycle. Each hidden layer transforms inputs, resulting in a new representation at each successive layer.

Some common non-linear transformations in weighted inputs include sigmoid, tanh, ReLU, selu, softmax, etc. Feed Forward Neural Networks can be used for classification use-cases and in unsupervised learning as auto-encoders.

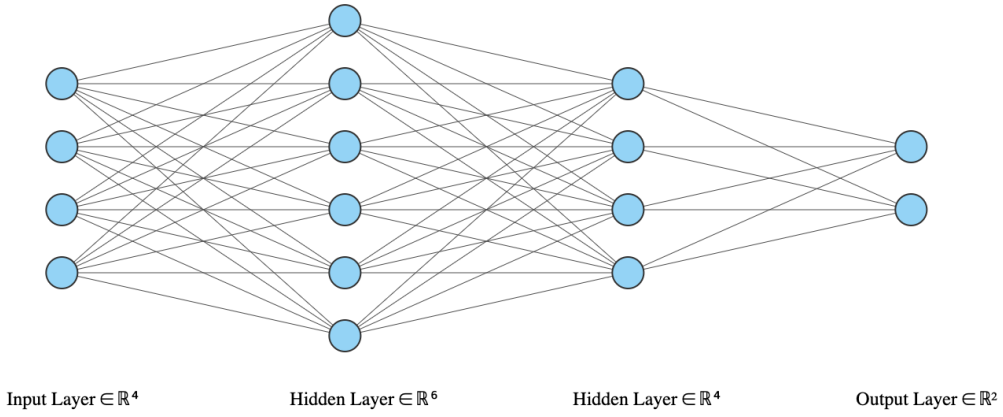


Figure 14: A basic FFNN architecture

Convolutional Neural Networks (ConvNets or CNNs)

Convolutional Neural Networks use convolutional layers to perform hierarchical feature extraction. One common difference between the FFNN and CNNs is that only the last layer in a CNN is fully connected. In an FFNN, each layer is a fully connected layer, i.e., all the nodes of a hidden layer are connected to each node in the neighbor layer. The CNNs or ConvNets are commonly used for image classification, image segmentation, object detection, and text classification.

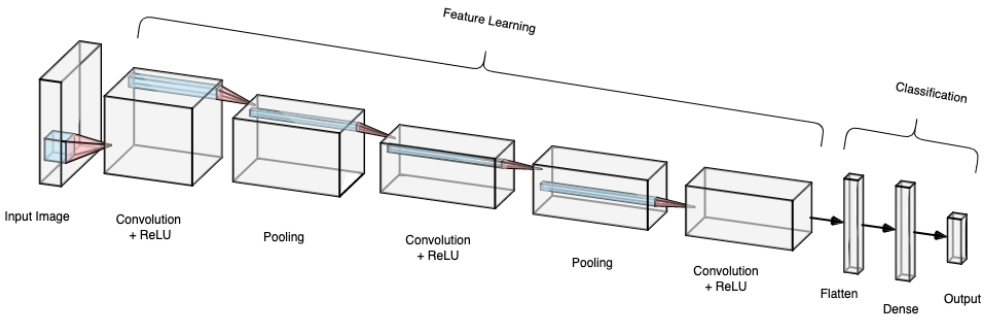


Figure 15: A basic CNN architecture

Recurrent Neural Networks (RNNs)

Recurrent Neural Networks have one hidden layer per time slice, thus enable to store past information for long time. In comparison to other Neural Networks, RNNs are difficult to train. Long Short Term Memory (LSTMs) and Gated Recurrent Networks (GRUs) are two counter-parts of Recurrent Neural Networks.

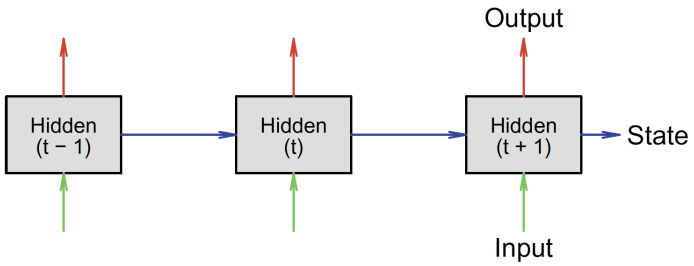


Figure 16: A basic RNN architecture - $References^{1}$

III.II Forward Propagation

Forward propagation is one of the core phases in the learning phase. The input data moves forward through the successive hidden layers and the data transformation happens at each hidden layer. The output of from each layer feeds as input into the next successive layer.

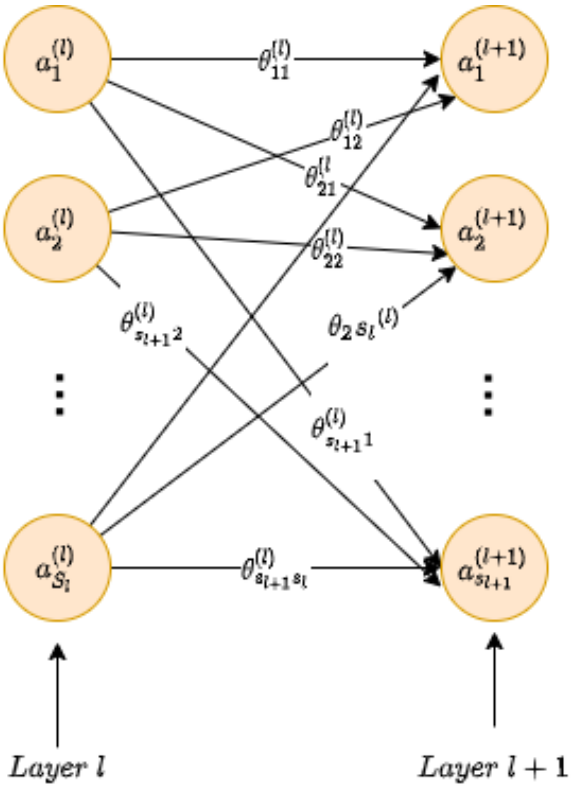


Figure 17: Forward propagation

The above diagram shows two adjacent hidden layers l of size S_l and $l+1$ of size S_{l+1} in a NN model.All the nodes in layer l are connected to the nodes of successive layer $l+1$. The notation representation is as follow:-

- Each neuron/node is represented as a_i^{layer}, where subscript i is the index of the node and superscript $layer$ refers to the index of the hidden layer.

- Weights are represented as θ_{ij}^{l} where j is the starting node on layer l and i is the target node on $l+1$

$$a_j^{(l)} \xrightarrow{\theta_{ij}^{(l)}} a_i^{(l+1)}$$

At each neuron in a hidden or an output layer, the following two operations happen:-

- Weighted sum of the inputs $(\theta_{i0}^{(l)} + \theta_{i1}^{(l)}.a_1^{(l)} + \cdots + +\theta_{is_l}^{(l)}.a_{s_l}^{(l)})$
  Here $\theta_{i0}^{(l)}$ is the bias unit.
- Activation - Evaluate the weighted sum with an activation function φ like sigmoid, tanh, ReLU, softmax, etc.

The overall operation looks like

$$a_i^{(l+1)} = \varphi(\theta_{i0}^{(l)} + \theta_{i1}^{(l)}.a_1^{(l)} + \cdots + +\theta_{is_l}^{(l)}.a_{s_l}^{(l)})$$

We can further elaborate this operation using Linear algebra. Let us convert the layer inputs into vectors.

$$\text{Vector of units in layer } l = V_l = \begin{pmatrix} a_1^{(l)} \\ a_2^{(l)} \\ \vdots \\ a_{s_l}^{(l)} \end{pmatrix} \in \mathbf{R}^{s_l} \tag{4}$$

The extended vector with the bias $a_0^{(l)} = 1$ from layer l is given by

$$\text{Extended vector for layer } l = \bar{V}_l = \begin{pmatrix} 1 \\ a_1^{(l)} \\ a_2^{(l)} \\ \vdots \\ a_{s_l}^{(l)} \end{pmatrix} \in \mathbf{R}^{s_l+1} \tag{5}$$

$$\text{Vector of units in layer } l+1 = V_{l+1} = \begin{pmatrix} a_1^{(l+1)} \\ a_2^{(l+1)} \\ \vdots \\ a_{s_{l+1}}^{(l+1)} \end{pmatrix} \in \mathbf{R}^{s_{l+1}} \tag{6}$$

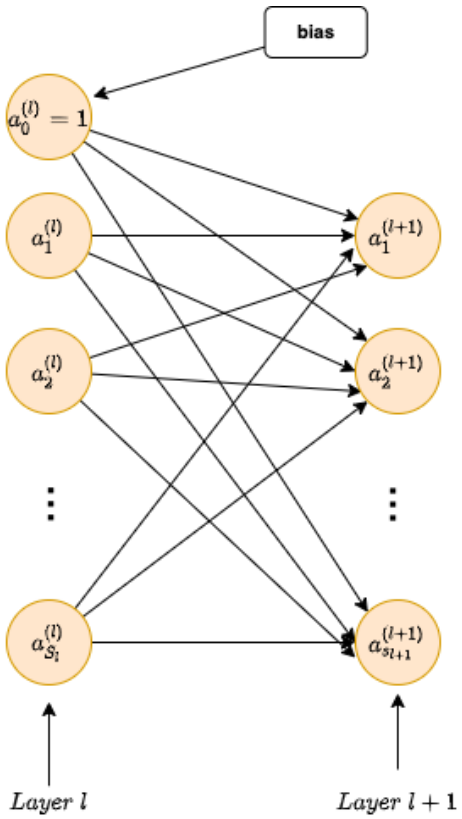


Figure 18: Forward propagation with bias

The arrows connecting the neurons from layer l to $l+1$ carries weight $\theta_{ij}^{(l)}$

$$1 \leq i \leq s_{l+1}$$

$$0 \leq j \leq s_l$$

All the weights can be written in a matrix format as follow:

$$\theta^{(l)} = \begin{pmatrix} \theta_{10}^{(l)} & \theta_{11}^{(l)} & \theta_{12}^{(l)} \dots & \theta_{1s_l}^{(l)} \\ & a_2^{(l+1)} & & \\ \theta_{20}^{(l)} & \theta_{21}^{(l)} & \theta_{22}^{(l)} \dots & \theta_{2s_l}^{(l)} \\ \vdots & \vdots & \vdots & \vdots \\ \theta_{s_{l+1}0}^{(l)} & \theta_{s_{l+1}1}^{(l)} & \theta_{s_{l+1}2}^{(l)} \dots & \theta_{s_{l+1}s_l}^{(l)} \end{pmatrix} \longrightarrow \text{Size of matrix} = s_{l+1}.(s_l + 1) \tag{7}$$

The forward propagation can be represented in a lucid manner using Linear Algebra

$$v_{l+1} = \varphi(\theta^{(l)}.\bar{V_l})$$

We can consider the basic format of a Neural Network with the n input features and s output units. Hyperparameters

k - no. of layers ($k-2$ hidden layers)

s_i - no. of units in i^{th} layer

φ - activation function (may differ for each layer)

Parameters

$\theta^{(l)}$ - weight matrix, here $l = 1, 2, \ldots, k-1$

Total parameters

$$\sum_{l=1}^{k-1}(s_l + 1).(s_{l+1})$$

All these parameters need to be optimized as part of the training process. As an outcome of successful training, we select an optimal set of parameters $\hat{\theta}$.

As an example, we can count the parameters for a NN architecture with one hidden layer (4 nodes), two inputs and one output. As we can see from the Figure 19, The total parameters are

$$(s_1 + 1)s_2 + (s_2 + 1)s_3 = 3 \times 4 + 5 \times 1 = 17$$

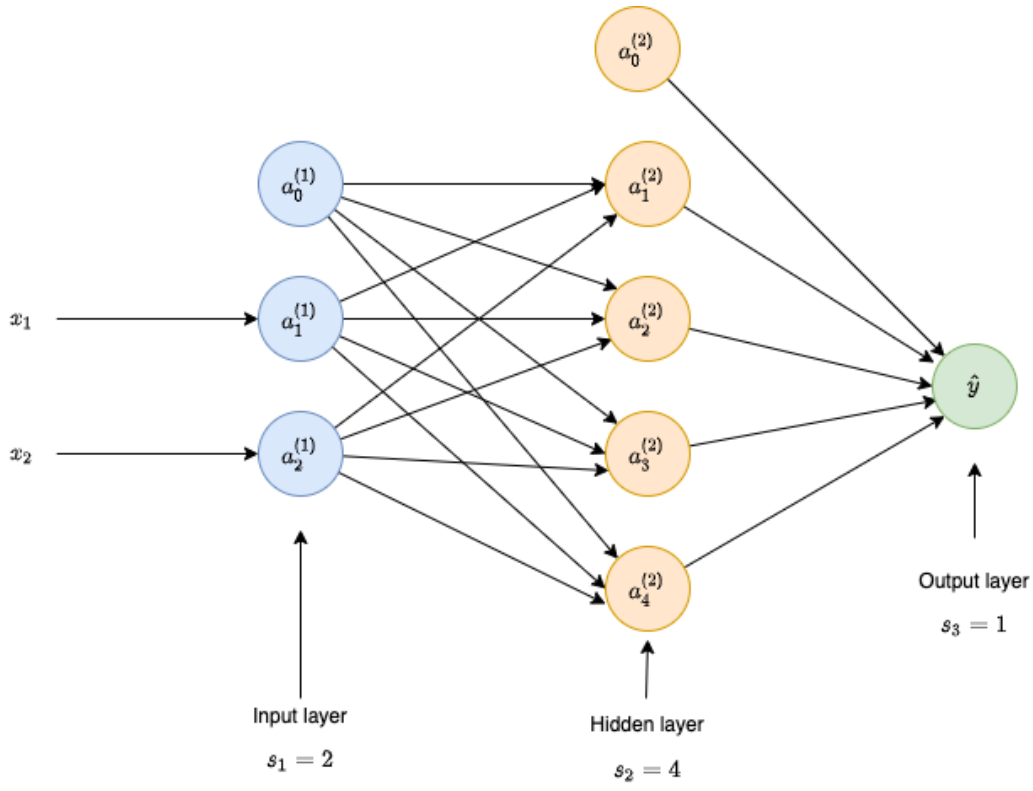


Figure 19: NN Architecture with one hidden layer (size=4)

The Power of Hidden Layer - Example of NN for XOR

XOR is a logical function, which outputs true when fed with odd number of true inputs. Lets us consider two binary inputs x_1 and x_2, then x_1 XOR x_2 is

$$x_1 \bigoplus x_2 \equiv (x_1 \vee x_2) \wedge \neg(x_1 \wedge x_2)$$

x_1	x_2	y
0	0	0
1	0	1
0	1	1
1	1	0

Table 3: XOR: Truth table

The geometrical representation of the data looks like below

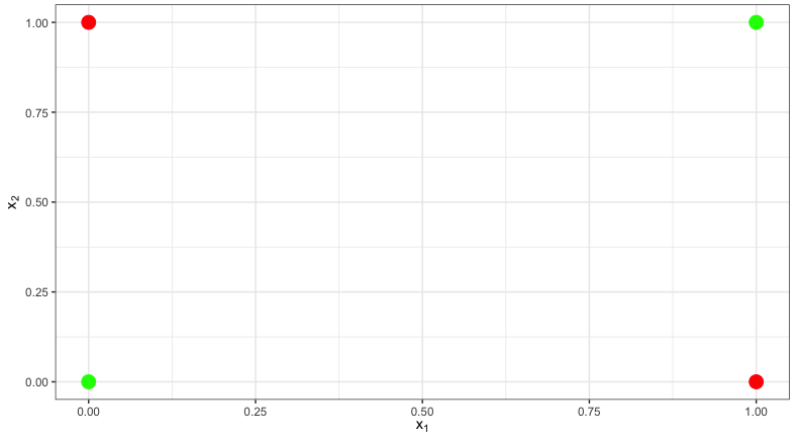


Figure 20: XOR values representation

As we notice, no linear solution can segregate the red and green points or the outcome of a logical XOR operation. We can solve this problem using a NN with a hidden layer.

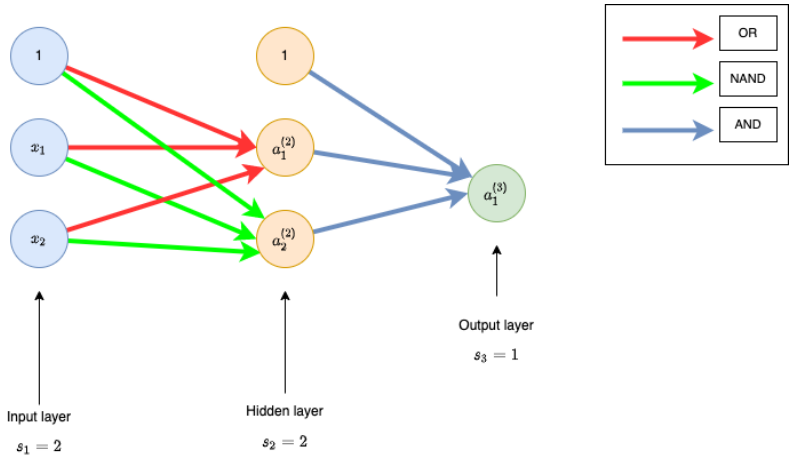


Figure 21: NN for XOR Operation

We can use the sigmoid activation function at layer 2 and layer 3 along with the following weight matrix to solve this problem

$$\theta^1 = \begin{pmatrix} -10 & 20 & 20 \\ 30 & -20 & -20 \end{pmatrix}$$

$$\theta^2 = \begin{pmatrix} -30 & 20 & 20 \end{pmatrix}$$

III.III Activation Functions

Activation functions play a crucial role of transforming the inputs from the previous layer to a more meaningful representation. On each successive layer, this transformation of data gets closer to the expected output from the Neural Network model.
The output vector for a layer l in a NN is given as $v_{l+1} = \varphi(\theta^{(l)}.\bar{V_l})$, where φ represents the activation function applied to the dot product of the weights matrix $\theta^{(l)}$, and the extended vector $\bar{V_l}$. The following section covers some of the commonly used activation functions along with their properties:-

1. Identity Activation Function: This activation function do not perform any modification to the Linear function. This function appears in traditional Linear regression, where we have the linear function, but we do not perform any operation on the outcome of the linear function.

$$\varphi(Z) = Z, \quad \varphi : \mathbf{R} \to \mathbf{R}$$

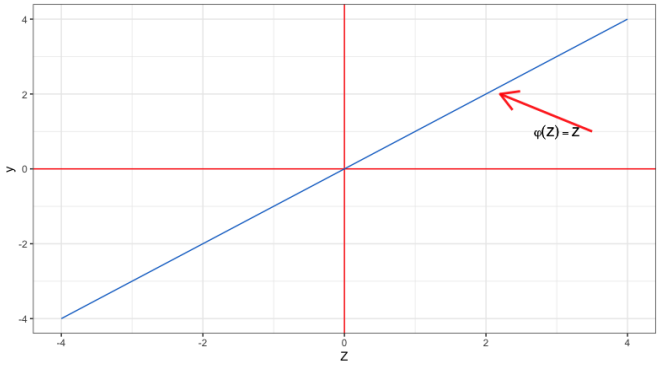


Figure 22: Identity Activation Function

2. Rectified Linear Unit or ReLU: This is the most commonly used activation function on the hidden layers in a NN. ReLU return a 0 if the input is less than or equal to 0, and returns the input if the input is greater than 0. So, it is used when we want the output of a neuron be always $+ve$.

$$\varphi(Z) = max(Z, 0), \quad \varphi : \mathbf{R} \to [0, \infty)$$

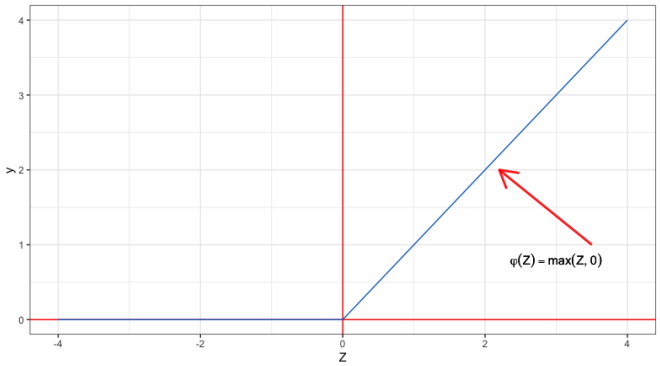


Figure 23: ReLU Activation Function

3. Leaky Rectified Linear Unit or LReLU: This function allows some values on the $-ve$ side and behaves like an Identity function for $+ve$ values.

$$\varphi : \mathbf{R} \to \mathbf{R}$$

$$\varphi(Z) = \begin{cases} z, & \text{if } z > 0 \\ az, & \text{if } z \leq 0; \text{usually } a = 0.01 \end{cases}$$

This function is called as **randomized relu** when $a \neq 0.01$

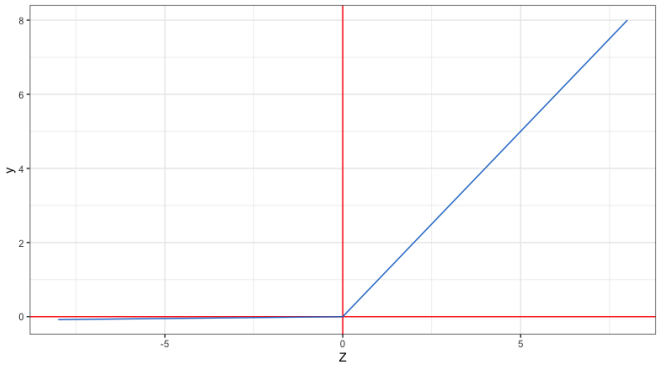


Figure 24: Leaky ReLU Activation Function

Both ReLU and LReLU are widely used for Regression problems due to their nature of Linearity.

4. Sigmoid: Sigmoid activation function is most commonly used in the Binary classification problems in Logistic Regression. This function takes inputs of real numbers and return a probability.

$$\varphi : \mathbf{R} \to [0, 1)$$

$$\varphi(Z) = \frac{1}{1 + e^{-Z}}$$

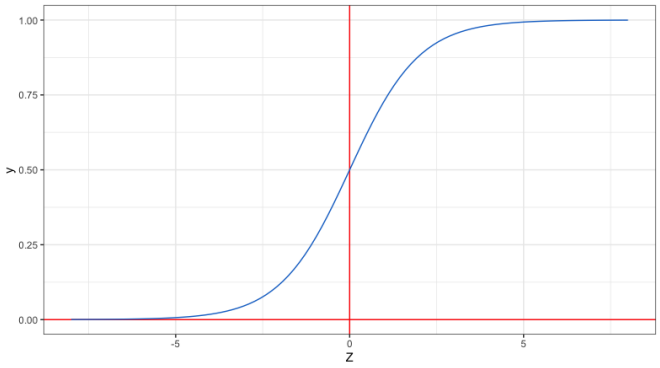


Figure 25: Sigmoid Activation Function

5. Hyperbolic Tangent: This activation looks like the Sigmoid activation function in shape but the output value ranges between -1 and $+1$. This is also used in the Binary classification problems, but does not return a probability.

$$\varphi : \mathbf{R} \to [-1, 1)$$

$$\varphi(Z) = \frac{e^{Z} - e^{-Z}}{e^{Z} + e^{-Z}}$$

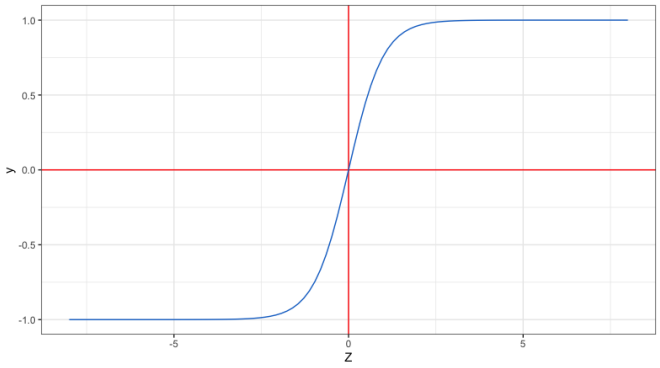


Figure 26: Hyperbolic Tangent Activation Function

6. Softmax: This activation function is used for a multi-class classification problem. It takes k numerical entries and transforms into k probabilities.

$$p : \mathbf{R}^k \to (0, 1)^k$$

$$p\begin{pmatrix} z_1 \\ z_2 \\ \vdots \\ z_k \end{pmatrix} = \begin{pmatrix} p_1 \\ p_2 \\ \vdots \\ p_k \end{pmatrix} \tag{8}$$

$$p_i : \mathbf{R}^k \to (0, 1)$$

The i^{th} probability is given by

$$p_i = \frac{e^{z_i}}{e^{z_1} + e^{z_2} + \cdots + e^{z_k}}$$

All the probabilities for the k classes sum up to 1.

$$p_1 + p_2 + \cdots + p_k = 1$$

III.IV Derivatives of Activation Functions

Derivative of a function is the rate of change of the function with respect to a variable. For a function $f(x)$, the derivative is written as $f'(x)$ or $\frac{df(x)}{dx}$. We train Neural Networks with gradient descent, so partial derivatives come into play. The goal is to minimize the error, and if we know how the error changes with a change in weights, we can change the weights in a direction to minimize the error. The derivatives for commonly used Activation functions are as follow: -

- Derivative of Sigmoid: Sigmoid function is normally used in a binary classification scenario's final layer of a NN model. It provides the probability score for the output. The Sigmoid function is given as

$$\varphi(Z) = \frac{1}{1+e^{-Z}}$$

The derivative of the Sigmoid function is as follows:-

$$\frac{d}{dZ}\varphi(Z) = \varphi(Z)(1-\varphi(Z))$$

- Derivative of $tanh$: Hyperbolic tangent function is given as follow:-

$$\varphi(Z) = \tanh Z = \frac{\sinh Z}{\cosh Z} = \frac{e^{Z}-e^{-Z}}{e^{Z}+e^{-Z}}$$

The derivative of $\tanh$ is represented as:-

$$\frac{d}{dZ}\varphi(Z) = \frac{d}{dZ}\tanh(Z) = 1-\tanh^2(Z)$$

- Derivative of Rectified Linear Unit: ReLU function is represented as

$$\varphi(Z) = relu(Z) = max(Z,0)$$

The derivative of ReLU is given as

$$\frac{d}{dZ}\varphi(Z) = \frac{d}{dZ}relu(Z)\begin{cases}1, & \text{if } z>0\\ 0, & \text{otherwise}\end{cases}$$

- Derivative of Leaky Rectified Linear Unit: The Leaky ReLU function is represented as

$$\varphi(Z) = lrelu(Z) = \begin{cases}z, & \text{if } z>0\\ az, & \text{if } z\leq 0; \text{usually } a=0.01\end{cases}$$

The derivative of Leaky ReLU is given as

$$\frac{d}{dZ}\varphi(Z) = \frac{d}{dZ}lrelu(Z)\begin{cases}1, & \text{if } z>0\\ a, & \text{otherwise}\end{cases}$$

- Derivative of Softmax is given as follows

$$\frac{\partial p_i}{\partial Z_j} = \begin{cases}p_i(1-p_j), & \text{if } i=j\\ -p_i p_j, & \text{if } i\neq j\end{cases}$$

Here p_i is the output vector and z_j is the input vector.

III.V Cross-Entropy Loss

The training data set has the pairs as (V_t, y_t) and the output $Y = y \in \{0, 1\}$. The NN returns a probability for output class for each input record V_t i.e. the probability $p(V_t) \in (0, 1)$. The Cross-Entropy Loss for a binary classification problem is given by

$$\mathcal{J}(\theta) = \sum_{t=1}^{m} [-y_t.log(p(V_t)) - (1 - y_t).log(1 - p(V_t))]$$

The value of the Cross-Entropy Loss changes as we change the weights of the Neural Network.

In case of a multi-class classification problem, suppose we have s classes $\{k_1, k_2, \ldots, k_s\}$ The output $Y_t = (y_1^t\ y_2^t \ldots y_s^t) \in \{0, 1\}^s$ with

$$y_q^t = \begin{cases} 1, & \text{if } v_t \in k_q \\ 0, & \text{if } v_t \notin k_q \end{cases}$$

Multi-class cross-entropy loss is given by :

$$\mathcal{J}(\theta) = \sum_{q=1}^{s} \sum_{t=1}^{m} [-y_q^t.log(p^q(V_t)) - (1 - y_q^t).log(1 - p^q(V_t))]$$

III.VI Back Propagation

The history of the backpropagation goes back to a paper published by Rumelhart, Hinton, and Williams in 1986. The algorithms apply to FFNN, and an enhanced version can be used on other NNs like ConvNets. Back Propagation or Backprop is the critical part of a Neural Network, which involves optimization of the network parameters or the training phase to minimize the loss function $\mathcal{J}$. At a high level, minimizing a function involves taking a derivative of the function and equating it to zero, i.e., $\frac{\partial \mathcal{J}}{\partial w} = 0$. However, finding the minima of the cost function in a NN is complicated due to the following:-

- Multiple equations need to be optimized as we deal with weights from different layers.
- Number of weights could be different for each layer depending upon the no. of nodes in a layer.
- We need to find a global minimum while multiple local minima can exist.

The Backpropagation Algorithm computes all the partial derivative of the total Loss function w.r.t the weight directions $\frac{\partial \mathcal{J}}{\partial \theta_{ij}^{(l)}}$. The backpropagation is performed by establishing a cost function, e.g., choosing a mean square error (MSE) for a regression problem or Cross-Entropy for a binary or multi-class classification problem. We measure the performance of the NN on each forward pass based on the cost function. The network computes the gradient of the cost with respect to the weights and works backward to adjust the weights. This process is repeated to reduce the cost function to its minimum possible value.

To illustrate the working of the Backpropagation algorithm, let us consider a binaryy classification problem with n input features $(x_1, x_2, \ldots, x_n)$ and output $y \in \{0, 1\}$. The NN in this case predicts the probability p of $y = 1$. We choose Sigmoid activation functions for all layers i.e. $l = 2, 3, \ldots, k$ ($l = 1$ is the input layer). And we choose cross-entropy loss function to measure the performance of the NN.

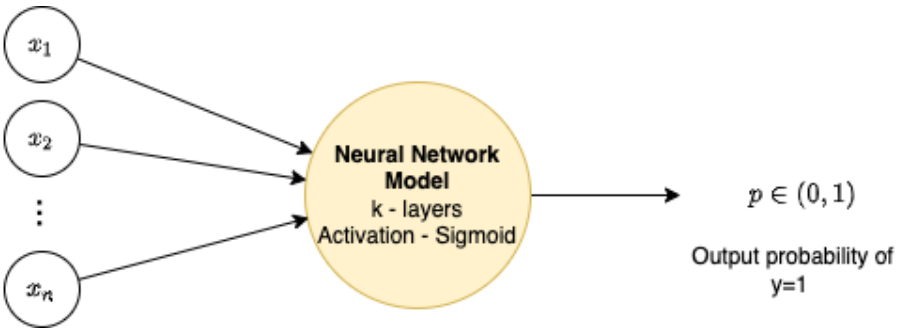


Figure 27: Network Illustration for Binary Classification problem

Suppose we have m records in the training set i.e. $(V_t, y_t), t = 1, 2, \ldots, m$. The cross-entropy loss function is given as

$$\mathcal{J}(\theta) = \sum_{t=1}^{m}[-y_t.log(p(V_t)) - (1 - y_t).log(1 - p(V_t))]$$

Backpropagation Algorithm - Case K=2

We start with a case of Logistic regression with just an input and output layer $(k = 2)$. As we discussed earlier, there are two operations happening at each hidden layer namely Linear operation with weights and inputs followed by the activation function Sigmoid over the Linear sum. To put it in mathematical equation, it looks like the follow:-

$$\text{Weights: } \theta_0, \theta_1, \theta_2, \ldots, \theta_n$$

$$\text{Linear Sum: } z = \theta_0.x_0 + \theta_1.x_1 + \theta_2.x_2 + \cdots + \theta_n.x_n = \theta_0 + \theta_1.x_1 + \theta_2.x_2 + \cdots + \theta_n.x_n, \quad x_0 = 1$$

$$\text{Activation: } p = g(z) = \frac{1}{1 + e^{-z}}$$

$$\text{Loss function: } \mathcal{J}(\theta) = \sum_{t=1}^{m}[-y_t.log(g(z_t)) - (1 - y_t).log(1 - g(z_t))]$$

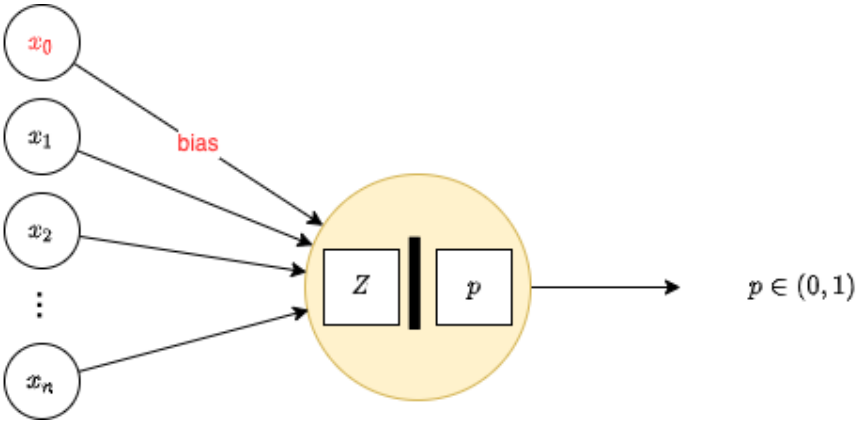


Figure 28: NN Overview for Logistic Regression

The derivative of the Cost function involves partial derivative of the Activation function and the Linear sum function. We noticed in the earlier section that the derivative of the Sigmoid function is

given by:

$$\frac{d}{dz}\varphi(z) = \frac{d}{dz}g(z) = g(z)(1 - g(z)) \tag{9}$$

And the derivative of the Linear sum operation is given by:

$$\frac{d}{d\theta_i}(Z) = \frac{d}{d\theta_i}(\theta_0.x_0 + \theta_1.x_1 + \theta_2.x_2 + \cdots + \theta_i.x_i + \ldots, \theta_n.x_n) = x_i \tag{10}$$

The derivative of the cost function w.r.t. weight θ_0:

$$\begin{aligned}
\frac{d}{d\theta_0}\mathcal{J}(\theta) &= \sum_{t=1}^{m}\frac{d}{d\theta_0}[-y_t.log(g(z_t)) - (1 - y_t).log(1 - g(z_t))] \\
&= -\sum_{t=1}^{m}\frac{d}{d\theta_0}[y_t.log(g(z_t)) + (1 - y_t).log(1 - g(z_t))] \\
&= -\sum_{t=1}^{m}\left(\frac{y_t}{g(z_t)}.g'z(t).\frac{dz_t}{d\theta_0}\right) + \left(\frac{1 - y_t}{1 - g(z_t)}.-g'z(t).\frac{dz_t}{d\theta_0}\right) \\
&= -\sum_{t=1}^{m}\left(\frac{y_t}{g(z_t)}.g(z_t)(1 - g(z_t)).x_0\right) + \left(\frac{1 - y_t}{1 - g(z_t)}.(-g(z_t)(1 - g(z_t))).x_0\right), \text{using 9, 10.} \\
&= -\sum_{t=1}^{m}(y_t(1 - g(z_t)).1) + (1 - y_t).(-1).g(z_t).1), \text{using } x_0 = 1 \\
&= -\sum_{t=1}^{m}(y_t - y_t.g(z_t) - g(z_t) + y_t.g(z_t)) \\
&= \sum_{t=1}^{m}(g(z_t) - y_t)
\end{aligned} \tag{11}$$

Similarly, we can find the partial derivative of the cost function $\mathcal{J}$ w.r.t weights θ_i, $i \geq 1$

$$\begin{aligned}
\frac{d}{d\theta_i}\mathcal{J}(\theta) &= \sum_{t=1}^{m}\frac{d}{d\theta_i}[-y_t.log(g(z_t)) - (1 - y_t).log(1 - g(z_t))] \\
&= -\sum_{t=1}^{m}\frac{d}{d\theta_i}[y_t.log(g(z_t)) + (1 - y_t).log(1 - g(z_t))] \\
&= -\sum_{t=1}^{m}\left(\frac{y_t}{g(z_t)}.g'z(t).\frac{dz_t}{d\theta_i}\right) + \left(\frac{1 - y_t}{1 - g(z_t)}.-g'z(t).\frac{dz_t}{d\theta_i}\right) \\
&= -\sum_{t=1}^{m}\left(\frac{y_t}{g(z_t)}.g(z_t)(1 - g(z_t)).x_i^t\right) + \left(\frac{1 - y_t}{1 - g(z_t)}.(-g(z_t)(1 - g(z_t))).x_i^t\right), \text{using 9, 10.} \\
&= -\sum_{t=1}^{m}x_i^t(y_t - y_t.g(z_t) - g(z_t) + y_t.g(z_t)) \\
&= \sum_{t=1}^{m}x_i^t(g(z_t) - y_t)
\end{aligned} \tag{12}$$

We have these two final equation for the partial derivatives: -

$$\frac{d}{d\theta_0}\mathcal{J}(\theta) = \sum_{t=1}^{m}(g(z_t) - y_t)$$

$$\frac{d}{d\theta_i}\mathcal{J}(\theta) = \sum_{t=1}^{m}x_i^t(g(z_t) - y_t), \quad i \geq 1$$

The predicted output is represented as $p = g(z_t)$, and y_t is the actual output label from the training set. So, the error for the training pair (V_t, y_t) is given as:

$$\delta_t = g(z_t) - y_t$$

With this, we can re-write the above equations as:-

$$\frac{d}{d\theta_i}\mathcal{J}(\theta) = \begin{cases} \sum_{t=1}^{m} \delta_t, & \text{if } i = 0 \\ \sum_{t=1}^{m} x_i^t.\delta_t, & \text{if } i \geq 1 \end{cases}$$

We have a common pattern for the partial derivative i.e. $\frac{d}{d\theta_i}\mathcal{J}(\theta) = \sum_{t=1}^{m} x_i^t.\delta_t$, $x_0 = 1$ for the bias node.

At a high level, we have the following steps in the Back propagating algorithm:-

1. Identify the training set (V_t, y_t).
2. Feed the input variables V_t to the NN model.
3. Compute probability $p = g(v_t)$ using forward propagation.
4. Compute the error $\delta_t = g(v_t) - y_t$.
5. Backpropagate the error corresponding to θ_i. Multiply δ_i with the origin unit value x_i.
   (a) Sum over all the pairs in the training data.
   Algorithm to compute $\frac{d}{d\theta_i}\mathcal{J}$

General Backpropagation Algorithm

This section will discuss the backpropagation algorithm for a general binary classification problem. We consider a simple case with a $n - dimensional$ input vector with binary output. This network has several hidden layers (k), and we predict the probability of binary class 1 in the output layer. We can think of a training set with t records of labeled pairs (V_t, y_t).

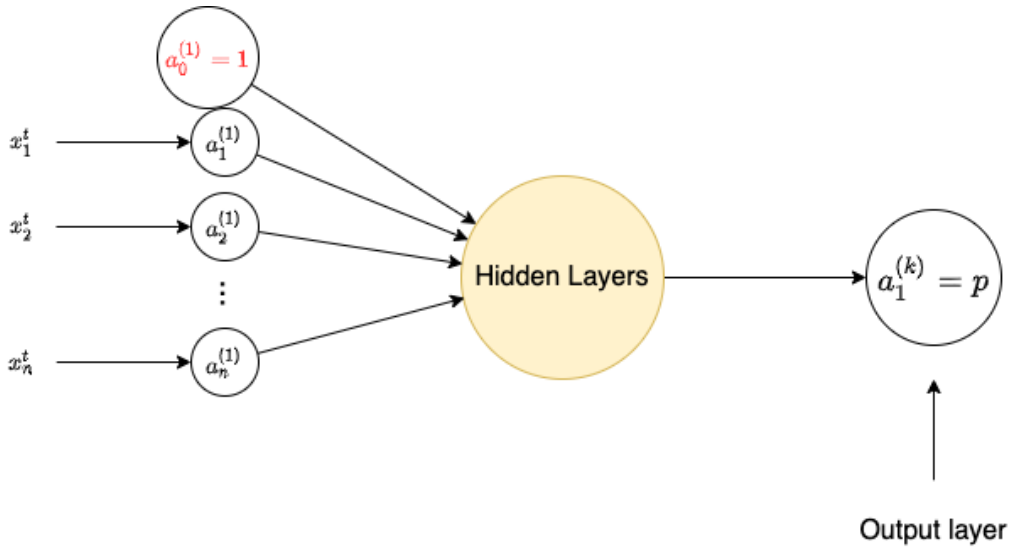


Figure 29: Generic NN for Binary classification

If we feed the input features for a record from the training set to the Neural Network, we would get a value at the output layer as a probability of a binary class, denoted as $a_1^{(k)}$ in the above figure. The final predicted class is compared with the target label or the given output (y_t) to find the error in the output layer. The error is given by:

$$\delta^{(k)} = a_1^{(k)} - y_t$$

This error term depends upon the weights of the network and the specific input pair (V_t, y_t) used. There are other hidden errors at each hidden layer that are propagating backward through the network. Suppose, we have a hidden layer l of size s_l, then the error vector will have $s_l + 1$.

$$\bar{\delta}^{(l)} = \begin{bmatrix} \delta_0^{(l)} \\ \delta_1^{(l)} \\ \delta_2^{(l)} \\ \vdots \\ \delta_{s_l}^{(l)} \end{bmatrix} \in \mathbf{R}^{s_l+1}$$

Here, $\delta_0^{(l)}$ is the error corresponding to the bias unit. So, $\bar{\delta}^{(l)}$ is called as the extended vector of error terms at layer l. The normal vector without the bias error is referred to as $\delta^{(l)}$.

The general formula to compute these intermediate error vectors in layer l is:

$$\bar{\delta}^{(l)} = [\theta^{(l)}]^t \delta^{(l+1)} \odot a^{(l)} \odot (1 - a^{(l)}),\ l \geq 2$$

In this formula, we have:

$[\theta^{(l)}]$ - Weight matrix for all the connections from $layer : l$ to $layer : l + 1$ with a size of s_{l+1} (rows) $\times$ $s_l + 1$ (columns). $[\theta^{(l)}]^t$ will be a weight matrix of size $s_l + 1 \times s_{l+1}$.

$\odot$ - Represents the element-wise multiplication, also called as Hadamard product of two vectors. Example:

$$\begin{bmatrix} 3 \\ 2 \\ 1 \end{bmatrix} \odot \begin{bmatrix} 1 \\ 4 \\ 2 \end{bmatrix} = \begin{bmatrix} 3 \\ 8 \\ 2 \end{bmatrix}$$

$\bar{a}^{(l)}$- Extended vector of all the entries in layer l (including bias $a_0^{(l)}$), with size $s_l + 1$. We get the values $a_i^{(l)}$ from forward-propagation.

$(1 - \bar{a}^{(l)})$: Performing a minus operation for all the entries in $\bar{a}^{(l)}$ resulting in a $s_l + 1$ dimensional vector.

As we can see, this is a backward propagating method to find the errors in layer l by using the errors in $l + 1$. To illustrate the working of Back-propagation as compared to forward-propagation, consider a NN with four layers.

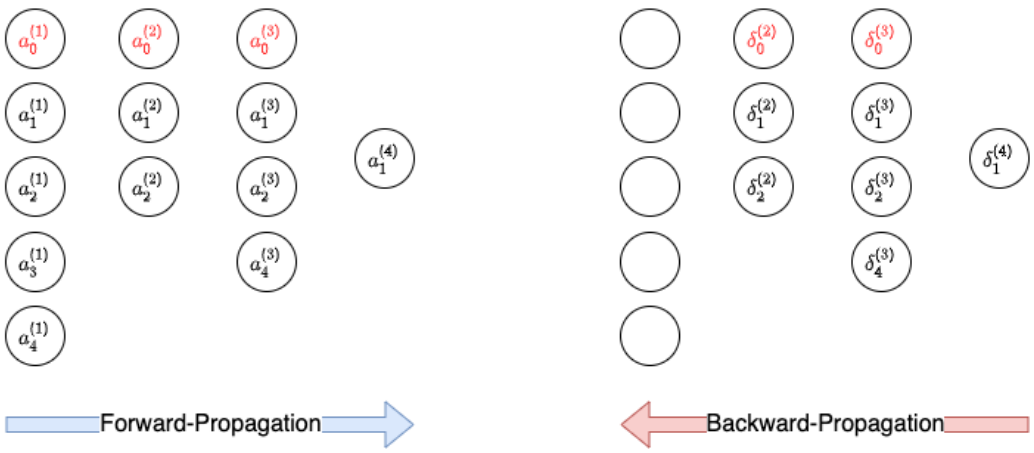


Figure 30: Forward-propagation v/s Backward-propagation

The above figure shows the network's forward and backward propagation values. Forward propagation moves from left to right, whereas backward propagation moves from the right to left except in the input layer. So, back propagation ensures that each hidden layer other than the input layer has the error value for every neuron.

The error vector $\delta_i^{(l+1)}$ is the key to calculating the partial derivatives of the total loss function $\mathcal{J}$, which is given by the following Backpropagation formula.

$$\frac{d\mathcal{J}}{d\theta_{ij}^{(l)}} = \sum_{t=1}^{m} \delta_i^{(l+1)} . a_j^{(l)}$$

The following figure shows a more simplistic picture of this formula.

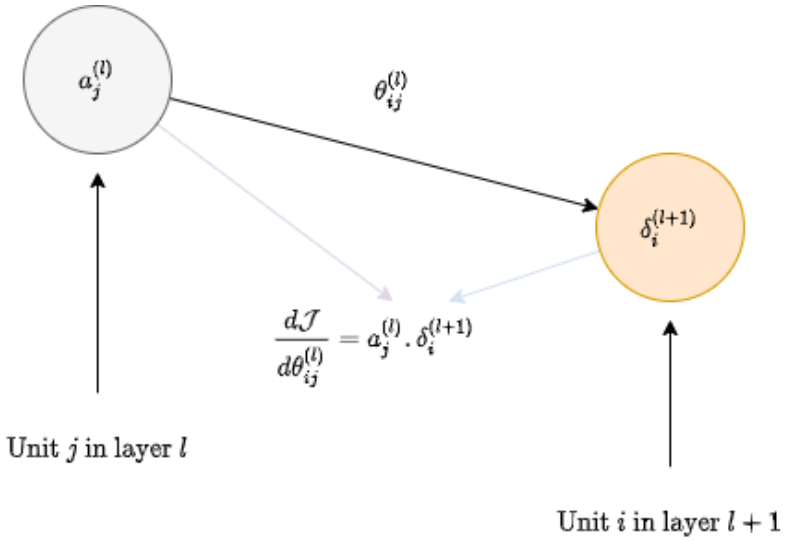


Figure 31: Backpropagation formula - breakdown

The RHS of this equation $\frac{d\mathcal{J}}{d\theta_{ij}^{(l)}} = \sum_{t=1}^{m} \delta_i^{(l+1)} . a_j^{(l)}$ is the sum of the values obtained from all the pairs (V_t, y_t) in the training set.

The following steps work in an ordered fashion as part of finding the partial derivatives in a NN:-

1. Feed the network model with the input from one pair (V_t, y_t) in the training set.
2. Perform forward propagation to compute all the values for the activation. For example, to calculate $a_i^{(l+1)}$.
   (a) Consider the bias unit where applicable $a_0^{(l)}$
   (b) Compute the linear sum $z_i^{(l+1)} = \theta_{i0}^{(l)}.a_0^{(l)} + \theta_{i1}.a_1^{(l)} + \theta_{i2}.a_2^{(l)} + \cdots + \theta_{is_l}.a_{s_l}^{(l)}$
   (c) Apply the activation function $a_j^{(l)} = g(z_i^{(l)})$
3. Compute the error at the final layer as the difference between the predicted output and the actual output $\delta_1^k = a_1^{(k)} - y$.
4. Perform backpropagation to find $\delta_i^{(l)}$, i.e. intermediate errors for all the hidden layers except the input layer.
$$\bar{\delta}^{(l)} = [\theta^{(l)}]^t \delta^{(l+1)} \odot a^{(l)} \odot (1 - a^{(l)})$$
5. Repeat the process (Steps 1-4) for all pairs in the training set.
6. Compute the partial derivatives using $\frac{d\mathcal{J}}{d\theta_{ij}^{(l)}} = \sum_{t=1}^{m} \delta_i^{(l+1)}.a_j^{(l)}$

Training

Training is the process of minimizing the Loss function $\mathcal{J}$ by finding the appropriate weights for the network. Gradient Descent is used as a method to minimize the loss function $\mathcal{J}$. Suppose, we have a training set $(V_t, y_t), t = 1, 2, 3, \ldots, m$, with an n dimensional input variables and an s class output.

$$V_t = (x_1^t \; x_2^t \; \ldots \; x_n^t)$$

$$y_t = (y_1^t \; y_2^t \; \ldots \; y_s^t)$$

The weights for the network represented as $\theta_{ij}^{(l)}$ for the connections from layer l to layer $l+1$.

$$V_t = \begin{pmatrix} x_1^t \\ x_2^t \\ x_3^t \\ \vdots \\ x_n^t \end{pmatrix} \Rightarrow \begin{pmatrix} \text{NN with weights} \\ \theta \end{pmatrix} \Rightarrow \begin{pmatrix} f^1(V_t) \\ f^2(V_t) \\ f^3(V_t) \\ \vdots \\ f^s(V_t) \end{pmatrix} = f_\theta(V_t) \tag{13}$$

The NN is going to predict the output using the weights θ for a s class target i.e. $f_\theta(V_t) \in \mathbf{R}^s$. We measure how close we are to the predicted output by using $\mathcal{L}(f_\theta(V_t), y_t)$. We define a loss function as

$$\mathcal{J}(\theta) = \sum_{t=1}^{m} \mathcal{L}(f_\theta(V_t), y_t)$$

As part of training, we start with some initialization weights say

$$\theta^0 = (\theta_{ij}^{(l),0})_{i,j,l} \longrightarrow \text{ Large collection of parameters}$$

$$\text{Layers } l = 1, 2, \ldots, k-1$$

$$i = 1, 2, \ldots, S_{l+1}$$

$$j = 1, 2, \ldots, S_l$$

The weights are adjusted with learning rate α (hyperparameter) using the following formula:

$$\theta^{d+1} = \theta^d - \alpha . \nabla \mathcal{J}(\theta^d)$$

$$(\theta_{ij}^{(l),d+1})_{i,j,l} = (\theta_{ij}^{(l),d})_{i,j,l} - \alpha . \frac{d\mathcal{J}}{d\theta_{ij}^{(l)}}(\theta^d)$$

We keep optimizing the weights in each iteration to minimize the loss function $\mathcal{J}(\theta)$.

$$\theta^0 \rightsquigarrow \theta^1 \rightsquigarrow \theta^2 \rightsquigarrow \cdots \rightsquigarrow \theta^d \rightsquigarrow \theta^{d+1} \ldots$$

These attempts of optimizations are called epochs and are controlled via hyperparameter.

IV Convolutional Neural Networks

Convolutional Neural Networks or ConvNets, a.k.a. CNNs, are most commonly used for the problems in the domain of computer vision. ConvNets use the spatially local correlation by enforcing regional connectivity between the adjacent layers.

IV.I Building Blocks of CNN

We will start with the basics of the Convolutional Neural Network before exploring a CNN model. Some of the building blocks of a CNN are as follow:-

Convolution Operation

Images can be represented as matrix values in different channels depending upon the color space. Some commonly used color spaces are Grayscale, CMYK, HSV, RGB, etc. Convolution operation is performed by applying a filter, a.k.a. kernel, over the image, which computes the dot product of the filter with the are of the image under the filter. The filter slides over the complete image to perform this activity and reduces the image to a readable format without losing features.

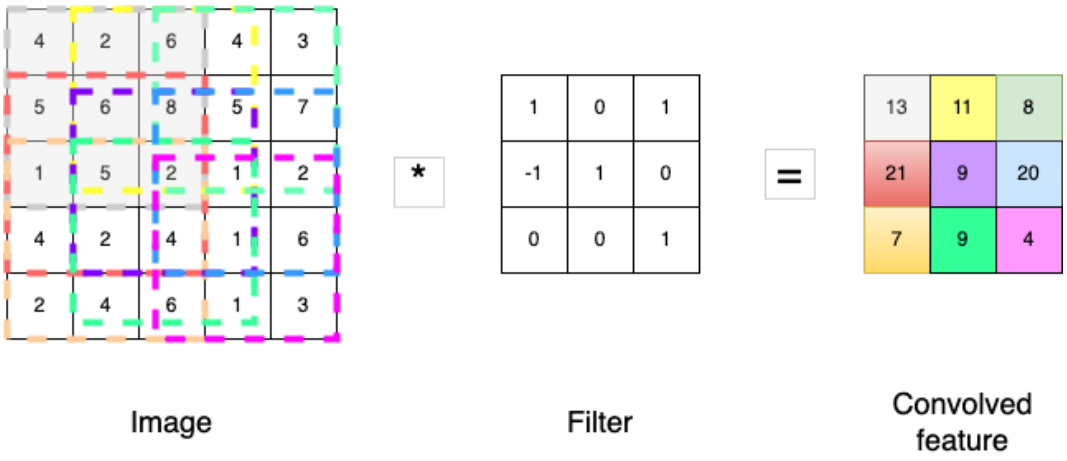


Figure 32: Convolution Operation - Overview

Suppose, we have an Image in the form of matrix values. We apply a filter of weights (kernel) 3×3 on the first row and first column of the matrix. As an example the first value of the convolution feature is calculated as follow: -
$4 \times 1 + 5 \times -1 + 1 \times 0 + 2 \times 0 + 6 \times 1 + 5 \times 0 + 6 \times 1 + 8 \times 0 + 2 \times 1 = 13$

We scan the whole matrix with the size matching the filter 3×3 to perform the convolution operation and generate a convolved features matrix of size 3×3 from a 5×5 matrix.

Edge Detection

Edges for the objects in an image are the areas where there is a drastic change in the brightness of the pixels. There are different filters to detect the vertical, horizontal or angle edges. An example of a vertical detector along with the convolution operation is as follow:-

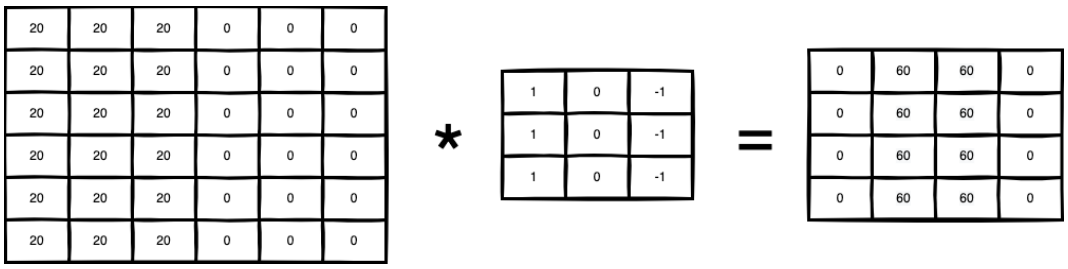


Figure 33: Convolution Example - Matrix operation ($References^1$)

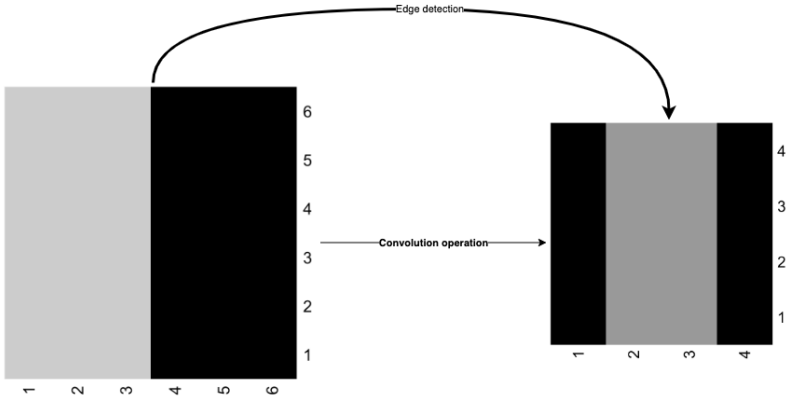


Figure 34: Convolution Example - Edge detection

Some commonly used edge detection filters are Canny, Sobel, Scharr, etc. The size of the output image matrix after applying a filter of size $f \times f$ on an image of matrix size $n \times n$ is $(n-f+1) \times (n-f+1)$. The value of f is odd to ensure that the filter always has a central position and to help with padding.

Padding

As we discussed in the filter operations above, the size of the output image is reduced to $n \times n$ is $(n - f + 1) \times (n - f + 1)$. In addition to the shrinkage in size, we also lose important information at the edges of the image. To overcome this problem, we pad extra pixels to the image border via a hyper-parameter called padding. If we apply a padding $p = 1$ on a $n \times n$, the resultant image size is $n + 2 \times n + 2$. Padding is configured using two modes in CNN models, namely:

1. Valid: No padding
2. Same: Pad to match the size of the output image with the input image.
   Example of Padding:

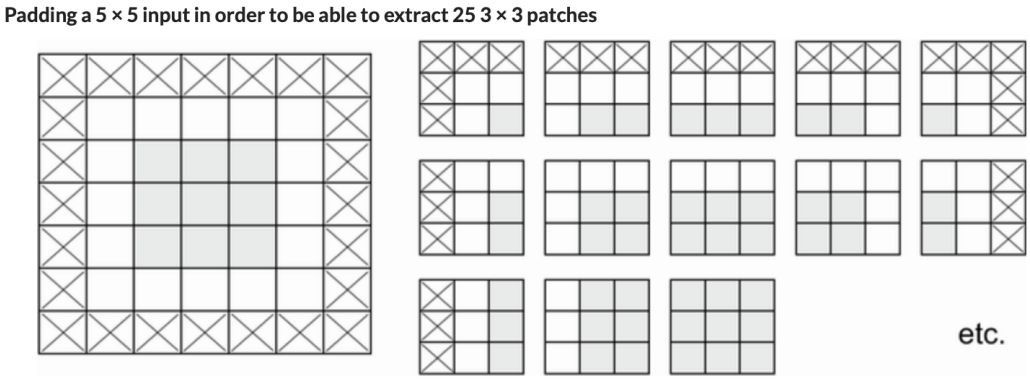


Figure 35: Padding Example - $References^2$

When a filter of size $f \times f$ is applied along with a padding of size p on an image of size $n \times n$, the resulting image size is given by $(n + 2p - f + 1) \times (n + 2p - f + 1)$.

Strided Convolution

Strides are another factor that can influence the output size of the convolution. Stride is the distance between two successive windows in a convolution operation. By default, the convolution windows are contiguous, i.e., a stride of size 1. Here is an example of a stride with size 2.

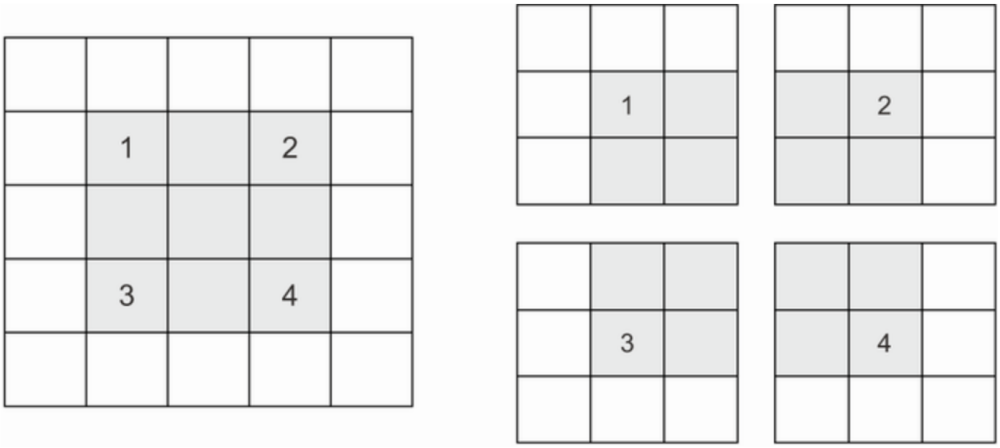


Figure 36: 3 × 3 convolution patches with 2 × 2 strides - $References^2$

The size of the convolution output with stride size s, padding p, filter $(f \times f)$, and original image size $(n \times n)$ is given by

$$(\frac{n + 2p - f}{s} + 1), (\frac{n + 2p - f}{s} + 1)$$

In the above formula, if the fraction is not an integer, we use the $floor$ function in R.

Convolutions over Volume

Color images are represented in multiple channels, i.e., more than one data matrix. For example, a common format for a color picture is the rgb channel, which means a stack of red, green, and blue

channels. To perform a convolution on a $3D$ image (with three channels), we need a $3D$ filter, i.e., with the depth of the filter matching the number of channels in the input image. The follow of the convolution operation on a volume $3D$ image with a $3D$ filter goes as follows:-

1. Take dot product of the filters with the corresponding values from the channels in the image, e.g., values from the red filter with the values from the red channel from the image, and so on.
2. Add up these dot products from each filter with the corresponding channel to produce a single value.

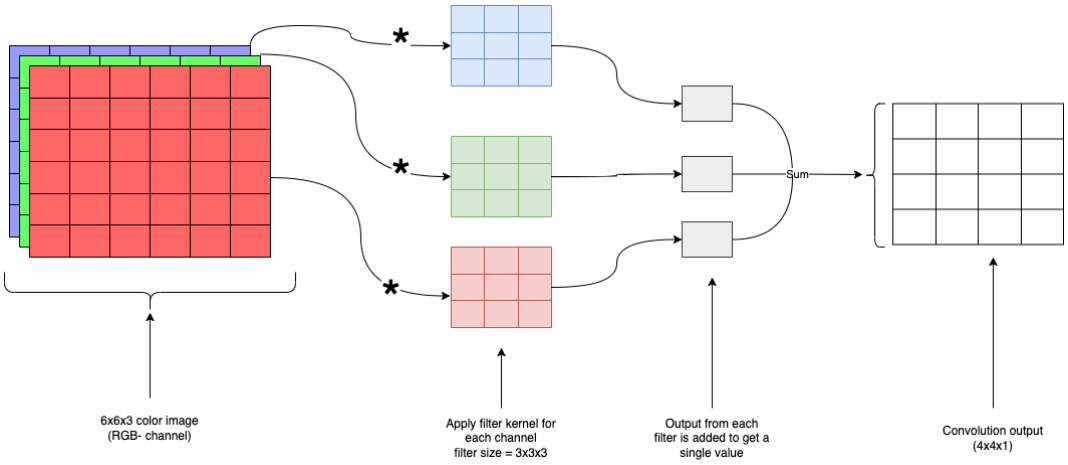


Figure 37: Volume convolution Example

If we want to detect the edges in a specific color channel, say red, we keep the other color channels, i.e., blue and green in this case, to zero and vice-versa. The edge filters are set to the same value for all channels when checking the edges across all the channels. We can use different filters to extract various images' features, like additional edge filters - horizontal, vertical, angular, etc.

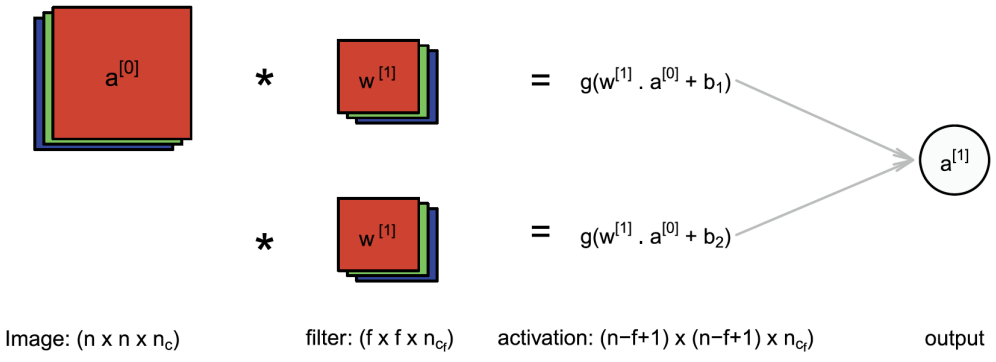


Figure 38: Using multiple filters to get a 2D output

Pooling

Pooling can help reduce the input size and, in turn, can lower the computation cost. It also makes the feature detection independent of its position in the image, known as invariant feature detection. Pooling is commonly performed in a window size of 2×2. Types of pooling layer:-

Max-pooling layer: Moves the Pool layer filter over the input image and stores the max value from input as output.

Average-pooling layer: Moves the Pool layer filter over the input image and stores the average value for input as output.

Max-pooling is most used as compared to Average-pooling. The pooling layers cannot be trained using the Backpropagation. However, these are controlled via hyper parameters like the window size of the pooling operation f, stride s, and the type of pooling max or average. If we apply the pooling layer with window size $f \times f$ on an input image of size $n_H \times n_W \times n_C$, and striding of size s, the output dimensions are $\frac{n_H - f}{s} + 1 \times \frac{n_H - f}{s} + 1 \times n_C$ (padding is usually not used with the max-pooling operation).

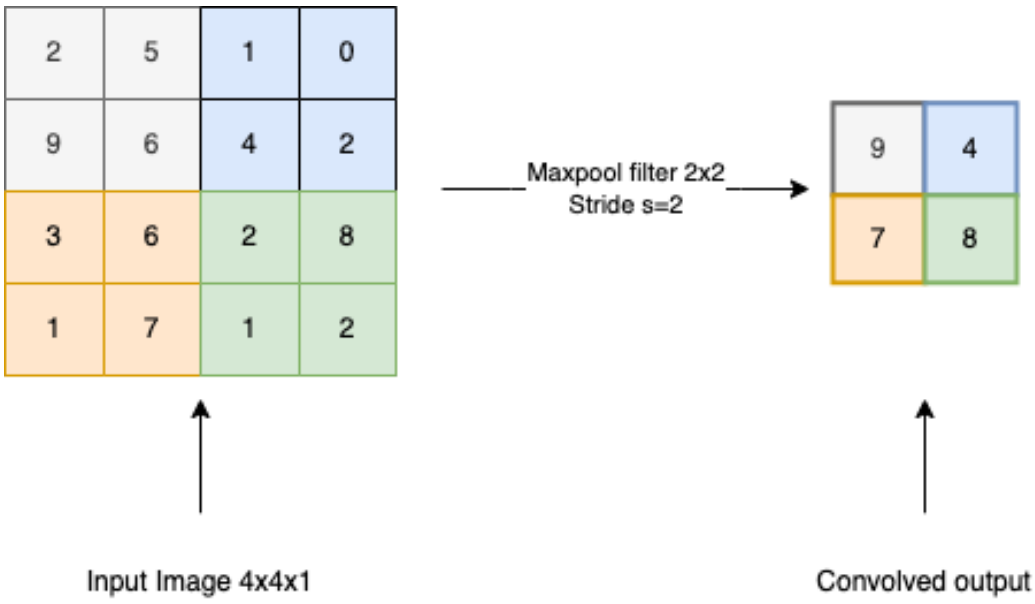


Figure 39: Max-pooling operation with filter size 2x2

Flattening

Flattening converts the last convolution layer output to a one-dimensional array which can then be used to make the actual predictions.

IV.II Visualizing the ConvNet Learning

So far, we have discussed the convolution operations in a ConvNet model that helps understand the features in the input image. We can visualize the outputs from the filters in a hidden layer to find the pictures that maximize a unit's activation. We can start with a CNN with 15 layers that take an image of size 150×150 as input and gives a probability using a single unit sigmoid Dense layer. The summary of the model is as follows:-

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                 Output Shape.              Param #
```

```
=================================================================
 conv2d_3 (Conv2D)              (None, 148, 148, 32)        896
 activation_3 (Activation)      (None, 148, 148, 32)        0
 max_pooling2d_3 (MaxPooling2D) (None, 74, 74, 32)          0
 conv2d_2 (Conv2D)              (None, 72, 72, 64)          18496
 activation_2 (Activation)      (None, 72, 72, 64)          0
 max_pooling2d_2 (MaxPooling2D) (None, 36, 36, 64)          0
 conv2d_1 (Conv2D)              (None, 34, 34, 128)         73856
 activation_1 (Activation)      (None, 34, 34, 128)         0
 max_pooling2d_1 (MaxPooling2D) (None, 17, 17, 128)         0
 conv2d (Conv2D)                (None, 15, 15, 128)         147584
 activation (Activation)        (None, 15, 15, 128)         0
 max_pooling2d (MaxPooling2D)   (None, 7, 7, 128)           0
 flatten (Flatten)              (None, 6272)                0
 dense_1 (Dense)                (None, 512)                 3211776
 dense (Dense)                  (None, 1)                   513
=================================================================
Total params: 3,453,121
Trainable params: 3,453,121
Non-trainable params: 0
_________________________________________________________________
```

We can notice from the model summary how output from each layer changes e.g. the output of the first-layer is $1 \times 148 \times 148 \times 32$. We take an image from Vincent Van Gogh arts to exhibit the working of this model. The original image looks like below:

Figure 40: Original Image - $References$[1]

As a requirement for the input feed to the model, we pre-process the image into a tensor of shape 4D.

```
img_path <- "~/Downloads/vangogh02.jpg"
img <- image_load(img_path, target_size = c(150, 150))

img_tensor <- image_to_array(img)
img_tensor <- array_reshape(img_tensor, c(1, 150, 150, 3))
img_tensor <- img_tensor / 255
dim(img_tensor)
[1]   1 150 150   3
```

To visualize the outputs from the layers and filters, we create a function act_model to get the model output provided the image tensor as input.

```
layer_outputs <- lapply(model\$layers[1:15], function(layer) layer\$output)
act_model <- keras_model(inputs = model\$input, outputs = layer_outputs)
activations <- act_model %>% predict(img_tensor)
```

We can check the dimension for the $first - layer$ and the $tenth - layer$ activation.

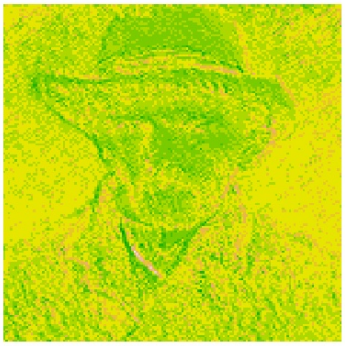
```
first_layer_act <- activations[[1]]
tenth_layer_act <- activations[[10]]
dim(first_layer_act)
[1]   1 148 148  32

dim(tenth_layer_act)
[1]   1  15  15 128
```

We create a $plot\_channel$ function to plot the channel of the respective layer.

```
plot_channel <- function(channel) {
    rotate <- function(x) t(apply(x, 2, rev))
    image(rotate(channel), axes = FALSE, asp = 1, col = terrain.colors(12))
}
```

We use the $plot\_channel$ function to plot the output for channels 1, 16, and 32 from the $first - layer$.

```
par(mfrow = c(1, 3))
plot_channel(first_layer_act[1, , , 1])
plot_channel(first_layer_act[1, , , 16])
plot_channel(first_layer_act[1, , , 32])
```



(a) Channel-1

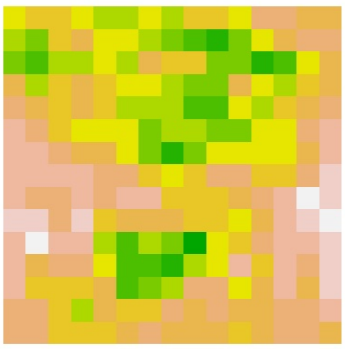
(b) Channel-16

(c) Channel-32

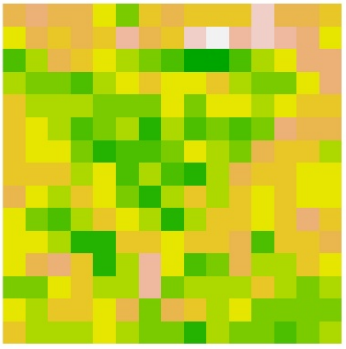
Figure 41: First Layer Activation - Output

We can observe that the first layer activation retains most of the image information and perform exploration on the edges of the image input.
The information gets abstracted as we move higher in the layers stack. For example, we can plot the outputs from the $tenth-layer$ for channels $2, 64, 128$ and observe that the layer represents more global and abstract features.

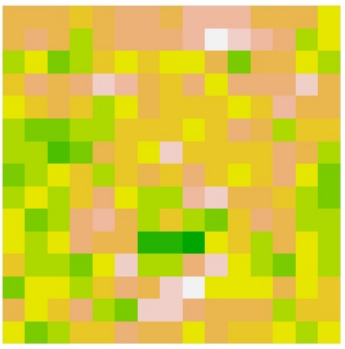
```
par(mfrow = c(1, 3))
plot_channel(tenth_layer_act[1, , , 2])
plot_channel(tenth_layer_act[1, , , 64])
plot_channel(tenth_layer_act[1, , , 128])
```



(a) Channel-2



(b) Channel-64



(c) Channel-128

Figure 42: Tenth Layer Activation - Output

V ConvNet by Example

Keras provides a set of easy to use APIs for model development which runs on top of Tensorflow. Tensorflow can run on different type of hardware e.g. CPU, GPU, and TPU.

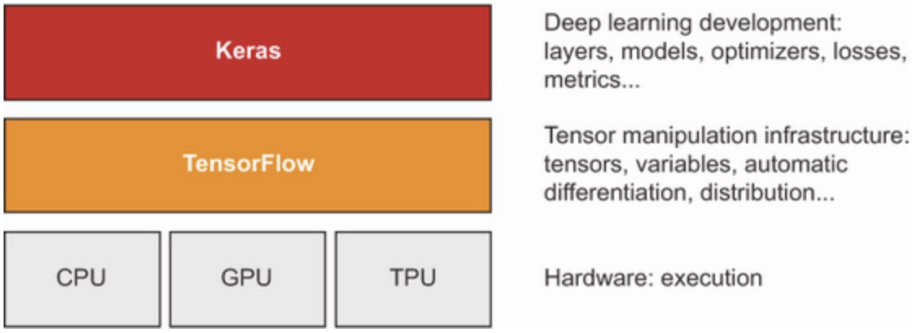


Figure 43: Keras and Tensorflow - $References^2$

Keras Deep Learning API can be used to develop and train model for basic to advanced scenarios. In this section, we will use the *keras* library in RStudio (2022.02.3+492).

- **Data set** - We use a dataset containing Chest X-ray images with opacity(pneumonia) and normal cases. This curated dataset can be accessed on Kaggle, which is already distributed into training, validation and test subsets for machine learning exercise. The information on the data sets and no. of images is in Table 4.

Data set	Normal	Opacity
Training	1082	3110
Validation	267	773
Test	234	390

Table 4: Dataset details

Some sample images from the data set along with the labels are as follow:-

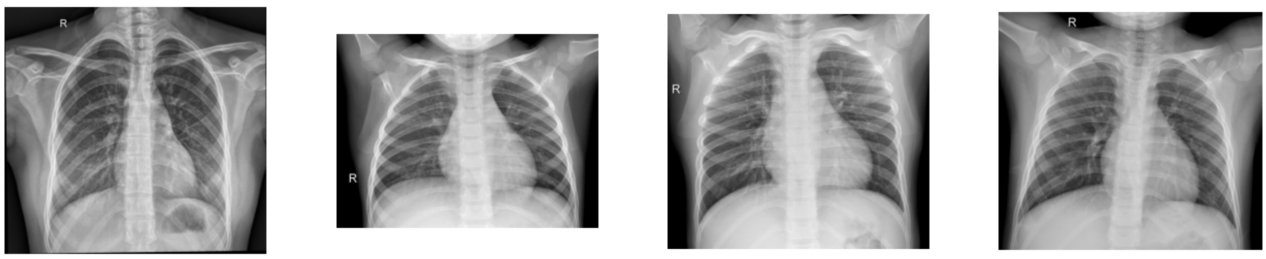


Figure 44: Images - Normal category

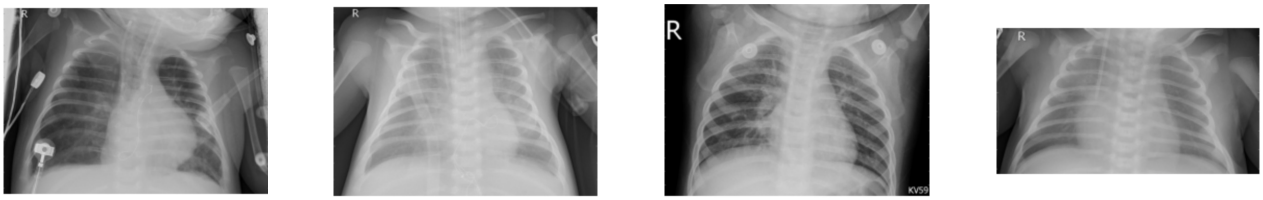


Figure 45: Images - Opacity(Pneumonia) category

These are grayscale images (single channel) with different sizes but in good resolution. As an example, we can see the details for one random image.

```
library(EBImage)
img <- readImage(file.path(train_image_files_path,"normal","IM-0688-0001.jpeg"))
print(img)
```

```
> print(img)
    Image
    colorMode    : Grayscale
    storage.mode : double
    dim          : 1514 1310
    frames.total : 1
    frames.render: 1

    imageData(object)[1:5,1:6]
    [,1] [,2] [,3] [,4] [,5] [,6]
    [1,]    0    0    0    0    0    0
    [2,]    0    0    0    0    0    0
    [3,]    0    0    0    0    0    0
    [4,]    0    0    0    0    0    0
    [5,]    0    0    0    0    0    0
```

- **Data Augmentation** - Image data generators allow altering images with zoom, shear, rotation, etc. to improve the learning process. We use basic Image generators to read the images from the data set directories.

```
train_data_gen = image_data_generator(
    rescale = 1/255
)

valid_data_gen <- image_data_generator(
    rescale = 1/255
)
```

We use the $flow\_image\_files\_path$ function to input images in the model. We'll also standardize the image size to 120×120

```
target_labels <- c("Normal", "Opacity")
img_width <- 120
img_height <- 120
target_size <- c(img_width, img_height)
channels = 1

# Training images
train_image_array_gen <- flow_images_from_directory(
                            train_image_files_path,
                            train_data_gen,
                            target_size = target_size,
                            class_mode = 'binary',
                            classes = target_labels,
                            color_mode = "grayscale",
                            seed = 1994)
# Validation images
valid_image_array_gen <- flow_images_from_directory(
                            valid_image_files_path,
                            valid_data_gen,
                            target_size = target_size,
                            class_mode = 'binary',
                            classes = target_labels,
                            color_mode = "grayscale",
                            seed = 1994)
```

- **Model Training** - This is a binary classification problem with the outcome as normal or opacity health condition. The baseline accuracy is 50%, so a good model should be able to predict with an accuracy higher than the baseline accuracy. We start with a simple CNN model to learn the data representation in the images, and then modify it to observe the impact. We set some parameters required for the model training.

```
batch_size <- 132
num_classes <- 2
epochs <- 12
```

  - $model\_0$– The base ConvNet $model\_0$ has two Conv2D layers, one Flatten layer and two Dense layers. We use $relu$ activation in the model and $sigmoid$ activation at the end layer to address this binary classification problem. The model summary looks as follow:-

```
 Layer (type)                 Output Shape              Param #
=================================================================
 conv2d_79 (Conv2D)           (None, 118, 118, 4)       40
```

```
 activation_59 (Activation)    (None, 118, 118, 4)         0
 conv2d_78 (Conv2D)            (None, 116, 116, 2)         74
 flatten_33 (Flatten)          (None, 26912)               0
 dense_67 (Dense)              (None, 10)                  269130
 activation_58 (Activation)    (None, 10)                  0
 dense_66 (Dense)              (None, 1)                   11
 activation_57 (Activation)    (None, 1)                   0
=================================
Total params: 269,255
Trainable params: 269,255
Non-trainable params: 0
_________________________________
```

We use $binary\_crossentropy$ as the loss function and optimizer $optimizer\_adadelta$ when compiling the model. We also record $accuracy$ in addition to the $loss$ values when training the model

```
# Compile model
model_0 %>% compile(
  loss = 'binary_crossentropy',
  optimizer = optimizer_adadelta(),
  metrics = c('accuracy')
)

# Fit model
history_0 <- model_0 %>% fit(
  train_image_array_gen,
  batch_size = batch_size,
  epochs = epochs,
  validation_data = valid_image_array_gen
)
```

The model achieves an accuracy of 94.52% on the validation data in 12 epochs. The training accuracy is on the rise throughout the model fit, while the validation accuracy stays almost same. A similar patter can be seen in the loss graph, with the validation loss rises towards the end of the model fit. This indicates that the model is over-fitting in the later epochs. The metrics plots for accuracy and loss looks as follow:-

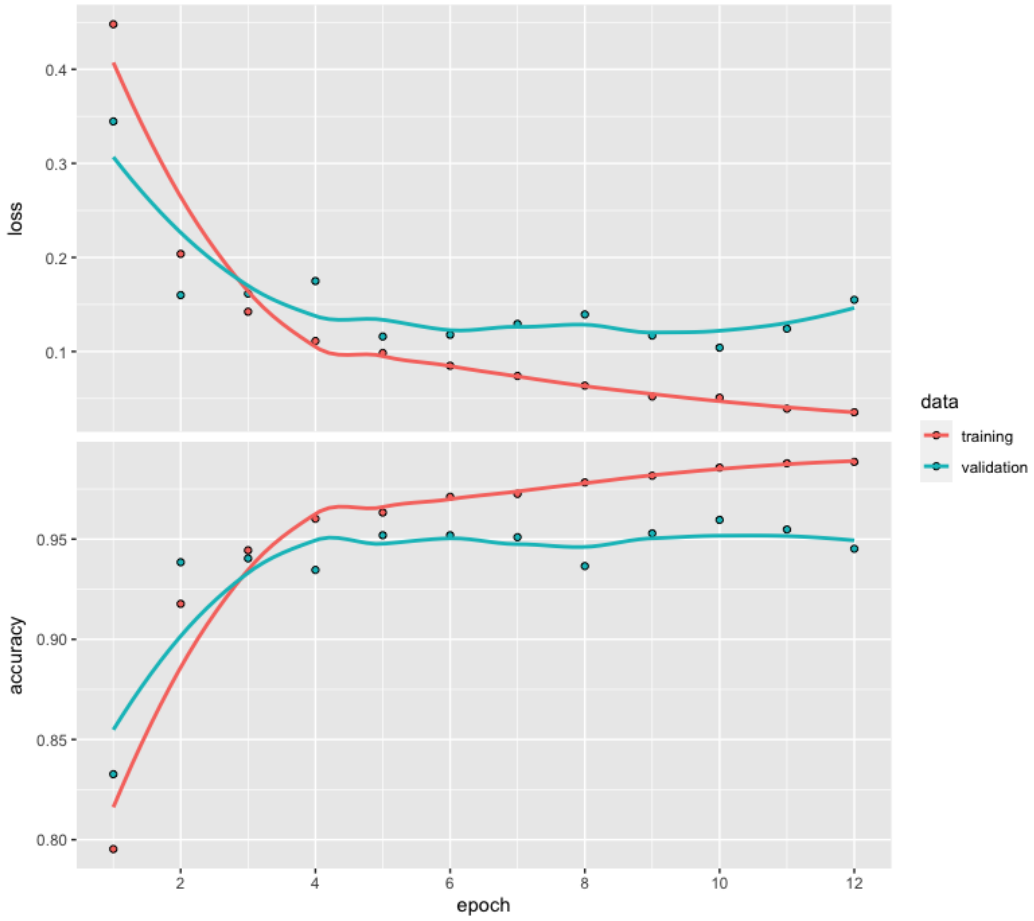


Figure 46: CNN $model\_0$ Fit - Metrics

– $model\_1$ We increase the model complexity by introducing another Conv2D layer and increasing the number of filters. This change almost doubles the number of parameters in the model from $269,255$ to $521,473$. Model summary looks as follow:-

```
 Layer (type)                 Output Shape              Param #
=================================================================
 conv2d_82 (Conv2D)           (None, 118, 118, 16)      160
 activation_63 (Activation)   (None, 118, 118, 16)      0
 conv2d_81 (Conv2D)           (None, 116, 116, 8)       1160
 conv2d_80 (Conv2D)           (None, 114, 114, 4)       292
 flatten_33 (Flatten)         (None, 51984)             0
 dense_67 (Dense)             (None, 10)                519850
 activation_58 (Activation)   (None, 10)                0
 dense_66 (Dense)             (None, 1)                 11
 activation_57 (Activation)   (None, 1)                 0
=================================================================
Total params: 521,473
Trainable params: 521,473
Non-trainable params: 0
_________________________________________________________________
```

The model learning drops with a validation accuracy of 74.33% in the second epoch and it stays there. This seems to be a problem of under-fitting as the model is not able to learn the representations of the data in the images.

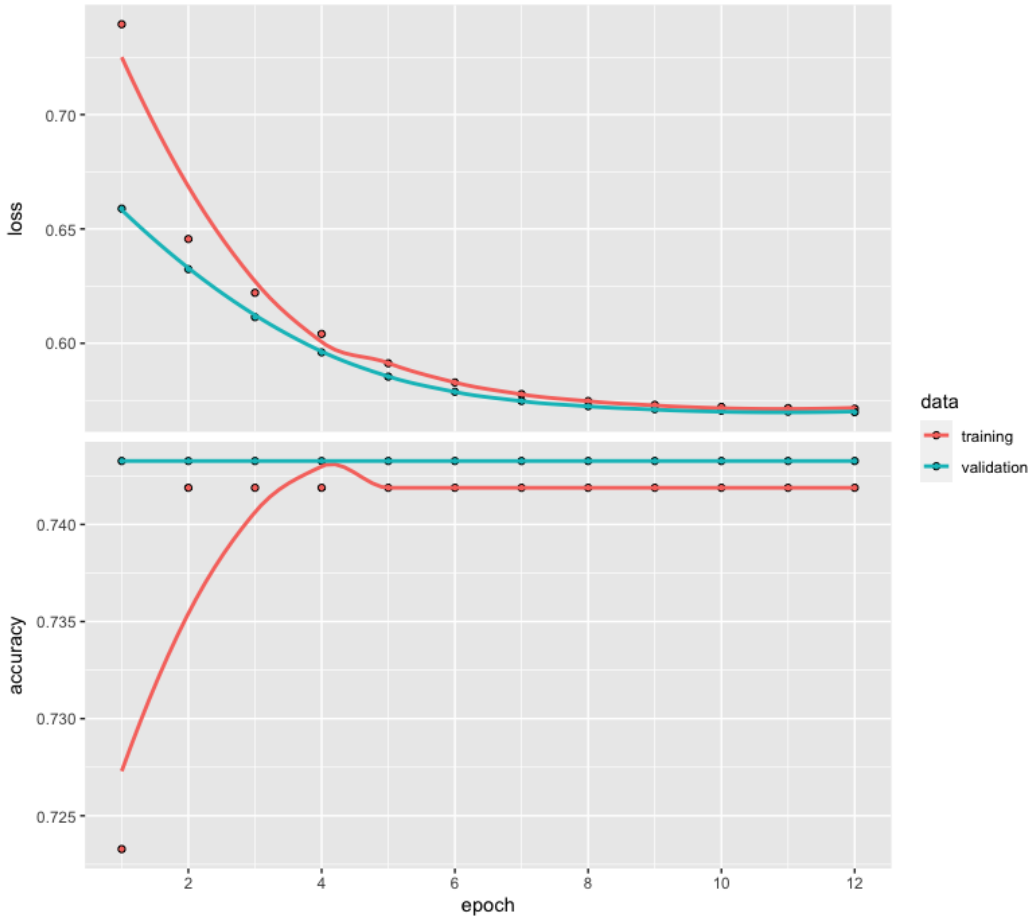


Figure 47: CNN $model\_1$ Fit - Metrics

– $model\_2$– We further increase the filters of Conv2D layers in model to increase the parameters to $10,403,105$. The model summary looks as follow:-

```
 Layer (type)                 Output Shape              Param #
=================================================================
 conv2d_82 (Conv2D)           (None, 118, 118, 32)      320
 activation_63 (Activation)   (None, 118, 118, 32)      0
 conv2d_81 (Conv2D)           (None, 116, 116, 8)       4624
 conv2d_80 (Conv2D)           (None, 114, 114, 4)       1160
 flatten_33 (Flatten)         (None, 103968)             0
 dense_67 (Dense)             (None, 10)                10396900
 activation_58 (Activation)   (None, 10)                0
 dense_66 (Dense)             (None, 1)                 101
 activation_57 (Activation)   (None, 1)                 0
=================================================================
Total params: 10,403,105
```

```
Trainable params: 10,403,105
Non-trainable params: 0
_________________________________
```

The validation accuracy improves to 96.15% in the ninth epoch and the validation loss increases after that.

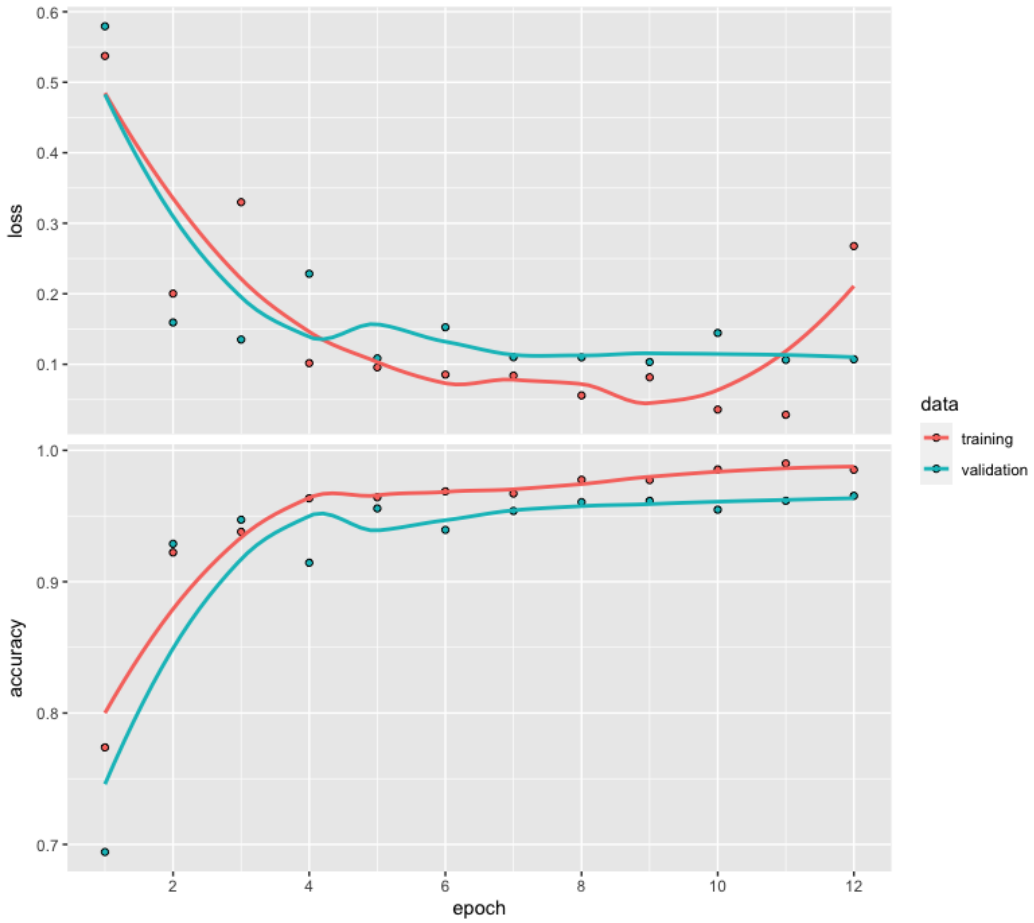


Figure 48: CNN $model\_2$ Fit - Metrics

– $model\_3$– We introduce MaxPooling layers after each Conv2d layer. This reduces the model parameters to $141,505$. The model summary looks as follow:-

```
 Layer (type)                 Output Shape              Param #
=================================================================
 conv2d_82 (Conv2D)           (None, 118, 118, 32)      320
 activation_63 (Activation)   (None, 118, 118, 32)      0
 max_pooling2d_1 (MaxPooling2D) (None, 59, 59, 32).     0
 conv2d_81 (Conv2D)           (None, 57, 57, 16)        4624
 max_pooling2d_2 (MaxPooling2D) (None, 28, 28, 16).     0
 conv2d_80 (Conv2D)           (None, 26, 26, 8)         1160
 max_pooling2d_2 (MaxPooling2D) (None, 13, 13, 8).      0
 flatten_33 (Flatten)         (None, 1352)              0
```

```
 dense_67 (Dense)              (None, 100)                135300
 activation_58 (Activation)    (None, 100)                0
 dense_66 (Dense)              (None, 1)                  101
 activation_57 (Activation)    (None, 1)                  0
=================================
Total params: 141,505
Trainable params: 141,505
Non-trainable params: 0
_________________________________
```

The validation accuracy improves to 96.83% in the ninth epoch with a validation loss at 0.0912.

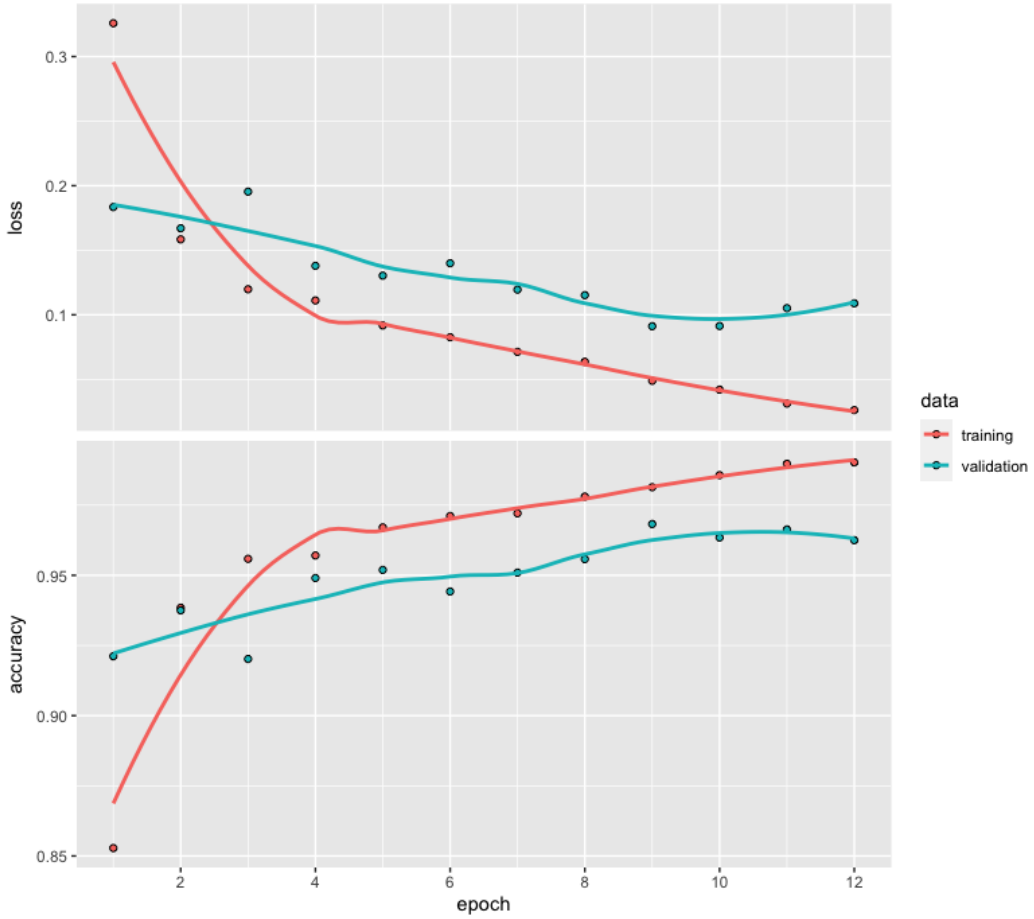


Figure 49: CNN $model\_3$ Fit - Metrics

Observations - We tried four different models with different range of complexities and we can observe that the model complexity does not always help improve the learning. Introducing a new layer or changing the filters updates the number of parameters to be trained. An increase in number of parameters may also slightly increases the epoch run-time.

- Regularization - We choose the best model $model\_2$ from the initial testing and try the regularization techniques to study the effect on the accuracy of the model. Regularization provides

a simplistic approach to control the problem of over-fitting by using some constraints on the weights to choose a smaller value. Some of the standard regularization methods are **BatchNormalization, Dropout, and L2 regularization**. Similar to what we followed in previous phases, we start with a base model and test the impact of regularization by enabling one parameter at a time. The metrics like accuracy, loss, and convergence rate are noted for each regularization change for the base model.

1. **BatchNormalization** - We modify the model by adding *BatchNormalization* layer after each Conv2D layer. This layer would normalize the data coming out from the Conv2D. The model is able to achieve a validation accuracy of 96.15% in ninth epoch and the minimum value of the validation loss at 0.1584.

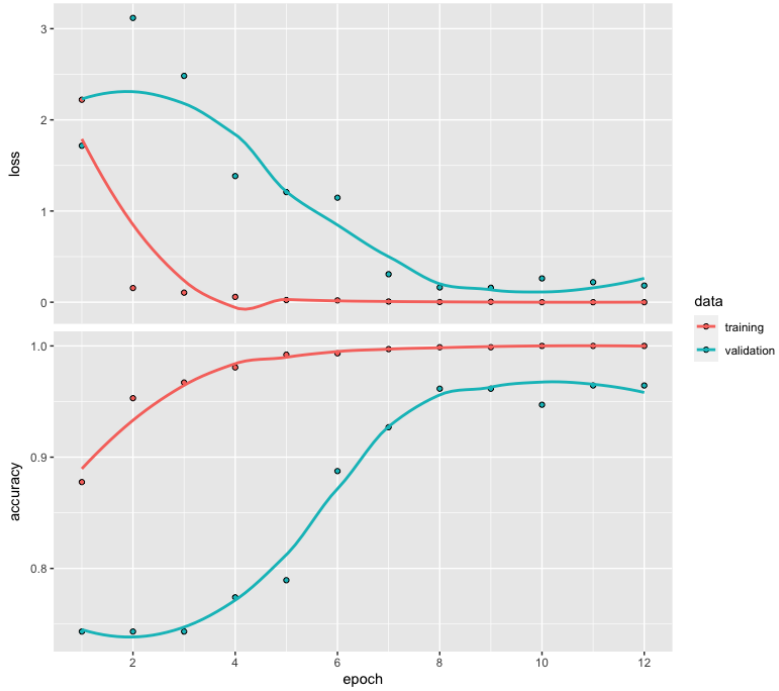


Figure 50: Using BatchNormalization

2. **Adding L2 Regularization** - We further modify the model by adding **L2 regularization** in each Conv2D layer. L2 regularization forces the model to take smaller weights. The validation accuracy of remains approx. same (96.06%), however the validation loss increases to .2788.

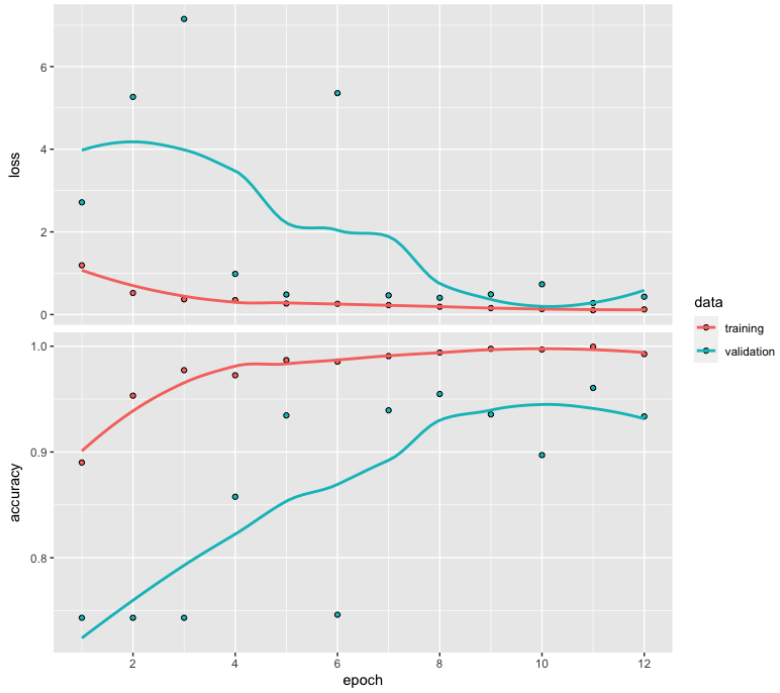


Figure 51: Using L2 Regularization

3. **Adding Dropout** - Finally, we add the **Dropout** layer with a rate=0.5 after the Flatten layer. The model achieves a a validation accuracy of 96.25% and the validation loss changes to 0.1696

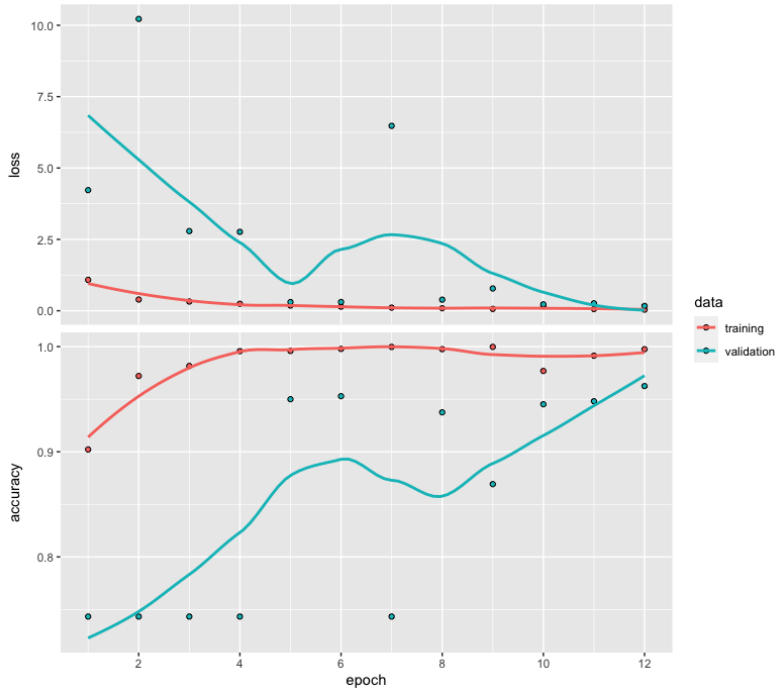


Figure 52: Using Dropout

Observations - We tried out three **Regularization** techniques, namely BatchNormalization,

L2 Regularization, and Dropout. There are the following observations from these tryouts:-

- **Slow convergence** - The final model with all the regularization layers was slow to converge as compared to the initial/base model, which is expected behavior.
- **Impact on Accuracy** - The following table shows the impact on Training and Validation Accuracy as we applied regularization.

Regularization	Training Accuracy	Validation Accuracy
None	97.73%	96.15%
BatchNormalization	99.88%	96.15%
BatchNormalization, L2 Regularization	99.95%	96.06%
BatchNormalization, L2 Regularization, Dropout	99.58%	96.25%

Table 5: Impact of Regularization

As we can notice from the above table, we don't see much improvements on the validation accuracy by applying the regularization.

Evaluate model on Test set - We use the model with all three the regularization methods to predict the labels for the test set and verify the model's performance.

```
scores <- model %>% evaluate(
       x_test, y_test, verbose = 0
       )

# Output metrics
cat('Test loss:', scores[[1]], '\n')
>Test loss: 0.1695605
cat('Test accuracy:', scores[[2]], '\n')
>Test accuracy: 0.9625
```

As we can see from the model evaluation, the ConvNet model delivers an impressive accuracy of 96.25% on the test set.

References

1. Ghatak, A. (2019). Deep learning with R. Springer Singapore.
2. Chollet, F. (2021). Deep learning with python. Manning Publications.
3. Boyd, S. P., amp; Vandenberghe, L. (2021). Unconstrained minimization. In Convex optimization. essay, Cambridge University Press.
4. MNIST_CNN. Keras. (n.d.). from https://keras.rstudio.com/articles/examples/mnist_cnn.html
5. Chest X-ray dataset from https://www.kaggle.com/datasets/paultimothymooney/chest-xray-pneumonia
6. Gradient Descent Algorithm Image from https://medium.com/@rndayala/gradient-descent-algorithm-2553ccc79750
7. Newton Raphson method image from https://www.quora.com/What-is-the-difference-between-Newtons-method-and-secant-method-Explain-your-answer-by-example
8. Local and Global minimum image from https://vitalflux.com/local-global-maxima-minima-explained-examples/