UNIT 2: Training Neural Network

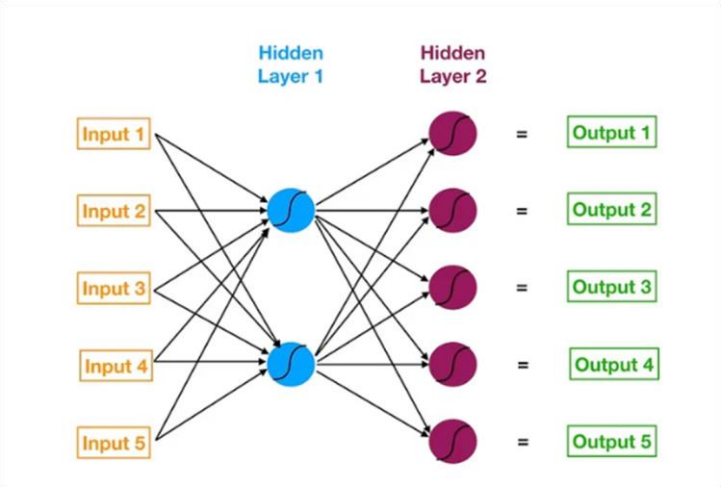
Training Neural Network: Risk minimization, loss function, back propagation, regularization, model selection, and optimization.

Deep Neural Networks: Difficulty of training deep neural networks, Greedy layer wise training.

I. Training Neural Network:

- Like any other model, it's trying to make a good prediction.

- Consider a Neural network and see how it goes from input to output.

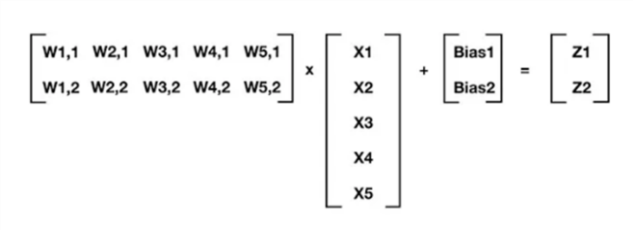


- The first hidden layer consists of two neurons.
- So to connect all five inputs to the neurons in Hidden Layer 1, we need ten connections.
- Note our notation for the weights that live in the connections — $W_{1,1}$ denotes the weight that lives in the connection between Input 1 and Neuron 1 and $W_{1,2}$ denotes the weight in the connection between Input 1 and Neuron 2.
- Now let's calculate the outputs of each neuron in Hidden Layer 1.
- We do so using the following formulas (**W** denotes weight, **In** denotes input).

$$Z1 = W1*In1 + W2*In2 + W3*In3 + W4*In4 + W5*In5 + Bias\_Neuron1$$

$$Neuron\ 1\ Activation = \underline{Sigmoid}(Z1)$$

- We can use matrix math to summarize this calculation — for example, $W_{4,2}$ denotes the weight that lives in the connection between Input 4 and Neuron 2)



- For any layer of a neural network where the prior layer is **m** elements deep and the current layer is **n** elements deep, this generalizes to

$$[W] @ [X] + [Bias] = [Z]$$

- Where [W] is **n by m** matrix of weights, [X] is **m by 1** matrix of activations from the prior layer, [Bias] is **n by 1** matrix of neuron biases, and [Z] is **n by 1** matrix of intermediate outputs.
- Once we have [Z], we can apply the activation function (sigmoid) to each element of [Z] and that gives us our neuron outputs (activations) for the current layer.
- By repeatedly calculating [Z] and applying the activation function to it for each successive layer, we can move from **input to output**.
- The training process of a neural network— **define a cost function and use gradient descent optimization to minimize it**.

2.1 Risk Minimization:

- A major concept in this problem is the empirical risk, which is used to measure the loss or discrepancy L(y, f(x,α)) between the response y and the response f(x,α) provided by the algorithm.
- Consider simple supervised learning classification problem.
- Let us say that we want to classify spam emails, either spam or not spam.
- We denote the domain space with X and the label space with Y, we also need a function for mapping the domain set space to the label set space f: X -> Y.
- we need a model to make predictions: spam or not spam.
- The synonym for **model** is the hypothesis ***h.***
- The hypothesis, is nothing else than a function which takes input from domain ***X*** and produces a label 0 or 1, i.e. a function ***h: X -> Y***.
- In the end, we actually want to find the hypothesis that minimizes our error.
- With this, we come to the term empirical risk minimization.
- The term empirical implies that we minimize our error based on a **sample set *S*** from the domain set **X**.

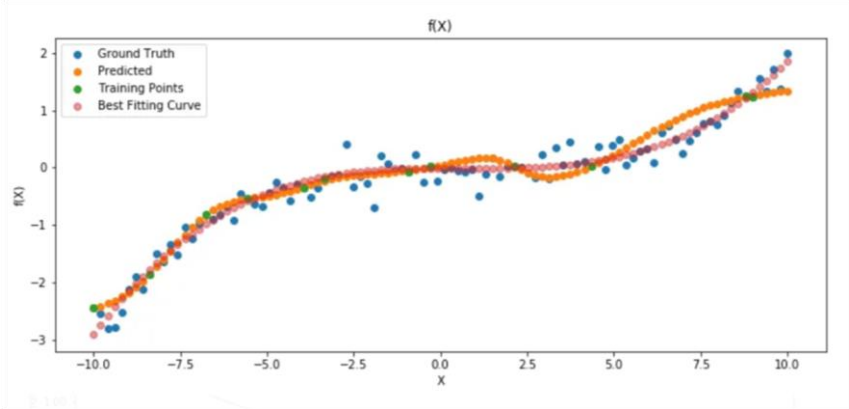
- Looking at it from a probabilistic perspective we say that we sample ***S*** from the domain set **X**, with ***D*** being the distribution over **X**.
- we define the **true error**, which is based on the whole domain **X**

$$L_{\mathcal{D},f}(h) \stackrel{\text{def}}{=} \mathbb{P}_{x\sim\mathcal{D}}[h(x) \neq f(x)] \stackrel{\text{def}}{=} \mathcal{D}(\{x : h(x) \neq f(x)\}).$$

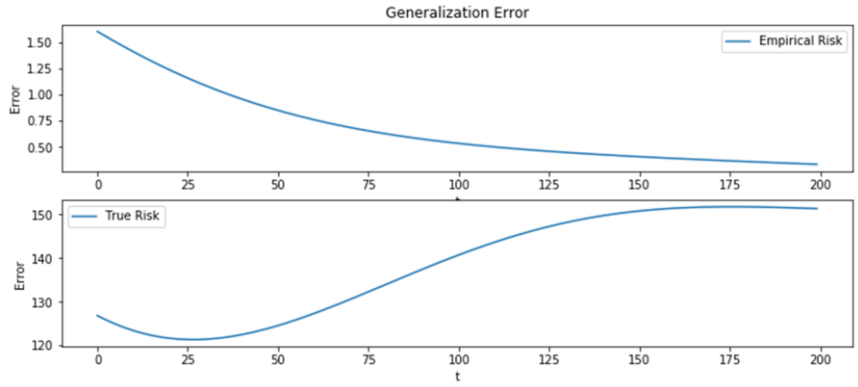
- The error for hypothesis h.we calculate the **error L** based on a **domain distribution D** and a label mapping f.
- The error is equal to the probability of sampling x from d such that the label produced by the hypothesis is different from the actual label mapping.
- Since we only have access to ***S***, a subset of the input domain, we learn based on that sample of training examples.
- We don't have access to the **true error**, but to the **empirical error**:

$$L_S(h) \stackrel{\text{def}}{=} \frac{|\{i \in [m] : h(x_i) \neq y_i\}|}{m}$$

- m denotes the number of training examples.
- we effectively define the empirical error as the fraction of misclassified examples in the set S.
- The empirical error is also sometimes called the generalization error.
- *W*e want to generalize based on ***S***,This error is also called the **risk.**
- Now we can talk about the problem of overfitting.
- Namely, since we have only a subsample of the data it can happen that we minimize the empirical error but actually increase the true error.
- This result can be observed in a simple curve fitting problem.
- Let us imagine that we have some robot that we want to control, we want to map some sensor data X
- We fit a neural network to do this and we obtain the following result:



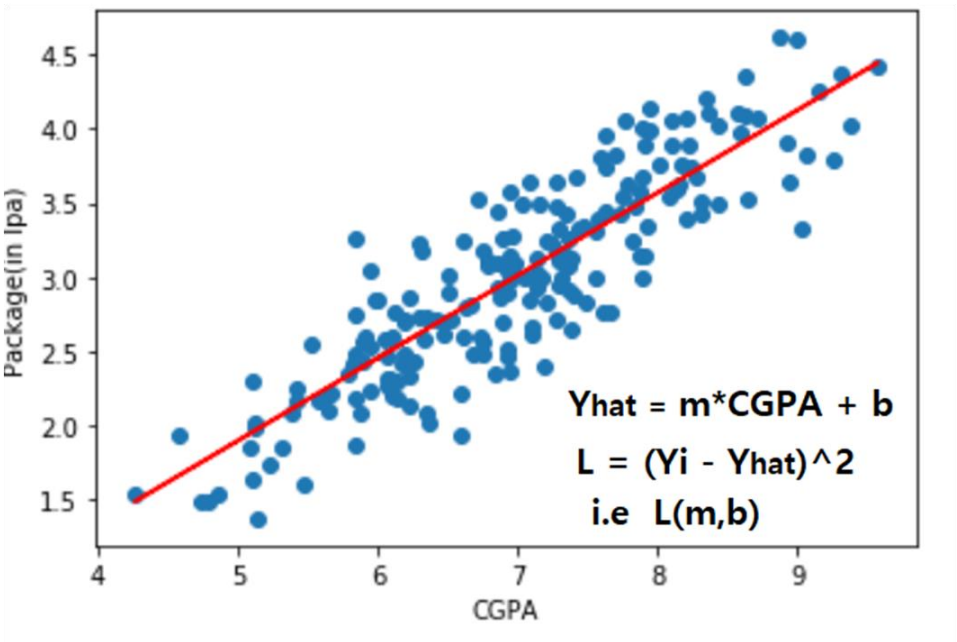
- We can see this generalization error looking at another plot, notice how at **some point the true error starts increasing** while the **empirical error decreases** further.
- This is the consequence of the model overfitting on the training data.



2.2 Loss function:

- A loss function is a mathematical function that quantifies the difference between predicted and actual values in a machine learning model.
- It measures the model's performance and guides the optimization process by providing feedback on how well it fits the data.
- The Loss function is a method of evaluating how well the algorithm is modeling dataset.
- In simple linear regression, prediction is calculated using slope(m) and intercept(b).
- The loss function for this is the (Yi – Yihat)^2 i.e loss function is the function of slope and intercept.

Linear Regression



Loss Function in Deep Learning

1. Regression
    - MSE(Mean Squared Error)
    - MAE(Mean Absolute Error)

2. Classification
    - Binary cross-entropy
    - Categorical cross-entropy

A. Regression Loss

1. Mean Squared Error/Squared loss/ L2 loss

- The Mean Squared Error (MSE) is the simplest and most common loss function.
- To calculate the MSE, you take the difference between the actual value and model prediction, square it, and average it across the whole dataset.

$$MSE = \frac{1}{N}\sum_{i}^{N}(Yi - \hat{Y}i)^2$$

2. Mean Absolute Error/ L1 loss

- The Mean Absolute Error (MAE) is also the simplest loss function.
- To calculate the MAE, you take the difference between the actual value and model prediction and average it across the whole dataset.

$$MAE = \frac{1}{N}\sum_{i=1}^{N}|Yi - \hat{Y}i|$$

B. Classification Loss

1. Binary Cross Entropy/log loss

- It is used in binary classification problems like two classes. for example , whether a person has covid or not.
- Binary cross entropy compares each of the predicted probabilities to the actual class output which can be either 0 or 1.
- It then calculates the score that penalizes the probabilities based on the **distance from the expected value**. That means how close or far from the actual value.

$$\text{Log Loss} = -\frac{1}{N}\sum_{i=1}^{N} y_i \log \hat{y}_i + (1-y_i)\log(1-\hat{y}_i)$$

- yi – actual values
- yihat – Neural Network prediction

2. Categorical Cross Entropy

- Categorical Cross entropy is used for Multiclass classification.

  loss function = -sum up to k($y_j$log$y_j$hat)

  where k is classes

$$\text{Loss} = -\sum_{j=1}^{K} y_j \log(\hat{y}_j)$$

where k is number of classes in the data

- **cost function** = -1/n(sum upto n(sum j to k (y_{ij}logh_{ij}hat))

$$\text{Cost} = \frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{k}[y_{ij}\log(\hat{y}_{ij})]$$

where

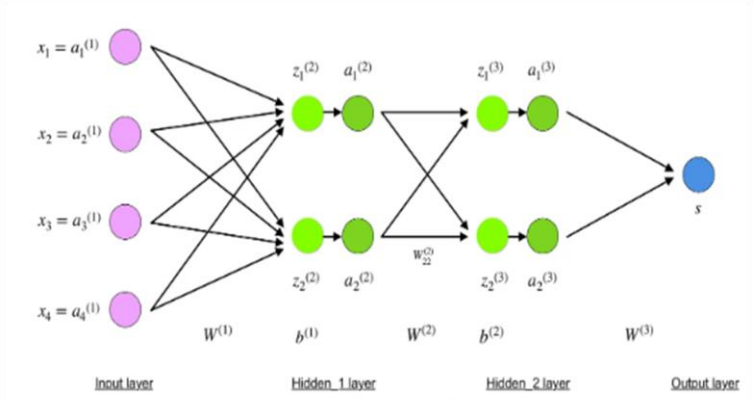
k is classes,

y = actual value

yhat – Neural Network prediction

2.3 Back-Propagation:

- When we use a feedforward neural network to accept an input x and produce an output yˆ, information flows forward through the network.
- The inputs x provide the initial information that then propagates up to the hidden units at each layer and finally produces yˆ.
- This is called *forward propagation*.
- The *back-propagation* algorithm, often simply called *backprop*, allows the information from the cost to then flow backwards through the network, in order to compute the gradient.
- Backpropagation is the learning mechanism that allows the Multilayer Perceptron to iteratively adjust the weights in the network, with the goal of minimizing the cost function.



Simple 4-layer neural network illustration

Input layer

- These can be as simple as scalars or more complex like vectors

$$x_i = a_i^{(1)}, i \in 1,2,3,4$$

Equation for input x_i

- The first set of activations (*a*) are equal to the input values.

Hidden layers

•The final values at the hidden neurons, are computed using ***z^l*** — weighted inputs in layer *l*,

and ***a^l***— activations in layer *l*.

For layer 2 and 3 the equations are:

- $l = 2$

$$z^{(2)} = W^{(1)}x + b^{(1)}$$
$$a^{(2)} = f(z^{(2)})$$

Equations for z^2 and a^2

- $l = 3$

$$z^{(3)} = W^{(2)}a^{(2)} + b^{(2)}$$
$$a^{(3)} = f(z^{(3)})$$

Equations for z^3 and a^3

- Let's pick layer 2 and its parameters as an example.
- W^1 is a weight matrix of shape (n, m) where n is the number of output neurons (neurons in the next layer) and m is the number of input neurons (neurons in the previous layer).
- For us, **n = 2 and m = 4.**

$$W^{(1)} = \begin{bmatrix} W_{11}^{(1)} & W_{12}^{(1)} & W_{13}^{(1)} & W_{14}^{(1)} \\ W_{21}^{(1)} & W_{22}^{(1)} & W_{23}^{(1)} & W_{24}^{(1)} \end{bmatrix}$$

Equation for W^1

- *x* is the input vector of shape *(m, 1)* where *m* is the number of input neurons. For us, *m = 4*.

$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}$$

Equation for x

- b^1 is a bias vector of shape (n , 1) where n is the number of neurons in the current layer. For us, n = 2.

$$b^{(1)} = \begin{bmatrix} b_1^{(1)} \\ b_2^{(1)} \end{bmatrix}$$

Equation for b^1

- Following the equation for z^2, we can use the above definitions of W^1, *x* and b^1 to derive "*Equation for z^2*":

$$z^{(2)} = \begin{bmatrix} W_{11}^{(1)}x_1 + W_{12}^{(1)}x_2 + W_{13}^{(1)}x_3 + W_{14}^{(1)}x_4 \\ W_{21}^{(1)}x_1 + W_{22}^{(1)}x_2 + W_{23}^{(1)}x_3 + W_{24}^{(1)}x_4 \end{bmatrix} + \begin{bmatrix} b_1^{(1)} \\ b_2^{(1)} \end{bmatrix} \qquad z^{(2)} = \begin{bmatrix} z_1^{(2)} \\ z_2^{(2)} \end{bmatrix}$$

Equation for z^2

Output layer

- The final part of a neural network is the output layer which produces the predicated value.

$$s = W^{(3)}a^{(3)}$$

Equation for output s

- The final step in a forward pass is to evaluate the predicted output s against an expected output y.

$$C = cost(s, y)$$

Equation for cost function C

Backpropagation and computing gradients

- Backpropagation aims to minimize the cost function by adjusting network's weights and biases.
- The level of adjustment is determined by the gradients of the cost function with respect to those parameters.
- We first need to revisit some calculus terminology:
- Gradient of a function C(x_1, x_2, …, x_m) in point x is a vector of the partial derivatives of C in x.

$$\frac{\partial C}{\partial x} = [\frac{\partial C}{\partial x_1}, \frac{\partial C}{\partial x_2}, \ldots, \frac{\partial C}{\partial x_m}]$$

Equation for derivative of C in x

- The gradient shows how much the parameter x needs to change (in positive or negative direction) to minimize C.
- Compute those gradients using a technique called chain rule.
- For a single weight *(w_jk)^l*, the gradient is:

$$\frac{\partial C}{\partial w_{jk}^l} = \frac{\partial C}{\partial z_j^l}\frac{\partial z_j^l}{\partial w_{jk}^l} \qquad chain\ rule$$

$$z_j^l = \sum_{k=1}^{m} w_{jk}^l a_k^{l-1} + b_j^l \qquad by\ definition$$

$$m\ -\ number\ of\ neurons\ in\ l-1\ layer$$

$$\frac{\partial z_j^l}{\partial w_{jk}^l} = a_k^{l-1} \qquad by\ differentiation\ (calculating\ derivative)$$

$$\frac{\partial C}{\partial w_{jk}^l} = \frac{\partial C}{\partial z_j^l}a_k^{l-1} \qquad final\ value$$

Equations for derivative of C in a single weight (w_jk)^l

- The gradients allow us to optimize the model's parameters:

$$w := w - \epsilon \frac{\partial C}{\partial w}$$

$$b := b - \epsilon \frac{\partial C}{\partial b}$$

end

- Initial values of *w* and *b* are randomly chosen.
- Epsilon (*e*) is the learning rate. It determines the gradient's influence.
- *w* and *b* are matrix representations of the weights and biases.
- Termination condition is met once the cost function is minimized.

The BACKPROPAGATION Algorithm:

BACKPROPAGATION(*training_examples*, η, n_{in}, n_{out}, n_{hidden})

Each training example is a pair of the form $\langle \vec{x}, \vec{t} \rangle$, *where* $\vec{x}$ *is the vector of network input values, and* $\vec{t}$ *is the vector of target network output values.*

η *is the learning rate (e.g., .05).* n_{in} *is the number of network inputs,* n_{hidden} *the number of units in the hidden layer, and* n_{out} *the number of output units.*

The input from unit i into unit j is denoted x_{ji}, *and the weight from unit i to unit j is denoted* w_{ji}.

- Create a feed-forward network with n_{in} inputs, n_{hidden} hidden units, and n_{out} output units.
- Initialize all network weights to small random numbers (e.g., between −.05 and .05).
- Until the termination condition is met, Do

- For each $\langle \vec{x}, \vec{t} \rangle$ in *training_examples*, Do

  Propagate the input forward through the network:

  1. Input the instance $\vec{x}$ to the network and compute the output o_u of every unit u in the network.

  Propagate the errors backward through the network:

  2. For each network output unit k, calculate its error term δ_k

$$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k) \tag{T4.3}$$

  3. For each hidden unit h, calculate its error term δ_h

$$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in outputs} w_{kh}\delta_k \tag{T4.4}$$

  4. Update each network weight w_{ji}

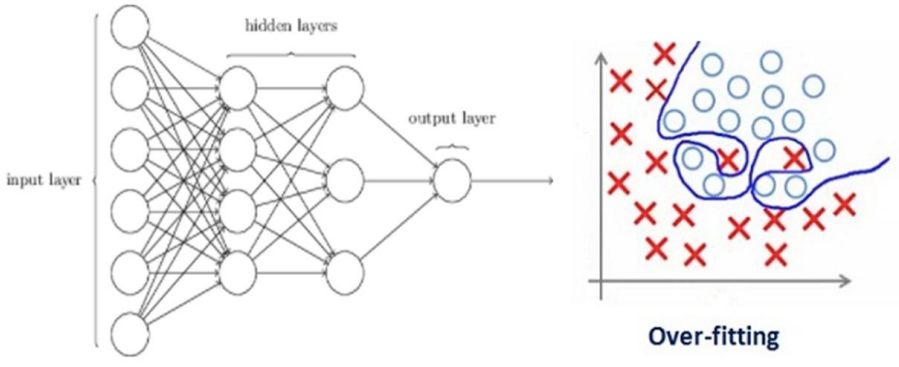
$$w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$$

  where

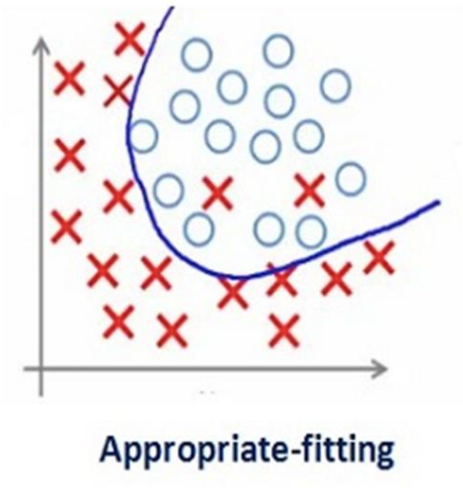
$$\Delta w_{ji} = \eta \, \delta_j \, x_{ji} \tag{T4.5}$$

2.4 Regularization:

- Regularization is a technique used in machine learning and deep learning to prevent overfitting and improve the generalization performance of a model.
- It involves adding a penalty term to the loss function during training.
- This penalty discourages the model from becoming too complex or having large parameter values, which helps in controlling the model's ability to fit noise in the training data.
- Regularization methods include L1 and L2 regularization.
- By applying regularization, models become more robust and better at making accurate predictions on unseen data.
- Let's consider a neural network which is overfitting on the training data.



- large value of the regularization coefficient is not that useful.
- We need to optimize the value of regularization coefficient in order to obtain a well-fitted model



- we'll learn a few different techniques in order to apply regularization in deep learning.

L2 & L1 regularization:

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

Cost function = Loss (say, binary cross entropy) + Regularization term

- Due to the addition of this regularization term, the values of weight matrices decrease because it assumes that a neural network with **smaller weight** matrices leads to simpler models.
- Therefore, it will **reduce overfitting**.
- However, this regularization term differs in L1 and L2.
- In L2, we have:

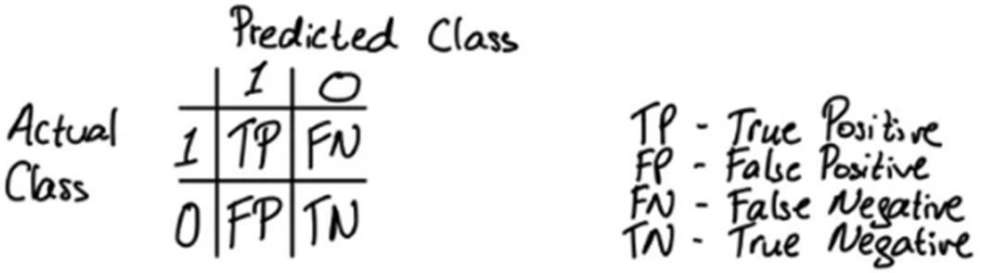
$$Cost\,function = Loss + \frac{\lambda}{2m} * \sum ||w||^2$$

- Here, **lambda** is the regularization parameter.
- It is the hyperparameter whose value is optimized for better results.
- L2 regularization is also known as *weight decay* as it forces the weights to **decay towards zero** (but not exactly zero).
- In L1, we penalize the absolute value of the weights.

$$Cost\,function = Loss + \frac{\lambda}{2m} * \sum \|w\|$$

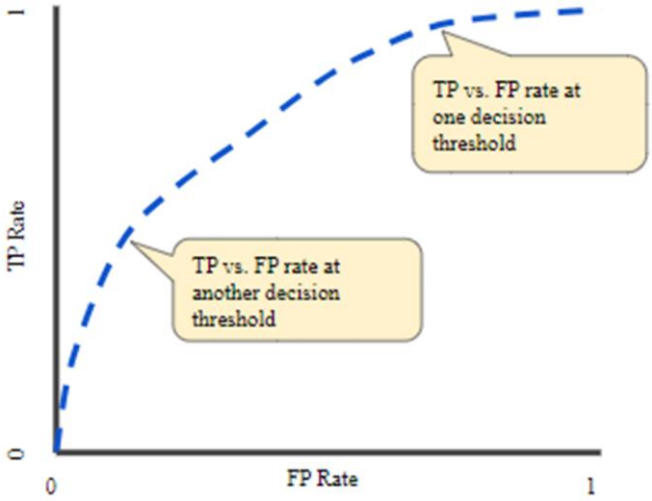
2.5 Model Selection:

- Model Selection and Evaluation is a important procedure in the ML workflow.
- We analyze performance and decide what actions to take in order to improve this model.
- This step is usually the difference between a model that performs well and a model that performs very well.
- When we have a classification task, we will consider the accuracy of our model by its ability to assign an instance to its correct class.
- Consider this on a binary level.



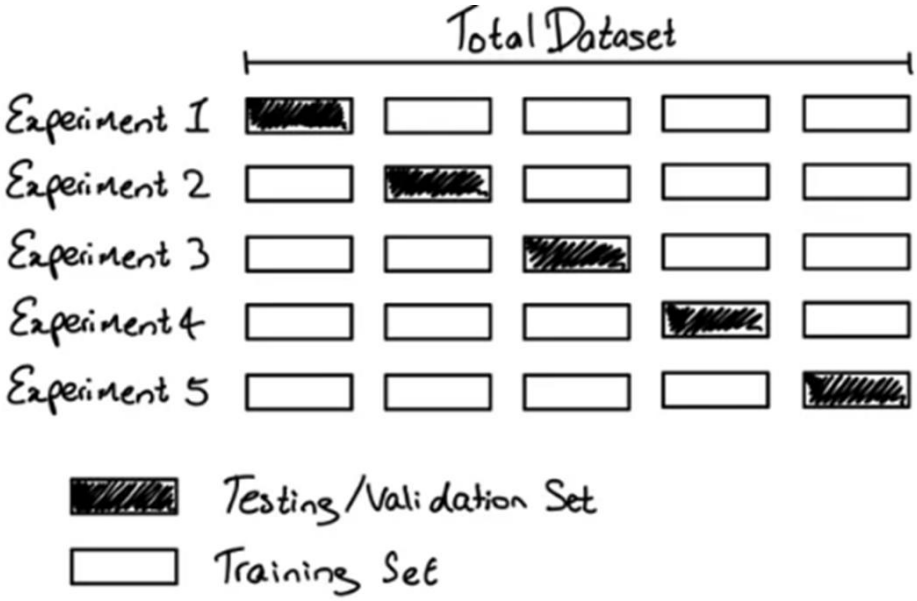
- Our True Positive and True Negative are correct classifications, as we can see in both cases, the actual class and the predicted class are the same.
- The other two classes, the model predicts incorrectly.
- False Positive — when the model predicts 1, but the actual class is 0, also known as **Type I Error**
- False Negative — when the model predicts 0, but the actual class is 1, also known as **Type II Error**
- one more metric that is used in understanding how well a classifier performs is the **Receiver Operator Characteristic** (ROC) curves.
- These plots the **True Positive rate on the y axis** vs the **False Positive rate** on the x axis.
- An ROC curve plots TPR vs. FPR at different classification thresholds.
- **Lowering** the classification threshold classifies more items as **positive**, thus increasing both False Positives and True Positives.

Receiver Operator Characteristic (ROC) curves



Cross Validation:

- **Cross Validation can be considered under the model improvement section.**
- In cross validation, we run our modelling process on different subsets of the data to get multiple measures of model quality.
- Consider the following dataset, split into 5 sections, called folds.

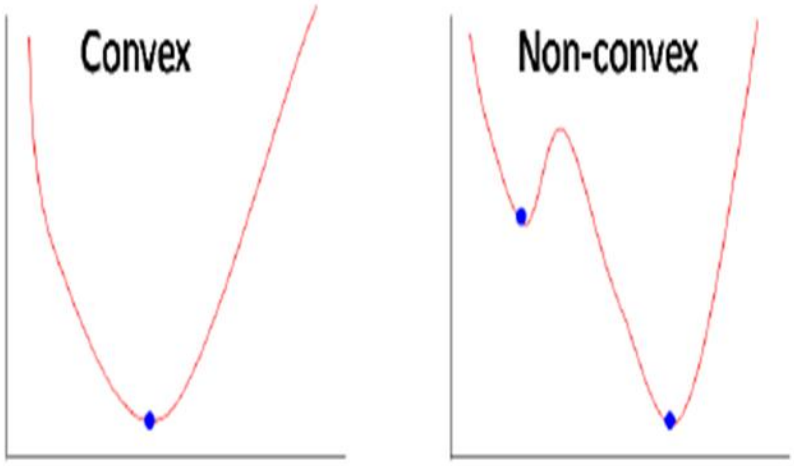


2.6 Optimization:

- Optimizers **update the parameters** of neural networks such as weights and learning rate to minimize the loss function.
- Different instances of Gradient descent based Optimizers are as follows:
- Batch Gradient Descent or Gradient Descent (GD)
- Stochastic Gradient Descent (SGD)
- Mini batch Gradient Descent (MB-GD)

Batch Gradient Descent:

- Gradient descent is an optimization algorithm that's used when training deep learning models.
- It's based on a convex function and updates its parameters iteratively to minimize a given function to its local minimum.



Gradient Descent

$$\Theta_j = \Theta_j - \alpha \frac{\partial}{\partial \Theta_j} J(\Theta_0, \Theta_1)$$

Learning Rate

The notation used in the above Formula is,
α is the learning rate,
J is the cost function, and
Θ is the parameter to be updated.
the gradient represents the partial derivative of J(cost function) with respect to Θ_j

- It is the most basic but most used optimizer that directly uses the **derivative of the loss function** and learning rate to reduce the loss function and tries to reach the global minimum.

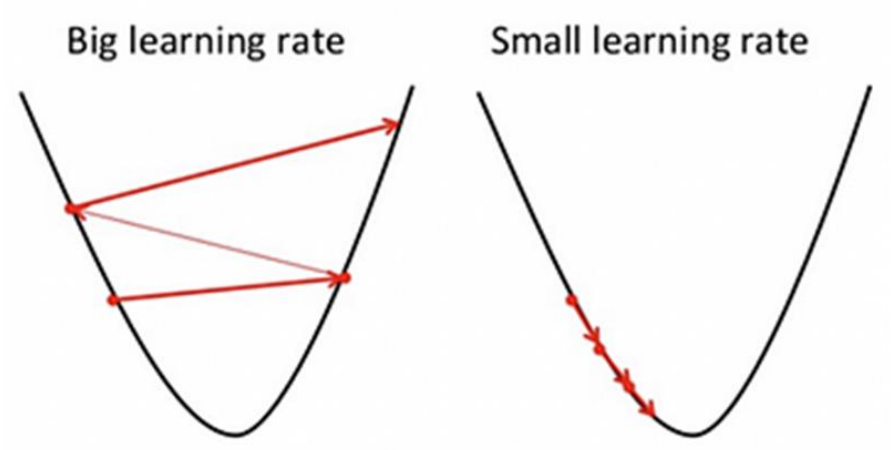
Role of Learning Rate:

- Learning rate represents the size of the steps our optimization algorithm takes to reach the global minima.
- To ensure that the gradient descent algorithm reaches the local minimum we must set the learning rate to an appropriate value, which is neither too low nor too high

Learning Rate:

In deep learning, the learning rate is a hyperparameter that controls how much to adjust the weights of the model with respect to the gradient of the loss function during each iteration of training. It plays a crucial role in the convergence of the model and affects the training speed and model performance.

- Big learning rate: If the learning rate is too high, the model might overshoot the optimal values during weight updates, leading to unstable training or failure to converge.
- Small learning rate: If the learning rate is too low, the model may converge very slowly or get stuck in a suboptimal solution.



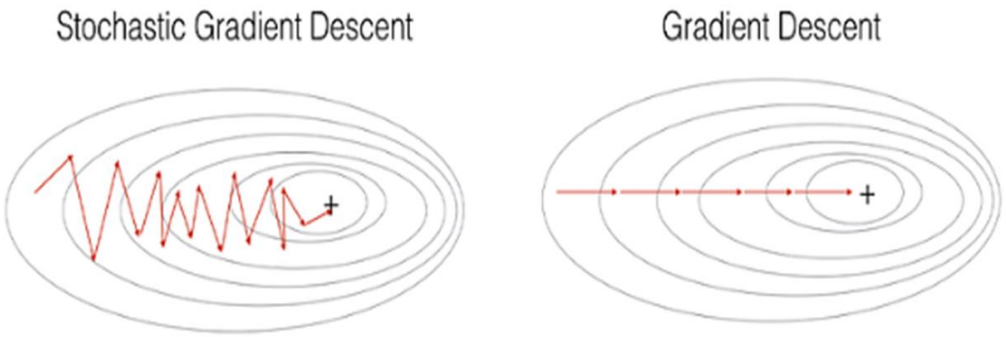
Stochastic Gradient Descent (SGD) and Mini-Batch Gradient Descent are optimization algorithms that use the learning rate, but they are not types of learning rates themselves. Instead, they define how the learning rate is applied during the weight update process.

Here's how learning rate interacts with these methods:

Stochastic Gradient Descent

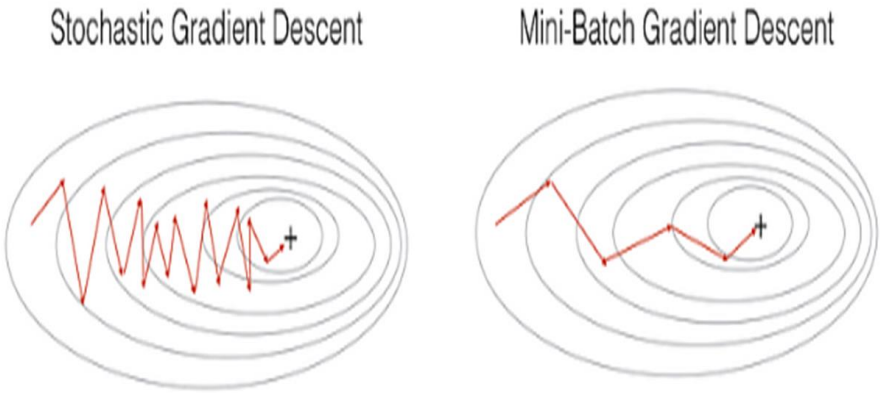
- One of the disadvantages of the Gradient Descent algorithm is that it requires a **lot of memory to load the entire dataset** at a time to compute the derivative of the loss function.
- So, In the SGD algorithm, we compute the derivative by taking one data point at a time i.e, tries to update the model's parameters more frequently.
- Therefore, the model parameters are updated after the computation of loss on each training example.

- So, let's have a dataset that contains 1000 rows, and when we apply SGD it will update the model parameters 1000 times in one complete cycle of a dataset instead of one time as in Gradient Descent.



Mini-Batch Gradient Descent

- To overcome the problem of large time complexity in the case of the SGD algorithm.
- MB-GD algorithm takes a batch of points or subset of points from the dataset to compute derivate**.**



Deep Neural Networks:

A Deep Neural Network (DNN) is a type of artificial neural network with multiple layers between the input and output layers. In the context of deep learning, it refers to any neural network that has more than one hidden layer.

2.7 Difficulty of training Deep Neural Networks:

- When we are training a Neural Network, we use Gradient Descent Algorithm to train the model.
- our goal is to minimize the cost function to achieve the optimal values for the model parameters i.e. to reach global optima.
- And depending on the type of Neural Network, there are hyperparameters ,need to tune to reach the global optimum.

Training deep neural networks (DNNs) is challenging due to several factors that arise as the network's depth and complexity increase. Here are some key difficulties:

1. Vanishing/Exploding Gradients

- **Vanishing gradients** occur when the gradients used to update the weights during backpropagation become extremely small, causing the early layers to learn very slowly or not at all.

- **Exploding gradients** happen when gradients grow exponentially, leading to instability and large updates that make training difficult.

Solution: Techniques like ReLU activation functions, weight initialization methods (e.g., Xavier or He initialization), and normalization techniques like batch normalization help mitigate this issue.

2. Overfitting:

- As networks become deeper, they may have a high capacity to learn the training data too well, memorizing details rather than generalizing to new data. This leads to overfitting, where the network performs well on training data but poorly on unseen data.

Solution: Regularization methods such as dropout, L1/L2 regularization, and data augmentation help reduce overfitting.

3. Computational Complexity:

- Deep networks have millions of parameters, requiring substantial computational resources for training. Training can take days or weeks even on powerful hardware like GPUs or TPUs, especially with large datasets.

Solution: Optimization techniques like mini-batch gradient descent, using GPUs/TPUs for parallel processing, and model compression (e.g., pruning, quantization) can alleviate this issue.

4. Need for Large Datasets:

- DNNs generally require large amounts of labeled data to perform well. Insufficient data can lead to poor generalization, as the network cannot effectively learn meaningful patterns.

Solution: Data augmentation, transfer learning (pretrained models), and synthetic data generation can help when the dataset is small.

5. Hyperparameter Tuning:

- DNNs have many hyperparameters (e.g., learning rate, batch size, number of layers, activation functions) that need to be tuned carefully. Finding the right combination is often a trial-and-error process, requiring significant experimentation.

Solution: Techniques like grid search, random search, and more advanced methods like Bayesian optimization or hyperband can aid in efficient hyperparameter tuning.

6.Weight Symmetry:

- Weight Symmetry is when all the weights in Neural Network have identical values for successive steps of backpropagation.
- This phenomenon occurs when we initialize all the weights & biases with a common value.

- Thus, all the hidden units will have identical activation. This in turn will result in identical derivatives during backpropagation .
- This is problematic because the network will not be able to distinguish between the different hidden units & hence no learning will happen.
- **Solution:** the problem can be solved by random initialization of these parameters.

7. **Slow Progress:**

- When we choose the learning rate that is too small, the gradient descent makes slow progress towards the global optimum thus increasing the computational cost.
- **Solution:** If you plot the training curve, you will observe that the error is decreasing linearly but very slowly. The solution is to increase the learning rate.

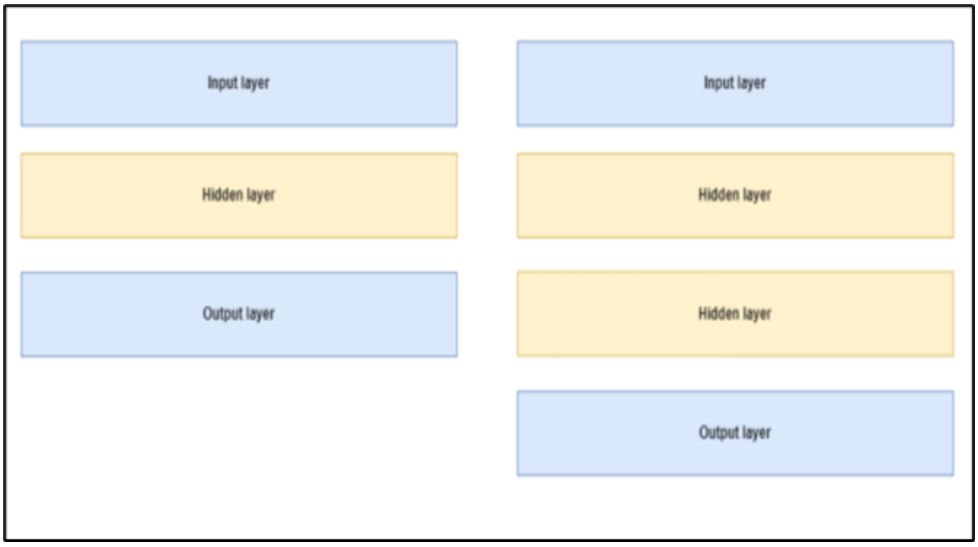
8. **Instability and Oscillations:**

- If we set the learning rate too high, the gradient descent will overshoot.
- Sometimes, the overshoot gets larger with each step & quickly blows up. This phenomenon is known as Instability. In the training curve, you will observe that the cost will suddenly shoot to infinity.
- This, in turn, will cause the weights to diverge & will result in Oscillations over subsequent training epochs.
- **Solution:** The first solution is to tune the learning rate by gradually decreasing it.
  In the case of Oscillations, another solution to compromising on training speed is **Momentum**.

2.8 Greedy layer wise training :

- In the early days of deep learning, an abundance of resources was not available when training a deep learning model.
- In addition, deep learning practitioners suffered from the vanishing gradients problem and the exploding gradients problem.
- Vanishing gradient problem is a phenomenon that occurs during the training of deep neural networks, where the gradients that are used to update the network become extremely small or "vanish" as they are backpropogated from the output layers to the earlier layers.
- These gradients are used to update the weights. If the gradients are large, the multiplication of these gradients will become huge over time.
- This results in the model being unable to learn and its behavior becomes unstable. This problem is called the exploding gradient problem.
- This was an unfortunate combination when one wanted to train a model with increasing depth.
- What depth would be best?
- From what depth would we suffer from vanishing and/or exploding gradients?
- And how can we try to find out without *wasting* a lot of resources?
- **Greedy layer-wise training** of a neural network is one of the answers that was posed for solving this problem.

- By adding a hidden layer every time the model finished training, it becomes possible to find what depth is adequate given your training set.
- The idea behind this strategy is to find an optimum number of layers for training your neural network.
- You start with a simple neural network - an input layer, a hidden layer, and an output layer.
- You train it for a fixed number of epochs - say, 25.
- Then, after training, you **freeze all the layers**, except for the last one.
- In addition, you cut it off the network. At the tail of your cutoff network, you now add a new layer - for example, a densely-connected one.



optimum number of layers

- You then re-add the trained final layer, and you end up with a network that is one layer deeper.
- In addition, because all layers except for the last two are frozen, your progress so far will help you to train the final two better.

The key benefits of pretraining are:

- Simplified training process.
- Facilitates the development of deeper networks.
- Useful as a weight initialization scheme.
- Perhaps lower generalization error.