NEURAL NETWORKS

AND

DEEP LEARNING

-VAIBHAV SINGH PANWAR

What is a neural network?

- are the subset of machine learning and are at the heart of deep learning algorithms.
- the human brain inspires name and structure
- mimicking how biological neurons work and signal to one another.
- ANNs are comprised of node layers, containing
  - *an input layer,*
  - *one or more hidden layers,*
  - *and an output layer*

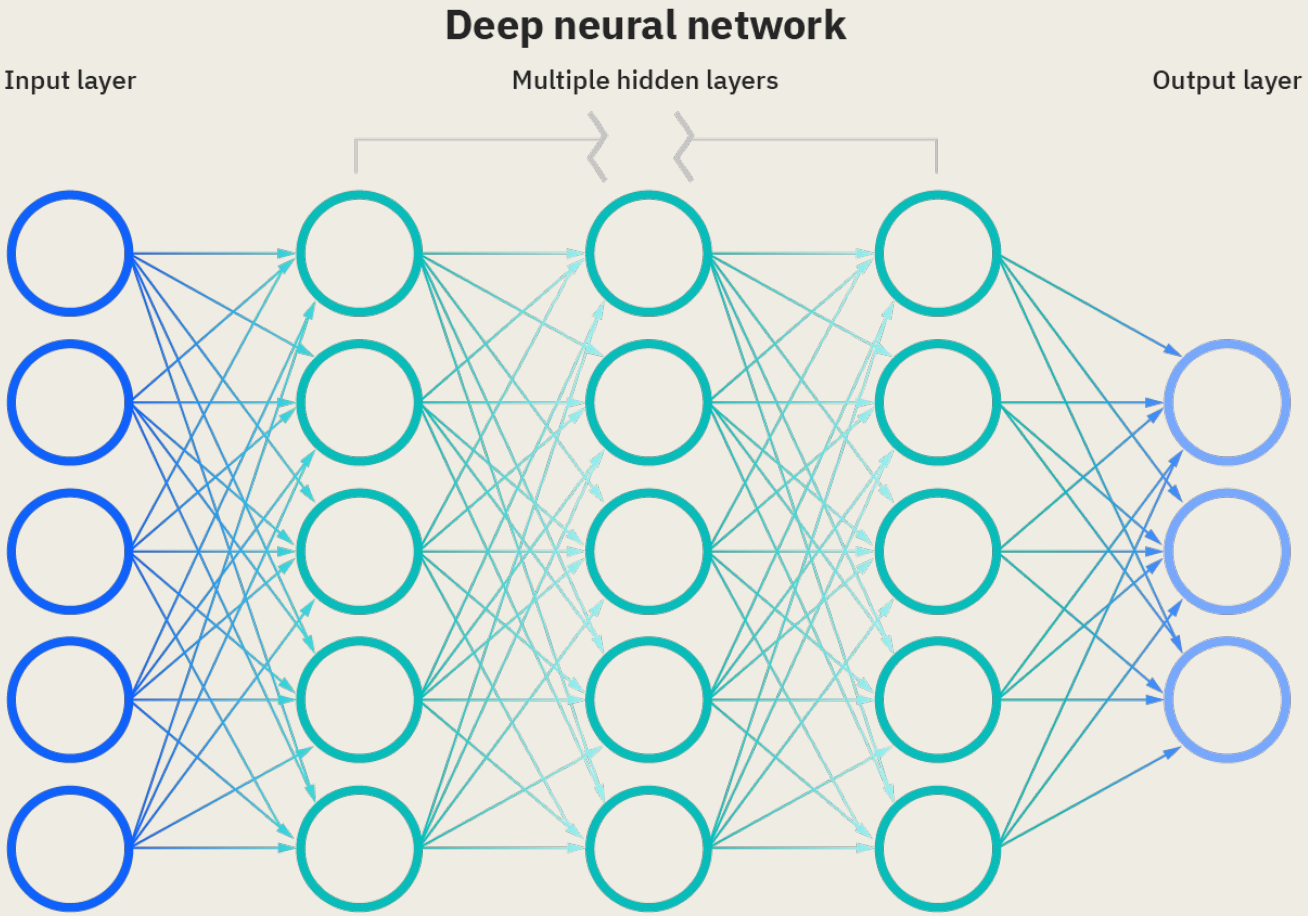
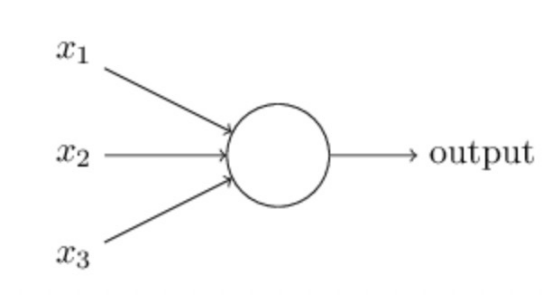


Fig. 1- Deep Neural network
Ref- shorturl.at/jku49

- Each node, or artificial neuron, connects to another and has an associated weight and threshold.
- A node is activated and sends data to the network's next layer if its output rises to a certain threshold value.
  - *Otherwise, no data is transmitted to the network's next layer.*
- Neural networks rely on training data to learn and improve their accuracy over time.

How a perceptron works-

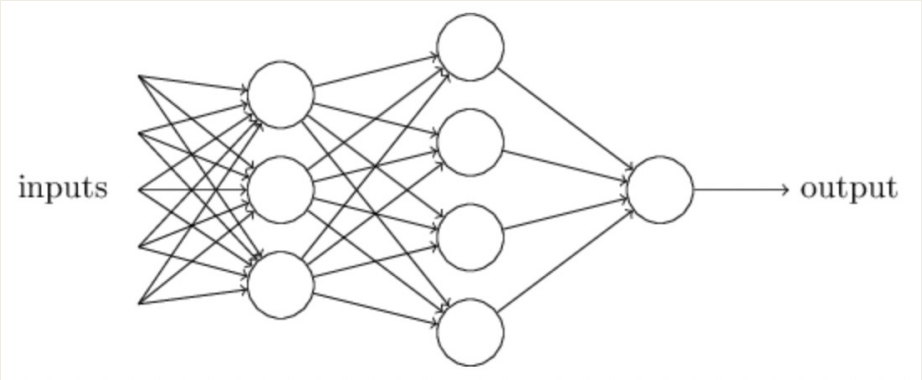
- A perceptron is a type of artificial neuron.
- A perceptron generates a single binary output from several binary inputs, such as x_1, x_2, x_3 …and so on.



- The neuron's output, 0 or 1, is determined by whether the weighted sum $\sum_j w_j x_j$ is less than or greater than some *threshold value.*
  - $$Output = \begin{cases} 0 \; if \;\; \sum_j w_j x_j \;\leq threshold \\ 1 \; if \;\; \sum_j w_j x_j \; > threshold \end{cases} \qquad (1)$$

- Just like the weights, a threshold is a real number which is a parameter of the neuron.

- Different decision-making models can be obtained through a perceptron by adjusting the weights and the threshold.



  - *Fig.3 Neural Network [1]*

- In this network, the first layer of perceptrons, by weighing the inputs, are making three very simple decisions.
- The neurons in the second layer are weighing the results from the first layer,
  - *and so, perceptrons in the second layer can make a decision at a more complex level than perceptrons in the first layer.*
- And even more complex decisions can be made by the perceptron in the third layer.
- In this way, a many-layer network of perceptrons can engage in sophisticated decision-making.

- We can change the $\sum_j w_j x_j$ in the first equation as a dot product,

  $w.x = \sum_j w_j x_j$ *where* w *and* x *are vectors whose components are the weights and inputs, respectively.*

  The second change is to move the threshold to the other side of the inequality and to replace it with what's known as the perceptron's *bias*, b = −threshold

- Using the bias instead of the threshold, the perceptron rule can be rewritten:
  - $$Output = \begin{cases} 0 \;\; if \;\; w.x + b \leq 0 \\ 1 \; if \;\; w.x + b > \; 0 \end{cases}$$

- Bias can be thought of as a measure of how easy it is to get the perceptron to output a 1.
  - If the bias is very large, then it is easy for the perceptron to output a 1.

- Perceptrons can be used to compute the elementary logical functions such as AND, OR, and NAND.

The problem with perceptron-

- For a neural network, we want it to learn weights and biases during training so that it can deliver the required output. This happens due to the *learning algorithm.*
- During the training of the network, we want that a small change in weights or biases should cause only a small change in the output of the network.
- The problem is that this isn't what happens when our network contains perceptrons.
- Sometimes, a tiny adjustment to the weights or bias of a single perceptron in the network might totally reverse that perceptron's output, changing it from 0 to 1, for example.
- By using a new kind of synthetic neuron called a sigmoid neuron, we can solve this issue.

The Sigmoid neuron-

- Sigmoid neurons are similar to perceptrons, but modified.
- The input can be anything between 0 and 1 for a sigmoid neuron.
- The output for the sigmoid neuron is $\sigma(w.x + b)\ where\ \sigma(z) = \frac{1}{1+e^{-z}}$
- If $z = (w.x + b)$ is a large number, then $e^{-z} \approx 0\ and\ so\ \sigma(z) \approx 1$.
  - *And if* $z = (w.x + b)$ *is a very negative, then* $e^{-z} \rightarrow \infty\ and\ so\ \sigma(z) \approx 0$.

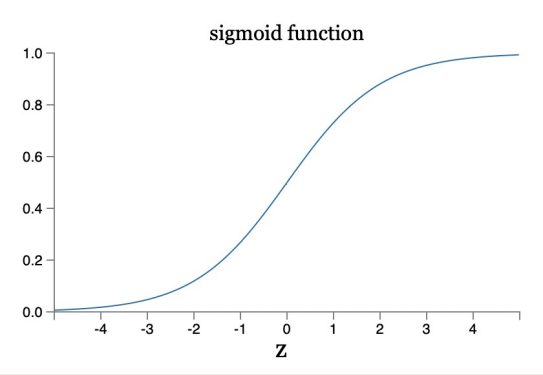


Fig.4 Output graph of sigmoid function. [1]

- small changes in the weights and bias of sigmoid neurons causes only a small change in their output.
- The change in output can be calculated by-
- $\Delta output \approx \sum_j \frac{\partial\, output}{\partial w_j} \Delta w_j + \frac{\partial\, output}{\partial b} \Delta b$
- A sigmoid neurons output can be any real number between 0 and 1.

The architecture of neural networks-

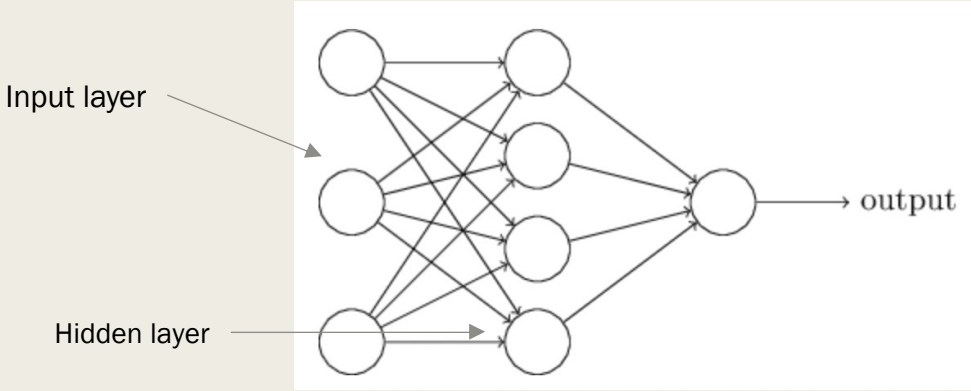


Fig 5. The architecture of NN [1]

- The neural networks where the output from one layer is used as input to the next layer are called the *Feed Forward neural networks.*
- While designing the NNs, we already know the size of input layer and the output layer.
- But, there is no fix rule to design the number of hidden layers, size of hidden layers.
  - *So, these type of parameters are called as hyperparameters of a NN.*

Training a Neural Network-

- We want our NN to find weights and biases so that the output from the network approximates $y(x)$ for all training inputs x.
- To achieve this, we define a cost/objective function and we need to minimize this.
  - $C(w,b) = \frac{1}{2n}\sum_x ||y(x) - a||^2$
- We'll use the gradient descent approach to accomplish it. We are looking for a set of weights and biases that minimise the cost.

Gradient Descent-

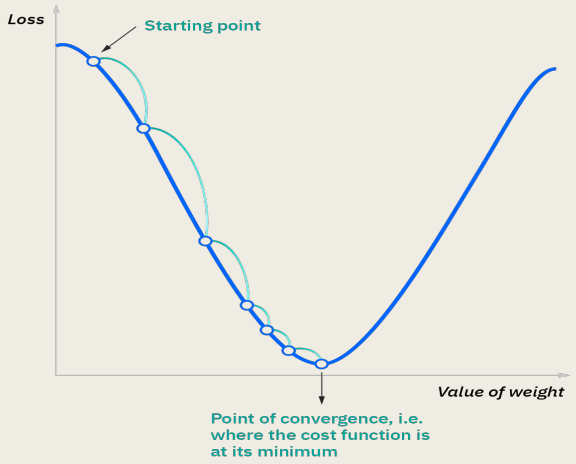


Fig 6. Gradient Descent procedure. [2]

- Gradient descent is an optimization algorithm which is commonly used to train machine learning models and neural networks.

- From that starting point, we will find the derivative (or slope). The slope will then update the parameters.

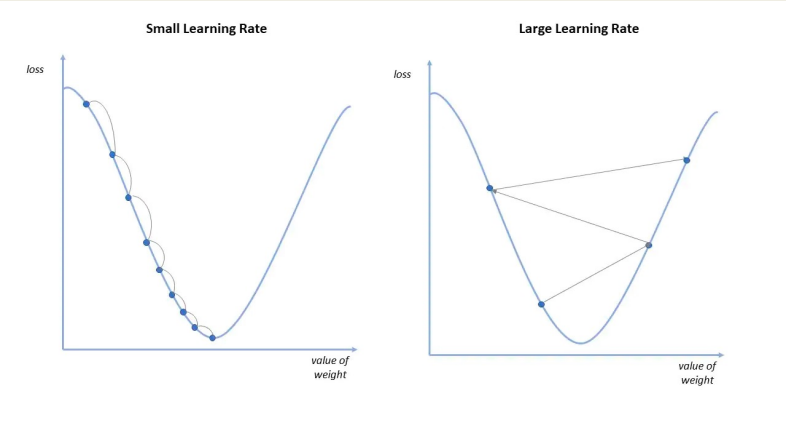


Fig 7. How learning rate effects GD. [2]

- Learning Rate or step size or alpha is the size of steps taken to reach the minimum.

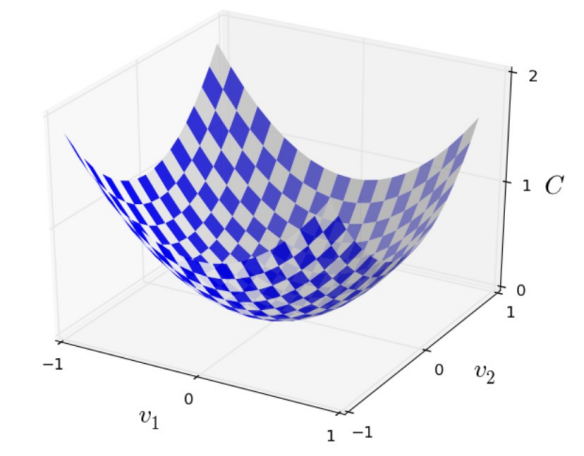


Fig.8 Shape of MSE loss function. [1]

- To reach the minima of the cost function (C), we decrease the value of the cost function at every step. If C is a function of m variables $v_1, v_2 \ldots\ldots v_m$, then C changes as follows-
  - $\Delta C \approx \nabla C . \Delta v$ where the gradient ∇C is the vector $\equiv \left(\frac{\partial C}{\partial v_1} \ldots . \frac{\partial C}{\partial v_m}\right)$
- For the two variable case, we can choose $\Delta v = -\eta \nabla C$.
  - *Where Δv is the step size and η is learning rate.*
- So we get the update rule as-
  - $v \rightarrow v' = v - \eta \nabla C$

Stochastic Gradient Descent in NN-

- As vanilla gradient descent is slower on large data, SGD is introduced.

- Suppose w_k and b_l denote the weights and biases in our neural network, then stochastic gradient descent works by picking out a randomly chosen mini-batch of training inputs, and training with those-
  - $w_k \rightarrow w'_k = w_k - \frac{\eta}{m}\sum_j \frac{\partial C_{X_j}}{\partial w_k}$
  - $b_l \rightarrow b'_l = b_l - \frac{\eta}{m}\sum_j \frac{\partial C_{X_j}}{\partial b_l}$

- where the sums are over all the training examples X_j in the current mini-batch.

The Backpropagation Algorithm-

- Computing the partial derivatives $\frac{\partial C}{\partial w}$ and $\frac{\partial C}{\partial w}$ of the cost function C with respect to any weights or biases in the network is the aim of backpropagation.
- We know that activation ${a^l}_j$ of the jth neuron in the lth layer is related to the activations in the $(l-1)^{th}$ layer by the equation-
  - ${a^l}_j = \sigma(\sum_k {w^l}_{jk} . {a^{l-1}}_k + {b^l}_j)$ *where the sum is over all k neurons in the $(l-1)^{th}$ layer.*
- We can rewrite this in a matrix form as --- $a^l = \sigma(w^l a^{l-1} + b^l)$
- the intermediate quantity $z_l \equiv w^l a^{l-1} + b^l$ is called the weighted inputs to the neurons in the layer *l.*
- z^l has components ${z^l}_j = \sum_k {w^l}_{jk} . {a^{l-1}}_k + {b^l}_j \; for\ the\ jth\ neuron\ in\ lth\ layer.$

The four equations of Backpropagation-

- $\delta^L = \nabla_a C \odot \sigma'(z^L)$
- $\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$
- $\frac{\partial C}{\partial b^l{}_j} = \delta^l{}_j$
- $\frac{\partial C}{\partial w^l{}_{jk}} = a^{l-1}{}_k \delta^l{}_j$

Backpropagation Algo. with SGD-

Given a mini-batch of m training examples, the following algorithm applies a gradient descent learning step based on that mini-batch:

- Step 1- **Input a set of training examples**
- Step 2- **For each training example $\boldsymbol{x}$:** Set the corresponding input activation $a^{x,1}$, and perform the following steps:
  - **Feedforward:** For each $l = 2,3 \ldots L$, compute $z^{x,l} = w^l a^{x,l-1} + b^l$ and $a^{x,l} = \sigma(z^{x,l})$
  - **Output error $\boldsymbol{\delta^{x,l}}$**: Compute the vector $\delta^{x,L} = \nabla_a C_x \odot \sigma'(z^{x,L})$.
  - **Backpropagate the error:** For each $l = L-1, L-2, .., 2$ compute- $\delta^{x,l} = ((w^{l+1})^T \delta^{x,l+1}) \odot \sigma'(z^{x,l})$
- Step 3- **Gradient descent-** For each $l = L, L-1, \ldots 2$ update the weights and biases according to the rule-
  - $w^l \rightarrow w^l - \frac{\eta}{m} \sum_x \delta^{x,l} (a^{x,l-1})^T$ and. $b^l \rightarrow b^l - \frac{\eta}{m} \sum_x \delta^{x,l}$

Using Neural Nets to Recognize handwritten digits-

- Considering the image below –

We can easily read it as 504192.

- But for a simple computer program to understand this digit, is a very hard task.
- Neural networks approach this problem by taking a large dataset of handwritten digits, called a training dataset and developing a system out of it.
- Now, we will make a neural network comprising of three layers to recognize handwritten digits.
- We are taking the MNIST dataset to train our neural network which contains 60000 training examples of images of handwritten digits in greyscale and 10000 testing examples.

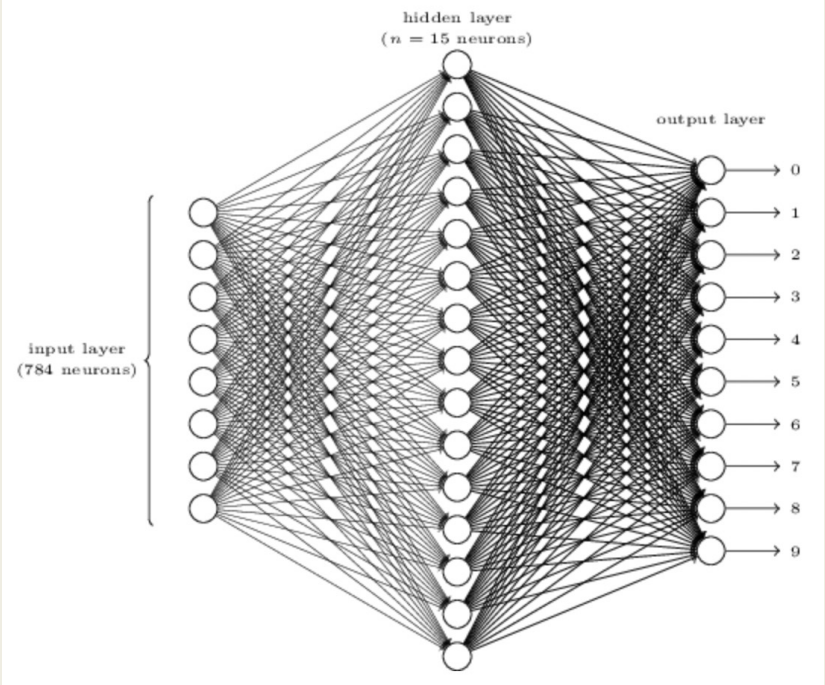


Fig.9 Neural network for handwritten digit recognition. [1]

- As the images in the dataset are of 28*28 pixels, so we will have 784 neurons in the input layer.
- We need to recognize the digits from 0 to 9, so 10 neurons are in the output layer.
- Here, we are using Quadratic cost as the loss function.
- Basic code in Colab.

The Problem of Learning Slowdown-

- We know, that neurons learn by changing the weight and bias at a rate determined by the partial derivatives of the cost function- $\frac{\partial C}{\partial w}$, and $\frac{\partial C}{\partial b}$.
- If we are using the quadric cost as the loss function, then, for a single neuron,the cost can be written as- $C = \frac{(y-a)^2}{2}$ where a is the neuron's output, y is desired o/p.
- Using the chain rule to differentiate with respect to the weight and bias we get-

$$\frac{\partial C}{\partial w} = (a - y)\sigma'(z)x \text{ and } \frac{\partial C}{\partial b} = (a - y)\sigma'(z)$$

- To understand the behaviour of these expressions, we look at the shape of sigmoid function.

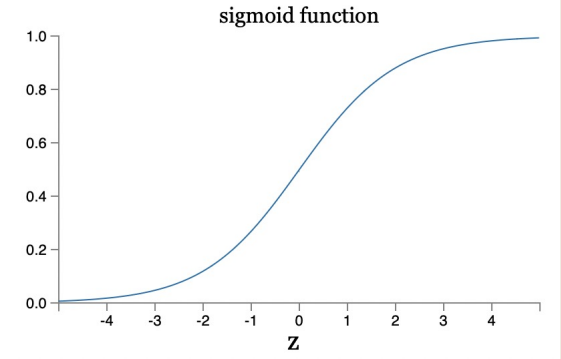


Fig 10. The sigmoid function. [1]

- when the neuron's output is close to 1, the curve gets very flat, and so $\sigma'(z)$ gets very small.
- And so, the $\frac{\partial C}{\partial w}, \text{and} \frac{\partial C}{\partial b}$ too gets very small.
- This is the origin of the learning slowdown.
- How can we address the learning slowdown?

- We discovered that the origin of learning slowdown was the term $\sigma'(z)$ in prev. eqns.
- Is it possible to choose a cost function so that the term $\sigma'(z)$ disappear?
- If yes, then, the cost $C = C_x$ for a single training example would be-
  - $\frac{\partial C}{\partial w_j} = x_j(a - y)$.....(1) and $\frac{\partial C}{\partial b} = (a - y)$ ....(2)
- If any cost function exists, then it would mean that greater the initial error, the faster the neuron learns. Hence, the problem of learning slowdown will be eliminated.
- Now, from the chain rule, we know that $\frac{\partial C}{\partial b} = \frac{\partial C}{\partial a}\sigma'(z)$.
- And this becomes $\frac{\partial C}{\partial b} = \frac{\partial C}{\partial a}a(1 - a)\ because\ \sigma'(z) = \sigma(z)\big(1 - \sigma(z)\big) = a(1 - a)$.
- Comparing this to eqn. 2, we have- $\frac{\partial C}{\partial a} = \frac{a-y}{a(1-y)}$

- If we integrate the result of prev. eqn with respect to a, then we get-

$$C = -[y \ln a + (1 - y) \ln(1 - y)] + constant$$

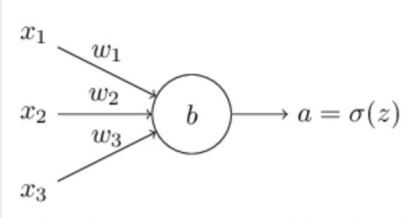
- This is the cost from a single training example. To get the full cost function, we average over all the training examples-

$$C = -\frac{1}{n}\sum_x [y \ln a + (1 - y) \ln(1 - y)] + constant$$ *where the constant here is the average of the individual constants for each training example.*

- The above is called the Cross-Entropy Loss function.
- Cross-entropy comes from the field of information theory.
- Cross-entropy is a measure of surprise.
- We get low surprise if the output is what we expect, and high surprise if the output is unexpected

The Cross-Entropy loss function-

- The cross-entropy function for a single neuron is defined as-



$C = -\frac{1}{n}\sum_x [y \ \ln a + (1-y)\ln(1-a)]$ where n is the total number of items of training data, the sum is over all training inputs, x, and y is the corresponding desired output.

- How does this solve the problem of Learning slowdown?
- To see this, let's compute the partial derivative of the cross-entropy cost with respect to the weights. We substitute $\mathrm{a} = \sigma(z)$ into the above equation and apply the chain rule twice, obtaining:

$$\frac{\partial C}{\partial w_j} = -\frac{1}{n}\sum_x \left(\frac{y}{\sigma(z)} - \frac{(1-y)}{1-\sigma(z)}\right)\frac{\partial \sigma}{\partial w_j}$$

$$= -\frac{1}{n}\sum_x \left(\frac{y}{\sigma(z)} - \frac{(1-y)}{1-\sigma(z)}\right)\sigma'(z)x_j$$

- Putting everything over a common denominator and simplifying, this becomes:
  - $\frac{\partial C}{\partial w_j} = \frac{1}{n}\sum_x \frac{\sigma'(z)x_j}{\sigma(z)(1-\sigma(z))}(\sigma(z) - y)$
- We see that the $\sigma'(z)$ and $\sigma(z)(1 - \sigma(z)$ terms cancel in the equation just above, and it simplifies to become:

$$\frac{\partial C}{\partial w_j} = \frac{1}{n}\sum_x x_j(\sigma(z) - y)$$

- The above expression tells us that rate of learning the weights is controlled by $\sigma(z) - y$ i.e , by the error in the output. The larger the error, the faster the neuron will learn.
- In particular, suppose $y = y_1, y_2, ..$are the desired values at the output neurons, i.e., the neurons in the final layer, $a^L{}_1, a^L{}_2, ..$are the actual output values. Then we define the cross-entropy by-

$$C = -\frac{1}{n}\sum_x \sum_j [y_j \ln a^L{}_j + (1 - y_j)\ln(1 - a^L{}_j)]$$

SoftMax Activation Function-

- **Softmax** is a mathematical function that converts a vector of numbers into a vector of probabilities.
- softmax function is used to normalize the outputs, converting them from weighted sum values into probabilities that sum to one.

$$\sum_j a^L{}_j = \frac{\sum_j e^{z^L{}_j}}{\sum_{k=1}^{k=n} e^{z^L{}_k}}$$

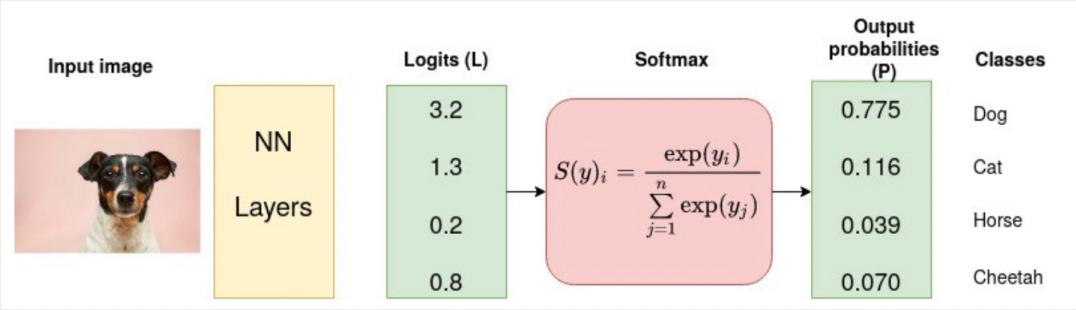


Fig. 1. The output of SoftMax function. [3]

Overfitting and Underfitting-

- Overfitting means that the neural network performs very well on training data, but fails as soon it sees some new data from the problem domain.
- Training accuracy is very high in overfitting, but the validation accuracy is too low.
- Underfitting, means that the model performs poorly on both datasets.

Neural Network Complexity-

- We can consider a neural network as a function, that performs a mathematical mapping from input **x** to output **y**. A mathematical function can take the form of a polynomial of a certain degree *n*.

$$f(\theta) = a_0 + a_1\theta + a_2\theta^2 + a_3\theta^3 + \ldots\ldots = \sum_{n=0}^{n=N} a_n\theta^n$$

- A polynomial function with a higher degree is considered to have more complexity than a polynomial function of a lesser degree.
- Bias- Bias is the error rate of the training data.
  - *High error rate* $\longrightarrow$ *High Bias*
- Variance- The error rate of the testing data is called variance.
  - *High error rate* $\longrightarrow$ *High Variance*
- What is the consequence of having much or less complexity?

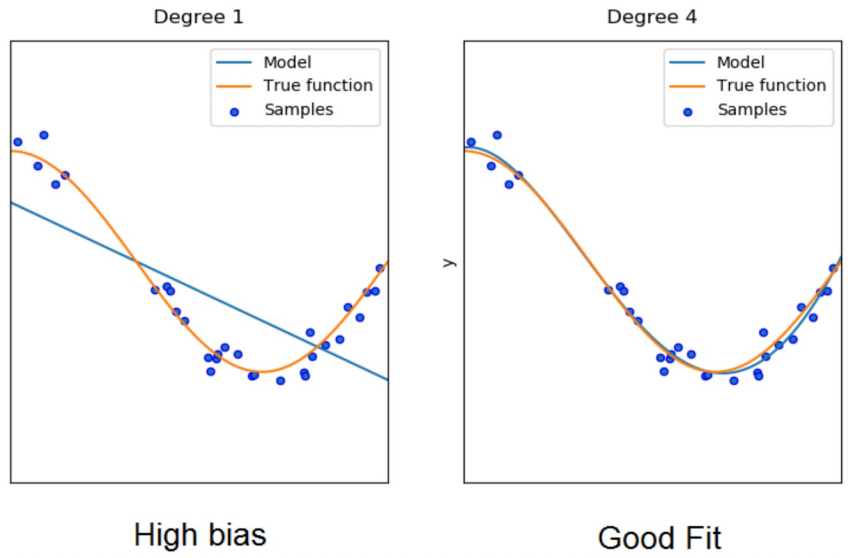


Fig. 2 Underfitting [4]

- **Reasons for Underfitting:**
  - High bias and low variance
  - The size of the training dataset used is not enough.
  - The model is too simple.
  - Training data is not cleaned and also contains noise in it.
- **Techniques to reduce underfitting:**
  - Increase model complexity
  - Increase the number of features
  - Remove noise from the data.
  - Increase the number of epochs or increase the duration of training.

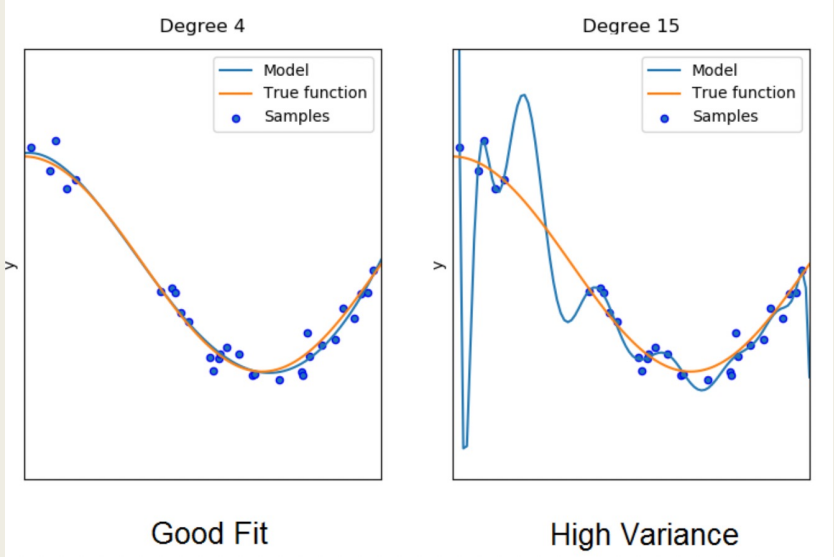


Fig.3 Overfitting [4]

- **Reasons for Overfitting:**
  - High variance and low bias
  - The model is too complex
  - The size of the training data

- **Techniques to reduce Overfitting:**
  - Increase training data.
  - Reduce model complexity.
  - Early stopping during the training phase
  - Ridge Regularization and Lasso Regularization
  - Using dropout for neural networks to prevent overfitting.

Variance Bias Tradeoff-

- This tradeoff means that the rising complexity of the model causes a lower bias error on the one side but causes a higher variance error on the other.

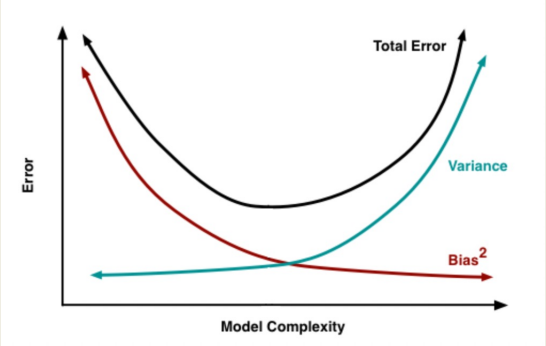


Fig.3 Graph of bias and variance v/s model complexicity [4]

- The overall error of a neural network has a minimum for a certain complexity.
- We need to find a good balance between bias and variance such that it minimizes the total error, and hence the problem of underfitting and overfitting can be prevented.

Regularization-

- Regularization refers to a set of different techniques that lower the complexity of a neural network model during training, and thus prevent the overfitting.
- There are three very popular and efficient regularization techniques called *L1*, *L2*, and dropout.

L2 regularization-

- commonly known as weight decay or Ride Regression.
- adding an extra term to the cost function, called the *regularization term.*
- So, the L2 regularised cross-entropy loss would look like-

$$C = -\frac{1}{n}\sum_{x}\sum_{j}[y_j \ln a_j^L + (1 - y_j)\ln(1 - a^L{}_j)] + \frac{\lambda}{2n}\sum_{w}||w||_2{}^2$$

- λ>0 is the regularization parameter.
- effect of regularization is to make it so the network prefers to learn small weights.

L1 Regularization-

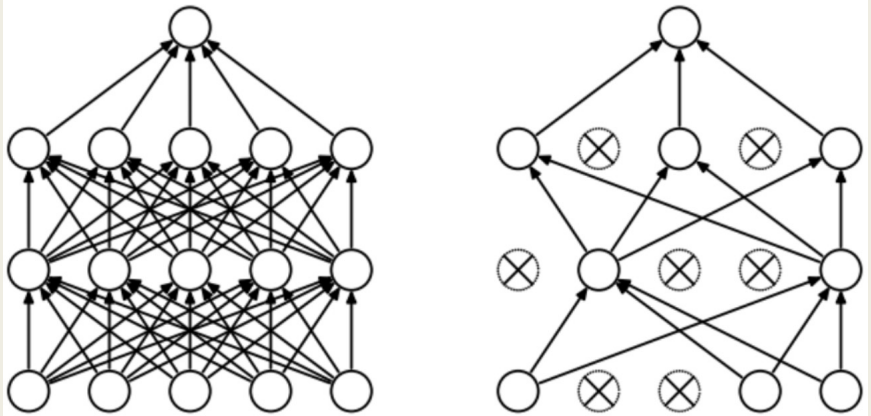
- In the case of L1 regularization (also knows as Lasso regression), we simply use another regularization term different than L2 regularization.

  $C = C_0 + \frac{\lambda}{n}\sum_w |w|$ *where,* C_0 *is the original cost function.*

- If the lambda value is too high, model will be simple, but there is a risk of *underfitting*.
- If the lambda value is too low, model will be more complex, and there is a risk of *overfitting*.

Dropout-

- Dropout refers to ignoring some neurons during the training phase which are chosen at random.
- We need dropout to prevent overfitting.
- Training Phase: For each hidden layer, for each training sample, for each iteration, ignore a random fraction, *p*, of nodes and their corresponding activations.
- Testing Phase: Use all activations, but reduce them by a factor *p* (to account for the missing activations during training).



(a) Standard Neural Net (b) After applying dropout.

Weight Initialization-

- **Weight initialization** is an important consideration in the design of a neural network.

Techniques for weight Initialization-

- **Zero Initialization-** if we initialized all the weights with 0, then the derivative wrt loss function is the same for every weight, thus all weights have the same value in subsequent iterations.
  - *Problem of symmetry, model no better than linear model*
- **Random Initialization-** It is used to break the symmetry.
- **Xavier weight initialization- W =** U [-(1/sqrt(n)), 1/sqrt(n)] where *n* is the number of inputs to the node.
- **Normalised Xavier weight initialization-**

  weight = U [-(sqrt(6)/sqrt(n + m)), sqrt(6)/sqrt(n + m)] where *n* is the number of inputs to the node and *m* is the number of outputs from the layer
- **He Weight Initialization-** weight = G (0.0, sqrt(2/n)) where *n* is the number of inputs to the node.

References-

- [1.] http://neuralnetworksanddeeplearning.com/chap3.html
- [2.] https://www.ibm.com/cloud/learn/gradient-descent
- [3.] https://towardsdatascience.com/softmax-activation-function-how-it-actually-works-d292d335bd78
- [4.] https://medium.com/mlearning-ai/underfitting-and-overfitting-in-deep-learning-687b1b7eb738

Thank You!