Program : **B.Tech**

Subject Name: **Machine Learning**

Subject Code: **CS-601**

Semester: **6th**

UNIT-2

Linearity vs non linearity

A linear model uses a linear function for its prediction function or as a crucial part of its prediction function.

A linear function takes a fixed number of numerical inputs, let's call them x1,x2,…,xn and returns

$$w_0 + \sum_{i=1}^{n} w_i x_i$$

where the weights w0,…,wn are the parameters of the model.

If the prediction function is a linear function, we can perform regression, i.e. predicting a numerical label. We can also take a linear function and return the sign of the result (whether the result is positive or not) and perform binary classification that way: all examples with a positive output receive label A, all others receive label B. There are various other (more complex) options for a response function on top of the linear function, the logistic function is very commonly used (which leads to logistic regression, predicting a number between 0 and 1, typically used to learn the probability of a binary outcome in a noisy setting).

A non-linear model is a model which is not a linear model. Typically these are more powerful (they can represent a larger class of functions) but much harder to train.

Activation functions like sigmoid,ReLU

A neural network is comprised of layers of nodes and learns to map examples of inputs to outputs.

For a given node, the inputs are multiplied by the weights in a node and summed together. This value is referred to as the summed activation of the node. The summed activation is then transformed via an activation function and defines the specific output or “activation” of the node. It is also known as Transfer Function.

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

It is used to determine the output of neural network like yes or no. It maps the resulting values in between 0 to 1 or -1 to 1 etc. (depending upon the function).

Activation functions are an extremely important feature of the artificial neural networks. They basically decide whether a neuron should be activated or not. Whether the information that the neuron is receiving is relevant for the given information or should it be ignored.

$$\text{Y = Activation}(\Sigma(weight * input) + bias)$$

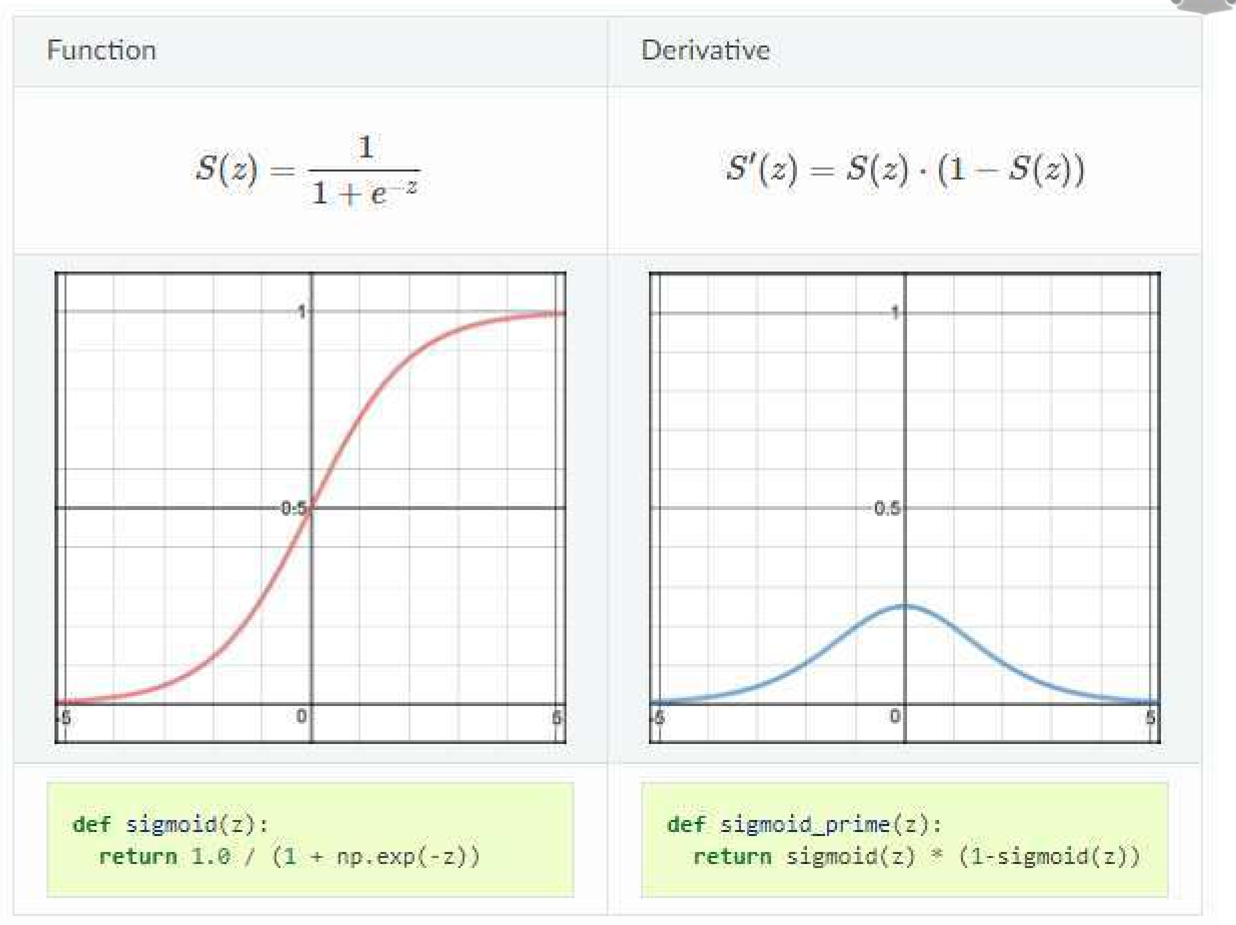
The activation function is the non linear transformation that we do over the input signal. This transformed output is then sen to the next layer of neurons as input.

The Activation Functions can be basically divided into 2 types-

- Linear Activation Function
- Non-linear Activation Functions

Sigmoid

Sigmoid takes a real value as input and outputs another value between 0 and 1. It’s easy to work with and has all the nice properties of activation functions: it’s non-linear, continuously differentiable, monotonic, and has a fixed output range.



Function	Derivative
$S(z) = \frac{1}{1+e^{-z}}$	$S'(z) = S(z) \cdot (1 - S(z))$

```
def sigmoid(z):
  return 1.0 / (1 + np.exp(-z))
```

```
def sigmoid_prime(z):
  return sigmoid(z) * (1-sigmoid(z))
```

Figure 2.1

Pros

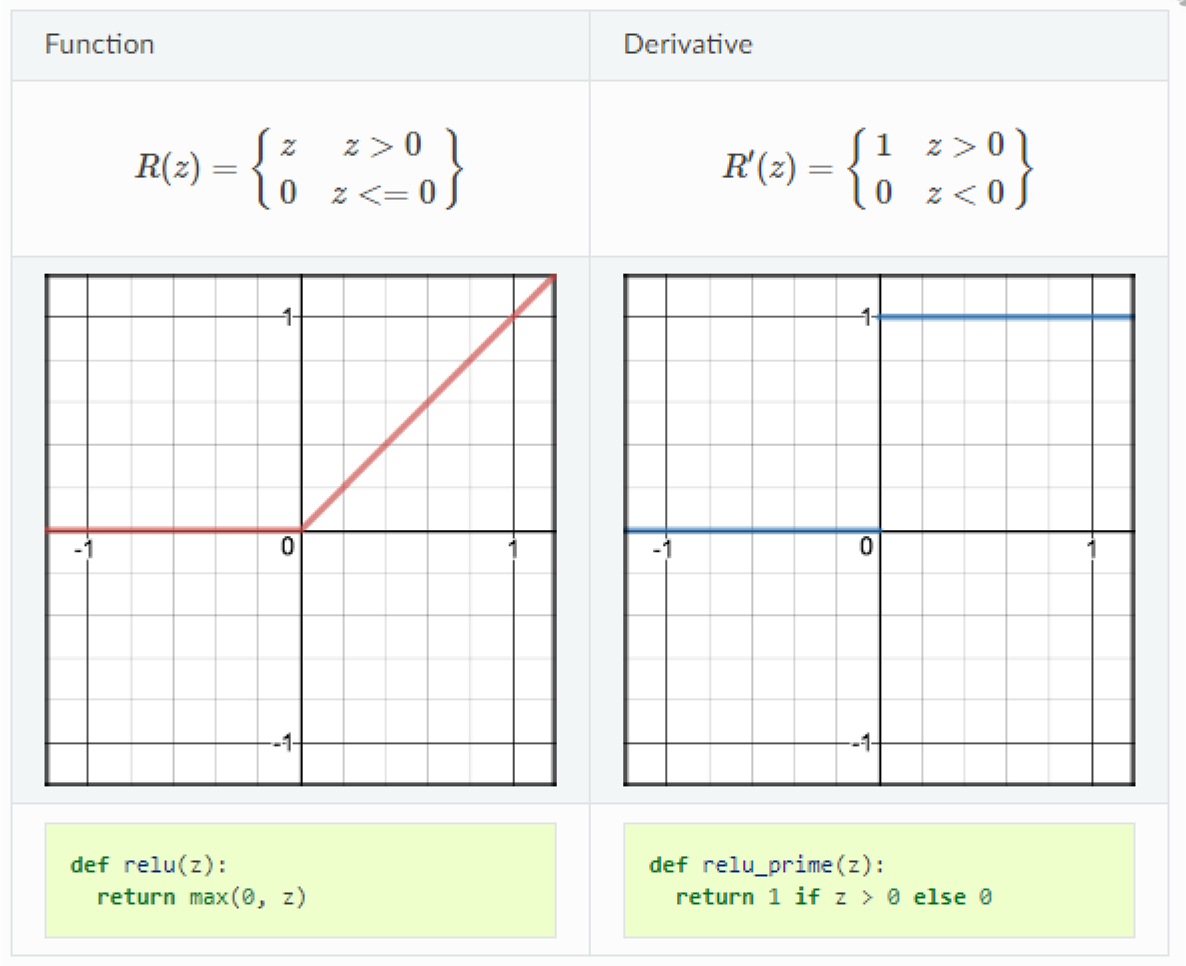
- It is nonlinear in nature. Combinations of this function are also nonlinear!
- It will give an analog activation unlike step function.
- It has a smooth gradient too.
- It's good for a classifier.
- The output of the activation function is always going to be in range (0,1) compared to (-inf, inf) of linear function. So we have our activations bound in a range.

Cons

- Towards either end of the sigmoid function, the Y values tend to respond very less to changes in X.
- It gives rise to a problem of "vanishing gradients".
- Its output isn't zero centered. It makes the gradient updates go too far in different directions. 0 < output < 1, and it makes optimization harder.
- Sigmoids saturate and kill gradients.
- The network refuses to learn further or is drastically slow (depending on use case and until gradient /computation gets hit by floating point value limits).

ReLU

A recent invention which stands for Rectified Linear Units. The formula is deceptively simple: max(0,z). Despite its name and appearance, it's not linear and provides the same benefits as Sigmoid but with better performance.



Function	Derivative
$R(z)=\begin{Bmatrix} z & z>0 \\ 0 & z<=0 \end{Bmatrix}$	$R'(z)=\begin{Bmatrix} 1 & z>0 \\ 0 & z<0 \end{Bmatrix}$

```
def relu(z):
  return max(0, z)
```

```
def relu_prime(z):
  return 1 if z > 0 else 0
```

Figure 2.2

Weights and bias

Neural networks are a class of machine learning algorithms used to model complex patterns in datasets using multiple hidden layers and non-linear activation functions. A neural network takes an input, passes it through multiple layers of hidden neurons (mini-functions with unique coefficients that must be learned), and outputs a prediction representing the combined input of all the neurons.

$$Y = \sum (weight * input) + bias$$

Neuron(Node) — It is the basic unit of a neural network. It gets certain number of inputs and a bias value. When a signal(value) arrives, it gets multiplied by a weight value. If a neuron has 4 inputs, it has 4 weight values which can be adjusted during training time.

$$z = x_1 * w_1 + x_2 * w_2 + \ldots\ldots + x_n * w_n + b * 1$$

$$\hat{y} = a_{out} = sigmoid(z)$$

$$sigmoid(z) = \frac{1}{1+e^{-z}}$$

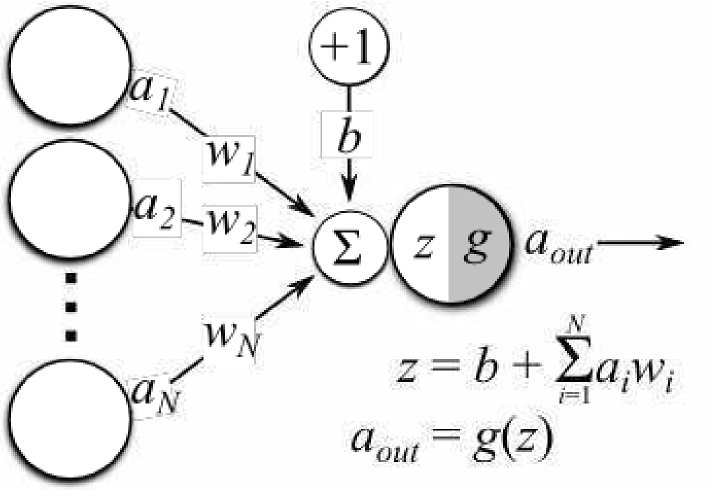


Figure 2.3 Operations at one neuron of a neural network

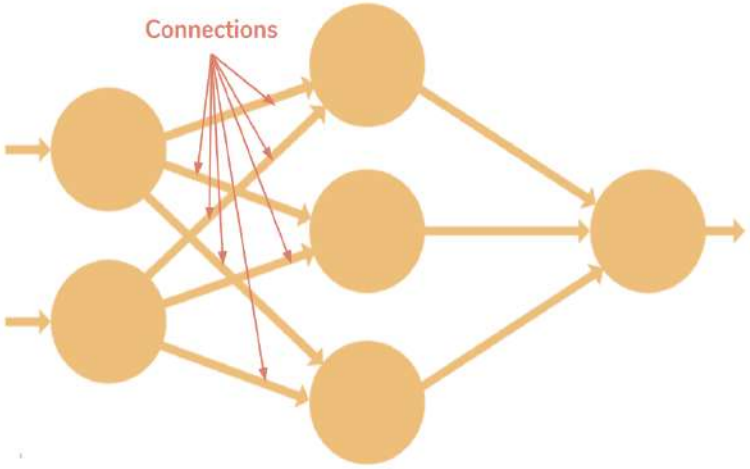


Figure 2.4

Connections — It connects one neuron in one layer to another neuron in other layer or the same layer. A connection always has a weight value associated with it. Goal of the training is to update this weight value to decrease the loss(error).

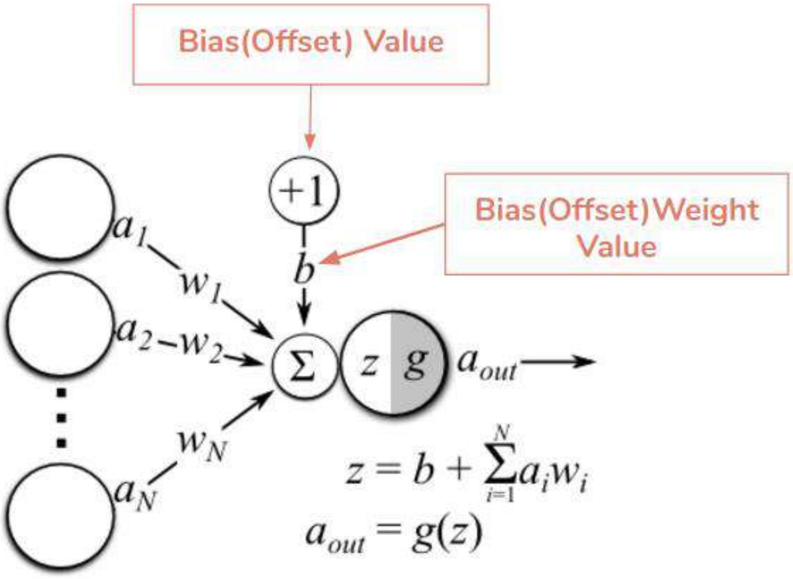


Figure 2.5

Bias (Offset) — It is an extra input to neurons and it is always 1, and has it's own connection weight. This makes sure that even when all the inputs are none (all 0's) there's gonna be an activation in the neuron. Bias terms are additional constants attached to neurons and added to the weighted input before the activation function is applied. Bias terms help models represent patterns that do not necessarily pass through the origin. For example, if all your features were 0, would your output also be zero? Is it possible there is some base value upon which your features have an effect? Bias terms typically accompany weights and must also be learned by your model.

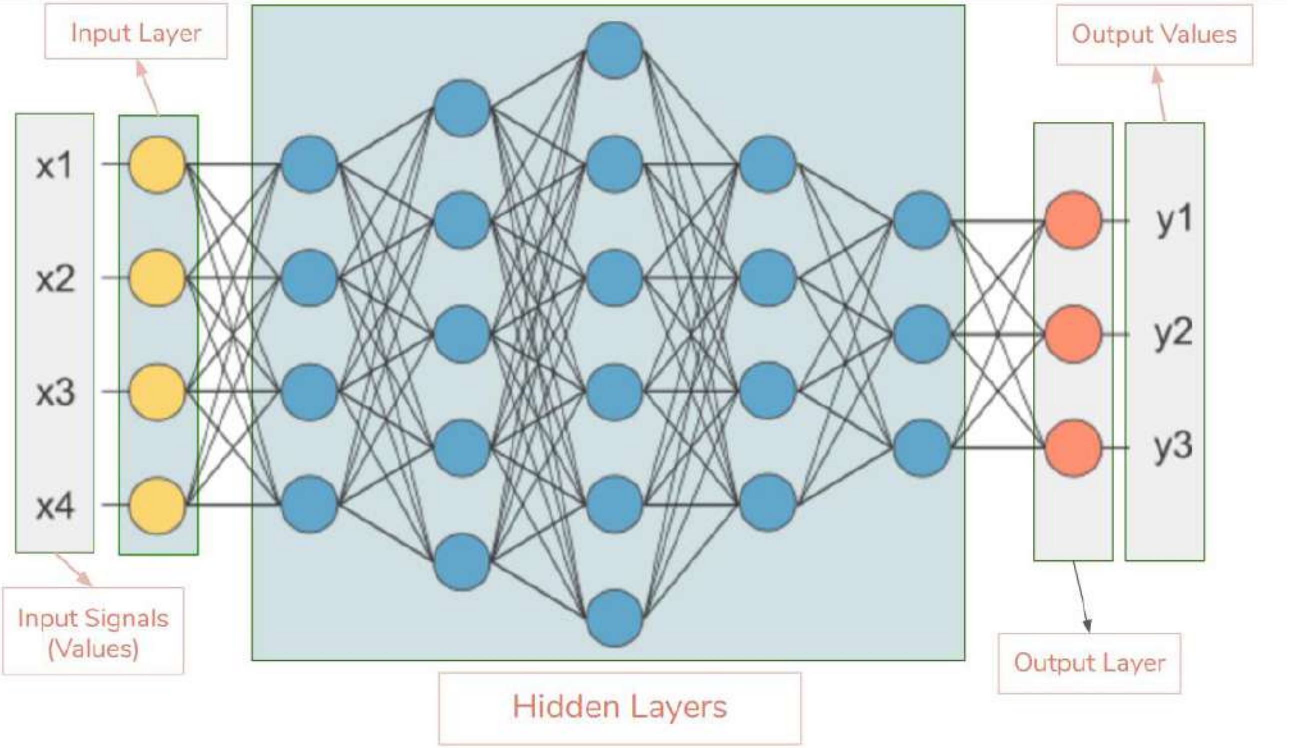


Figure 2.6

Input Layer — This is the first layer in the neural network. It takes input signals (values) and passes them on to the next layer. It doesn't apply any operations on the input signals (values) & has no weights and biases values associated. In our network we have 4 input signals x1, x2, x3, x4.

Hidden Layers — Hidden layers have neurons (nodes) which apply different transformations to the input data. One hidden layer is a collection of neurons stacked vertically (Representation). In our image given above we have 5 hidden layers. In our network, first hidden layer has 4 neurons (nodes), 2nd has 5 neurons, 3rd has 6 neurons, 4th has 4 and 5th has 3 neurons. Last hidden layer passes on values to the output layer. All the neurons in a hidden layer are connected to each and every neuron in the next layer, hence we have a fully connected hidden layers.

Output Layer — This layer is the last layer in the network & receives input from the last hidden layer. With this layer we can get desired number of values and in a desired range. In this network we have 3 neurons in the output layer and it outputs y1, y2, y3.

Input Shape — It is the shape of the input matrix we pass to the input layer. Our network's input layer has 4 neurons and it expects 4 values of 1 sample. Desired input shape for our network is (1, 4, 1) if we feed it one sample at a time. If we feed 100 samples input shape will be (100, 4, 1). Different libraries expect shapes in different formats.

Weights (Parameters) — A weight represent the strength of the connection between units. If the weight from node 1 to node 2 has greater magnitude, it means that neuron 1 has greater influence over neuron 2. A weight brings down the importance of the input value. Weights near zero means changing this input will not change the output. Negative weights mean increasing this input will decrease the output. A weight decides how much influence the input will have on the output.

A neuron’s input equals the sum of weighted outputs from all neurons in the previous layer. Each input is multiplied by the weight associated with the synapse connecting the input to the current neuron. If there are 3 inputs or neurons in the previous layer, each neuron in the current layer will have 3 distinct weights — one for each each synapse.

Single Input

$$\begin{aligned} Z &= Input \cdot Weight \\ &= XW \end{aligned}$$

$$\begin{aligned} Z &= \sum_{i=1}^{n} x_i w_i \\ &= x_1 w_1 + x_2 w_2 + x_3 w_3 \end{aligned}$$

Multiple Input

Notice, it’s exactly the same equation we use with linear regression! In fact, a neural network with a single neuron is the same as linear regression! The only difference is the neural network post-processes the weighted input with an activation function.

Loss Functions

A loss function, or cost function, is a wrapper around our model’s predict function that tells us “how good” the model is at making predictions for a given set of parameters. The loss function has its own curve and its own derivatives. The slope of this curve tells us how to change our parameters to make the model more accurate! We use the model to make predictions. We use the cost function to update our parameters. Our cost function can take a variety of forms as there are many different cost functions available. Popular loss functions include: MSE (L2) and Cross-entropy Loss.

The loss function computes the error for a single training example. The cost function is the average of the loss functions of the entire training set.

- ‘*mse*’: for mean squared error.
- ‘*binary_crossentropy*’: for binary logarithmic loss (logloss).
- ‘*categorical_crossentropy*’: for multi-class logarithmic loss (logloss).

Gradient Descent

Optimization is a big part of machine learning. Almost every machine learning algorithm has an optimization algorithm at it’s core.

Gradient descent is an optimization algorithm used to find the values of parameters (coefficients) of a function (f) that minimizes a cost function (cost).

Gradient descent is best used when the parameters cannot be calculated analytically (e.g. using linear algebra) and must be searched for by an optimization algorithm.

Gradient Descent Procedure

The procedure starts off with initial values for the coefficient or coefficients for the function. These could be 0.0 or a small random value.

coefficient = 0.0

The cost of the coefficients is evaluated by plugging them into the function and calculating the cost.

cost = f(coefficient)

or

cost = evaluate(f(coefficient))

The derivative of the cost is calculated. The derivative is a concept from calculus and refers to the slope of the function at a given point. We need to know the slope so that we know the direction (sign) to move the coefficient values in order to get a lower cost on the next iteration.

delta = derivative(cost)

Now that we know from the derivative which direction is downhill, we can now update the coefficient values. A learning rate parameter (alpha) must be specified that controls how much the coefficients can change on each update.

coefficient = coefficient – (alpha * delta)

This process is repeated until the cost of the coefficients (cost) is 0.0 or close enough to zero to be good enough.

Types of gradient Descent:

1. **Batch Gradient Descent:** This is a type of gradient descent which processes all the training examples for each iteration of gradient descent. But if the number of training examples is large, then batch gradient descent is computationally very expensive. Hence if the number of training examples is large, then batch gradient descent is not preferred. Instead, we prefer to use stochastic gradient descent or mini-batch gradient descent.
2. **Stochastic Gradient Descent:** This is a type of gradient descent which processes 1 training example per iteration. Hence, the parameters are being updated even after one iteration in which only a single example has been processed. Hence this is quite faster than batch gradient descent. But again, when the number of training examples is large, even then it processes only one example which can be additional overhead for the system as the number of iterations will be quite large.
3. **Mini Batch gradient descent:** This is a type of gradient descent which works faster than both batch gradient descent and stochastic gradient descent. Here *b* examples where *b<m* are processed per iteration. So even if the number of training examples is large, it is processed in batches of b training examples in one go. Thus, it works for larger training examples and that too with lesser number of iterations.

Multilayer network

A fully connected multi-layer neural network is called a Multilayer Perceptron (MLP).

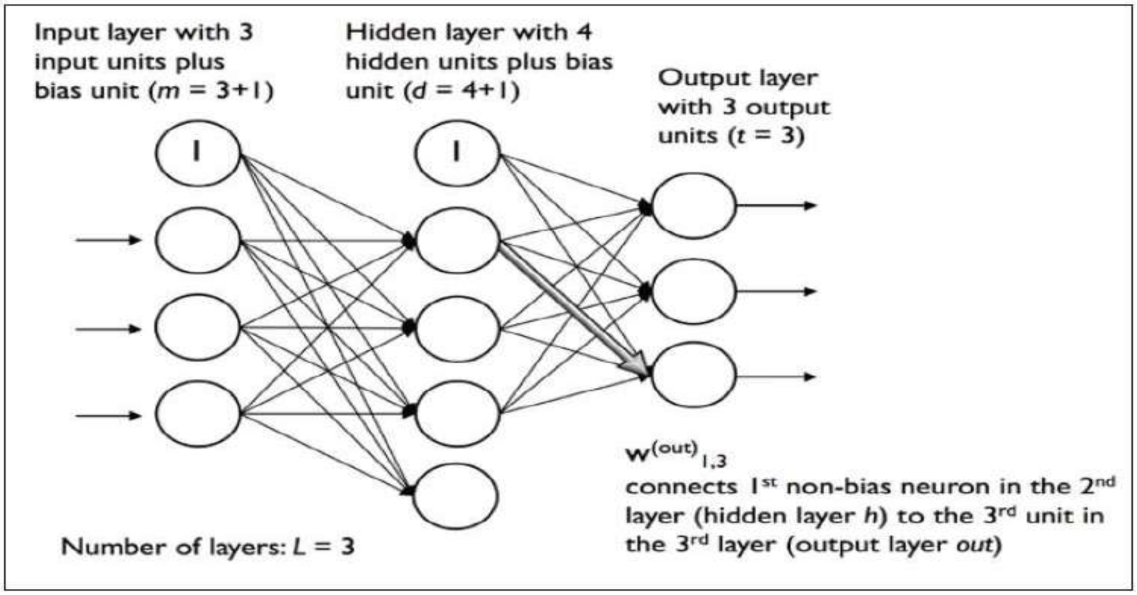

Input layer with 3 input units plus bias unit (m = 3+1)
Hidden layer with 4 hidden units plus bias unit (d = 4+1)
Output layer with 3 output units (t = 3)
1
1
w(out) 1,3
connects 1st non-bias neuron in the 2nd layer (hidden layer h) to the 3rd unit in the 3rd layer (output layer out)
Number of layers: L = 3

Figure 2.7

It has 3 layers including one hidden layer. If it has more than 1 hidden layer, it is called a deep ANN.
An MLP is a typical example of a feed-forward artificial neural network.
In this figure, the ith activation unit in the lth layer is denoted as $a_i^{(l)}$.
The number of layers and the number of neurons are referred to as hyper parameters of a neural network, and these need tuning. Cross-validation techniques must be used to find ideal values for these.
The weight adjustment training is done via backpropagation.

Notations

In the representation below:

$$a^{(in)} = \begin{bmatrix} a_0^{(in)} \\ a_1^{(in)} \\ \vdots \\ a_m^{(in)} \end{bmatrix} = \begin{bmatrix} 1 \\ x_1^{(in)} \\ \vdots \\ x_m^{(in)} \end{bmatrix}$$

- $a_i^{(in)}$ refers to the ith value in the input layer
- $a_i^{(h)}$ refers to the ith unit in the hidden layer
- $a_i^{(out)}$ refers to the ith unit in the output layer
- $a_o^{(in)}$ is simply the bias unit and is equal to 1; it will have the corresponding weight w_0
- The weight coefficient from layer l to layer l+1 is represented by $w_{k,j}^{(l)}$

Backpropagation

Backpropagation is a supervised learning technique for neural networks that calculates the gradient of descent for weighting different variables. It's short for the backward propagation of errors, since the error is computed at the output and distributed backwards throughout the network's layers.
When an artificial neural network discovers an error, the algorithm calculates the gradient of the error function, adjusted by the network's various weights. The gradient for the final layer of weights is calculated first, with the first layer's gradient of weights calculated last. Partial calculations of the gradient from one layer are reused to determine the gradient for the previous layer. This point of this backwards method of error checking is to more efficiently calculate the gradient at each layer than the traditional approach of calculating each layer's gradient separately.

Uses of Backpropagation

Backpropagation is especially useful for deep neural networks working on error-prone projects, such as image or speech recognition. Taking advantage of the chain and power rules allows backpropagation to function with any number of outputs and better train all sorts of neural networks.

Unstable gradient problem

The unstable gradient problem is a fundamental problem that occurs in a neural network, that entails that a gradient in a deep neural network tends to either explode or vanish in early layers. The unstable gradient problem is not necessarily the vanishing gradient problem or the exploding gradient problem, but is rather due to the fact that gradient in early layers is the product of terms from all proceeding layers. More layers make the network an intrinsically unstable solution. Balancing all products of terms is the only way each layer in a neural network can close at the same speed and avoid vanishing or exploding gradients. Balanced product of terms occurring by chance becomes more and more unlikely with more layers. Neural networks there for have layers that learn at different speeds, without being given any mechanisms or underlying reason for balancing learning speeds. When magnitudes of gradients accumulate, unstable networks are more likely to occur, which is a cause of poor prediction results.

Consider $y = f(x)$ that maps an input x to some output y. The function $f(\cdot)$ could be anything but here we consider a normal feed forward neural network.

Feedforward NN's have weight matrices and (non-linear) activation functions (that enable a non-linear decision surface, see also the XOR problem). In the elaboration below, let's look at the simplest imaginable "neural" network, with a single input (scalar x) and a single output (scalar y). Input x passes through several layers (let's take 3 in the example), in each of which it's multiplied with a weight matrix W_l (l for layer index) and the resulting vector is fed through an activation function (let's take a sigmoid, σ). In our example, the input is a scalar and our weights are scalars as well. So the expression for $f(x)$ is:

$$f(x) = \sigma(w_3 \times \sigma(w_2 \times \sigma(w_1 \times x)))$$

Now let's train this w.r.t. some cost E. Whichever E we choose, we want the $\frac{\partial E}{\partial w_l}$ that constitute our gradient of E w.r.t. weights w_l in order to find weights that minimize E. The choice of cost function doesn't matter for our purposes since $\frac{\partial E}{\partial w_l} = \frac{\partial E}{\partial y}\frac{\partial y}{\partial w_l}$. So let's differentiate our expression for $f(x)$ w.r.t. the weights:

$$\frac{\partial f}{\partial w_1} = \sigma'(h_2 w_3) \times w_3 \times \sigma'(h_1 w_2) \times w_2 \times \sigma'(x w_1) x$$

Where $h_2 = \sigma(w_2 \times \sigma(w_1 \times x))$ and $h_1 = \sigma(w_1 \times x)$.

The expression for the partial derivative of E w.r.t. a weight in the first layer of (w_1) shows a lot of product terms. This product chain can be a source of gradient instabilities.

First of all, the product of $\sigma'(\cdot)$ exponentially decreases when going deeper. The function $\sigma'(\cdot) = \sigma(\cdot) \times (1 - \sigma(\cdot))$ has a peak value of 0.25 at 0. Thus, with a deep FF NN, in the k-th layer, the partial derivative of E for each weight in that layer will have a $(1/4)^k$ contribution from the sigmoid. With three layers, this is 0.0156, compared to the 0.25 at the top layer. This is assumed to be the cause of the vanishing gradient problem.

On the other hand, when weights in each layer are large, we could get an exploding gradient (because of the product of many large numbers). To conclude, the chain of products in the derivative for deeper layers causes gradient instabilities.

Auto encoders

An auto encoder **neural network** is an **Unsupervised Machine learning** algorithm that applies backpropagation, setting the target values to be equal to the inputs. Auto encoders are used to reduce the size of our inputs into a smaller representation. If anyone needs the original data, they can reconstruct it from the compressed data.

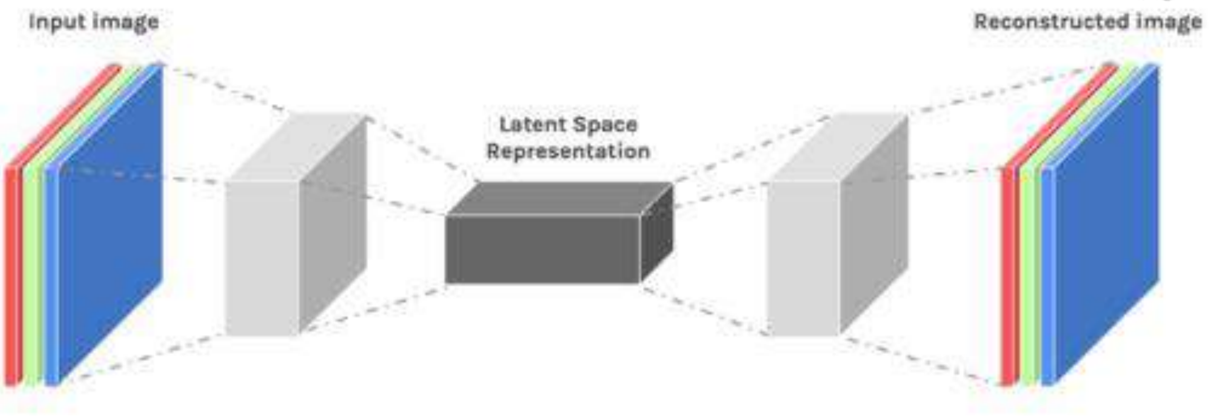


Figure 2.8

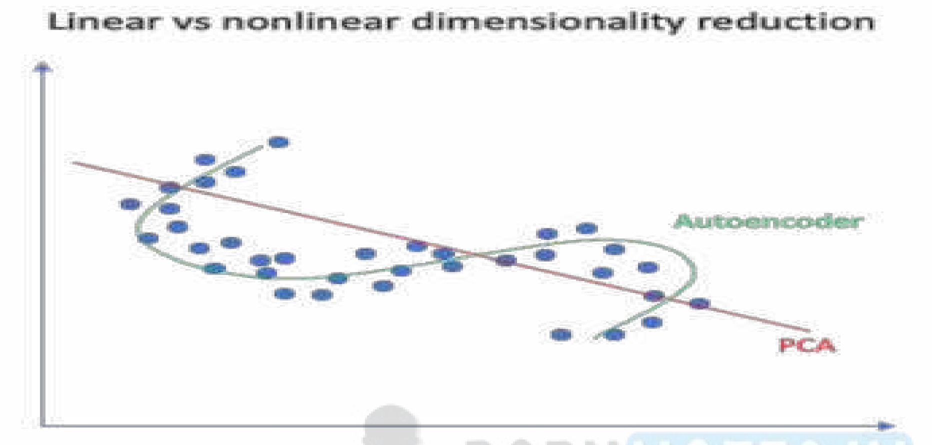


Figure 2.9

Auto encoders are preferred over PCA because:

- An auto encoder can learn **non-linear transformations** with a **non-linear activation function** and multiple layers.
- It doesn't have to learn dense layers. It can use **convolutional layers** to learn which is better for video, image and series data.
- It is more efficient to learn several layers with an auto encoder rather than learn one huge transformation with PCA.
- An auto encoder provides a representation of each layer as the output.
- It can make use of **pre-trained layers** from another model to apply transfer learning to enhance the encoder/decoder.

Applications of Auto encoders

1. **Image Coloring**

Auto encoders are used for converting any black and white picture into a colored image. Depending on what is in the picture, it is possible to tell what the color should be.

2. **Feature variation**

It extracts only the required features of an image and generates the output by removing any noise or unnecessary interruption.

3. **Dimensionality Reduction**

The reconstructed image is the same as our input but with reduced dimensions. It helps in providing the similar image with a reduced pixel value.

4. **Denoising Image**

The input seen by the auto encoder is not the raw input but a stochastically corrupted version. A denoising auto encoder is thus trained to reconstruct the original input from the noisy version.

5. **Watermark Removal**

It is also used for removing watermarks from images or to remove any object while filming a video or a movie.

Architecture of Auto encoders : An Auto encoder consist of three layers:

1. **Encoder**
2. **Code**
3. **Decoder**

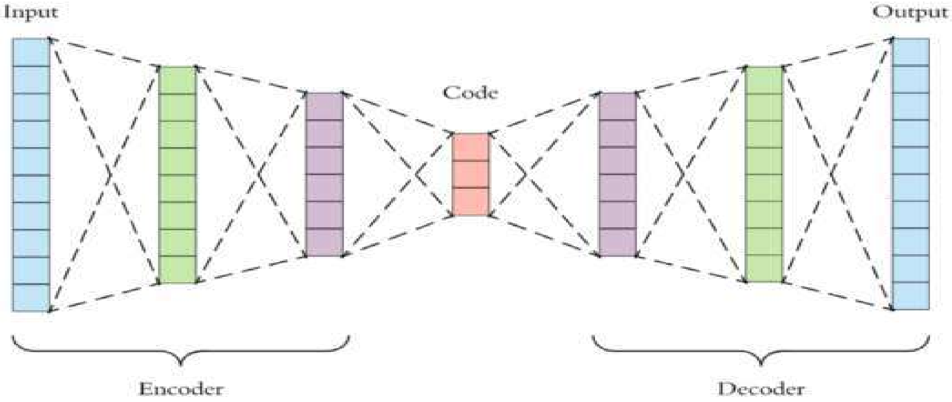


Figure 2.10

- **Encoder:** This part of the network compresses the input into a **latent space representation**. The encoder layer **encodes** the input image as a compressed representation in a reduced dimension. The compressed image is the distorted version of the original image.
- **Code:** This part of the network represents the compressed input which is fed to the decoder.
- **Decoder:** This layer **decodes** the encoded image back to the original dimension. The decoded image is a lossy reconstruction of the original image and it is reconstructed from the latent space representation.

The layer between the encoder and decoder, ie. the code is also known as **Bottleneck**. This is a well-designed approach to decide which aspects of observed data are relevant information and what aspects can be discarded. It does this by balancing two criteria :

- Compactness of representation, measured as the compressibility.
- It retains some behaviourally relevant variables from the input.

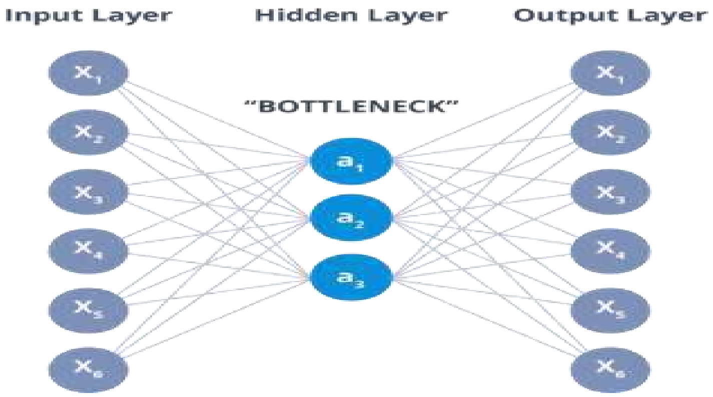


Figure 2.11

Types of Autoencoders

1. **Convolution Auto encoders**

Auto encoders in their traditional formulation does not take into account the fact that a signal can be seen as a sum of other signals. Convolutional Auto encoders use the convolution operator to exploit this observation. They learn to encode the input in a set of simple signals and then try to reconstruct the input from them, modify the geometry or the reflectance of the image.

2. **Sparse Auto encoders**

Sparse auto encoders offer us an alternative method for introducing an information bottleneck **without requiring a reduction in the number of nodes** at our hidden layers. Instead, we'll construct our loss function such that we penalize activations within a layer.

3. **Deep Auto encoders**

The extension of the simple Auto encoder is the Deep Auto encoder. The first layer of the Deep Auto encoder is used for first-order features in the raw input. The second layer is used for second-order features corresponding to patterns in the appearance of first-order features. Deeper layers of the Deep Auto encoder tend to learn even higher-order features.

A **deep auto encoder** is composed of two, symmetrical deep-belief networks-

1. First four or five shallow layers representing the encoding half of the net.
2. The second set of four or five layers that make up the decoding half.

Batch normalization

Batch normalization is a method we can use to normalize the inputs of each layer, in order to fight the internal covariate shift problem.

During training time, a batch normalization layer does the following:

1. Calculate the mean and variance of the layers input.

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{Batch mean}$$

$$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_B)^2 \qquad \text{Batch variance}$$

Batch statistics for step 1

2. Normalize the layer inputs using the previously calculated batch statistics.

$$\overline{x_i} = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

Normalization of the layers input in step 2

3. Scale and shift in order to obtain the output of the layer.

$$y_i = \gamma \overline{x_i} + \beta$$

Scaling and shifting the normalized input for step 3

Notice that *γ* and *β* are **learned during training** along with the original parameters of the network.

So, if each batch had *m* samples and there where *j* batches:

$$E_x = \frac{1}{m}\sum_{i=1}^{j}\mu_B^{(i)} \qquad \text{Inference mean}$$

$$Var_x = \left(\frac{m}{m-1}\right)\frac{1}{m}\sum_{i=1}^{j}\sigma_B^{2(i)} \qquad \text{Inference variance}$$

$$y = \frac{\gamma}{\sqrt{Var_x + \epsilon}}x + \left(\beta + \frac{\gamma E_x}{\sqrt{Var_x + \epsilon}}\right) \qquad \text{Inference scaling/shifting}$$

Inference formulas

Dropout

Dropout is implemented per-layer in a neural network.

It can be used with most types of layers, such as dense fully connected layers, convolutional layers, and recurrent layers such as the long short-term memory network layer.

Dropout may be implemented on any or all hidden layers in the network as well as the visible or input layer. It is not used on the output layer.

The term "dropout" refers to dropping out units (hidden and visible) in a neural network.

Simply , dropout refers to ignoring units (i.e. neurons) during the training phase of certain set of neurons which is chosen at random. By "ignoring", mean these units are not considered during a particular forward or backward pass.

More technically, At each training stage, individual nodes are either dropped out of the net with probability 1-p or kept with probability p, so that a reduced network is left; incoming and outgoing edges to a dropped-out node are also removed.

Neural networks are the building blocks of any machine-learning architecture. They consist of one input layer, one or more hidden layers, and an output layer.

When we training our neural network (or model) by updating each of its weights, it might become too dependent on the dataset we are using. Therefore, when this model has to make a prediction or classification, it will not give satisfactory results. This is known as over-fitting. We might understand this problem through a real-world example: If a student of mathematics learns only one chapter of a book and then takes a test on the whole syllabus, he will probably fail.

To overcome this problem, we use a technique that was introduced by Geoffrey Hinton in 2012. **This technique is known as dropout.**

L1 and L2 regularization

L1 and L2 regularisation owes its name to L1 and L2 norm of a vector ***w*** respectively. Here's a primer on norms:

$$||\mathbf{w}||_1 = |w_1| + |w_2| + \ldots + |w_N|$$

1-norm (also known as L1 norm)

$$||\mathbf{w}||_2 = (w_1^2 + w_2^2 + \ldots + w_N^2)^{\frac{1}{2}}$$

2-norm (also known as L2 norm or Euclidean norm)

$$||\mathbf{w}||_p = (w_1^p + w_2^p + \ldots + w_N^p)^{\frac{1}{p}}$$

p-norm

A linear regression model that implements L1 norm for regularisation is called **lasso regression**, and one that implements (squared) L2 norm for regularisation is called **ridge regression**. To implement these two, note that the linear regression model stays the same:

$$\hat{y} = w_1x_1 + w_2x_2 + \ldots + w_Nx_N + b$$

but it is the calculation of the loss function that includes these regularisation terms:

$$Loss = Error(y, \hat{y})$$

Loss function with no regularisation

$$Loss = Error(y, \hat{y}) + \lambda \sum_{i=1}^{N} |w_i|$$

Loss function with L1 regularisation

$$Loss = Error(y, \hat{y}) + \lambda \sum_{i=1}^{N} w_i^2$$

Loss function with L2 regularisation

The regularisation terms are 'constraints' by which an optimisation algorithm must 'adhere to' when minimising the loss function, apart from having to minimise the error between the true *y* and the predicted *ŷ*.

Loss Functions

To demonstrate the effect of L1 and L2 regularisation, let's fit our linear regression model using 3 different loss functions/objectives:

1. **L**
2. **L1**
3. **L2**

Our objective is to minimise these different losses.

1. **Loss function with no regularisation**

We define the loss function L as the squared error, where error is the difference between *y* (the true value) and *ŷ* (the predicted value).

$$\begin{aligned} L &= (\hat{y} - y)^2 \\ &= (wx + b - y)^2 \end{aligned}$$

Let's assume our model will be overfitted using this loss function.

2. **Loss function with L1 regularisation**

Based on the above loss function, adding an L1 regularisation term to it looks like this:

$$L_1 = (wx + b - y)^2 + \lambda|w|$$

where the regularisation parameter *λ* > 0 is manually tuned. Let's call this loss function L1. Note that |*w*/ is differentiable everywhere except when *w*=0, as shown below. We will need this later.

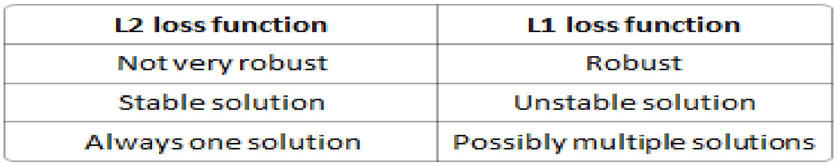
$$\frac{d|w|}{dw} = \begin{cases} 1 & w > 0 \\ -1 & w < 0 \end{cases}$$

3. **Loss function with L2 regularisation**

Similarly, adding an L2 regularisation term to L looks like this:

$$L_2 = (wx + b - y)^2 + \lambda w^2$$

where again, *λ* > 0.



L2 loss function	L1 loss function
Not very robust	Robust
Stable solution	Unstable solution
Always one solution	Possibly multiple solutions

Momentum

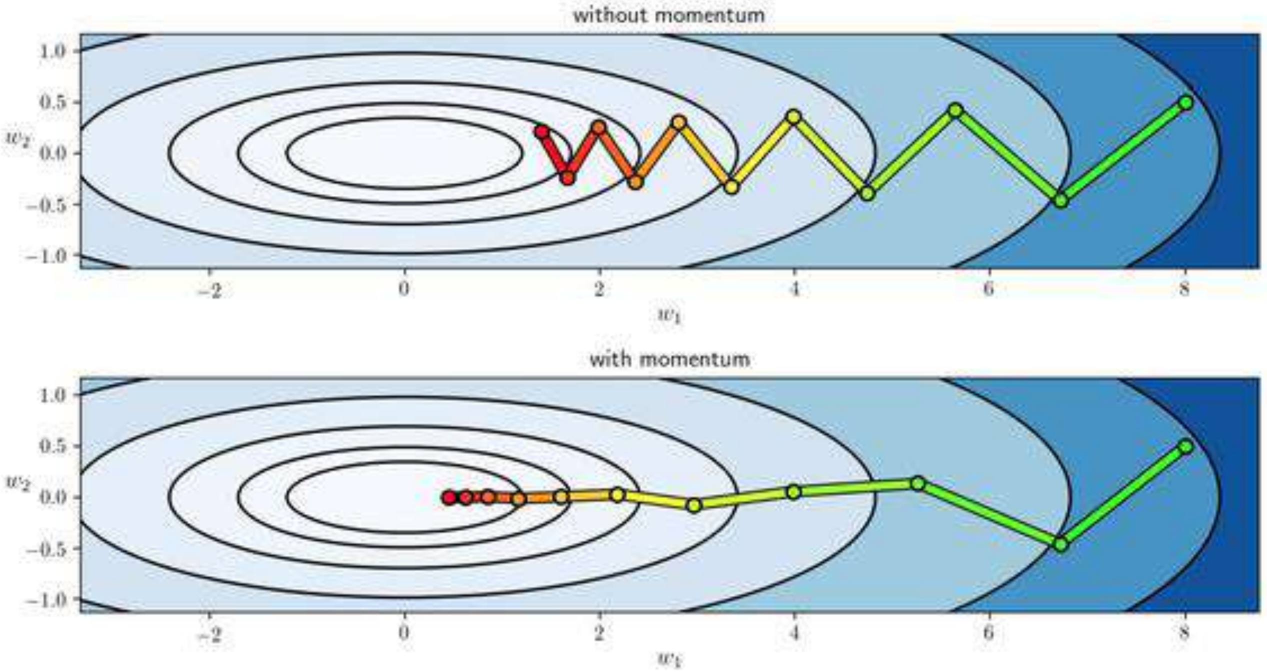


Figure 2.12

Momentum methods in the context of machine learning refer to a group of tricks and techniques designed to speed up convergence of first order optimization methods like gradient descent (and its many variants).

They essentially work by adding what's called *the momentum term* to the update formula for gradient descent, thereby ameliorating its natural "zigzagging behavior," especially in long narrow valleys of the cost function.

The figure below shows the progress of gradient descent - with and without momentum - towards reaching the minimum of a quadratic cost function, located at the center of the concentric elliptical contours.

Let's say your first update to the weights is a vector θ1θ1. For the second update (which would be θ2θ2 without momentum) you update by θ2+αθ1θ2+αθ1. For the next one, you update by θ3+αθ2+α2θ1θ3+αθ2+α2θ1, and so on. Here the parameter 0≤α<10≤α<1 indicates the amount of momentum we want.

The practical way of doing that is keeping an update vector vivi, and updating it as vi+1=αvi+θi+1vi+1=αvi+θi+1.

The reason we do this is to avoid the algorithm getting stuck in a local minimum. Think of it as a marble rolling around on a curved surface. We want to get to the lowest point. The marble having momentum will allow it to avoid a lot of small dips and make it more likely to find a better local solution.

Having momentum too high means you will be more likely to overshoot (the marble goes through the local minimum but the momentum carries it back upwards for a bit). This will lead to longer learning times. Finding the correct value of the momentum will depend on the particular problem: the smoothness of the function, how many local minima you expect, how "deep" the sub-optimal local minima are expected to be, etc.

Tuning hyper parameters

Hyper parameters, that cannot be directly learned from the regular training process. They are usually fixed before the actual training process begins. These parameters express important properties of the model such as its complexity or how fast it should learn.

Some examples of model hyper parameters include:

1. The penalty in Logistic Regression Classifier i.e. L1 or L2 regularization
2. The learning rate for training a neural network.
3. The C and sigma hyper parameters for support vector machines.
4. The k in k-nearest neighbors.

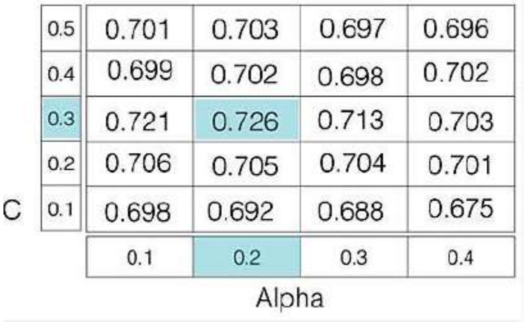
Models can have many hyper parameters and finding the best combination of parameters can be treated as a search problem. Two best strategies for Hyper parameter tuning are:

- Grid Search CV
- Randomized Search CV

Grid Search CV

In Grid Search CV approach, machine learning model is evaluated for a range of hyper parameter values. This approach is called Grid Search CV, because it searches for best set of hyper parameters from a grid of hyper parameters values.

For example, if we want to set two hyper parameters C and Alpha of Logistic Regression Classifier model, with different set of values. The grid search technique will construct many versions of the model with all possible combinations of hyper parameters, and will return the best one.



	0.5	0.701	0.703	0.697	0.696
	0.4	0.699	0.702	0.698	0.702
	0.3	0.721	0.726	0.713	0.703
	0.2	0.706	0.705	0.704	0.701
C	0.1	0.698	0.692	0.688	0.675
		0.1	0.2	0.3	0.4
			Alpha		

As in the image, for C = [0.1, 0.2, 0.3, 0.4, 0.5] and Alpha = [0.1, 0.2, 0.3, 0.4].

For a combination ***C=0.3 and Alpha=0.2***, performance score comes out to be **0.726(Highest)**, therefore it is selected.

Drawback : Grid Search CV will go through all the intermediate combinations of hyper parameters which makes grid search computationally very expensive.

Randomized Search CV

Randomized Search CV solves the drawbacks of Grid Search CV, as it goes through only a fixed number of hyper parameter settings. It moves within the grid in random fashion to find the best set hyper parameters. This approach reduces unnecessary computation.