Deep Learning

Neural Networks

A neural network has a large number of processors. These processors operate parallelly but are arranged as tiers. The first tier receives the raw input similar to how the optic nerve receives the raw information in human beings.

Each successive tier then receives input from the tier before it and then passes on its output to the tier after it. The last tier processes the final output.

Small nodes make up each tier. The nodes are highly interconnected with the nodes in the tier before and after. Each node in the neural network has its own sphere of knowledge, including rules that it was programmed with and rules it has learnt by itself.

The key to the efficacy of neural networks is they are extremely adaptive and learn very quickly. Each node weighs the importance of the input it receives from the nodes before it. The inputs that contribute the most towards the right output are given the highest weight.

What are the Different Types of Neural Networks?

Different types of neural networks use different principles in determining their own rules. There are many types of artificial neural networks, each with their unique strengths. You can take a look at this video to see the different types of neural networks and their applications in detail.

A mostly complete chart of

Neural Networks

- Backfed Input Cell
- Input Cell
- Noisy Input Cell
- Hidden Cell
- Probablistic Hidden Cell
- Spiking Hidden Cell
- Output Cell
- Match Input Output Cell
- Recurrent Cell
- Memory Cell
- Different Memory Cell
- Kernel
- Convolution or Pool

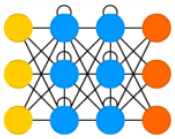
Perceptron (P)

Feed Forward (FF)

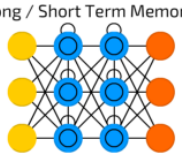
Radial Basis Network (RBF)

Deep Feed Forward (DFF)

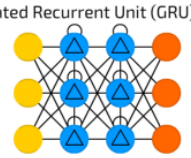
Recurrent Neural Network (RNN)



Long / Short Term Memory (LSTM)

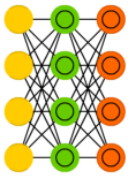


Gated Recurrent Unit (GRU)

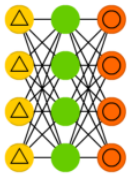


Auto Encoder (AE)

Variational AE (VAE)

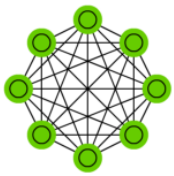


Denoising AE (DAE)

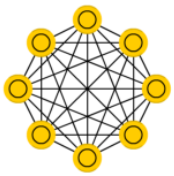


Sparse AE (SAE)

Markov Chain (MC)



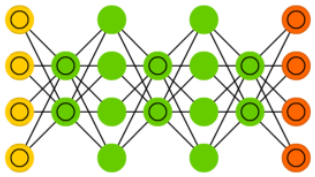
Hopfield Network (HN)



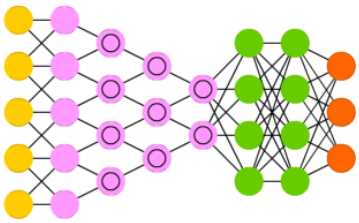
Boltzmann Machine (BM)

Restricted BM (RBM)

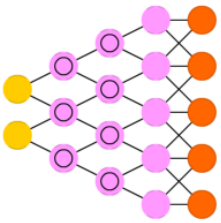
Deep Belief Network (DBN)



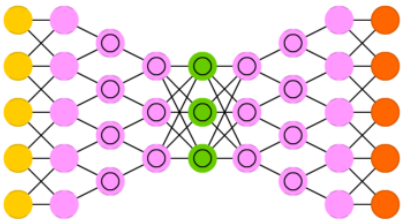
Deep Convolutional Network (DCN)



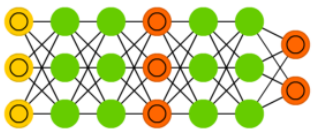
Deconvolutional Network (DN)



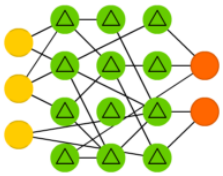
Deep Convolutional Inverse Graphics Network (DCIGN)



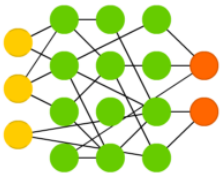
Generative Adversarial Network (GAN)



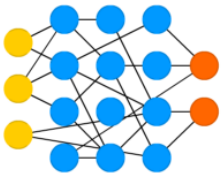
Liquid State Machine (LSM)



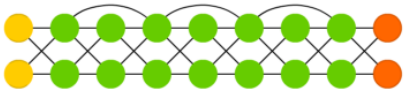
Extreme Learning Machine (ELM)



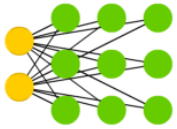
Echo State Network (ESN)



Deep Residual Network (DRN)

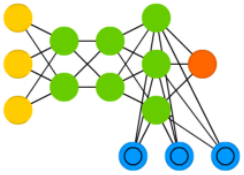


Kohonen Network (KN)



Support Vector Machine (SVM)

Neural Turing Machine (NTM)



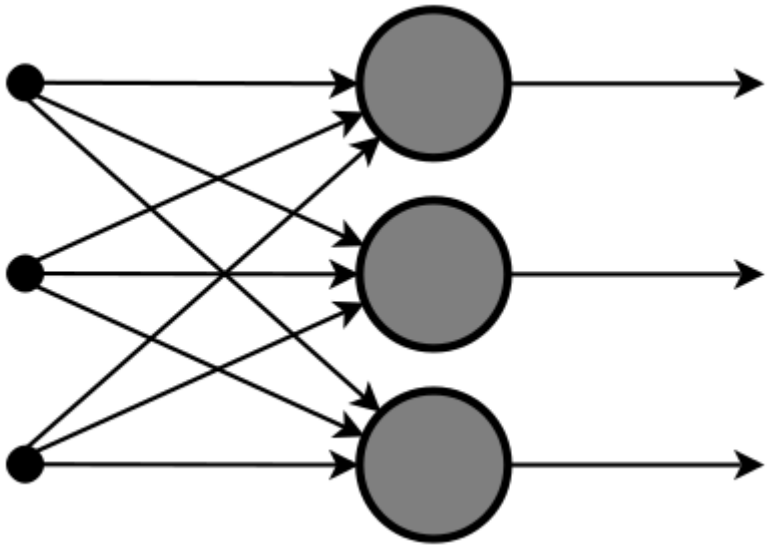
1. Feedforward Neural Network – Artificial Neuron

This is one of the simplest types of artificial neural networks. In a feedforward neural network, the data passes through the different input nodes till it reaches the output node.

In other words, data moves in only one direction from the first tier onwards until it reaches the output node. This is also known as a front propagated wave which is usually achieved by using a classifying activation function.

Unlike in more complex types of neural networks, there is no backpropagation and data moves in one direction only. A feedforward neural network may have a single layer or it may have hidden layers.

In a feedforward neural network, the sum of the products of the inputs and their weights are calculated. This is then fed to the output. Here is an example of a single layer feedforward neural network.



Feedforward Neural Network – Artificial Neuron

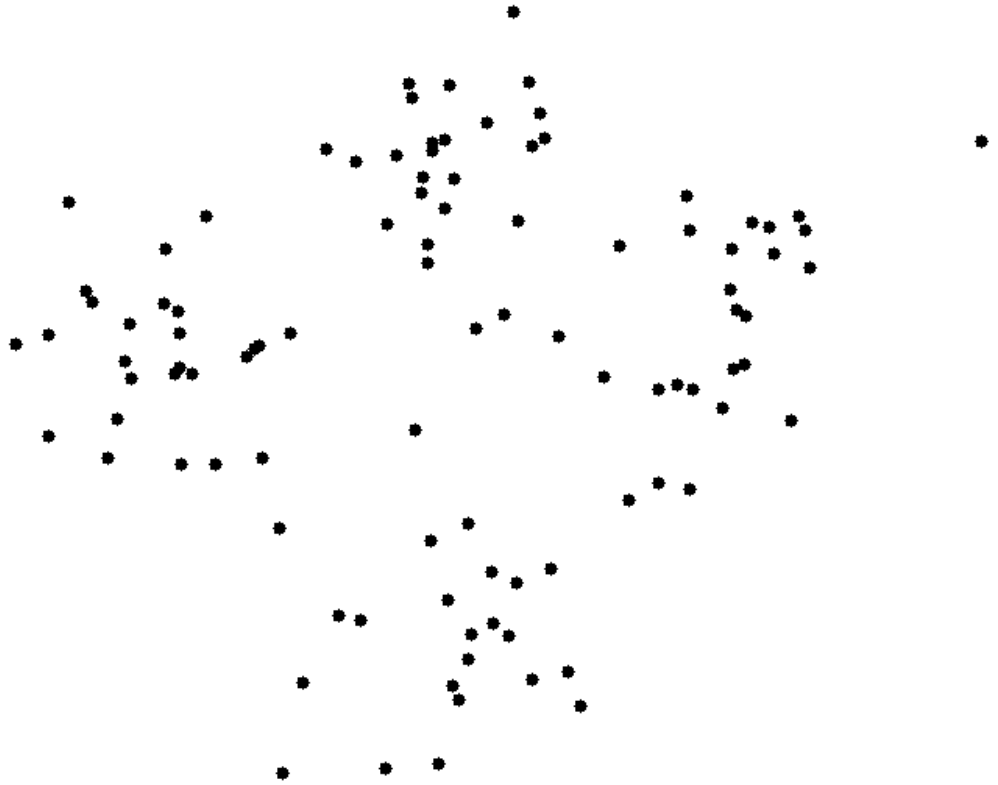
Feedforward neural networks are used in technologies like face recognition and computer vision. This is because the target classes in these applications are hard to classify.

A simple feedforward neural network is equipped to deal with data which contains a lot of noise. Feedforward neural networks are also relatively simple to maintain.

2. Radial Basis Function Neural Network

A radial basis function considers the distance of any point relative to the centre. Such neural networks have two layers. In the inner layer, the features are combined with the radial basis function.

Then the output of these features is taken into account when calculating the same output in the next time-step. Here is a diagram which represents a radial basis function neural network.



Radial Basis Function Neural Network

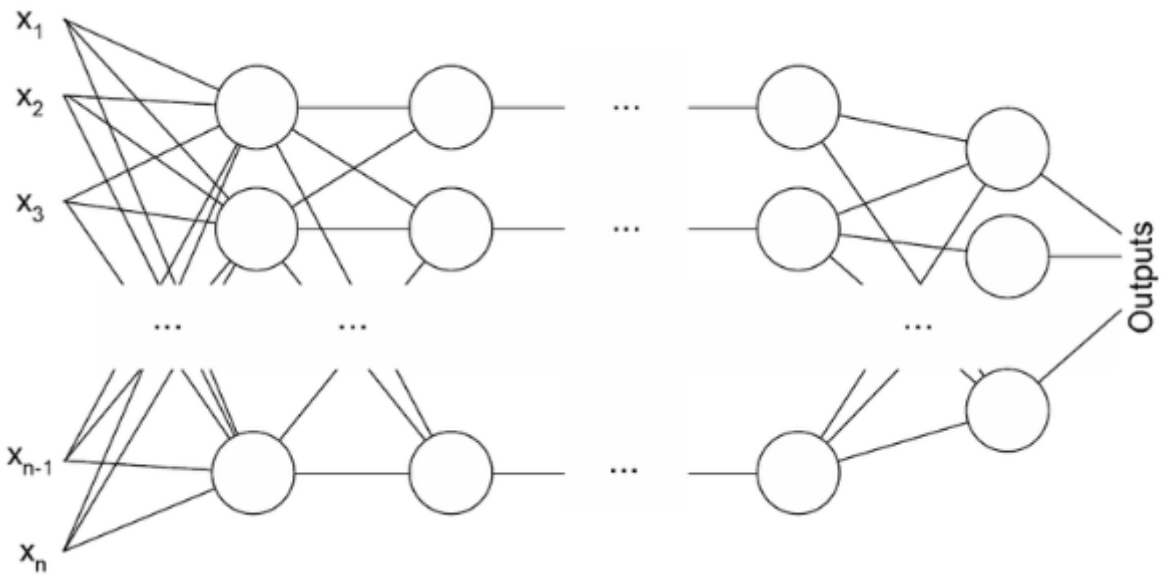
The radial basis function neural network is applied extensively in power restoration systems. In recent decades, power systems have become bigger and more complex.

This increases the risk of a blackout. This neural network is used in the power restoration systems in order to restore power in the shortest possible time.

3. Multilayer Perceptron

A multilayer perceptron has three or more layers. It is used to classify data that cannot be separated linearly. It is a type of artificial neural network that is fully connected. This is because every single node in a layer is connected to each node in the following layer.

A multilayer perceptron uses a nonlinear activation function (mainly hyperbolic tangent or logistic function). Here's what a multilayer perceptron looks like.



Multilayer Perceptron

This type of neural network is applied extensively in speech recognition and machine translation technologies.

4. Convolutional Neural Network

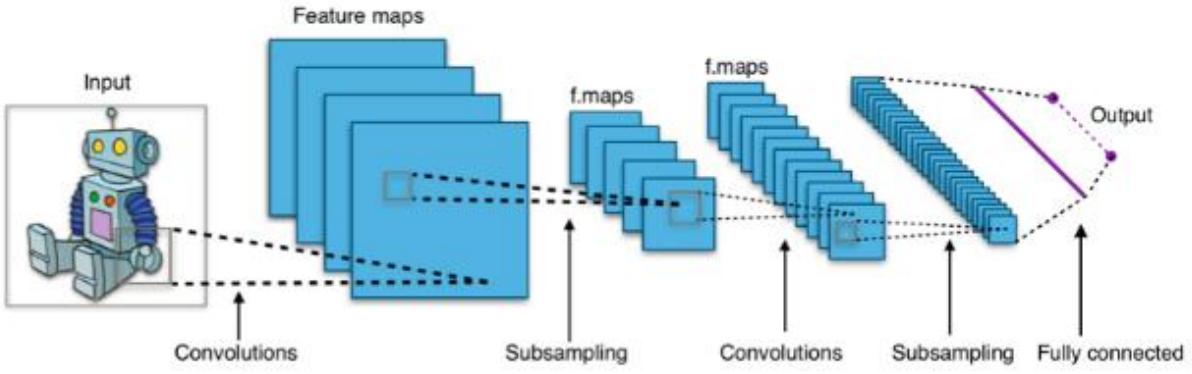
A convolutional neural network(CNN) uses a variation of the multilayer perceptrons. A CNN contains one or more than one convolutional layers. These layers can either be completely interconnected or pooled.

Before passing the result to the next layer, the convolutional layer uses a convolutional operation on the input. Due to this convolutional operation, the network can be much deeper but with much fewer parameters.

Due to this ability, convolutional neural networks show very effective results in image and video recognition, natural language processing, and recommender systems.

Convolutional neural networks also show great results in semantic parsing and paraphrase detection. They are also applied in signal processing and image classification.

CNNs are also being used in image analysis and recognition in agriculture where weather features are extracted from satellites like LSAT to predict the growth and yield of a piece of land. Here’s an image of what a Convolutional Neural Network looks like.



Convolutional Neural Network

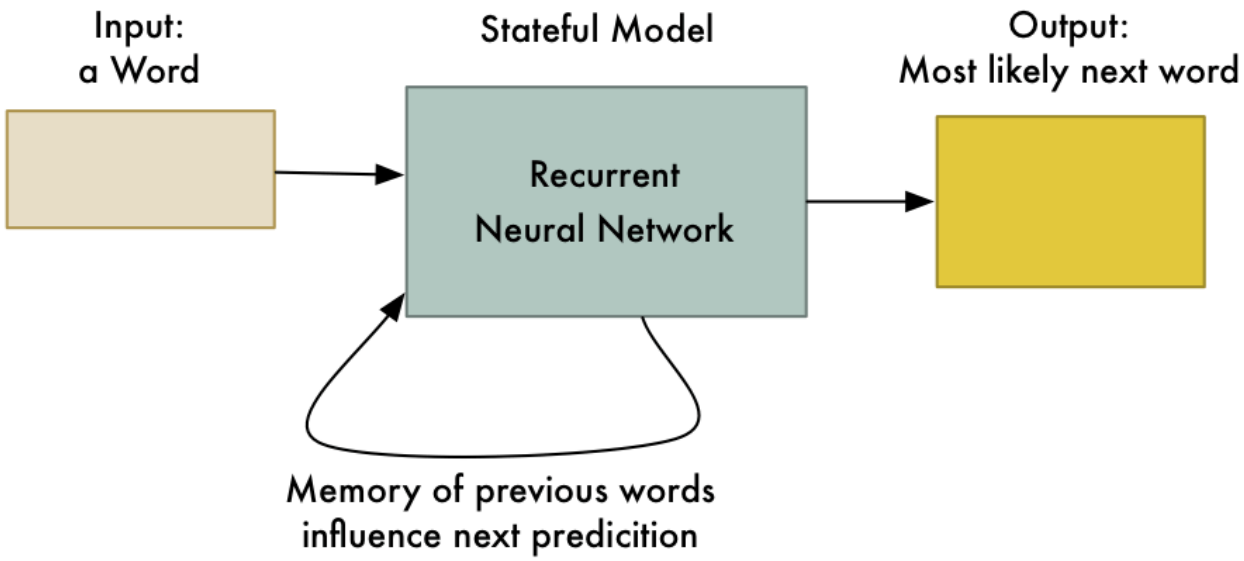
5. Recurrent Neural Network(RNN) – Long Short Term Memory

A Recurrent Neural Network is a type of artificial neural network in which the output of a particular layer is saved and fed back to the input. This helps predict the outcome of the layer.

The first layer is formed in the same way as it is in the feedforward network. That is, with the product of the sum of the weights and features. However, in subsequent layers, the recurrent neural network process begins.

From each time-step to the next, each node will remember some information that it had in the previous time-step. In other words, each node acts as a memory cell while computing and carrying out operations. The neural network begins with the front propagation as usual but remembers the information it may need to use later.

If the prediction is wrong, the system self-learns and works towards making the right prediction during the backpropagation. This type of neural network is very effective in text-to-speech conversion technology. Here's what a recurrent neural network looks like.



Output so far:

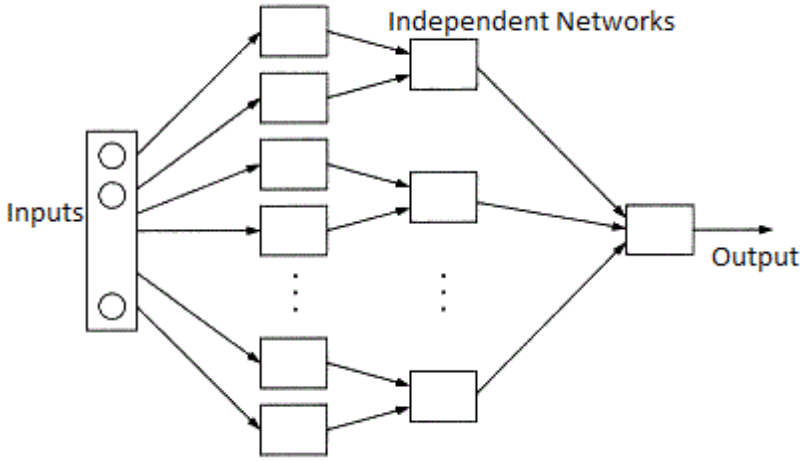
Machine

Recurrent Neural Network(RNN) – Long Short Term Memory

6. Modular Neural Network

A modular neural network has a number of different networks that function independently and perform sub-tasks. The different networks do not really interact with or signal each other during the computation process. They work independently towards achieving the output.

As a result, a large and complex computational process can be done significantly faster by breaking it down into independent components. The computation speed increases because the networks are not interacting with or even connected to each other. Here's a visual representation of a Modular Neural Network.



Modular Neural Network

7. Sequence-To-Sequence Models

A sequence to sequence model consists of two recurrent neural networks. There's an encoder that processes the input and a decoder that processes the output. The encoder and decoder can either use the same or different parameters. This model is particularly applicable in those cases where the length of the input data is not the same as the length of the output data.

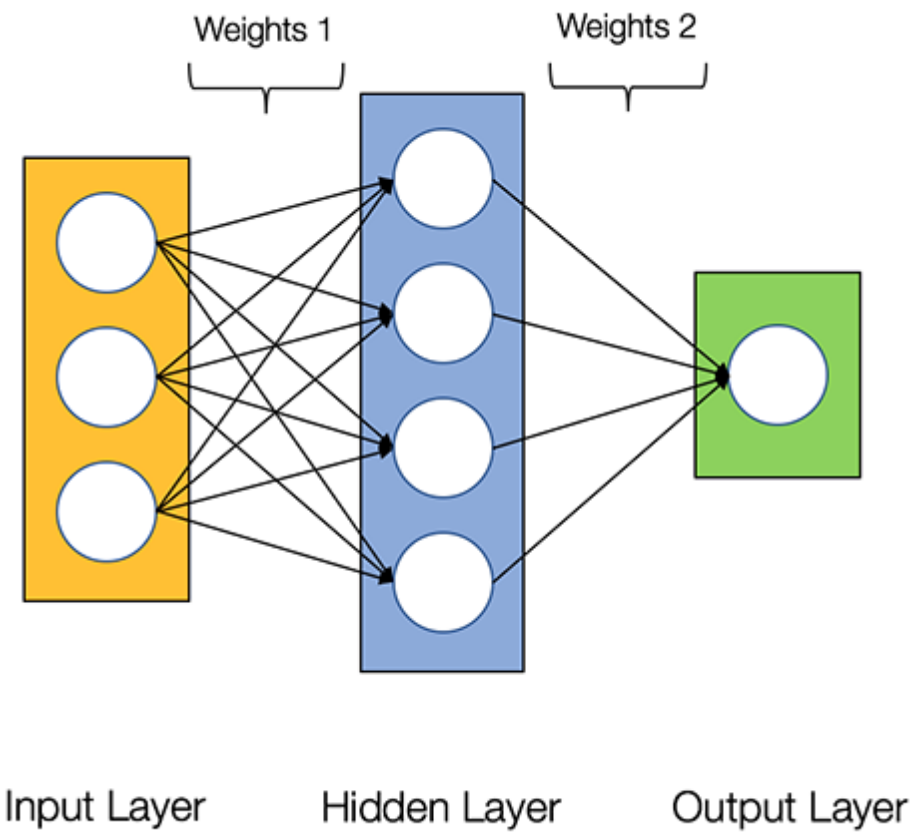
Sequence-to-sequence models are applied mainly in chatbots, machine translation, and question answering systems.

Building your own Neural Network

Neural Networks consist of the following components

- An **input layer**, $\boldsymbol{x}$
- An arbitrary amount of **hidden layers**
- An **output layer**, $\boldsymbol{\hat{y}}$
- A set of **weights** and **biases** between each layer, $\boldsymbol{W}$ ***and*** $\boldsymbol{b}$
- A choice of **activation function** for each hidden layer, $\boldsymbol{\sigma}$. In this tutorial, we'll use a Sigmoid activation function.

The diagram below shows the architecture of a 2-layer Neural Network (*note that the input layer is typically excluded when counting the number of layers in a Neural Network*)



The output $\hat{y}$ of a simple 2-layer Neural Network is:

$$\hat{y} = \sigma(W_2 \sigma(W_1 x + b_1) + b_2)$$

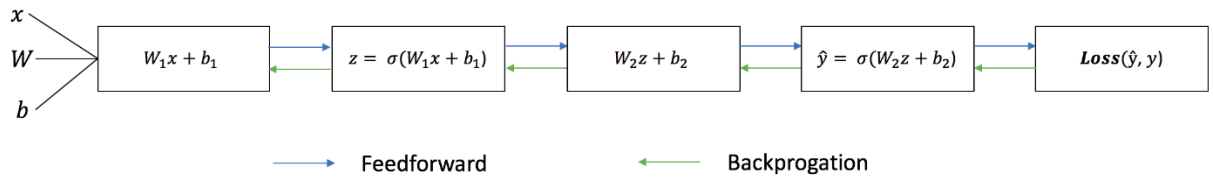
You might notice that in the equation above, the weights W and the biases b are the only variables that affects the output $\hat{y}$.

Naturally, the right values for the weights and biases determines the strength of the predictions. The process of fine-tuning the weights and biases from the input data is known as **training the Neural Network.**

Each iteration of the training process consists of the following steps:

- Calculating the predicted output $\hat{y}$, known as **feedforward**
- Updating the weights and biases, known as **backpropagation**

The sequential graph below illustrates the process.



Feedforward

As we've seen in the sequential graph above, feedforward is just simple calculus and for a basic 2-layer neural network, the output of the Neural Network is:

$$\hat{y} = \sigma(W_2\sigma(W_1x + b_1) + b_2)$$

Let's add a feedforward function in our python code to do exactly that. Note that for simplicity, we have assumed the biases to be 0.

However, we still need a way to evaluate the "goodness" of our predictions (i.e. how far off are our predictions)? The **Loss Function** allows us to do exactly that.

Loss Function

There are many available loss functions, and the nature of our problem should dictate our choice of loss function. In this tutorial, we'll use a simple **sum-of-sqaures error** as our loss function.

$$Sum - of - Squares\ Error = \sum_{i=1}^{n}(y - \hat{y})^2$$

That is, the sum-of-squares error is simply the sum of the difference between each predicted value and the actual value. The difference is squared so that we measure the absolute value of the difference.

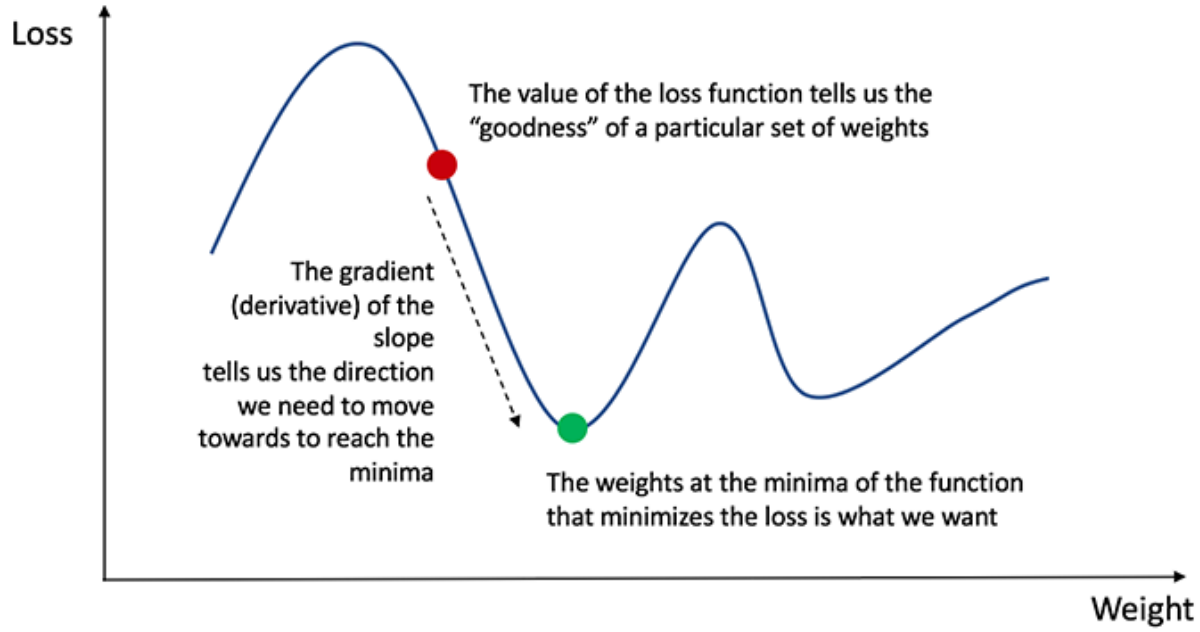
Our goal in training is to find the best set of weights and biases that minimizes the loss function.

Backpropagation

Now that we've measured the error of our prediction (loss), we need to find a way to **propagate** the error back, and to update our weights and biases.

In order to know the appropriate amount to adjust the weights and biases by, we need to know the **derivative of the loss function with respect to the weights and biases**.

Recall from calculus that the derivative of a function is simply the slope of the function.



If we have the derivative, we can simply update the weights and biases by increasing/reducing with it(refer to the diagram above). This is known as **gradient descent**.

However, we can't directly calculate the derivative of the loss function with respect to the weights and biases because the equation of the loss function does not contain the weights and biases. Therefore, we need the **chain rule** to help us calculate it.

$$Loss(y, \hat{y}) = \sum_{i=1}^{n} (y - \hat{y})^2$$

$$\frac{\partial\, Loss(y,\hat{y})}{\partial W} = \frac{\partial Loss(y,\hat{y})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z} * \frac{\partial z}{\partial W} \qquad \text{where } z = Wx + b$$

$$= 2(y - \hat{y}) * \text{derivative of sigmoid function} * x$$

$$= 2(y - \hat{y}) * z(1\text{-}z) * x$$

Chain rule for calculating derivative of the loss function with respect to the weights. Note that for simplicity, we have only displayed the partial derivative assuming a 1-layer Neural Network.

That was ugly but it allows us to get what we needed — the derivative (slope) of the loss function with respect to the weights, so that we can adjust the weights accordingly.

Now that we have that, we can add the backpropagation function into our python code.

Creating a Neural Network in TensorFlow

Create the Model

```
model = tf.keras.models.Sequential([tf.keras.layers.Flatten(),
          tf.keras.layers.Dense(128, activation=tf.nn.relu),
          tf.keras.layers.Dense(10, activation=tf.nn.softmax)])
```

Sequential: That defines a SEQUENCE of layers in the neural network

Flatten: Remember earlier where our images were a square when you printed them out? Flatten just takes that square and turns it into a 1-dimensional set.

Dense: Adds a layer of neurons

Each layer of neurons needs an **activation function** to tell them what to do. There are lots of options, but just use these for now.

Relu effectively means "If X>0 return X, else return 0" -- so what it does it only passes values 0 or greater to the next layer in the network.

Softmax takes a set of values, and effectively picks the biggest one, so,

for example, if the output of the last layer looks like [0.1, 0.1, 0.05, 0.1, 9.5, 0.1, 0.05, 0.05, 0.05], it saves you from fishing through it looking for the biggest value, and turns it into [0,0,0,0,1,0,0,0,0] -- The goal is to save a lot of coding!

Compiling the Model

```
model.compile(optimizer = tf.train.AdamOptimizer(),
              loss = 'sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

To make the network ready for training, we need to pick three more things, as part of the compilation step:

- A loss function—How the network will be able to measure its performance on the training data, and thus how it will be able to steer itself in the right direction.

- An optimizer—The mechanism through which the network will update itself based on the data it sees and its loss function.

- Metrics to monitor during training and testing—Here, we'll only care about accuracy (the fraction of the images that were correctly classified).

Training the Model

```
model.fit(training_images, training_labels, epochs=5)
```

you train it by calling **model.fit ** asking it to fit your training data to your training labels -- i.e. have it figure out the relationship between the training data and its actual labels, so in future, if you have data that looks like the training data, then it can make a prediction for what that data would look like.