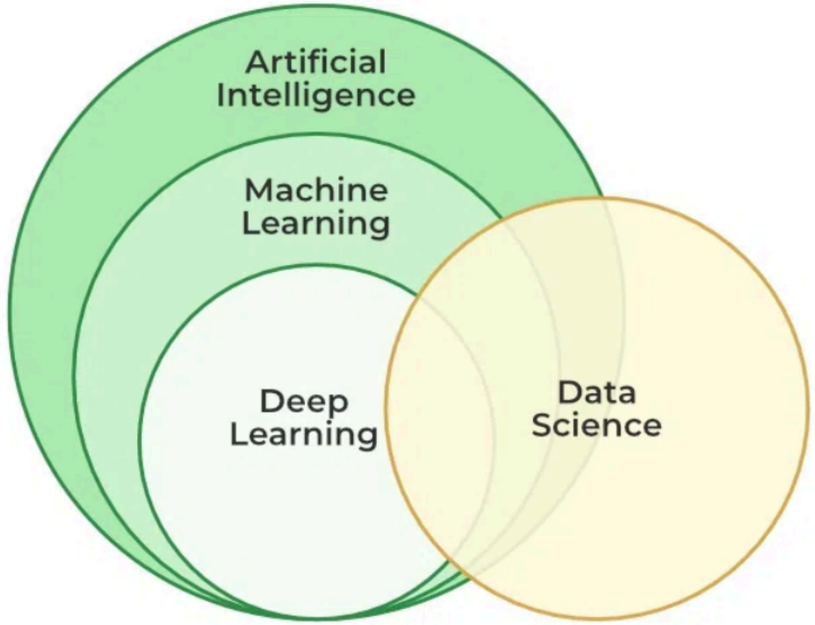
Introduction to Deep Learning

What is Deep Learning?

Deep learning is the branch of machine learning that is based on artificial neural network architecture. An artificial neural network or ANN uses layers of interconnected nodes called neurons that work together to process and learn from the input data.



Scope of Deep Learning

Deep learning can be used for:

- **Supervised Learning:** In this, the neural network learns to make predictions or classify data based on the labeled datasets. Here we input both input features along with the target variables. the neural network learns to make predictions based on the cost or error that comes from the difference between the predicted and the actual target, this process is known as backpropagation. Deep learning algorithms like Convolutional neural networks, Recurrent neural networks are used for many supervised tasks like image classifications and recognization, sentiment analysis, language translations, etc.

- **Unsupervised Learning**: technique in which the neural network learns to discover the patterns or to cluster the dataset based on unlabeled datasets. Here there are no target variables. while the machine has to self-determined the hidden patterns or relationships within the datasets. Deep learning algorithms like autoencoders and generative models are used for unsupervised tasks like clustering, dimensionality reduction, and anomaly detection.

- **Reinforcement Learning:** A technique in which an agent learns to make decisions in an environment to maximize a reward signal. The agent interacts with the environment by taking action and observing the resulting rewards. Deep learning can be used to learn policies, or a set of actions, that maximizes the cumulative reward over time. Deep reinforcement learning algorithms like Deep Q networks and Deep Deterministic Policy Gradient (DDPG) are used to reinforce tasks like robotics and game playing etc.

What is a Neural Network?

Neural networks are machine learning models that mimic the complex functions of the human brain. These models consist of interconnected nodes or neurons that process data, learn patterns, and enable tasks such as pattern recognition and decision-making.

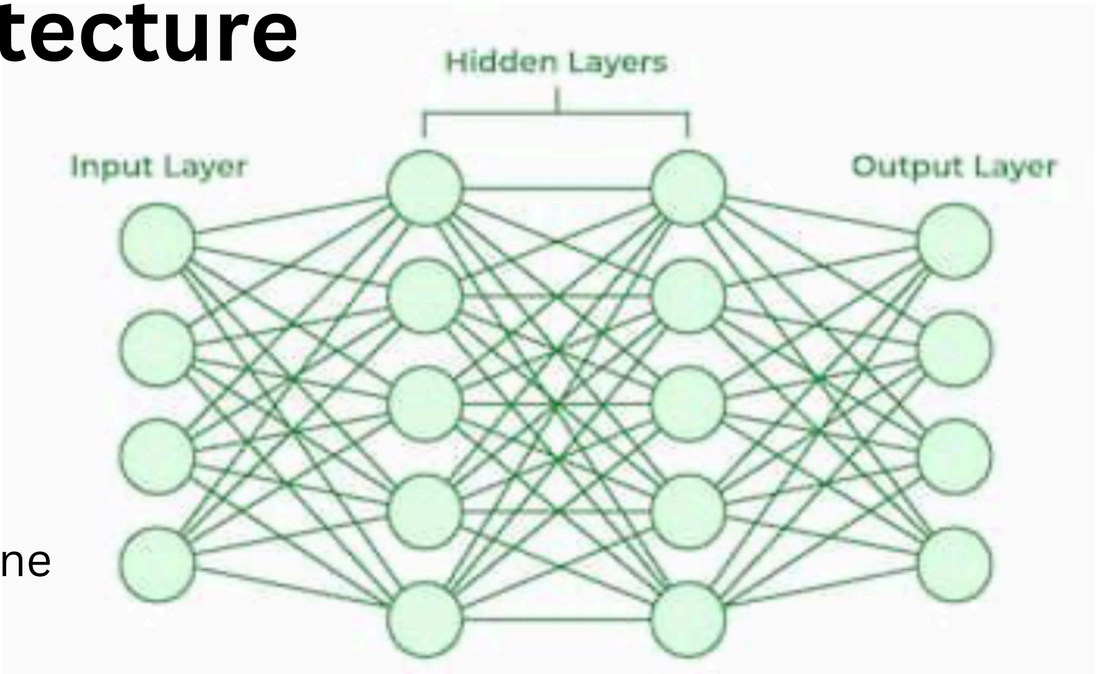
Understanding Neural Networks in Deep Learning

1. **Neurons**: The basic units that receive inputs, each neuron is governed by a threshold and an activation function.
2. **Connections**: Links between neurons that carry information, regulated by weights and biases.
3. **Weights and Biases**: These parameters determine the strength and influence of connections.
4. **Propagation Functions**: Mechanisms that help process and transfer data across layers of neurons.
5. **Learning Rule**: The method that adjusts weights and biases over time to improve accuracy.

Learning in neural networks follows a structured, three-stage process:

1. **Input Computation**: Data is fed into the network.
2. **Output Generation**: Based on the current parameters, the network generates an output.
3. **Iterative Refinement**: The network refines its output by adjusting weights and biases, gradually improving its performance on diverse tasks.

Layers in Neural Network Architecture



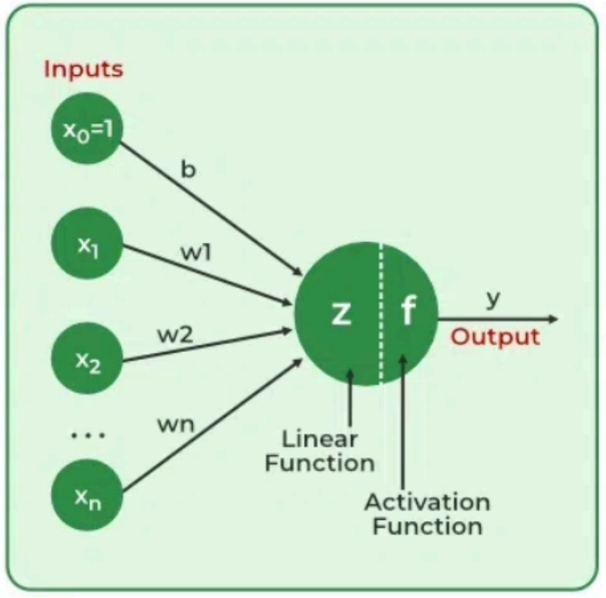
1. **Input Layer**: This is where the network receives its input data. Each input neuron in the layer corresponds to a feature in the input data.
2. **Hidden Layers**: These layers perform most of the computational heavy lifting. A neural network can have one or multiple hidden layers. Each layer consists of units (neurons) that transform the inputs into something that the output layer can use.
3. **Output Layer**: The final layer produces the output of the model. The format of these outputs varies depending on the specific task (e.g., classification, regression).

Working of Neural Networks

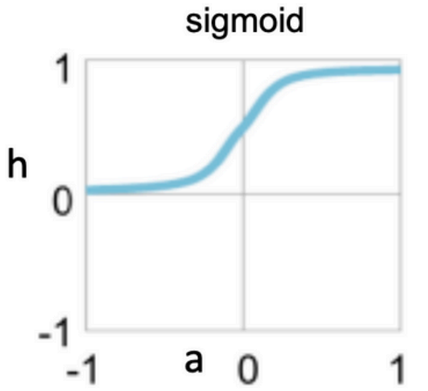
Forward Propagation

When data is input into the network, it passes through the network in the forward direction, from the input layer through the hidden layers to the output layer. This process is known as forward propagation. Here's what happens during this phase:

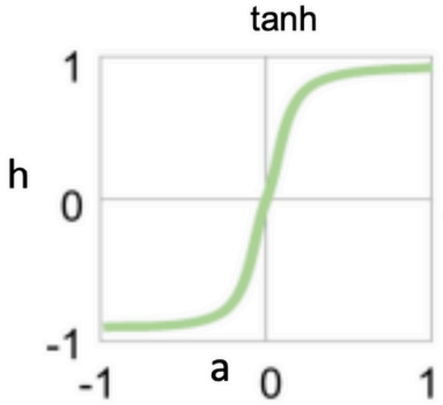
1. **Linear Transformation:** Each neuron in a layer receives inputs, which are multiplied by the weights associated with the connections. These products are summed together, and a bias is added to the sum. This can be represented mathematically as: $z = w_1x_1 + w_2x_2 + \ldots + w_nx_n + b$ where w represents the weights, x represents the inputs, and b is the bias.
2. **Activation:** The result of the linear transformation (denoted as z) is then passed through an activation function. The activation function is crucial because it introduces non-linearity into system, enabling the network to learn more complex patterns. Popular activation functions include ReLU, sigmoid, and tanh.



Activation Functions: Some Common Choices



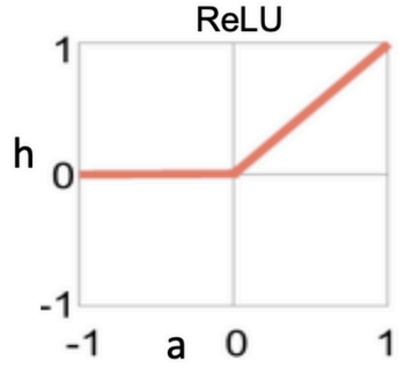
For sigmoid as well as tanh, gradients saturate (become close to zero as the function tends to its extreme values)



Preferred more than sigmoid. Helps keep the mean of the next layer's inputs close to zero (with sigmoid, it is close to 0.5)

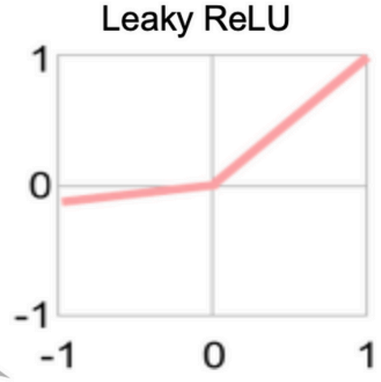
Sigmoid: $h = \sigma(a) = \frac{1}{1+\exp(-a)}$

tanh (tan hyperbolic): $h = \frac{\exp(a)-\exp(-a)}{\exp(a)+\exp(-a)} = 2\sigma(2a) - 1$



ReLU and Leaky ReLU are among the most popular ones (also efficient to compute)

Helps fix the dead neuron problem of ReLU when $\boldsymbol{a}$ is a negative number



$y = \boldsymbol{v}^\top(g(\boldsymbol{W}^\top x)) = \boldsymbol{v}^\top \boldsymbol{W}^\top x$

Still linear

Imp: Without nonlinear activation, a deep neural network is equivalent to a linear model no matter how many layers we use

Most activation functions are monotonic but there exist some non-monotonic activation functions as well (e.g., Swish: $\boldsymbol{a} \times \sigma(\boldsymbol{\beta a})$)

ReLU (Rectified Linear Unit): $h = \max(0, a)$

Leaky ReLU: $h = \max(\beta a, a)$
where β is a small postive number

Backpropagation

After forward propagation, the network evaluates its performance using a loss function, which measures the difference between the actual output and the predicted output. The goal of training is to minimize this loss. This is where backpropagation comes into play:

1. **Loss Calculation:** The network calculates the loss, which provides a measure of error in the predictions. The loss function could vary; common choices are mean squared error for regression tasks or cross-entropy loss for classification.
2. **Gradient Calculation:** The network computes the gradients of the loss function with respect to each weight and bias in the network. This involves applying the chain rule of calculus to find out how much each part of the output error can be attributed to each weight and bias.
3. **Weight Update:** Once the gradients are calculated, the weights and biases are updated using an optimization algorithm like stochastic gradient descent (SGD). The weights are adjusted in the opposite direction of the gradient to minimize the loss. The size of the step taken in each update is determined by the learning rate.

Iteration

This process of forward propagation, loss calculation, backpropagation, and weight update is repeated for many iterations over the dataset. Over time, this iterative process reduces the loss, and the network’s predictions become more accurate.

Through these steps, neural networks can adapt their parameters to better approximate the relationships in the data, thereby improving their performance on tasks such as classification, regression, or any other predictive modeling.

TYPES OF NEURAL NETWORKS

- **Feedforward Networks:** A feedforward neural network is a simple artificial neural network architecture in which data moves from input to output in a single direction.
- **Multilayer Perceptron (MLP):** MLP is a type of feedforward neural network with three or more layers, including an input layer, one or more hidden layers, and an output layer. It uses nonlinear activation functions.
- **Convolutional Neural Network (CNN):** A Convolutional Neural Network (CNN) is a specialized artificial neural network designed for image processing. It employs convolutional layers to automatically learn hierarchical features from input images, enabling effective image recognition and classification.
- **Recurrent Neural Network (RNN):** An artificial neural network type intended for sequential data processing is called a Recurrent Neural Network (RNN). It is appropriate for applications where contextual dependencies are critical, such as time series prediction and natural language processing, since it makes use of feedback loops, which enable information to survive within the network.
- **Long Short-Term Memory (LSTM):** LSTM is a type of RNN that is designed to overcome the vanishing gradient problem in training RNNs. It uses memory cells and gates to selectively read, write, and erase information.

Example of Email Classification

Let's consider a record of an email dataset:

Email ID	Email Content	Sender	Subject Line	Label
1	"Get free gift cards now!"	spam@example.com	"Exclusive Offer"	1

To classify this email, we will create a feature vector based on the analysis of keywords such as "free," "win," and "offer."

The feature vector of the record can be presented as:

- "free": Present (1)
- "win": Absent (0)
- "offer": Present (1)

Email ID	Email Content	Sender	Subject Line	Feature Vector	Label
1	"Get free gift cards now!"	spam@example.com	"Exclusive Offer"	[1, 0, 1]	1

1. Input Layer: The input layer contains 3 nodes that indicates the presence of each keyword.

2. Hidden Layer

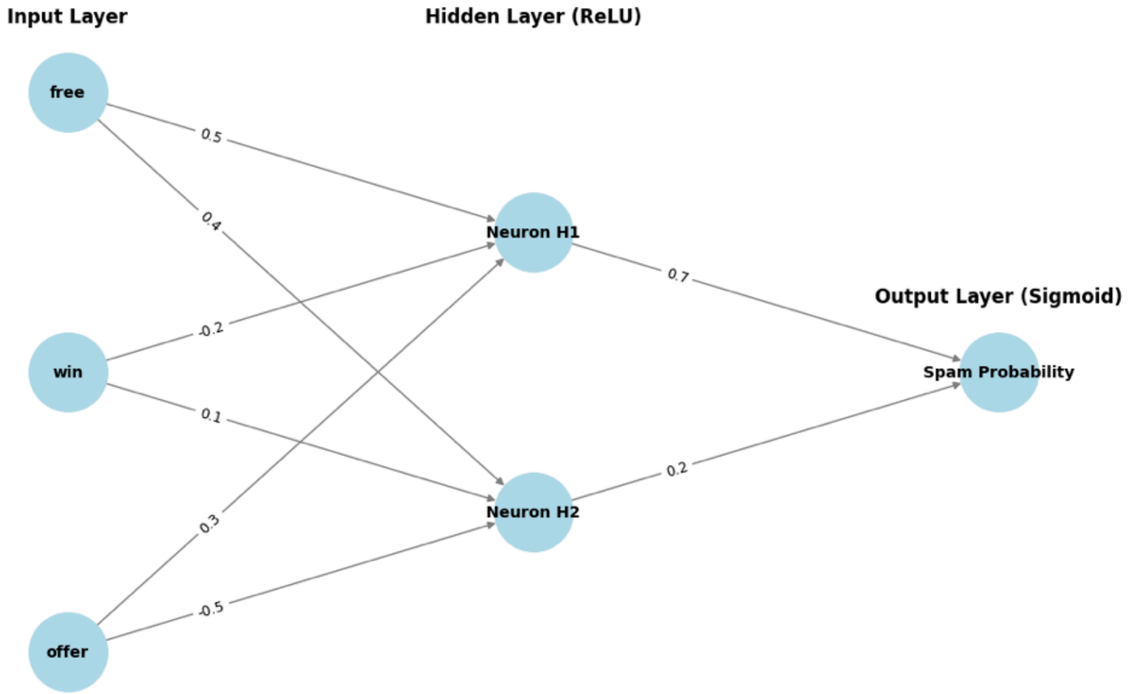
- The input data is passed through one or more hidden layers.
- Each neuron in the hidden layer performs the following operations:
  1. **Weighted Sum**: Each input is multiplied by a corresponding weight assigned to the connection. For example, if the weights from the input layer to the hidden layer neurons are as follows:
     - Weights for Neuron H1: `[0.5, -0.2, 0.3]`
     - Weights for Neuron H2: `[0.4, 0.1, -0.5]`
  2. **Calculate Weighted Input**:
     - For Neuron H1:
       - Calculation $= (1 \times 0.5) + (0 \times -0.2) + (1 \times 0.3) = 0.5 + 0 + 0.3 = 0.8$
     - For Neuron H2:
       - Calculation $= (1 \times 0.4) + (0 \times 0.1) + (1 \times -0.5) = 0.4 + 0 - 0.5 = -0.1$
  3. **Activation Function**: The result is passed through an activation function (e.g., ReLU or sigmoid) to introduce non-linearity.
     - For H1, applying ReLU: $\text{ReLU}(0.8) = 0.8$
     - For H2, applying ReLU: $\text{ReLU}(-0.1) = 0$

3. Output Layer

- The activated outputs from the hidden layer are passed to the output neuron.
- The output neuron receives the values from the hidden layer neurons and computes the final prediction using weights:
    - Suppose the output weights from hidden layer to output neuron are `[0.7, 0.2]`.
    - Calculation:
        - Input $= (0.8 \times 0.7) + (0 \times 0.2) = 0.56 + 0 = 0.56$
    - **Final Activation**: The output is passed through a sigmoid activation function to obtain a probability:
        - $\sigma(0.56) \approx 0.636$

4. Final Classification

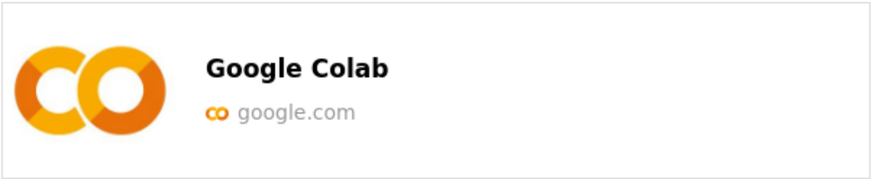
- The output value of approximately **0.636** indicates the probability of the email being spam.
- Since this value is greater than 0.5, the neural network classifies the email as spam (1).



Neural Network for Email Classification Example

Here is a Neural Network Model implementation

https://colab.research.google.com/drive/1pGc_54ypgoMg0onpQpysyv5cb8RDJhxt?authuser=1#scrollTo=7MZ6Py5HareO



Loss Functions and Optimizers

When compiling your model you need to choose a loss function and an optimizer. The loss function is the quantity that will be minimized during training. The optimizer determines how the network will be updated based on the loss function.

Example compile step:

```
model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['accuracy'])
```

Loss Functions

There are some simple guidlines for choosing the correct loss function:

binary crossentropy (`binary_crossentropy`) is used when you have a two-class, or binary, classification problem.

categorical crossentropy (`categorical_crossentropy`) is used for a multi-class classification problem.

mean squared error (`mean_squared_error`) is used for a regression problem.

In general, crossentropy loss functions are best to use when the model you use is outputting probabilities.

1.Mean Squared Error

The Mean Squared Error (MSE) Loss is one of the most widely used loss functions for regression tasks. It calculates the average of the squared differences between the predicted values and the actual values.

$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$

2. Binary Cross Entropy

Binary Cross-Entropy Loss, also known as Log Loss, is used for binary classification problems. It measures the performance of a classification model whose output is a probability value between 0 and 1.

$\text{Binary Cross-Entropy} = -\frac{1}{n}\sum_{i=1}^{n}[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)]$

where n is the number of data points, y_i is the actual binary label (0 or 1), and $\hat{y}_i$ is the predicted probability.

3. Categorical Cross Entropy

Categorical Cross-Entropy Loss is used for multiclass classification problems. It measures the performance of a classification model whose output is a probability distribution over multiple classes.

$\text{Categorical Cross-Entropy} = -\sum_{i=1}^{n}\sum_{j=1}^{k} y_{ij}\log(\hat{y}_{ij})$

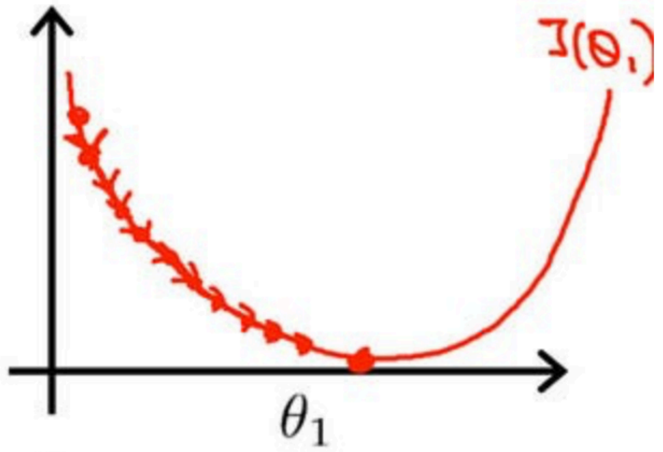
where n is the number of data points, k is the number of classes, y_{ij} is the binary indicator (0 or 1) if class label j is the correct classification for data point i, and $\hat{y}_{ij}$ is the predicted probability for class j.

Optimizers

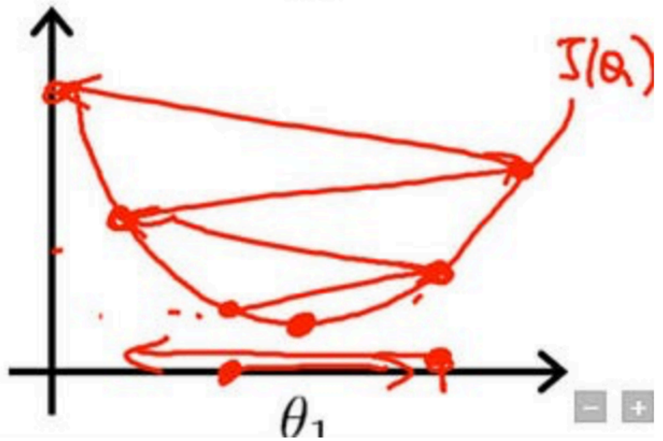
There are many optimizers you can use and many are a variant of stochastic gradient descent. For all of them you will be able to tune the **learning rate** parameter. The learning rate parameter tells the optimizer how far to move the weights of the layer in the direction opposite of the gradient. This parameter is very important, if it is too high then the training of the model may never converge. If it is too low, then the training is more relibable but very slow. It is best to try out multiple different learning rates to find which one is best.

$$\theta_1 := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_1)$$

If α is too small, gradient descent can be slow.



If α is too large, gradient descent can overshoot the minimum. It may fail to converge, or even diverge.



Stochastic Gradient Descent

```
keras.optimizers.SGD(lr=0.01, momentum=0.0, decay=0.0, nesterov=False)
```

This is a common 'basic' optimizer and many optimizers are variants of this. It can be adjusted by changing the learning rate, momentum and decay.

- learning rate (lr)
- momentum - accelerates SGD in the relevant direction and dampens oscillations. Basically it helps SGD push past local optima, gaining faster convergence and less oscillation. A typical choice of momentum is between 0.5 to 0.9.
- decay - you can set a decay function for the learning rate. This will adjust the learning rate as training progresses.
- nesterov - Nesterov momentum is a different version of the momentum method which has stronger theoretical converge guarantees for convex functions. In practice, it works slightly better than standard momentum

Adaptive learning rate optimizers

The following optimizers use a heuristic approach to tune some parameters automatically. Descriptions are mostly from the Keras documentation.

Adagrad

```
keras.optimizers.Adagrad(lr=0.01, epsilon=None, decay=0.0)
```

Adagrad is an optimizer with parameter-specific learning rates, which are adapted relative to how frequently a parameter gets updated during training. The more updates a parameter receives, the smaller the updates.

Keras recommends that you use the default parameters.

Adadelta

```
keras.optimizers.Adadelta(lr=1.0, rho=0.95, epsilon=None, decay=0.0)
```

Adadelta is a more robust extension of Adagrad that adapts learning rates based on a moving window of gradient updates, instead of accumulating all past gradients. This way, Adadelta continues learning even when many updates have been done. Compared to Adagrad, in the original version of Adadelta you don't have to set an initial learning rate. In this version, initial learning rate and decay factor can be set, as in most other Keras optimizers.

Keras recommends that you use the default parameters.

RMSprop

```
keras.optimizers.RMSprop(lr=0.001, rho=0.9, epsilon=None, decay=0.0)
```

RMSprop is similar to Adadelta and adjusts the Adagrad method in a very simple way in an attempt to reduce its aggressive, monotonically decreasing learning rate.

Keras recommends that you only adjust the learning rate of this optimzer.

Adam

```
keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999, epsilon=None, decay=0.0, amsgrad=False)
```

Adam is an update to the RMSProp optimizer. It is basically RMSprop with momentum.

Keras recommends that you use the default parameters.

The loss functions, metrics, and optimizers can be customized and configured like so:

Example implementation of optimizer, loss function, metrics

The loss functions, metrics, and optimizers can be customized and configured like so:

```
from keras import optimizers
from keras import losses
from keras import metrics

model.compile(optimizer=optimizers.RMSprop(lr=0.001), loss=losses.binary_crossentropy, metrics=[metrics.binary_accuracy])

#OR

loss = losses.binary_crossentropy
rmsprop = optimizers.RMSprop(lr=0.001)

model.compile(optimizer=rmsprop, loss=loss, metrics=[metrics.binary_accuracy])
```

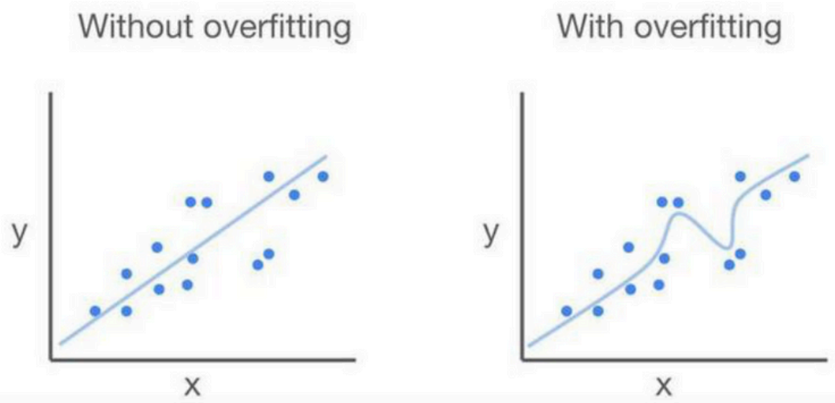
The Concept of Overfitting and Regularization

Overfitting in Machine Learning

In Machine learning, there is a term called **train data** and **test data** which machine learning model will learn from train data and try to predict the test data based on its learning. Overfitting is a concept in machine learning which states a common problem that occurs when a model **learns the train data too well** including the noisy data, resulting in **poor generalization performance on test data**. Overfit models don't generalize, which is the ability to apply knowledge to different situations.

Let's walk through an example of overfitting using the linear regression algorithm,

Suppose we are training a linear regression model to predict the price of a house based on its square feet and with few specifications. We collect a dataset of houses with their square feet and sale price. We then train our linear regression model on this dataset. Generally in linear regression algorithms, it draws a straight that best fits the data points by minimizing the difference between predicted and actual values. The goal is to make a straight line that captures the main pattern in the dataset . This way, it can predict new points more accurately. But sometimes we come across overfitting in linear regression as bending that straight line to fit exactly with a few points on the pattern which is shown below fig.1. This might look perfect for those points while training but doesn't work well for other parts of the pattern when come to model testing.

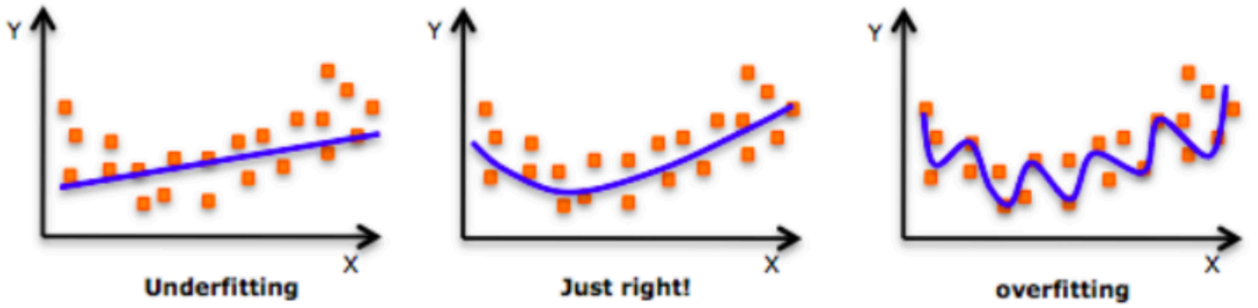


What is Regularization?

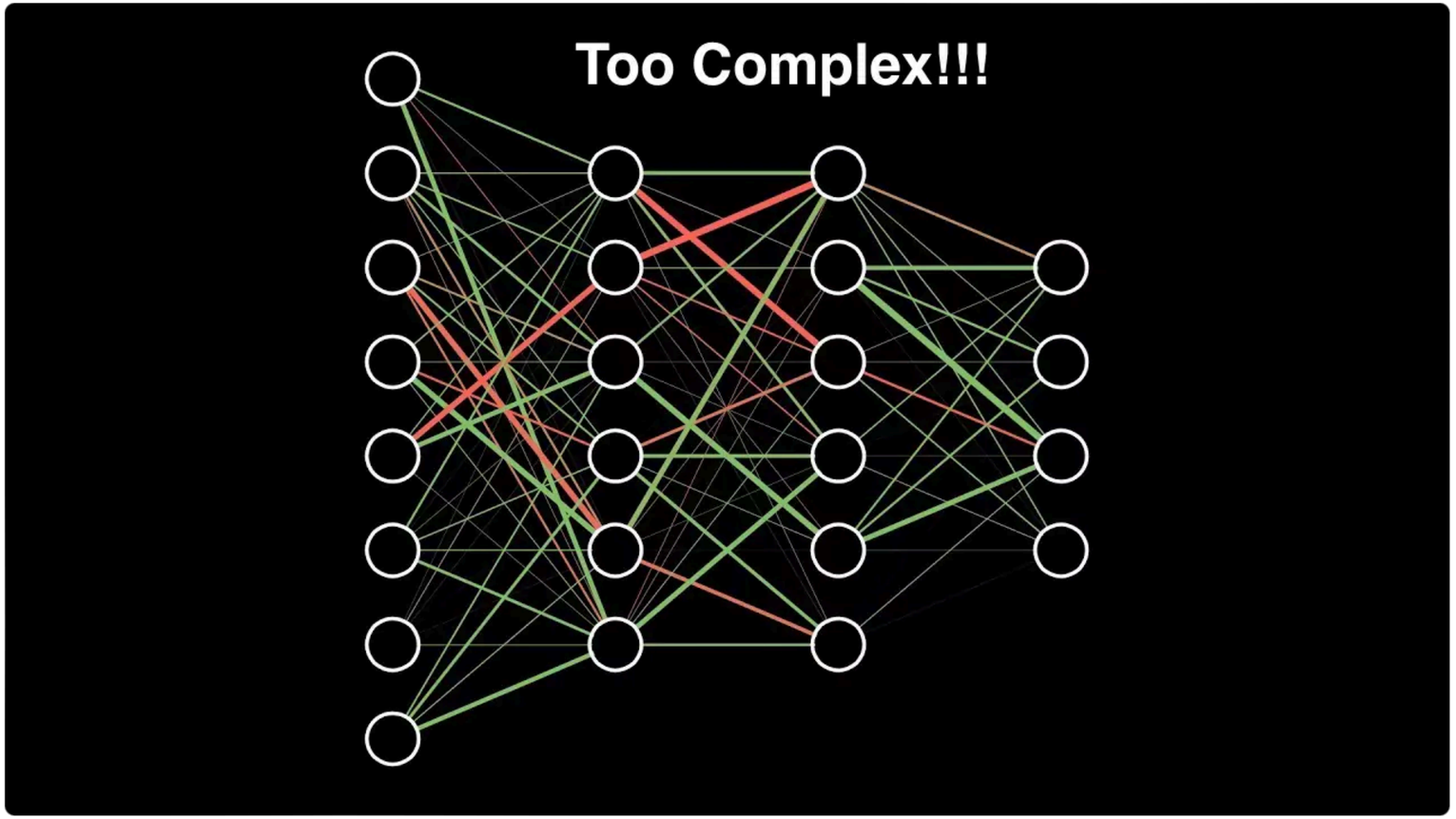
Regularization is a technique used in machine learning and deep learning to prevent overfitting and improve a model's generalization performance. It involves adding a penalty term to the **loss function** during training.

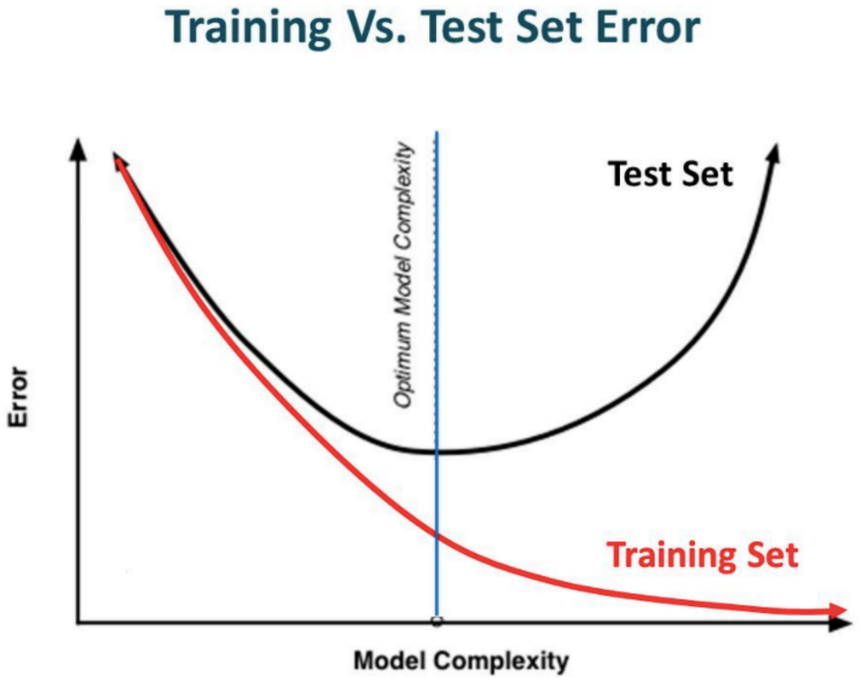
This penalty discourages the model from becoming too complex or having large parameter values, which helps in controlling the model's ability to fit noise in the training data. Regularization in deep learning **methods** includes L1 and L2 regularization, dropout, early stopping, and more. By applying regularization for deep learning, models become more robust and better at making accurate predictions on unseen data.

Before we deep dive into the topic, take a look at this image:



If you've built a neural network before, you know how complex they are. This makes them more prone to overfitting.





Different Regularization Techniques in Deep Learning

L2&L1 Regularization

L1 and L2 are the most common types of regularization deep learning. 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 also reduce overfitting to quite an extent.

However, this regularization term differs in L1 and L2.

For L2:

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

For L1:

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

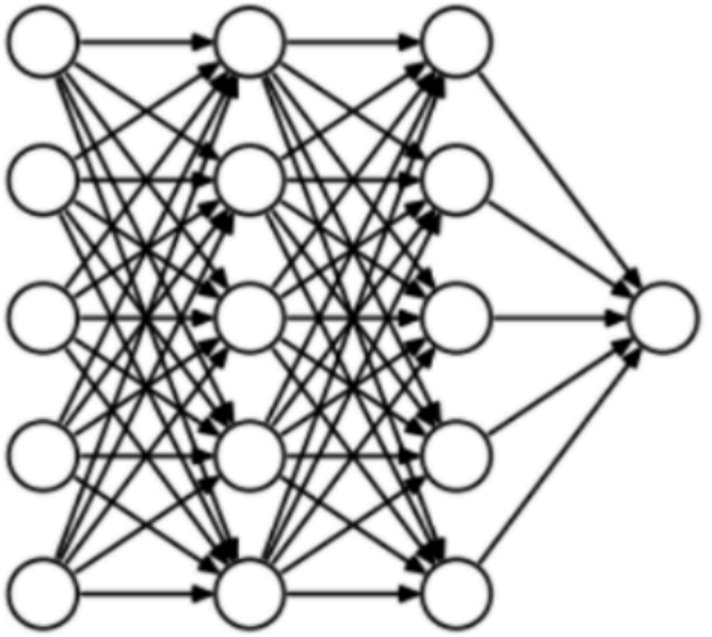
Copy Code

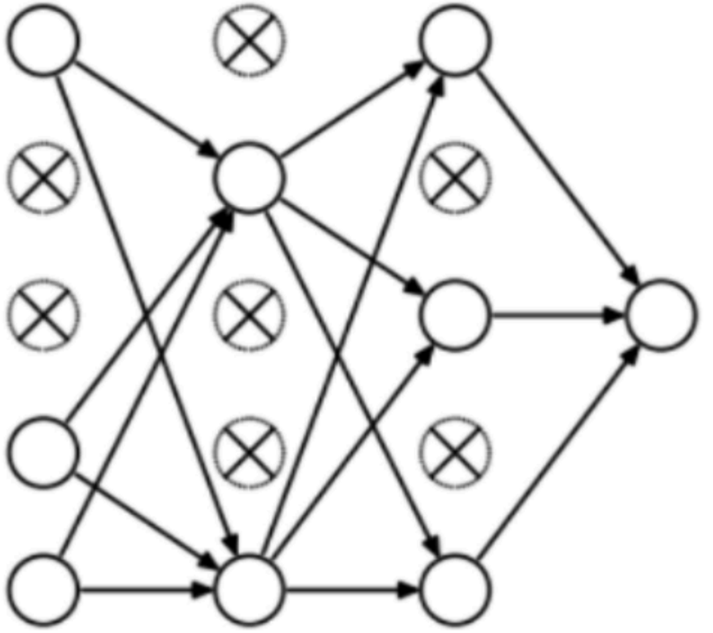
```
from keras import regularizers

model.add(Dense(64, input_dim=64,
                kernel_regularizer=regularizers.l2(0.01)))
```

Dropout

This is one of the most interesting types of **regularization techniques**. It also produces very good results and is consequently the most frequently used regularization technique in the field of deep learning.

So what does dropout do? At every iteration, it randomly selects some nodes and removes them along with all of their incoming and outgoing connections as shown below:





In **Keras**, we can implement dropout using the **Keras core layer**. Below is the Python code for it:

```
from keras.layers.core import Dropout

model = Sequential([
 Dense(output_dim=hidden1_num_units, input_dim=input_num_units, activation='relu'),
 Dropout(0.25),

Dense(output_dim=output_num_units, input_dim=hidden5_num_units, activation='softmax'),
 ])
```

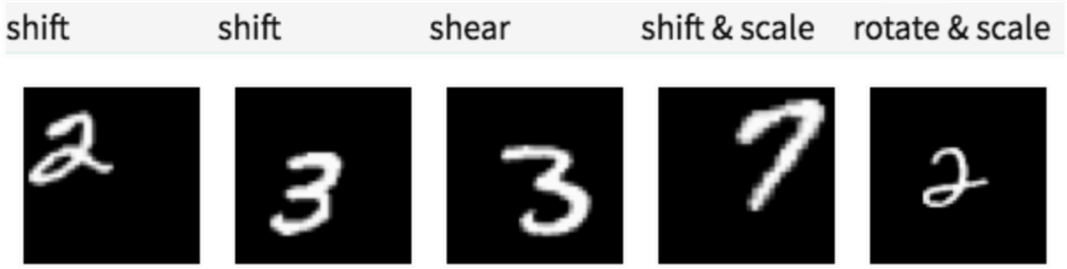
Copy Code

As you can see, we have defined 0.25 as the probability of dropping. We can tune it further for better results using the grid search method.

Data Augmentation

The simplest way to reduce overfitting is to increase the training data size. In machine learning, however, increasing the training data size was impossible as the labeled data was too costly.

But now, let's consider we are dealing with images. In this case, there are a few ways of increasing the size of the training data—rotating the image, flipping, scaling, shifting, etc. In the image below, some transformation has been done on the handwritten digits dataset.



This technique is known as data augmentation. It usually provides a big leap in improving the accuracy of the model, and it can be considered a mandatory trick to improve our predictions.

In *keras*, we can perform all of these transformations using **ImageDataGenerator**. It has a big list of arguments that you can use to pre-process your training data.

Below is the sample code to implement it:

Copy Code

```
from keras.preprocessing.image import ImageDataGenerator
datagen = ImageDataGenerator(horizontal flip=True)
datagen.fit(train)
```

Early Stopping

Early stopping is a **cross-validation** strategy in which we keep one part of the training set as the validation set. When we see that the performance on the validation set is getting worse, we immediately stop the training on the model.

In the above image, we will stop training at the dotted line since, after that, our model will start overfitting on the training data.

In *keras*, we can apply early stopping using the **callbacks** function. Below is the sample code for it.

```
from keras.callbacks import EarlyStopping

EarlyStopping(monitor='val_err', patience=5)
```

Here, monitor refers to the quantity that you need to keep track of, and '**val_err**' refers to the validation error.

Patience denotes the number of epochs with no further improvement, after which training stops. For a better understanding, let's look at the above image again. After the dotted line, each epoch will result in a higher validation error value. Therefore, our model will stop 5 epochs after the dotted line (since our patience equals 5) because it sees no further improvement

Note: *After 5 epochs (the value generally defined for patience), the model might start improving again, and the validation error may decrease. Therefore, we need to take extra care while tuning this hyperparameter.*