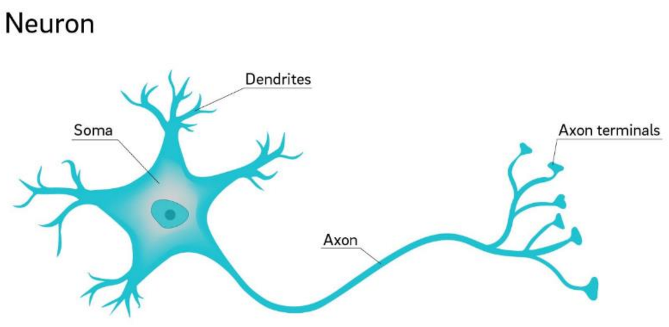
Unit-1

Biological Neural Networks:



1. **Terminologies:**

   a. **Dendrite:** Receives signal from neuron.
   b. **Soma:** Process information.
   c. **Axon:** Transmits the output of this neuron.
   d. **Synapse:** Point of connection to other neuron.

2. **Working:**

   - In living organisms, the brain is the control unit of the neural network, and it has different subunits that take care of vision, senses, movement, and hearing.
   - The neuron is the fundamental building block of neural networks.
   - In the biological systems, a neuron is a cell just like any other cell of the body, which has a DNA code and is generated in the same way as the other cells.
   - From above terminologies, dendrites receive the signals from surrounding neurons, and the axon transmits the signal to the other neurons.
   - At the ending terminal of the axon, the contact with the dendrite is made through a synapse.
   - Axon is a long fibre that transports the output signal as electric impulses along its length.
   - Each neuron has one axon. Axons pass impulses from one neuron to another like a domino effect.

Artificial Neural Network

- It is a type of neural network that is based on feed-forward strategy.
- It is called this because they pass information. through the nodes continuously till it reaches the output node.
- This is also known as simplest type of neural network.
- The ANN structure consist of input, weight, output, and hidden layer.

1) **Advantages:**
    - Ability to learn irrespective of type of data (linear or non-linear).
    - ANN is highly volatile and serves best in financial time series forecasting.
2) **Disadvantages:**
    - The simplest architecture make it difficult to explain behaviour of the network.
    - This network is dependent on hardware.

ANN VS BNN:

Parameters	ANN	BNN
Structure	input weight output hidden layer	dendrites synapse axon cell body
Learning	very precise structures and formatted data	they can tolerate ambiguity
Processor	complex high speed one or a few	simple low speed large number
Computing	centralized sequential stored programs	distributed parallel self-learning
Reliability	very vulnerable	robust

Traditional ML VS Deep Learning

Traditional Machine Learning	Deep Learning
Low hardware requirements on the computer: Given the limited computing amount, the computer does not need a GPU for parallel computing generally.	Higher hardware requirements on the computer: To execute matrix operations on massive data, the computer needs a GPU to perform parallel computing.
Applicable to training under a small data amount and whose performance cannot be improved continuously as the data amount increases.	The performance can be high when high-dimensional weight parameters and massive training data are provided.
Level-by-level problem breakdown	E2E learning
Manual feature selection	Algorithm-based automatic feature extraction
Easy-to-explain features	Hard-to-explain features

History of Deep Learning

• **1943:** McCulloch and Pitts proposed a model of a neuron.
• **1960s:** Widrow and Hoff explored Perceptron networks (which they called "Adelines") and the delta rule.
• **1962:** Rosenblatt proved the convergence of the perceptron training rule.
• **1969:** Minsky and Papert showed that the Perceptron cannot deal with nonlinearly separable data sets---even those that represent simple function such as X-OR.
• **1970-1985**: Very little research on Neural Nets.
• **1986:** Invention of Backpropagation [Rumelhart and McClelland, but also Parker and earlier on: Werbos] which can learn from nonlinearly-separable data sets.
• **Since 1985:** A lot of research in Neural Nets.

Perceptron model

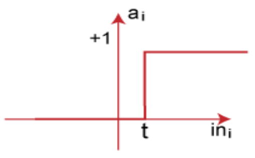
- Perceptron is Machine Learning algorithm for supervised learning of various binary classification tasks.
- A single neuron, the perceptron model detects whether any function is an input or not and classifies then in either of the classes.
- The model uses a hyperplane line that classifies two inputs and classifies then on the basis of the 2 classes that machine leams, thus implying that perceptron model is linear classification model.

The perception model consist of:

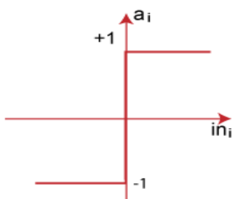
1. **Input Nodes or Input Layer:**
    - This is the primary component of Perceptron which accepts the initial data into the system for further processing.
    - Each input node contains a real numerical value.
2. **Wight and Bias:**
    - Weight parameter represents the strength of the connection between units.
    - This is another most important parameter of Perceptron components. Weight is directly proportional to the strength of the associated input neuron in deciding the output.
    - Further, Bias can be considered as the line of intercept in a linear equation.
3. **Activation Function:**
    - These are the final and important components that help to determine whether the neuron will fire or not.
    - Activation Function can be considered primarily as a step function.

    Types of Activation functions:

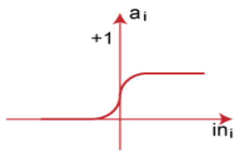
    - Sign function
    - Step function, and
    - Sigmoid function



Step Function



Sign Function



Sigmoid Function

Types of Perceptron Models

Based on the layers, Perceptron models are divided into two types. These are as follows:

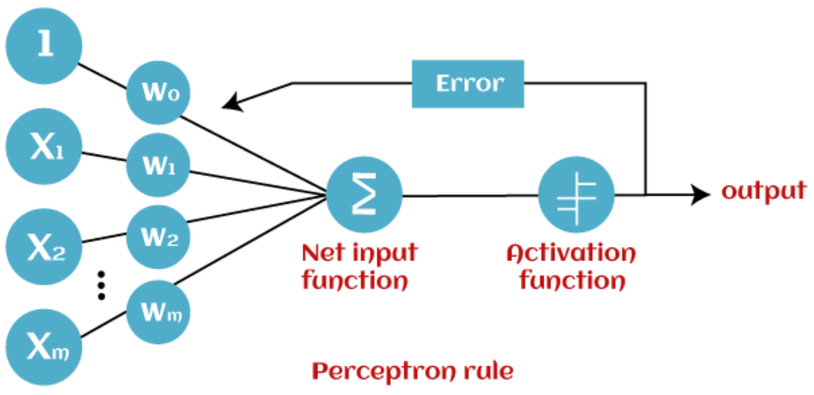
1. Single-layer Perceptron Model
2. Multi-layer Perceptron model

1. Single Layer Perceptron Model:

- This is one of the easiest Artificial neural networks (ANN) types.
- A single-layered perceptron model consists feed-forward network and also includes a threshold transfer function inside the model.
- The main objective of the single-layer perceptron model is to analyse the linearly separable objects with binary outcomes.
- In a single layer perceptron model, its algorithms do not contain recorded data, so it begins with inconstantly allocated input for weight parameters.
- Further, it sums up all inputs (weight). After adding all inputs, if the total sum of all inputs is more than a pre-determined value, the model gets activated and shows the output value as +1.
- If the outcome is same as pre-determined or threshold value, then the performance of this model is stated as satisfied, and weight demand does not change.

$$\begin{aligned} y &= 1 \quad if \sum_{i=0}^{n} w_i * x_i \geq 0 \\ &= 0 \quad if \sum_{i=0}^{n} w_i * x_i < 0 \end{aligned}$$

- It can only learn linearly separable patterns.



2. Multi-Layered Perceptron Model:

- A multi-layer perceptron model also has the same model structure but has a greater number of hidden layers.
- The multi-layer perceptron model is also known as the Backpropagation algorithm, which executes in two stages as follows:

Forward Stage:

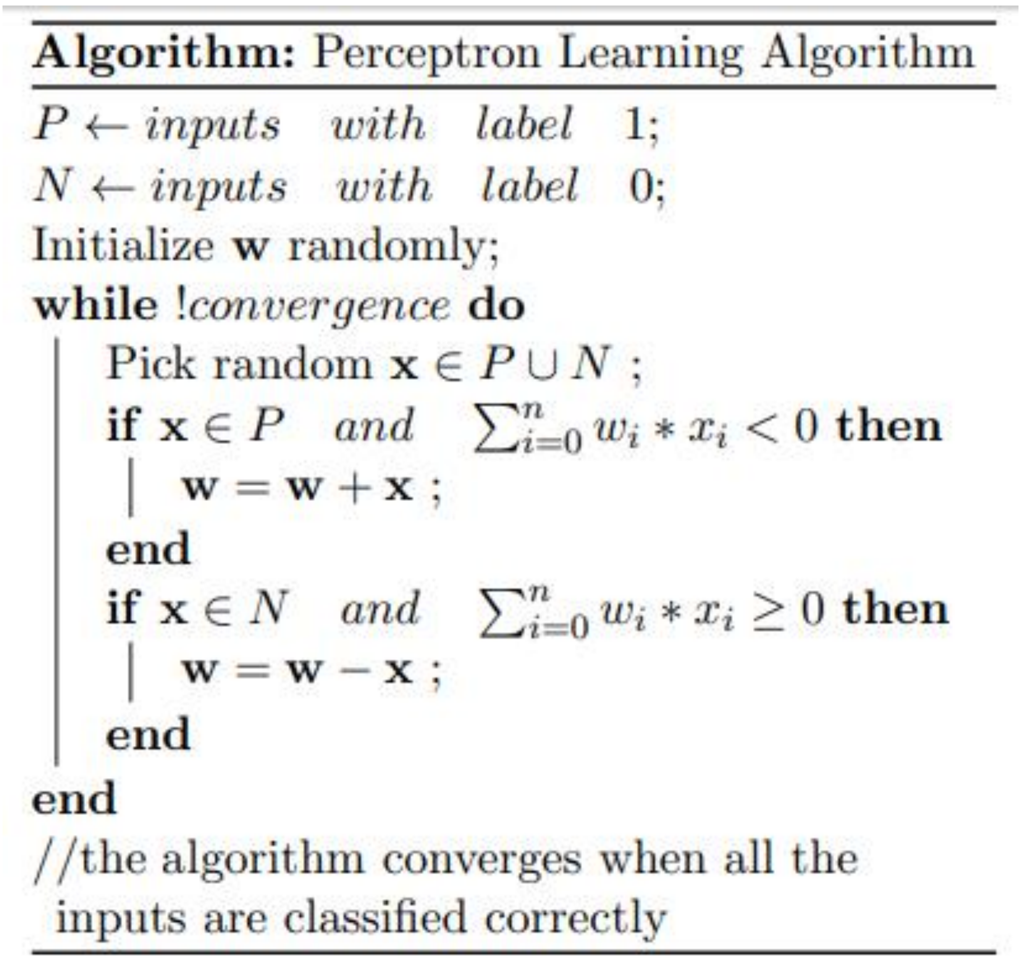
- Activation functions start from the input layer in the forward stage and terminate on the output layer.

Backward Stage:

- In the backward stage, weight and bias values are modified as per the model's requirement.
- In this stage, the error between actual output and demanded originated backward on the output layer and ended on the input layer.

- Hence, a multi-layered perceptron model has considered as multiple artificial neural networks having various layers in which activation function does not remain linear, similar to a single layer perceptron model.
- Instead of linear, activation function can be executed as sigmoid, TanH, ReLU, etc., for deployment.
- A multi-layer perceptron model has greater processing power and can process linear and non-linear patterns.
- Further, it can also implement logic gates such as AND, OR, XOR, NAND, NOT, XNOR, NOR.

Perceptron Learning Algorithm:



```
Algorithm: Perceptron Learning Algorithm
$P \leftarrow inputs \quad with \quad label \quad 1;$
$N \leftarrow inputs \quad with \quad label \quad 0;$
Initialize $\mathbf{w}$ randomly;
while !$convergence$ do
    Pick random $\mathbf{x} \in P \cup N$ ;
    if $\mathbf{x} \in P \quad and \quad \sum_{i=0}^{n} w_i * x_i < 0$ then
        $\mathbf{w} = \mathbf{w} + \mathbf{x}$ ;
    end
    if $\mathbf{x} \in N \quad and \quad \sum_{i=0}^{n} w_i * x_i \geq 0$ then
        $\mathbf{w} = \mathbf{w} - \mathbf{x}$ ;
    end
end
//the algorithm converges when all the
 inputs are classified correctly
```

Firing Rules:

• Threshold rules:

- Calculate weighted average of input.
- Fire if larger than threshold

• Perceptron rule:

- Calculate weighted average of input
- Output activation level is

Limitations of Perceptron Model

- The output of a perceptron can only be a binary number (0 or 1) due to the hard limit transfer function.
- Perceptron can only be used to classify the linearly separable sets of input vectors.
- If input vectors are non-linear, it is not easy to classify them properly.

Characteristics of Perceptron

1. Perceptron is a machine learning algorithm for supervised learning of binary classifiers.
2. In Perceptron, the weight coefficient is automatically learned.
3. Initially, weights are multiplied with input features, and the decision is made whether the neuron is fired or not.
4. The activation function applies a step rule to check whether the weight function is greater than zero.
5. The linear decision boundary is drawn, enabling the distinction between the two linearly separable classes +1 and -1.
6. If the added sum of all input values is more than the threshold value, it must have an output signal; otherwise, no output will be shown.

Backpropagation Algorithm

- It is widely used for training feed-forward neural networks.
- It computes the gradient of loss function with respect to network weights via chain rule.

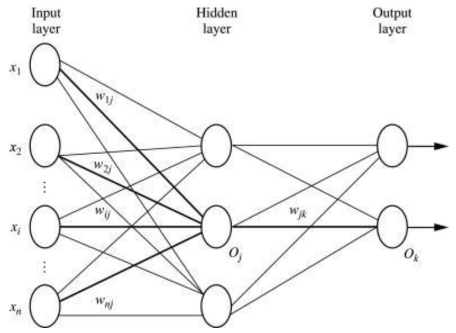
Working:

- Neural network uses supervised leaming to generate output vectors from input vectors that the network operates on it.
- It compares generated output to desired output and generates an error report if the result don't match generated output vector.
- Then adjusts the weights according bug to report to get your desired output.

Algorithm:

Step 1: Inputs X, arrive through the preconnected path.
Step 2: The input is modelled using true weights W. Weights are usually chosen randomly.
Step 3: Calculate the output of each neuron from the input layer to the hidden layer to the output layer.
Step 4: Calculate the error in the outputs
Backpropagation Error = Actual Output – Desired Output

Step 5: From the output layer, go back to the hidden layer to adjust the weights to reduce the error.
Step 6: Repeat the process until the desired output is achieved.

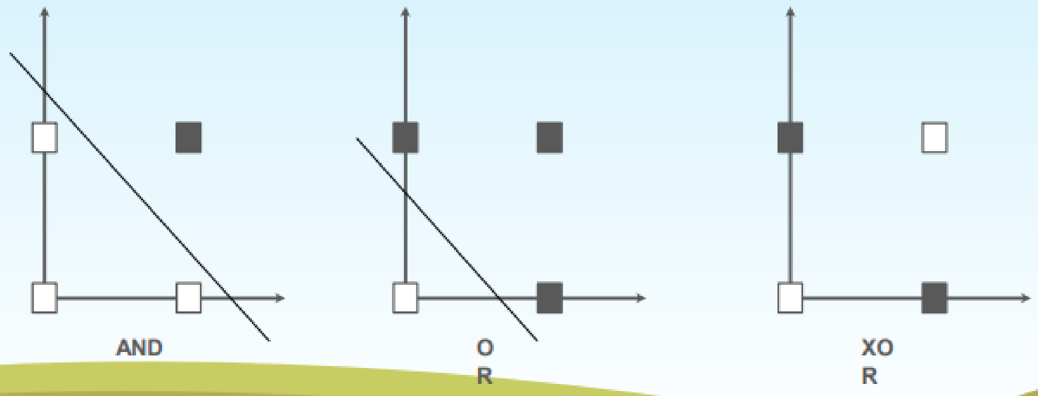


Limitation of Backpropagation Algorithm:

- It is sensitive to noisy data and irregularization noisy data can lead to inaccurate results.
- Performance highly dependent on input data.
- Spending too much time on training.
- The matrix-based approach is preferred over minibatch

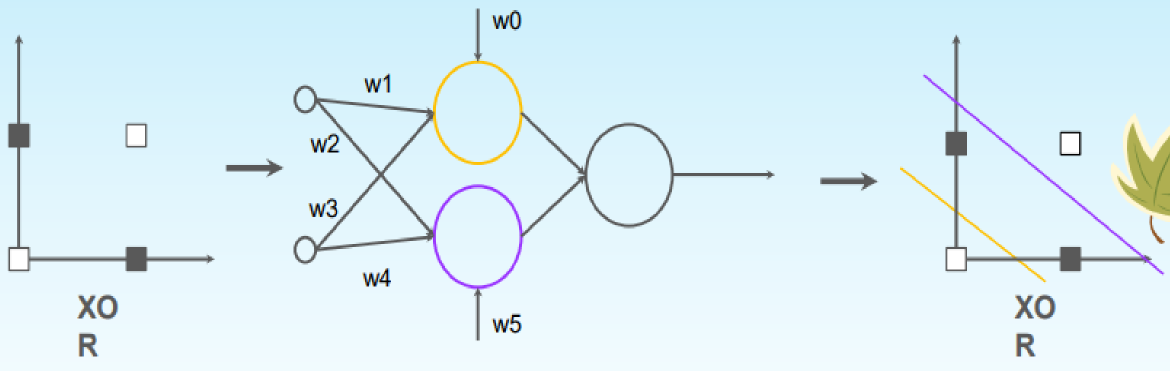
XOR Problem (Limitation of Perceptron)

- In 1969, Minsky, an American mathematician and AI pioneer, proved that a perceptron is essentially a linear model that can only deal with linear classification problems, but cannot process non-linear data.



Solution of XOR

To solve the problem of XOR we add an extra layer to perceptron called as Hidden Layer.



Loss Function

What is the Loss function?

The Loss function is a method of evaluating how well your algorithm is modelling your dataset.

Why Loss Function is important?

If the value of the loss function is lower then it's a good model otherwise, we have to change the parameter of the model and minimize the loss.

Loss Function	**Cost Function**
A loss function/error function is for a single training example/input.	A cost function, on other hand, is average loss over the entire training dataset.

Type of Loss Function:

A. Regression

1. MSE(Mean Squared Error)
2. MAE(Mean Absolute Error)
3. RMSE(Root Mean Squared Error)

B. Classification

1. Binary cross-entropy
2. Categorical cross-entropy

A. Regression Loss

1. Mean Squared Error/Squared loss/ L2 loss –

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

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

Advantage:

1. Easy to interpret.

2. Always differential because of the square.

3. Only one local minima.

Disadvantage:

1. Error unit in the square. because the unit in the square is not understood properly.

2. Not robust to outlier

2.Mean Absolute Error/ L1 loss-

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

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

Advantage:

1. Intuitive and easy.

2. Error Unit Same as the output column.

3. Robust to outlier.

Disadvantage:

1. Graph, not differential. we cannot use gradient descent directly, then we can subgradient calculation.

B. Classification Loss

1. Binary Cross Entropy/log loss:

- It is used in binary classification problems like two classes.

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

Where:

- yi – actual values
- yihat – Neural Network prediction

Advantage:

1. A cost function is a differential.

Disadvantage:

1. Multiple local minima.

2. Not intuitive.

2. Categorical Cross entropy:

- Categorical Cross entropy is used for Multiclass classification.
- Categorical Cross entropy is also used in SoftMax regression.

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

where k is number of classes in the data

Optimizers

What Are Optimizers in Deep Learning?

As mentioned in the introduction, optimizer algorithms are a type of optimization method that helps improve a deep learning model's performance. These optimization algorithms or optimizers widely affect the accuracy and speed training of the deep learning model. But first of all, the question arises of what an optimizer really is ?

While training the deep learning optimizers model, we need to modify each epoch's weights and minimize the loss function. An optimizer is a function or an algorithm that modifies the attributes of the neural network, such as weights and learning rates. Thus, it helps in reducing the overall loss and improving accuracy.

Before proceeding, there are a few terms that you should be familiar with.

Epoch – The number of times the algorithm runs on the whole training dataset.

Sample – A single row of a dataset.

Batch – It denotes the number of samples to be taken to for updating the model parameters.

Learning rate – It is a parameter that provides the model a scale of how much model weights should be updated.

Cost Function/Loss Function – A cost function is used to calculate the cost, which is the difference between the predicted value and the actual value.

Weights/ Bias – The learnable parameters in a model that controls the signal between two neurons.

1. Gradient Descent Optimizer

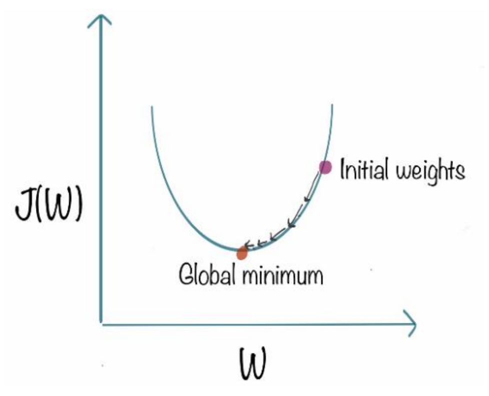
Gradient Descent can be considered the popular kid among the class of optimizers. This optimization algorithm uses calculus to modify the values consistently and to achieve the local minimum. Before moving ahead, you might have the question of what a gradient is.

x_new = x – alpha * f'(x)

The above equation means how the gradient is calculated. Here alpha is the step size that represents how far to move against each gradient with each iteration.

Gradient descent works as follows:

1. It starts with some coefficients, sees their cost, and searches for cost value lesser than what it is now.
2. It moves towards the lower weight and updates the value of the coefficients.
3. The process repeats until the local minimum is reached. A local minimum is a point beyond which it cannot proceed.



Gradient descent works best for most purposes. However, it has some downsides too. It is expensive to calculate the gradients if the size of the data is huge. Gradient descent works well for convex functions, but it doesn't know how far to travel along the gradient for nonconvex functions.

2. Stochastic Gradient Descent Optimizer (SGD)

At the end of the previous section, you learned why using gradient descent on massive data might not be the best option. To tackle the problem, we have stochastic gradient descent. The term stochastic means randomness on which the algorithm is based upon. In stochastic gradient descent, instead of taking the whole dataset for each iteration, we randomly select the batches of data. That means we only take a few samples from the dataset.

$$w := w - \eta \nabla Q_i(w).$$

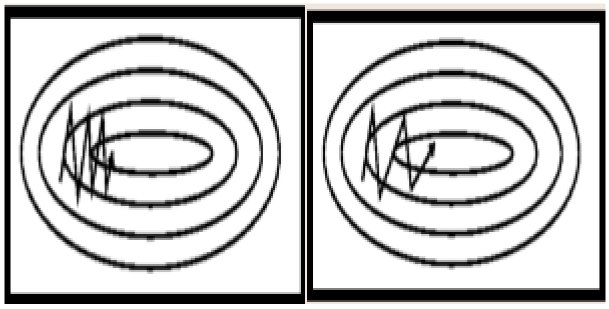
The procedure is first to select the initial parameters w and learning rate n. Then randomly shuffle the data at each iteration to reach an approximate minimum.

Since we are not using the whole dataset but the batches of it for each iteration, the path taken by the algorithm is full of noise as compared to the gradient descent algorithm. Thus, SGD uses a higher number of iterations to reach the local minima. Due to an increase in the number of iterations, the overall computation time increases. But even after increasing the number of iterations, the computation cost is still less than that of the gradient descent optimizer. So, the conclusion is if the data is enormous and computational time is an essential factor, stochastic gradient descent should be preferred over batch gradient descent algorithm.

3. Stochastic Gradient Descent with Momentum Deep Optimizer (SGD with Momentum Deep)

As discussed in the earlier section, you have learned that stochastic gradient descent takes a much more noisy path than the gradient descent algorithm. Due to this reason, it requires a more significant number of iterations to reach the optimal minimum, and hence computation time is very slow. To overcome the problem, we use stochastic gradient descent with a momentum algorithm.

What the momentum does is helps in faster convergence of the loss function. Stochastic gradient descent oscillates between either direction of the gradient and updates the weights accordingly. However, adding a fraction of the previous update to the current update will make the process a bit faster. One thing that should be remembered while using this algorithm is that the learning rate should be decreased with a high momentum term.



In the above image, the left part shows the convergence graph of the stochastic gradient descent algorithm. At the same time, the right side shows SGD with momentum. From the image, you can compare the path chosen by both algorithms and realize that using momentum helps reach convergence in less time. You might be thinking of using a large momentum and learning rate to make the process even faster. But remember that while increasing the momentum, the possibility of passing the optimal minimum also increases. This might result in poor accuracy and even more oscillations.

4. Mini Batch Gradient Descent Optimizer (Minibatch GD)

In this variant of gradient descent, instead of taking all the training data, only a subset of the dataset is used for calculating the loss function. Since we are using a batch of data instead of taking the whole dataset, fewer iterations are needed. That is why the mini-batch gradient descent algorithm is faster than both stochastic gradient descent and batch gradient descent algorithms. This algorithm is more efficient and robust than the earlier variants of gradient descent. As the algorithm uses batching, all the training data need not be loaded in the memory, thus making the process more efficient to implement. Moreover, the cost function in mini-batch gradient descent is noisier than the batch gradient descent algorithm but smoother than that of the stochastic gradient descent algorithm. Because of this, mini-batch gradient descent is ideal and provides a good balance between speed and accuracy.

Despite all that, the mini-batch gradient descent algorithm has some downsides too. It needs a hyperparameter that is “mini-batch-size,” which needs to be tuned to achieve the required accuracy. Although, the batch size of 32 is considered to be appropriate for almost every case. Also, in some cases, it results in poor final accuracy. Due to this, there needs a rise to look for other alternatives too.

5. Adagrad (Adaptive Gradient Descent) Deep Learning Optimizer

The adaptive gradient descent algorithm is slightly different from other gradient descent algorithms. This is because it uses different learning rates for each iteration. The change in learning rate depends upon the difference in the parameters during training. The more the parameters get changed, the more minor the learning rate changes. This modification is highly beneficial because real-world datasets contain sparse as well as dense features. So, it is unfair to have the same value of learning rate for all the features. The Adagrad algorithm uses the below formula to update the weights. Here the alpha(t) denotes the different learning rates at each iteration, n is a constant, and E is a small positive to avoid division by 0.

$$\mathrm{w_t} = \mathrm{w_{t-1}} - \eta'_{\mathrm{t}} \frac{\partial \mathrm{L}}{\partial \mathrm{w(t-1)}}$$

$$\eta'_{\mathrm{t}} = \frac{\eta}{\mathrm{sqrt}(\alpha_{\mathrm{t}} + \epsilon)}$$

The benefit of using Adagrad is that it abolishes the need to modify the learning rate manually. It is more reliable than gradient descent algorithms and their variants, and it reaches convergence at a higher speed.

One downside of the AdaGrad optimizer is that it decreases the learning rate aggressively and monotonically. There might be a point when the learning rate becomes extremely small. This is because the squared gradients in the denominator keep accumulating, and thus the denominator part keeps on increasing. Due to small learning rates, the model eventually becomes unable to acquire more knowledge, and hence the accuracy of the model is compromised.

6. RMS Prop (Root Mean Square) Deep Learning Optimizer

RMS prop is one of the popular optimizers among deep learning enthusiasts. This is maybe because it hasn’t been published but is still very well-known in the community. RMS prop is ideally an extension of the work RPPROP. RPPROP resolves the problem of varying gradients. The problem with the gradients is that some of them were small while others may be huge. So, defining a single learning rate might not be the best idea. RPPROP uses the sign of the gradient, adapting the step size individually for each weight. In this algorithm, the two gradients are first compared for signs. If they have the same sign, we're going in the right direction and hence increase the step size by a small fraction. Whereas if they have opposite signs, we have to decrease the step size. Then we limit the step size, and now we can go for the weight update.

The problem with RPPROP is that it doesn't work well with large datasets and when we want to perform mini-batch updates. So, achieving the robustness of RPPROP and the efficiency of mini-batches at the same time was the main motivation behind the rise of RMS prop. RMS prop can also be considered an advancement in AdaGrad optimizer as it reduces the monotonically decreasing learning rate.

The algorithm mainly focuses on accelerating the optimization process by decreasing the number of function evaluations to reach the local minimum. The algorithm keeps the moving average of squared gradients for every weight and divides the gradient by the square root of the mean square.

$$v(w,t) := \gamma v(w,t-1) + (1-\gamma)(\nabla Q_i(w))^2$$

where gamma is the forgetting factor. Weights are updated by the below formula

$$w := w - \frac{\eta}{\sqrt{v(w,t)}} \nabla Q_i(w)$$

The problem with RMS Prop is that the learning rate has to be defined manually, and the suggested value doesn't work for every application.

7. Adam Deep Learning Optimizer

The name adam is derived from adaptive moment estimation. This optimization algorithm is a further extension of stochastic gradient descent to update network weights during training. Unlike maintaining a single learning rate through training in SGD, Adam optimizer updates the learning rate for each network weight individually. The creators of the Adam optimization algorithm know the benefits of AdaGrad and RMSProp algorithms, which are also extensions of the stochastic gradient descent algorithms. Hence the Adam optimizers inherit the features of both Adagrad and RMS prop algorithms. In adam, instead of adapting learning rates based upon the first moment(mean) as in RMS Prop, it also uses the second moment of the gradients. We mean the uncentred variance by the second moment of the gradients(we don’t subtract the mean).

The adam optimizer has several benefits, due to which it is used widely. It is adapted as a benchmark for deep learning papers and recommended as a default optimization algorithm. Moreover, the algorithm is straightforward to implement, has a faster running time, low memory requirements, and requires less tuning than any other optimization algorithm.

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\left[\frac{\delta L}{\delta w_t}\right] v_t = \beta_2 v_{t-1} + (1-\beta_2)\left[\frac{\delta L}{\delta w_t}\right]^2$$

The above formula represents the working of adam optimizer. Here B1 and B2 represent the decay rate of the average of the gradients.

Activation functions

What is an activation function and why use them?

The activation function decides whether a neuron should be activated or not by calculating the weighted sum and further adding bias to it. The purpose of the activation function is to introduce non-linearity into the output of a neuron.

Why do we need Non-linear activation function?

A neural network without an activation function is essentially just a linear regression model. The activation function does the non-linear transformation to the input making it capable to learn and perform more complex tasks.

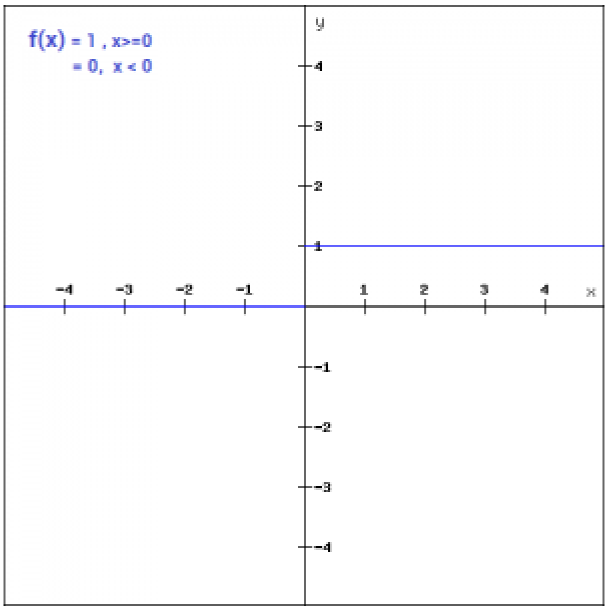
Popular types of activation functions and when to use them

1. Binary Step Function

The first thing that comes to our mind when we have an activation function would be a threshold-based classifier i.e., whether or not the neuron should be activated based on the value from the linear transformation.

In other words, if the input to the activation function is greater than a threshold, then the neuron is activated, else it is deactivated, i.e., its output is not considered for the next hidden layer. Let us look at it mathematically-

$f(x) = 1, x>=0$
$\quad = 0, x<0$



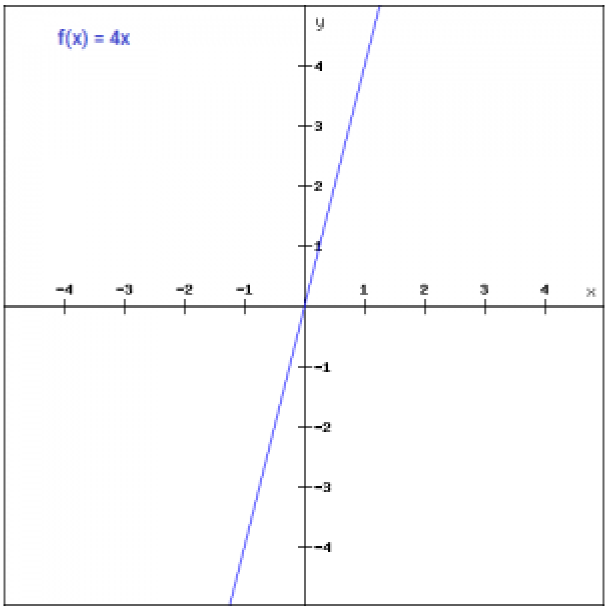
This is the simplest activation function, which can be implemented with a single if-else condition in python

```
def binary_step(x):
    if x<0:
        return 0
    else:
        return 1
binary_step(5), binary_step(-1)
```

2. Linear Function

We saw the problem with the step function, the gradient of the function became zero. This is because there is no component of x in the binary step function. Instead of a binary function, we can use a linear function. We can define the function as-

f(x)=ax



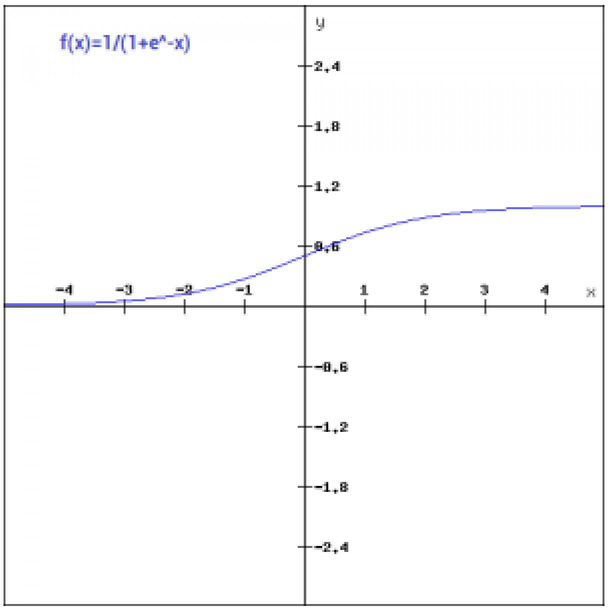
Here the activation is proportional to the input. The variable ‘a’ in this case can be any constant value. Let’s quickly define the function in python:

```
def linear_function(x):
    return 4*x
linear_function(4), linear_function(-2)
```

3. Sigmoid

The next activation function that we are going to look at is the Sigmoid function. It is one of the most widely used non-linear activation function. Sigmoid transforms the values between the range 0 and 1. Here is the mathematical expression for sigmoid-

f(x) = 1/(1+e^-x)



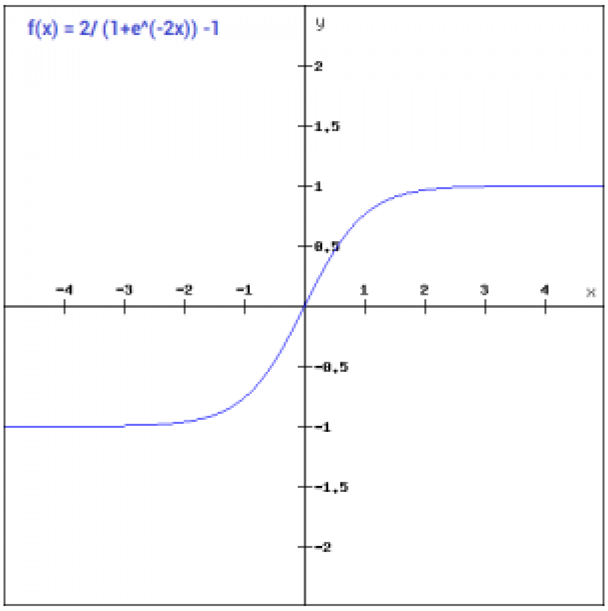
A noteworthy point here is that unlike the binary step and linear functions, sigmoid is a non-linear function. This essentially means -when I have multiple neurons having sigmoid function as their activation function, the output is nonlinear as well. Here is the python code for defining the function in python-

```
import numpy as np
def sigmoid_function(x):
    z = (1/(1 + np.exp(-x)))
    return z
sigmoid_function(7),sigmoid_function(-22)
```

4. Tanh

The tanh function is very similar to the sigmoid function. The only difference is that it is symmetric around the origin. The range of values in this case is from -1 to 1. Thus, the inputs to the next layers will not always be of the same sign. The tanh function is defined as-

tanh(x)=2sigmoid(2x)-1



In order to code this is python, let us simplify the previous expression.

tanh(x) = 2sigmoid(2x)-1
tanh(x) = 2/(1+e^(-2x)) -1

And here is the python code for the same:

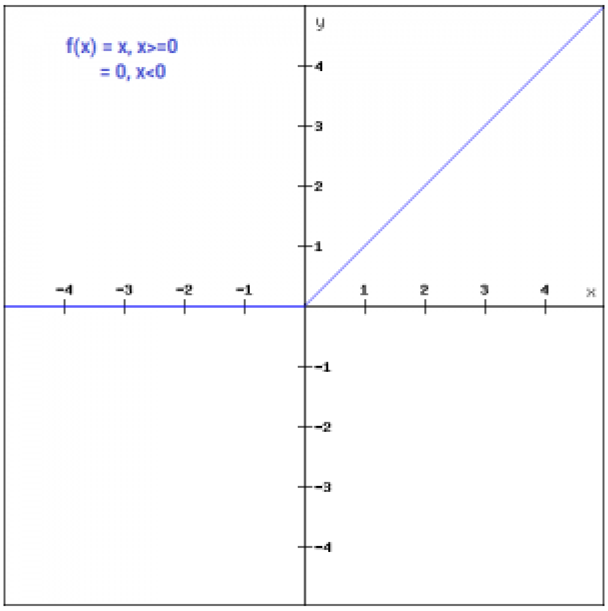
```
def tanh_function(x):
    z = (2/(1 + np.exp(-2*x))) -1
    return z
tanh_function(0.5), tanh_function(-1)
```

5. ReLU

The ReLU function is another non-linear activation function that has gained popularity in the deep learning domain. ReLU stands for Rectified Linear Unit. The main advantage of using the ReLU function over other activation functions is that it does not activate all the neurons at the same time.

This means that the neurons will only be deactivated if the output of the linear transformation is less than 0. The plot below will help you understand this better-

f(x)=max(0,x)



For the negative input values, the result is zero, that means the neuron does not get activated. Since only a certain number of neurons are activated, the ReLU function is far more computationally efficient when compared to the sigmoid and tanh function. Here is the python function for ReLU:

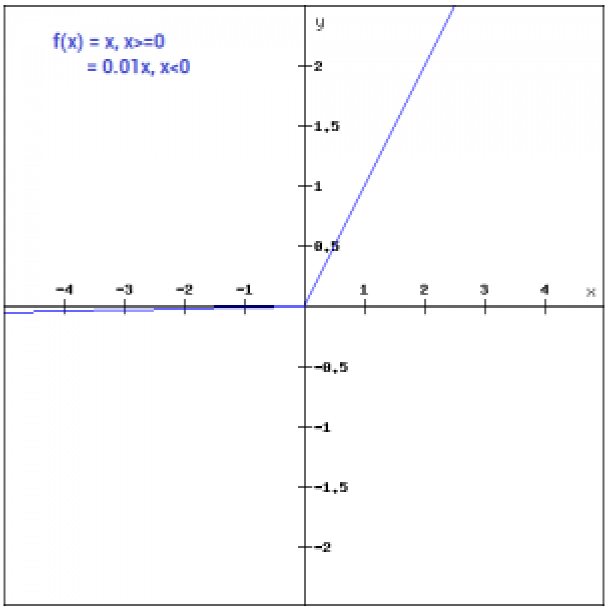
```
def relu_function(x):
    if x<0:
        return 0
    else:
        return x
relu_function(7), relu_function(-7)
```

6. Leaky ReLU

Leaky ReLU function is nothing but an improved version of the ReLU function. As we saw that for the ReLU function, the gradient is 0 for x<0, which would deactivate the neurons in that region.

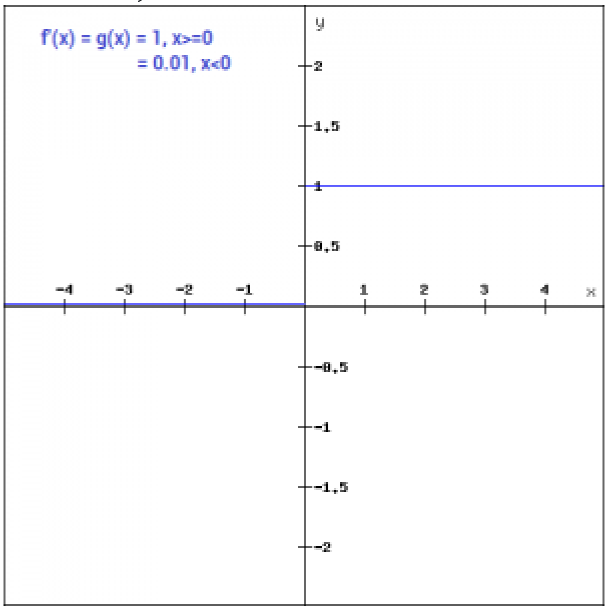
Leaky ReLU is defined to address this problem. Instead of defining the Relu function as 0 for negative values of x, we define it as an extremely small linear component of x. Here is the mathematical expression-

$$\begin{aligned} f(x) &= 0.01x, \; x<0 \\ &= x, \; x>=0 \end{aligned}$$



By making this small modification, the gradient of the left side of the graph comes out to be a non-zero value. Hence, we would no longer encounter dead neurons in that region. Here is the derivative of the Leaky ReLU function

f'(x) = 1, x>=0
=0.01, x<0



Since Leaky ReLU is a variant of ReLU, the python code can be implemented with a small modification-

```
def leaky_relu_function(x):
    if x<0:
        return 0.01*x
    else:
        return x
leaky_relu_function(7), leaky_relu_function(-7)
```

7. SoftMax

SoftMax function is often described as a combination of multiple sigmoids. We know that sigmoid returns values between 0 and 1, which can be treated as probabilities of a data point belonging to a particular class. Thus, sigmoid is widely used for binary classification problems.

The SoftMax function can be used for multiclass classification problems. This function returns the probability for a datapoint belonging to each individual class. Here is the mathematical expression of the same-

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \quad \text{for } j = 1, \ldots, K.$$

While building a network for a multiclass problem, the output layer would have as many neurons as the number of classes in the target. For instance, if you have three classes, there would be three neurons in the output layer. Suppose you got the output from the neurons as [1.2 , 0.9 , 0.75].

Applying the SoftMax function over these values, you will get the following result – [0.42 , 0.31, 0.27]. These represent the probability for the data point belonging to each class. Note that the sum of all the values is 1. Let us code this in python

```
def softmax_function(x):
    z = np.exp(x)
    z_ = z/z.sum()
    return z_
softmax_function([0.8, 1.2, 3.1])
```

Choosing the right Activation Function

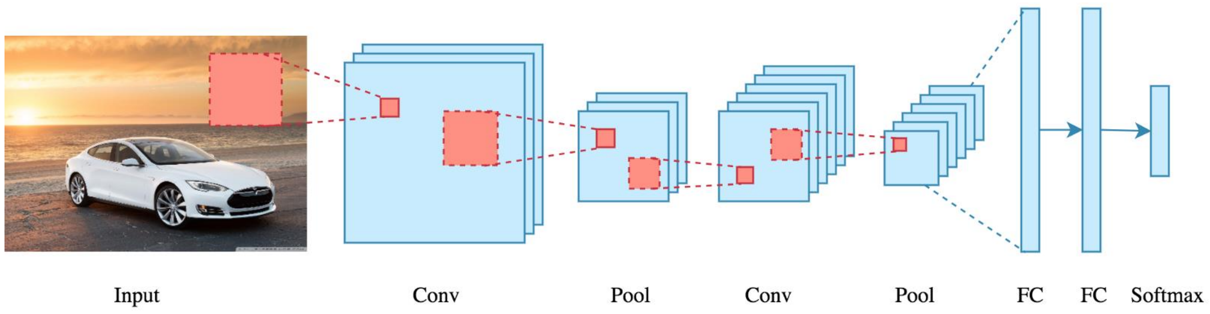
- ReLu – **Hidden Layers**
- Sigmoid / TanH – **Not for hidden layer. Generally, for output layer**
- Softmax / Swish – **Large network with 40+ layers**
- Regress – **Linear**
- Binary Classification – **Sigmoid / Logistic**
- Multiclass – **Softmax**
- Multilabel – **Sigmoid**
- CNN – **ReLu**
- RNN – **TanH / Sigmoid**

Unit-2

Convolutional Neural Networks (CNN)

What Is a CNN?

- A Convolutional Neural Network (CNN) is a deep learning algorithm widely used for computer vision tasks.
- It is specifically designed to process grid-like data, such as images, and automatically learn hierarchical representations of features from raw input.
- CNNs consist of convolutional layers that extract local patterns and spatial dependencies, pooling layers that down sample the data and reduce dimensions, fully connected layers that perform high-level feature learning, and activation functions that introduce non-linearities.
- CNNs excel at capturing visual information, enabling tasks like image classification, object detection, and image segmentation.
- By leveraging convolution and specialized layers, CNNs can analyse and interpret images, making them a powerful tool in computer vision with applications in various fields, including healthcare, autonomous vehicles, and more.



Fully Connected Network vs Convolutional Neural Network

Convolutional layers and fully connected layers are two different types of layers used in deep neural networks.

Convolutional layers are primarily used in computer vision tasks and are designed to extract features from images. They consist of a set of filters that are convolved with the input image to produce a set of feature maps. These feature maps capture different aspects of the input image, such as edges, textures, and shapes. Convolutional layers are characterized by their ability to learn spatially invariant representations of the input data, which makes them well-suited for tasks like image classification, object detection, and segmentation.

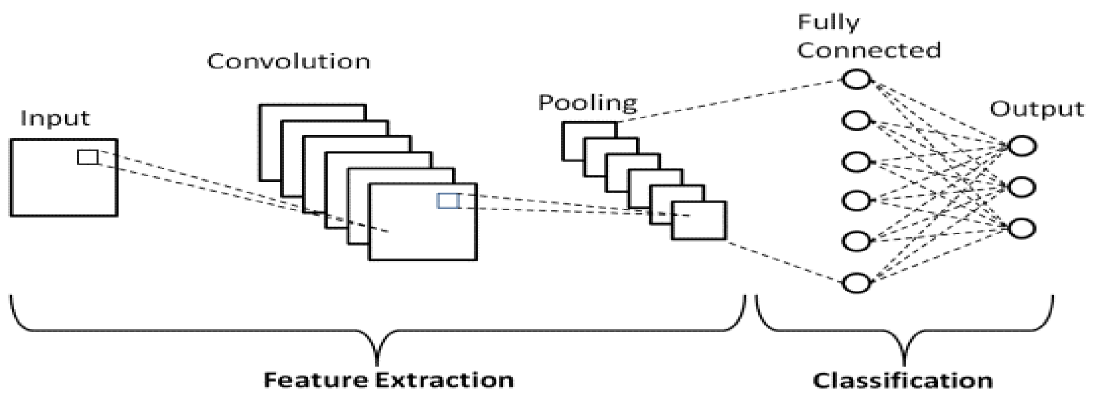
Fully connected layers, on the other hand, are used in a variety of deep learning applications, including computer vision, natural language processing, and speech recognition. They are typically used at the end of the network to map the extracted features to the output classes. Fully connected layers are characterized by their ability to capture complex relationships between features and produce high-dimensional output. They are often used in conjunction with convolutional layers in computer vision tasks, where the features extracted by the convolutional layers are passed through a series of fully connected layers to produce the final output.

In summary, convolutional layers are used for feature extraction in computer vision tasks, while fully connected layers are used for mapping the features to the output classes. Both types of layers are essential for building effective deep neural networks.

Basic Architecture

There are two main parts to a CNN architecture:

- A CNN architecture consists of two main parts: feature extraction and classification.
- Feature extraction involves using a convolutional tool to separate and identify different features in the input image.
- The feature extraction network consists of multiple pairs of convolutional or pooling layers.
- After feature extraction, the output is fed into a fully connected layer for classification.
- The fully connected layer uses the extracted features to predict the class of the image.
- The CNN model aims to reduce the number of features in the dataset and creates new summarized features.
- The architecture diagram of a CNN shows the arrangement of multiple CNN layer.



There are three types of layers that make up the CNN which are the convolutional layers, pooling layers, and fully-connected (FC) layers. When these layers are stacked, a CNN architecture will be formed. In addition to these three layers, there are two more important parameters which are the dropout layer and the activation function which are defined below.

A CNN architecture typically includes convolutional layers, pooling layers, fully connected layers, dropout, and activation functions. Here's an explanation of each component:

1. **Convolutional Layer:**

- The convolutional layer is the core component of a CNN. It applies a set of learnable filters (also known as kernels) to the input data.
- Each filter performs a convolution operation by sliding over the input and computing element-wise multiplications and summations.
- This process extracts local features and spatial patterns from the input data.
- The output of the convolutional layer is often referred to as feature maps or activation maps.

2. **Pooling Layer:**

- The pooling layer follows the convolutional layer and helps reduce the spatial dimensions of the feature maps.
- It aggregates information by down sampling and summarizing the feature maps.
- Common pooling operations include max pooling (selecting the maximum value within a region) and average pooling (calculating the average value).
- Pooling helps make the representations more robust to variations and reduces the computational requirements.

3. **Fully Connected Layer:**

- The fully connected layer is a traditional neural network layer where each neuron is connected to every neuron in the previous layer.
- It takes the flattened or reshaped output from the preceding layers and performs high-level feature learning and classification.
- The fully connected layers capture complex relationships and patterns in the extracted features.
- The final fully connected layer typically produces the output for classification or regression tasks.

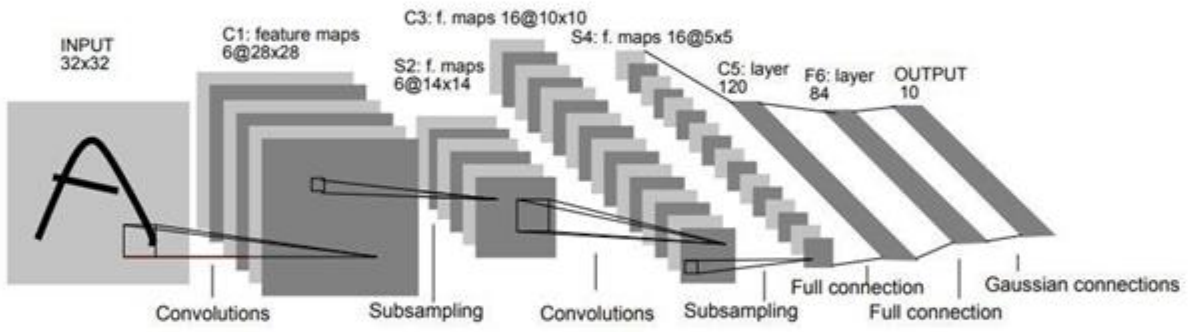
4. **Dropout:**

- Dropout is a regularization technique used to prevent overfitting in the network.
- It randomly sets a fraction of input units (neurons) to zero during training.
- Dropout helps to reduce the reliance of the network on specific neurons, forcing the network to learn more robust and generalized representations.

5. **Activation Function:**

- Activation functions introduce non-linearities into the network, enabling it to learn complex relationships between the inputs and outputs.
- Common activation functions include Rectified Linear Unit (ReLU), sigmoid, and tanh.
- ReLU is widely used in CNNs as it helps with efficient training and alleviates the vanishing gradient problem.
- The activation function is typically applied element-wise to the outputs of convolutional, pooling, or fully connected layers.

By combining these components in different arrangements and stacking multiple layers, CNN architectures can effectively learn and extract features from input data, enabling them to perform tasks like image classification, object detection, and image segmentation.



Understanding Border Effect of Padding

In Conv2D layers, padding is configurable via the padding argument, which takes two values:

- "valid", which means no padding (only valid window locations will be used); and
- "same", which means “pad in such a way as to have an output with the same width and height as the input.” The padding argument defaults to "valid".

Understanding Convolution Strides

- The other factor that can **influence output size** is the notion of **strides**.
- So far we assumed that the centre tiles of the convolution windows are all contiguous.
- The **distance between two successive windows** is a parameter of the convolution, called its stride, which defaults to 1.
- Strided convolutions: convolutions with **a stride higher than 1**.
- In figure you can see the patches extracted by a 3 × 3 convolution with stride 2 over a 5 × 5 input (without padding).
- Using stride 2 means the width and height of the feature map are **down sampled** by a factor of 2 (in addition to any changes induced by border effects).
- Strided convolutions are **rarely used in practice**, although they can come in handy for some types of models.
- To downsample feature maps, instead of strides, we tend to use the **max-pooling operation**.

Max-Pooling Operation

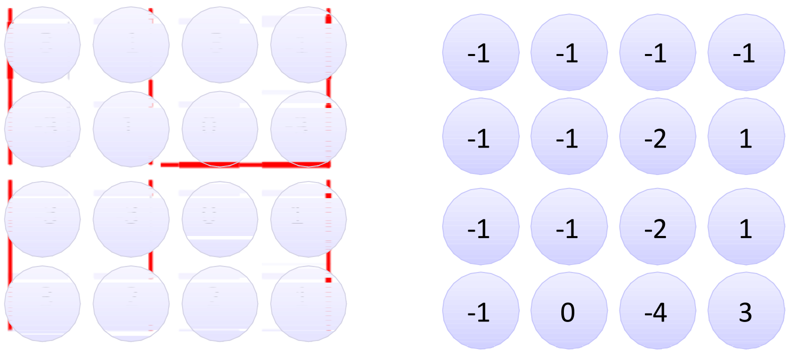
- the role of max pooling: to aggressively downsample feature maps, much like strided convolutions.
- Max pooling consists of extracting windows from the input feature maps and outputting the max value of each channel.
- It’s conceptually similar to convolution, except that instead of transforming local patches via a learned linear transformation (the convolution kernel), they’re transformed via a hardcoded max tensor operation.
- Why downsample feature maps this way?
- Why not remove the max-pooling layers and keep fairly large feature maps all the way up?
- the reason to use downsampling is to reduce the number of feature-map coefficients to process, as well as to induce spatial-filter hierarchies by making successive convolution layers look at increasingly large windows (in terms of the fraction of the original input they cover).

1	-1	-1
-1	1	-1
-1	-1	1

Filter 1

-1	1	-1
-1	1	-1
-1	1	-1

Filter 2



Max Pooling Vs Average Pooling

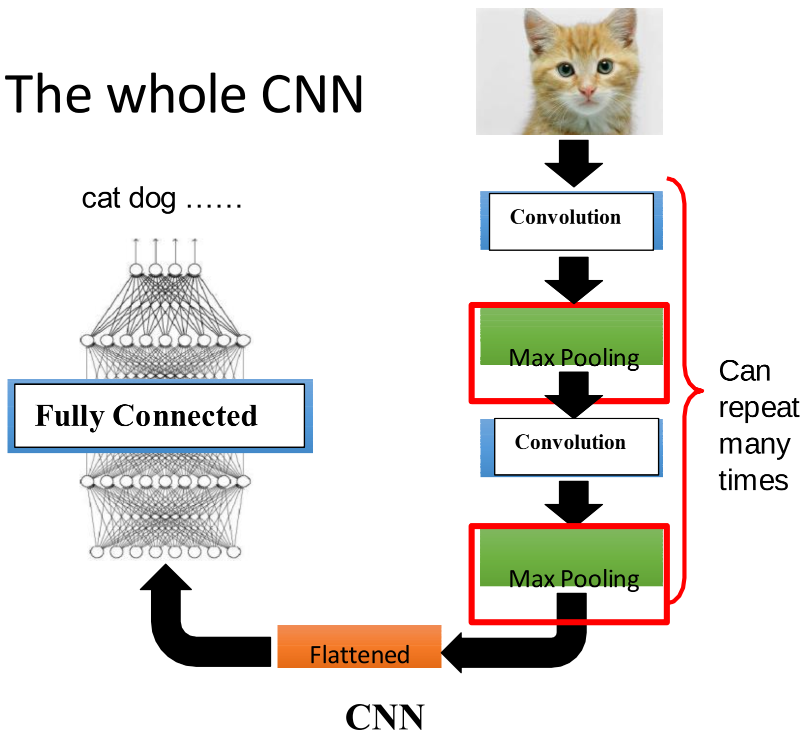
Max pooling and average pooling are two common types of pooling layers used in convolutional neural networks (CNNs) for down-sampling and feature reduction.

Max pooling is a pooling operation that selects the maximum value from a pool of values. In this operation, a sliding window is moved over the feature map and for each window, the maximum value is selected and used to create a new down-sampled feature map. Max pooling is often used to capture the most important features in an input image, and it is particularly useful for object detection tasks, where the location of the object is not known in advance.

Average pooling is another type of pooling operation that calculates the average value of the pool of values instead of selecting the maximum value. It is also applied by sliding a window over the input feature map, but instead of selecting the maximum value, the average value of the pool is used to create the down-sampled feature map. Average pooling is often used in CNNs to reduce the spatial dimension of the feature maps while retaining the important features. It is particularly useful for image classification tasks, where the input image contains multiple objects and the location of the objects is not as important as the features of the objects.

In summary, max pooling and average pooling are two common types of pooling operations used in CNNs. Max pooling is used to select the most important features in an input image and is useful for object detection tasks. Average pooling, on the other hand, calculates the average value of the pool of values and is useful for image classification tasks, where the spatial location of the features is less important.

The whole CNN



Different CNN architecture

There are several popular CNN architectures that have been widely used and have achieved significant success in various computer vision tasks. Here are a few notable CNN architectures:

1. **LeNet-5:** One of the earliest CNN architectures, LeNet-5 was developed by Yann LeCun. It consists of multiple convolutional layers followed by pooling layers and fully connected layers. LeNet-5 was primarily used for handwritten digit recognition.

2. **AlexNet:** AlexNet is a deep CNN architecture that gained prominence after winning the ImageNet Large Scale Visual Recognition Challenge in 2012. It introduced the use of ReLU activation functions, dropout regularization, and large-scale parallel computing on GPUs. AlexNet has eight layers, including five convolutional layers and three fully connected layers.

3. **VGGNet:** The VGGNet architecture, developed by the Visual Geometry Group at the University of Oxford, is known for its simplicity and depth. It features multiple convolutional layers, each with small-sized filters (3x3), followed by pooling layers. VGGNet is known for its deeper network configurations, such as VGG16 and VGG19, with 16 and 19 layers, respectively.

4. **GoogLeNet (Inception):** GoogLeNet, also known as the Inception architecture, introduced the concept of "Inception modules" that perform multiple parallel convolutions at different scales and concatenate their outputs. This architecture is highly efficient in terms of computational resources and achieved high accuracy on the ImageNet challenge.

5. **ResNet:** ResNet (Residual Network) is a groundbreaking CNN architecture that addresses the problem of vanishing gradients in deep networks. It utilizes skip connections, allowing information to flow directly from one layer to deeper layers, thus facilitating training of very

deep networks. ResNet architectures, such as ResNet-50 and ResNet-101, have been widely used and have achieved state-of-the-art results in various computer vision tasks.

6. **MobileNet:** MobileNet is designed specifically for mobile and embedded devices with limited computational resources. It employs depth wise separable convolutions that significantly reduce the number of parameters and computations while maintaining reasonable accuracy.

7. **DenseNet:** DenseNet introduces dense connections, where each layer is connected to every other layer in a feed-forward fashion. This architecture encourages feature reuse and facilitates gradient flow throughout the network, enabling better information propagation and improved performance.

These are just a few examples of CNN architectures, and there are many more variations and extensions. Each architecture has its own strengths and is suited for specific tasks or computational requirements. Researchers and practitioners continue to explore and develop new CNN architectures to further advance the field of computer vision.

Unit-3

Transfer Learning

What Is Transfer Learning and It’s Working

Transfer learning is a machine learning technique where knowledge gained from training one model on a specific task is transferred and applied to another related task. It leverages pre-trained models that have been trained on large datasets and generalizes their learned features to new, similar tasks.

The working of transfer learning typically involves the following steps:

1. **Pre-training:** Initially, a deep neural network model, such as a Convolutional Neural Network (CNN), is trained on a large dataset (source task) for a specific task, such as image classification. This pre-training phase allows the model to learn rich features and hierarchical representations from the data.

2. **Feature Extraction:** After pre-training, the learned features are extracted from the last layer or intermediate layers of the pre-trained model. These features capture meaningful representations of the input data.

3. **Transfer:** The extracted features are then used as inputs for a new model (target task) that is specifically designed for the new task at hand. The new model may consist of additional layers, such as fully connected layers or a classifier, to adapt the learned features to the target task.

4. **Fine-tuning:** In some cases, the new model's parameters and pre-trained layers are further fine-tuned using the target task's data. This step allows the model to adapt to the specific characteristics and nuances of the target task. Fine-tuning is typically performed on the later layers of the model, while keeping the earlier layers fixed or with a lower learning rate.

By using transfer learning, the new model can benefit from the knowledge and features learned during the pre-training phase.

This approach offers several advantages:

1. **Reduced Training Time:** Transfer learning saves significant computational resources and training time compared to training a model from scratch, as the pre-trained model has already learned useful representations.

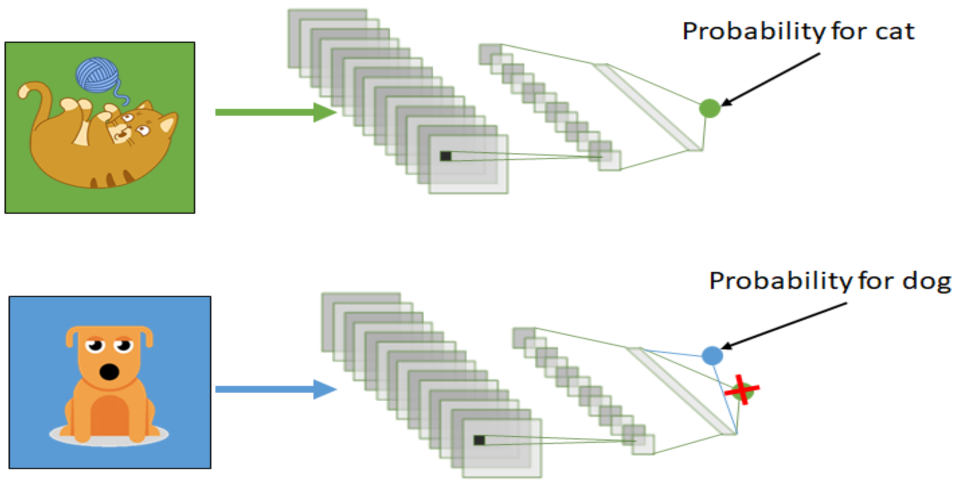
2. **Improved Generalization:** Pre-trained models have learned rich features from diverse datasets, enabling better generalization and improved performance on the target task, especially when the target task has limited training data.

3. **Effective in Low-Data Scenarios:** Transfer learning is particularly useful in situations where the target task has a small amount of labelled data. The pre-trained model's knowledge helps to overcome the data scarcity issue.

4. **Adaptability to New Domains:** Transfer learning allows the knowledge gained from one domain to be transferred and applied to another related domain. This enables leveraging pre-existing expertise and accelerating model development.

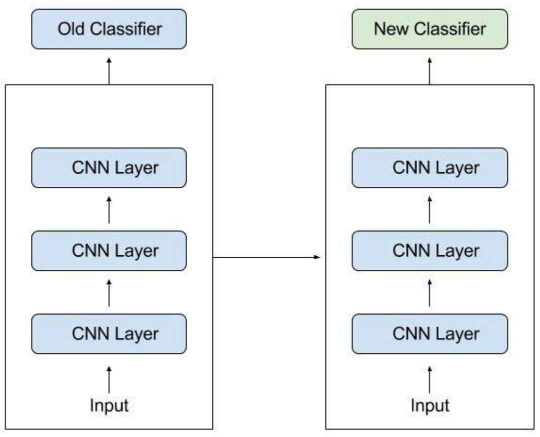
However, it's important to consider that transfer learning is effective when the pre-trained model and the target task are related or have similar features. If the pre-trained model's knowledge is not applicable to the target task, or the data distributions differ significantly, the performance may be limited, and training from scratch may be more suitable.

Overall, transfer learning is a powerful technique that allows models to leverage pre-existing knowledge and generalize well to new tasks, benefiting from the collective learning achieved across different domains.



Why Should You Use Transfer Learning?

- **Transfer learning offers a number of advantages, the most important of which are reduced training time, improved neural network performance (in most circumstances), and the absence of a large amount of data**.
- To train a neural model from scratch, a lot of data is typically needed, but access to that data is not always possible – this is when transfer learning comes in handy.



- Because the model has already been pre-trained, a good machine learning model can be generated with fairly little training data using transfer learning.

- This is especially useful in natural language processing, where huge labelled datasets require a lot of expert knowledge.
- Additionally, training time is decreased because building a deep neural network from the start of a complex task can take days or even weeks.

When to Use Transfer Learning

- **When we don’t have enough annotated data to train our model with.**
- **When there is a pre-trained model that has been trained on similar data and tasks.**
- **If you used TensorFlow to train the original model, you might simply restore it and retrain some layers for your job**.
- Transfer learning, on the other hand, only works if the features learnt in the first task are general, meaning they can be applied to another activity. Furthermore, the model’s input must be the same size as it was when it was first trained.
- If you don’t have it, add a step to resize your input to the required size.

1. PRE-TRAINED MODEL

  - **The second option is to employ a model that has already been trained.**
  - **There are a number of these models out there, so do some research beforehand.**
  - The number of layers to reuse and retrain is determined by the task.
  - Keras consists of nine pre-trained models used in transfer learning, prediction, fine-tuning.

- These models, as well as some quick lessons on how to utilise them, may be found here. Many research institutions also make trained models accessible.
- The most popular application of this form of transfer learning is deep learning.

1.1 Feature Extraction

1.2 Fine Tuning

1.1 Feature Extraction

Feature extraction consists of using the representations learned by a previous network to extract interesting features from new samples. These features are then run through a new classifier, which is trained from scratch.

As you saw previously, convnets used for image classification comprise two parts: they start with a series of pooling and convolution layers, and they end with a densely connected classifier. The first part is called the convolutional base of the model.

In the case of convnets, feature extraction consists of taking the convolutional base of a previously trained network, running the new data through it, and training a new classifier on top of the output.

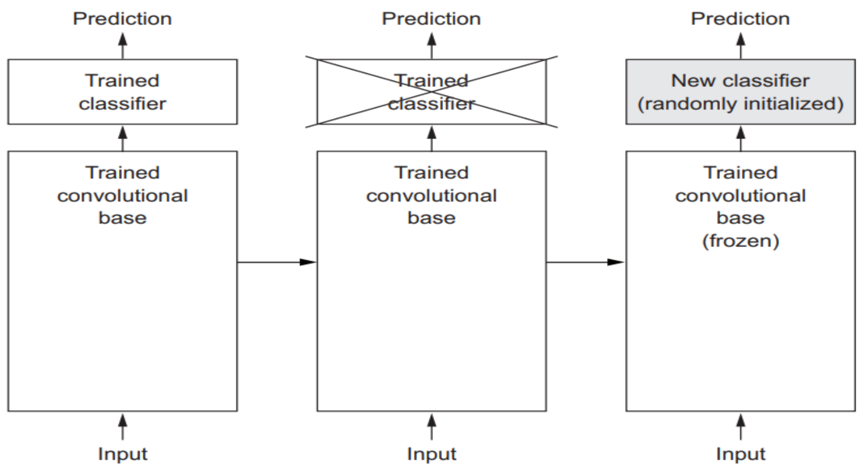

Prediction
Trained classifier
Trained convolutional base
Input
Prediction
~~Trained classifier~~
Trained convolutional base
Input
Prediction
New classifier (randomly initialized)
Trained convolutional base (frozen)
Input

Figure 5.14 Swapping classifiers while keeping the same convolutional base

- Here we take only convolutional base of previously trained network.

- Don’t reuse classifier.

- Because representation learned by convolutional (extracted features patterns) are likely to be more generic and reusable.

- Some of pretrained model are:

  a. LeNet
  b. AlexNet
  c. GoogleNet
  d. ResNet
  e. Xception
  f. Inception V3
  g. ResNet50
  h. VGG16
  i. VGG19
  j. MobileNet

Example: Consider VGG16 model pre-trained on ImageNet dataset

```
from keras.applications import VGG16

conv_base = VGG16(weights='imagenet',
include_top=False, input_shape=(150, 150, 3))
```

1.2 Fine Tuning

Fine-tuning consists of unfreezing a few of the top layers of a frozen model base used for feature extraction, and jointly training both the newly added part of the model (in this case, the fully connected classifier) and these top layers. This is called fine-tuning because it slightly adjusts the more abstract representations of the model being reused, in order to make them more relevant for the problem at hand.

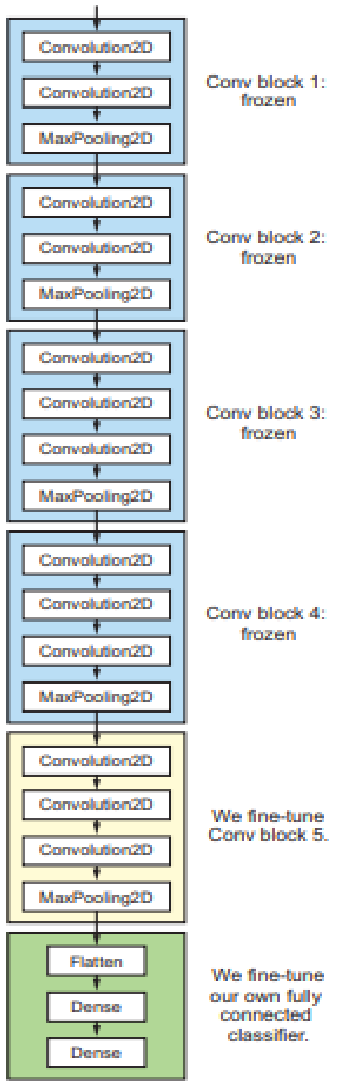


Figure 5.19 Fine-tuning the last convolutional block of the VGG16 network

The steps for fine-tuning a network are as follow:

1. Add your custom network on top of an already-trained base network.
2. Freeze the base network.
3. Train the part you added.
4. Unfreeze some layers in the base network.
5. Jointly train both these layers and the part you added.

Overfitting and Underfitting:

Overfitting and underfitting are two distinct problems in deep learning that arise when training a model. Here are the key differences between overfitting and underfitting:

1. **Definition:**

   a. **Overfitting:** Overfitting occurs when a model performs well on the training data but fails to generalize to new, unseen data. It means the model has learned the noise and outliers in the training set instead of the underlying patterns.

   b. **Underfitting:** Underfitting, on the other hand, happens when a model is too simplistic and fails to capture the underlying patterns and relationships in the training data. It leads to poor performance both on the training data and new data.

2. **Training and Validation Errors:**

   a. **Overfitting:** In the case of overfitting, the model achieves low error on the training data but high error on the validation or test data. The model fits the training data too closely, resulting in a large discrepancy between training and validation errors.

   b. **Underfitting:** For underfitting, both the training and validation errors are typically high. The model is not able to learn the patterns present in the training data and, as a result, performs poorly on both the training and new data.

3. **Model Complexity:**

   a. **Overfitting:** Overfitting often occurs when the model becomes overly complex. It has too many parameters or layers, allowing it to memorize the details of the training data, including noise and outliers.

   b. **Underfitting:** Underfitting, on the other hand, is associated with insufficient model complexity. The model is too simple to capture the complexity and nuances present in the data, resulting in poor performance.

4. **Generalization:**

   a. **Overfitting:** Overfitting leads to poor generalization. The model fails to generalize well to unseen data, performing poorly in real-world scenarios where the data distribution might differ from the training set.
   b. **Underfitting:** Underfitting also indicates poor generalization. The model fails to capture the underlying patterns, leading to suboptimal performance on both the training and new data.

5. **Solutions:**

   a. **Overfitting:** To address overfitting, techniques such as regularization (e.g., L1 or L2 regularization, dropout), cross-validation, or model simplification can be employed to reduce model complexity and prevent memorization of noise.
   b. **Underfitting:** Underfitting can be mitigated by increasing the model's complexity (adding layers, neurons, or parameters), adjusting hyperparameters, or improving feature engineering to capture more relevant information from the data.

In summary, overfitting and underfitting represent opposite problems in deep learning. Overfitting occurs when the model is overly complex and memorizes noise, while underfitting arises from a lack of model complexity and an inability to capture the underlying patterns. Proper regularization, model design, and finding the right balance of complexity are important for achieving optimal performance and generalization in deep learning models.

Data Augmentation :

Data augmentation is a technique widely used in deep learning to increase the diversity and size of training data without collecting new samples. It involves applying various transformations or modifications to the existing data to create new artificial samples that still retain the characteristics of the original data. By introducing these modified samples during training, the model learns to generalize better and becomes more robust to variations in the input data.

The primary goal of data augmentation is to reduce overfitting, which occurs when a model memorizes the training data instead of learning the underlying patterns. Overfitting can happen when the training dataset is small or lacks diversity, leading to poor generalization on unseen data. Data augmentation helps overcome this limitation by artificially expanding the dataset, providing the model with more examples to learn from and preventing it from relying too heavily on specific features or patterns.

Data augmentation techniques can be applied to various types of data, such as images, text, audio, or time series data. In this explanation, let's focus on data augmentation for image data, as it is one of the most common applications.

Here are some commonly used data augmentation techniques in deep learning for image data:

1. **Horizontal and Vertical Flipping:**

   Flipping images horizontally or vertically can create new samples while maintaining the same class label. For example, flipping an image of a cat horizontally still represents a cat.

2. **Rotation:**

   Rotating images by a certain angle (e.g., 90 degrees or 180 degrees) introduces new perspectives and viewpoints, making the model more robust to rotations in the test data.

3. **Scaling and Cropping:**

   Rescaling images to different sizes or cropping them to focus on specific regions can simulate variations in object size, position, or aspect ratio.

4. **Translation:**

   Shifting an image horizontally or vertically can simulate different object locations within the image, allowing the model to learn translation invariance.

Example: In the following function augmentation methodology is represented as

```
datagen =  ImageDataGenerator( rotation_range=40,
width_shift_range=0.2, height_shift_range=0.2, shear_range=0.2,
zoom_range=0.2, horizontal_flip=True, fill_mode='nearest')
```

Dropout:

Dropout is a regularization technique commonly used in transfer learning to mitigate overfitting and improve the generalization of deep learning models. In transfer learning, dropout can be applied to the fully connected or dense layers of the model during the fine-tuning process.

The dropout technique involves randomly dropping out a fraction of the neurons in a layer during training. This means that during each training iteration, a subset of neurons is temporarily ignored, and their outputs are set to zero. By doing so, dropout prevents the network from relying too heavily on specific neurons or learning redundant patterns.

In transfer learning, dropout helps to regularize the model and reduce the over-reliance on the pre-existing learned features from the source task or pre-trained model. It encourages the model to explore and utilize other relevant features present in the target task's data.

By introducing dropout, the model becomes more robust, as it learns to make accurate predictions even when certain neurons are dropped out. This regularization technique reduces the model's sensitivity to noise and helps prevent overfitting by encouraging the model to generalize better to new, unseen data.

During inference or prediction, dropout is typically turned off or scaled down to allow all neurons to contribute to the final output. This ensures that the model utilizes the entire network's learned knowledge without any dropout-induced randomness.

Overall, dropout is a valuable technique in transfer learning as it improves the model's generalization ability, reduces overfitting, and enhances performance on the target task by encouraging the exploration of different feature representations.

Regularization:

The processing of fighting against overfitting this way is called regularization.

1. **Reduce Network Size:**

   Reducing the amount of learnable parameters in the model (which is dictated by the number of layers and the number of units per layer) is the simplest technique to avoid overfitting.

   By reducing the network's size, we limit its capacity to memorize the training data and force it to focus on the most important features and patterns.

   By reducing the size of the network, we aim to simplify its structure and reduce its capacity to memorize the training data. This regularization technique is also known as network pruning or model compression.

2. **Add Weight Regularization:**

   Thus, a common way to mitigate overfitting is to put constraints on the complexity of a network by forcing its weights to take only small values, which makes the distribution of weight values more regular. This is called weight regularization, and it's done by adding to the loss function of the network a cost associated with having large weights.

   This cost comes in two flavours:

   - **L1 regularization:** The cost is proportional to the absolute value of the weight coefficients (the L1 norm of the weights).

   - **L2 regularization:** The cost is proportional to the square of the value of the weight coefficients (the L2 norm of the weights). Don't be misled by the alternative name; weight decay and L2 regularisation are mathematically equivalent.

Unit-4

Sequential or Temporal Data

Sequential or temporal data refers to a type of data that is ordered and dependent on the preceding or succeeding data points in a sequence. It represents information that evolves over time or has a specific order of occurrence. Unlike independent data points, sequential data relies on the context and relationship between neighbouring elements for proper analysis and understanding.

Examples of sequential data include time series data, such as stock prices, weather data, or sensor readings over time. Natural language sentences and speech signals are also sequential data, where the order of words or phonemes matters in determining the meaning.

Analysing and modelling sequential data require methods that can capture the temporal dependencies and patterns within the data. Traditional machine learning algorithms are typically not well-suited for handling sequential data due to their assumption of independence between data points. Instead, specialized techniques have been developed, including recurrent neural networks (RNNs) and long short-term memory (LSTM) networks.

RNNs are designed to process sequential data by maintaining an internal memory or hidden state that is updated at each time step. This allows the network to retain information about the past context while processing new inputs. LSTMs, a variant of RNNs, address the vanishing gradient problem by incorporating gating mechanisms that control the flow of information within the network.

By leveraging the temporal dependencies in sequential data, models can make predictions, generate new sequences, or extract meaningful patterns. Applications of sequential data analysis span various domains, including speech recognition, natural language processing, time series forecasting, and gesture recognition, among others.

What is The Sequential Learning?

Machine learning models that input or output data sequences are known as sequence models. Text streams, audio clips, video clips, time-series data, and other types of sequential data are examples of sequential data. Recurrent Neural Networks (RNNs) are a well-known method in sequence models.

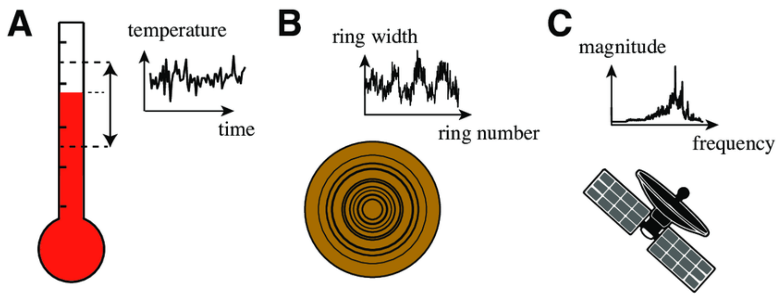
The analysis of sequential data such as text sentences, time-series, and other discrete sequence data prompted the development of Sequence Models. These models are better suited to handle sequential data, whereas Convolutional Neural Networks are better suited to treat spatial data.

The crucial element to remember about sequence models is that the data we’re working with are no longer independently and identically distributed (i.i.d.) samples, and the data are reliant on one another due to their sequential order. For speech recognition, voice recognition, time series prediction, and natural language processing, sequence models are particularly popular.

What is Sequential Data?

When the points in the dataset are dependent on the other points in the dataset, the data is termed sequential. A Timeseries is a common example of this, with each point reflecting an observation at a certain point in time, such as a stock price or sensor data. Sequences, DNA sequences, and meteorological data are examples of sequential data.

In other words, sequential we can term video data, audio data, and images up to some extent as sequential data. Below are a few basic examples of sequential data.



Below I have listed some popular machine learning applications that are based on sequential data:

- Time Series: a challenge of predicting time series, such as stock market projections.
- Text mining and sentiment analysis are two examples of natural language processing (e.g., Learning word vectors for sentiment analysis)
- Machine Translation: Given a single language input, sequence models are used to translate the input into several languages. Here’s a recent poll.
- Image captioning is assessing the current action and creating a caption for the image.
- Deep Recurrent Neural Network for Speech Recognition Deep Recurrent Neural Network for Speech Recognition
- Recurrent neural networks are being used to create classical music.

- Recurrent Neural Network for Predicting Transcription Factor Binding Sites based on DNA Sequence Analysis

In order to efficiently model with this data or to get as much information, it contains a traditional machine algorithm that will not help as much. To deal with such data there are some sequential models available and you might have heard some of those.

Sequential Models:

Sequential models in deep learning refer to a class of models designed to handle sequential or time-dependent data, such as text, speech, or time series data. These models are specifically designed to capture and understand the temporal dependencies and patterns present in the input sequences. They are widely used in tasks like natural language processing (NLP), speech recognition, machine translation, sentiment analysis, and more.

The fundamental characteristic of sequential models is their ability to process input data in a sequential manner, considering the order and relationships between elements within the sequence. Unlike traditional feed-forward neural networks, which treat each input independently, sequential models consider the context and history of the sequence.

The two primary types of sequential models are Recurrent Neural Networks (RNNs) and Transformer models. Let's explore each of them:

1. **Recurrent Neural Networks (RNNs):**

   RNNs are a class of models that process sequential data by maintaining an internal state or memory. They operate on the input sequence one element at a time, where the output at each step depends on the current input and the previous internal state. RNNs are particularly suitable for tasks that require an understanding of the sequence's temporal dynamics. The key component of an RNN is the recurrent layer, which consists of recurrent units (such as the Long Short-Term Memory, LSTM, or Gated Recurrent Unit, GRU) that maintain and update the hidden state as new elements of the sequence are processed. This hidden state captures the contextual information from previous steps and influences the predictions at each step. RNNs can process input sequences of arbitrary length and have the ability to model both short-term and long-term dependencies.

   However, standard RNNs suffer from vanishing or exploding gradient problems, which limit their ability to capture long-term dependencies effectively. To address this issue, variants like LSTM and GRU were introduced, which incorporate gating mechanisms to better control the flow of information through time.

2. **Transformer Models:**

   Transformer models have gained significant attention in recent years, especially in NLP tasks. Unlike RNNs, Transformers do not explicitly maintain an internal state or rely on sequential processing. Instead, they process the entire input sequence simultaneously in parallel. Transformers are based on a self-attention mechanism that allows them to capture dependencies between different positions in the sequence effectively.

   The core building blocks of Transformers are multi-head self-attention layers and position-wise feed-forward networks. Self-attention allows each position in the sequence to attend to all other positions, capturing the relationships between different elements. This mechanism enables Transformers to model long-range dependencies more effectively compared to traditional sequential models. Additionally, Transformers incorporate positional encoding to convey the order of the input sequence.

   Transformers have shown remarkable performance in various natural language processing tasks, such as machine translation (e.g., Google's Transformer-based model, "Transformer"), text generation (e.g., OpenAI's GPT), and language understanding (e.g., BERT).

   Both RNNs and Transformers have their strengths and are applicable in different scenarios. RNNs are better suited for tasks that require modelling sequential dependencies, while Transformers excel in capturing global relationships in the sequence.

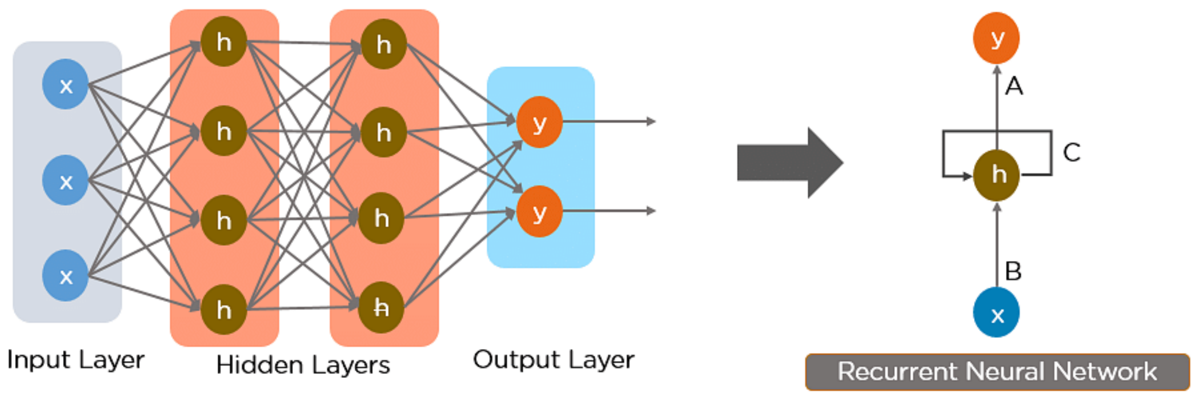
   It's worth noting that sequential models can be combined with other architectural components, such as convolutional layers, to handle both local and global dependencies simultaneously, as in Convolutional Recurrent Neural Networks (CRNNs) or Convolutional Transformers (CTs). These hybrid models leverage the strengths of both convolutional and sequential processing to tackle a wide range of tasks.

Recurrent Neural Network:

What are recurrent neural networks?

RNNs are a powerful and robust type of neural network, and belong to the most promising algorithms in use because it is the only one with an internal memory.

Because of their internal memory, RNNs can remember important things about the input they received, which allows them to be very precise in predicting what's coming next. This is why they're the preferred algorithm for sequential data like time series, speech, text, financial data, audio, video, weather and much more. Recurrent neural networks can form a much deeper understanding of a sequence and its context compared to other algorithms.



How RNN works:

The Recurrent Neural Network consists of multiple fixed activation function units, one for each time step. Each unit has an internal state which is called the hidden state of the unit. This hidden state signifies the past knowledge that the network currently holds at a given time step. This hidden state is updated at every time step to signify the change in the knowledge of the network about the past.

The hidden state is updated using the following recurrence relation:-

The formula for calculating the current state:

$$h_t = f(h_{t-1}, x_t)$$

where:

h_t -> current state
h_{t-1} -> previous state
x_t -> input state

Long Short-Term Memory(LSTM):

LSTM (Long Short-Term Memory) is a type of recurrent neural network (RNN) architecture that is widely used in deep learning for sequential data processing. It was specifically designed to overcome the limitations of traditional RNNs, such as the vanishing gradient problem, which hampers their ability to capture long-term dependencies in sequences.

In an LSTM, information is stored and passed through a series of memory cells. Each memory cell has three main components: an input gate, a forget gate, and an output gate. These gates control the flow of information into, out of, and within the memory cell.

Here's a step-by-step explanation of the LSTM architecture:

1. **Input Processing:**
   - At each time step in the sequence, the LSTM receives an input vector (x_t) representing the current element of the sequence.
   - The input vector is then combined with the previous hidden state (h_{t-1}) and passed through a linear transformation to produce an input activation vector (a_t).

2. **Forget Gate:**
   - The forget gate (f_t) determines how much information from the previous memory cell (c_{t-1}) should be forgotten.
   - It takes the input activation vector (a_t) and passes it through a sigmoid function, which outputs values between 0 and 1.
   - The forget gate output is element-wise multiplied with the previous memory cell (c_{t-1}) to determine which information to discard.

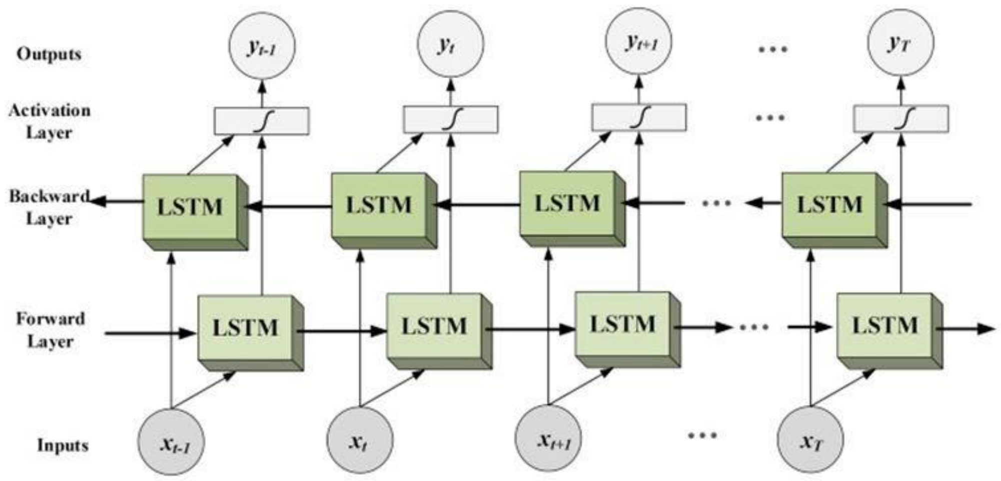
3. **Input Gate:**
   - The input gate (i_t) decides which new information should be stored in the memory cell.
   - The input activation vector (a_t) is passed through a sigmoid function to generate the input gate output.
   - Additionally, the input activation vector (a_t) is also passed through a tanh function to produce a new candidate cell state (C_tilde).
   - The input gate output and the candidate cell state are element-wise multiplied to obtain the information to be added to the memory cell.

4. **Update Memory Cell:**
    - The memory cell (c_t) is updated by combining the forget gate output and the new information from the input gate.
    - The forget gate output (f_t) is multiplied with the previous memory cell (c_{t-1}) to forget irrelevant information.
    - The input gate output (i_t) is multiplied with the candidate cell state (C_tilde) to add new relevant information.
    - The forget gate output and the input gate output are summed to update the memory cell: c_t = f_t * c_{t-1} + i_t * C_tilde.

5. **Output Gate:**
    - The output gate (o_t) determines how much of the memory cell should be exposed as the output of the LSTM.
    - The input activation vector (a_t) is passed through a sigmoid function to generate the output gate output.
    - The memory cell (c_t) is passed through a tanh function to squash the values between -1 and 1.
    - The output gate output is multiplied element-wise with the squashed memory cell to obtain the LSTM's hidden state (h_t).

The LSTM architecture allows the network to learn when to forget or remember information from previous time steps, effectively capturing long-term dependencies in the data. The presence of gates and the memory cell separate the memory and control flow, allowing gradients to propagate more easily during training and mitigating the vanishing gradient problem.

LSTMs have been successfully applied to a wide range of sequential tasks, including speech recognition, machine translation, sentiment analysis, time series forecasting, and more. They have proven particularly effective when dealing with sequences that exhibit long-term dependencies or require modelling of context over extended periods.



Outputs
y_{t-1}
y_t
y_{t+1}
y_T
Activation Layer
Backward Layer
LSTM
Forward Layer
Inputs
x_{t-1}
x_t
x_{t+1}
x_T

Text generation with LSTM:

In this section, we’ll explore how recurrent neural networks can be used to generate sequence data.

We’ll use text generation as an example, but the exact same techniques can be generalized to any kind of sequence data: you could apply it to sequences of musical notes in order to generate new music, to timeseries of brushstroke data (for example, recorded while an artist paints on an iPad) to generate paintings stroke by stroke, and so on. Sequence data generation is in no way limited to artistic content generation.

It has been successfully applied to speech synthesis and to dialogue generation for chatbots. The Smart Reply feature that Google released in 2016, capable of automatically generating a selection of quick replies to emails or text messages, is powered by similar techniques.

Gated Recurrent Unit(GRU):

GRU stands for Gated Recurrent Unit, which is a type of recurrent neural network (RNN) architecture. GRU was introduced as a variation of the traditional RNN model that addresses the vanishing gradient problem and allows for better modelling of long-term dependencies in sequential data.

To understand GRU, let's first briefly review the working principle of a standard RNN. In an RNN, information flows from one step to the next in a sequence, and each step maintains an internal hidden state that captures the information from previous steps. However, the simple RNN suffers from the vanishing gradient problem, where the gradients diminish exponentially as they propagate through time, making it difficult to learn dependencies over long sequences.

The GRU architecture introduces two main changes compared to the standard RNN: the introduction of gating mechanisms and the use of a more complex hidden state representation.

1. **Hidden State Representation:**

    - In a standard RNN, the hidden state at each step captures the entire history of the sequence. However, GRU introduces a new hidden state representation, which consists of two components: the previous hidden state (h_t-1) and the new candidate activation (h~_t). This allows the model to selectively update and reset information at each step.

2. **Gating Mechanisms:**

    GRU incorporates two gating mechanisms, namely the "reset gate" and the "update gate," which control the flow of information within the model.

    - Reset Gate (r_t): The reset gate determines how much of the previous hidden state should be forgotten or reset. It takes the current input (x_t) and the previous hidden state (h_t-1) as inputs and outputs a value between 0 and 1 for each element of the hidden state. The reset gate is computed as follows:
      r_t = sigmoid(W_r * [h_t-1, x_t])

- Update Gate (z_t): The update gate determines how much of the previous hidden state should be carried forward to the current step. Similar to the reset gate, it takes the current input (x_t) and the previous hidden state (h_t-1) as inputs and outputs a value between 0 and 1 for each element of the hidden state. The update gate is computed as follows:
  $z_t = \mathrm{sigmoid}(W_z * [h_{t-1}, x_t])$

The sigmoid function ensures that the values of the reset gate and update gate lie between 0 and 1, acting as control signals for information flow.

3. **Candidate Activation:**

   The candidate activation (h~_t) is computed by combining the current input (x_t) with a transformed version of the previous hidden state (h_t-1) that is controlled by the reset gate.

   The candidate activation is computed as follows:

   $\tilde{h}_t = \tanh(W_h * [r_t \odot h_{t-1}, x_t])$

   Here, $\odot$ denotes element-wise multiplication, and tanh is the hyperbolic tangent function that squashes the values between -1 and 1.

4. **Update Hidden State:**

   Finally, the new hidden state (h_t) is computed by interpolating between the previous hidden state (h_t-1) and the candidate activation (h~_t) using the update gate.

   The update hidden state is computed as follows:

   $h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t$

   This equation determines how much of the previous hidden state should be retained (controlled by 1 - z_t) and how much of the candidate activation should be integrated (controlled by z_t).

By incorporating the reset gate and update gate, the GRU can selectively retain or discard information from the previous hidden state, allowing it to capture long-term dependencies in the sequence while avoiding the vanishing gradient problem. The gating mechanisms also enable the GRU to handle varying time scales and adaptively update its hidden state based on the input.

GRU has become a popular choice in various sequence modelling tasks, including natural language processing, speech recognition, machine translation, and time series analysis, due to its effectiveness in capturing long-term dependencies and mitigating the vanishing gradient problem.

Compare LSTM vs RNN:

Here is a comparison of long short-term memory (LSTM) and recursive neural networks (RNNs).

	Long Short-Term Memory (LSTM)	Recurrent Neural Network (RNN)
Type of network	Recurrent neural network.	Recurrent neural network.
How it works	Uses gates to control the flow of information through the network.	Uses recursive connections to process sequential data.
Suitable for	Long-term dependencies.	Short-term dependencies.
Common applications	Language modelling, time series prediction, sentiment analysis, speech recognition, image captioning.	Language modelling, time series prediction, speech recognition.

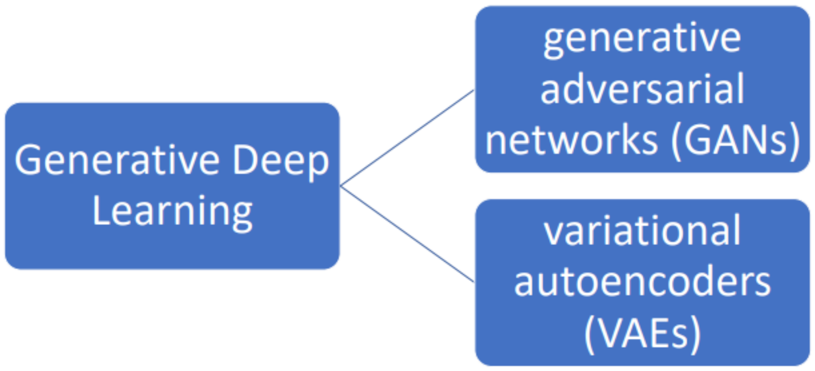
Unit-5

Generative Deep Learning:

Generative deep learning refers to the application of deep learning models to generate new data that has similar characteristics to the training data. In other words, generative deep learning models are designed to learn the underlying patterns and distributions of a given dataset, and then generate new data samples that follow these same patterns.

Generative deep learning models can be used for a wide range of tasks, including image and audio synthesis, text generation, and data augmentation. Some common types of generative deep learning models include generative adversarial networks (GANs), variational autoencoders (VAEs), and autoregressive models.

Generative deep learning is a rapidly growing area of research, with many exciting applications across a wide range of industries. Some potential use cases include generating realistic images or audio for virtual reality or gaming applications, generating synthetic data to augment training sets for machine learning models, and generating text for natural language processing applications such as chatbots or language translation.


Generative Deep Learning
generative adversarial networks (GANs)
variational autoencoders (VAEs)

How do you generate sequence data?

The universal way to generate sequence data in deep learning is to train a network (usually an RNN or a convnet) to predict the next token or next few tokens in a sequence, using the previous tokens as input.

For instance, given the input “the cat is on the ma,” the network is trained to predict the target t, the next character. As usual when working with text data, tokens are typically words or characters, and any network that can model the probability of the next token given the previous ones is called a language model. A language model captures the latent space of language: its statistical structure.

Once you have such a trained language model, you can sample from it (generate new sequences): you feed it an initial string of text (called conditioning data), ask it to generate the next character or the next word (you can even generate several tokens at once), add the generated output back to the input data, and repeat the process many times (see figure 8.1).

This loop allows you to generate sequences of arbitrary length that reflect the structure of the data on which the model was trained: sequences that look almost like human-written sentences.

In the example we present in this section, you’ll take a LSTM layer, feed it strings of N characters extracted from a text corpus, and train it to predict character N + 1. The output of the model will be a softmax over all possible characters: a probability distribution for the next character. This LSTM is called a character-level neural language model.

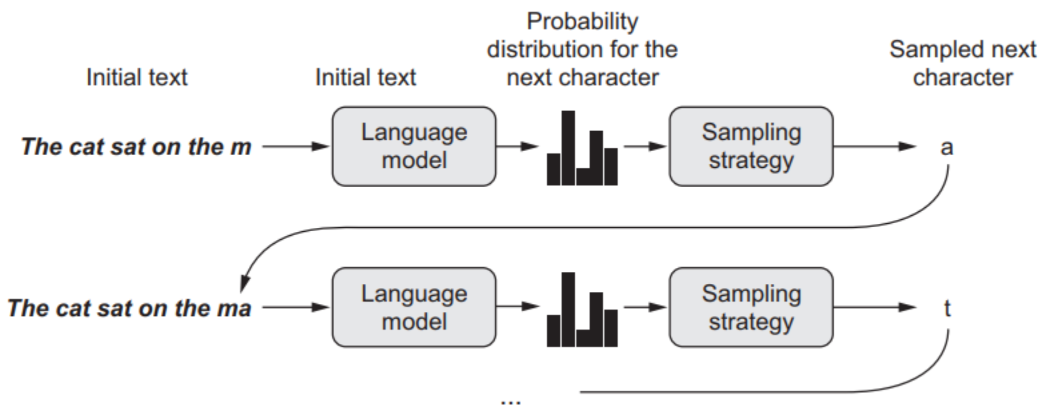


Figure 8.1 The process of character-by-character text generation using a language model

Neural Style Transfer:

Neural Style Transfer is a deep learning technique that combines the content of one image with the style of another image to create a new image. The basic idea behind neural style transfer is to use a pre-trained convolutional neural network (CNN) to extract the content and style features from the input images, and then optimize a new image that preserves the content features of the input image while matching the style features of the style image.

The process of neural style transfer typically involves the following steps:

1. **Pre-processing:**
   The input image and style image are pre-processed to remove any noise or artifacts that could interfere with the style transfer process.

2. **Feature Extraction:**
   A pre-trained CNN such as VGG is used to extract the feature maps of the input image and the style image.

3. **Style Transfer:**
   The style features of the style image are extracted and matched with the content features of the input image to create a new image that preserves the content of the input image while adopting the style of the style image.

4. **Post-processing:**
   The generated image is post-processed to enhance its visual quality and remove any artifacts.

Neural style transfer has many applications, including artistic style transfer, image colorization, and image synthesis. It has also been used in the fashion industry to generate new designs based on existing fashion styles, and in the film industry to create visual effects and generate synthetic scenes.

Auto-Encoder:

An autoencoder is a type of neural network that is used for unsupervised learning of compressed representations of input data. Autoencoders consist of two main components: an encoder and a decoder.

The encoder takes an input data sample and maps it to a lower-dimensional representation, often called the latent space or code. The latent space is typically smaller in dimension than the input data and contains a compressed representation of the input data. The decoder then takes the encoded representation and maps it back to the original input data space. The autoencoder is trained to minimize the difference between the input data and the reconstructed output data.

Autoencoders are used in a variety of applications such as image denoising, data compression, anomaly detection, and feature extraction. One key benefit of autoencoders is their ability to learn useful representations of high-dimensional input data in an unsupervised manner, without the need for labelled data.

Autoencoders can be extended and modified in various ways, such as stacked autoencoders, denoising autoencoders, variational autoencoders, and convolutional autoencoders, to name a few.

Generative Adversarial Network (GAN):

A Generative Adversarial Network (GAN) is a type of generative deep learning model that consists of two neural networks: a generator network and a discriminator network.

The generator network takes a random noise vector as input and generates a new sample in the data space, while the discriminator network takes a sample from either the training data or the generator network and classifies it as real or fake.

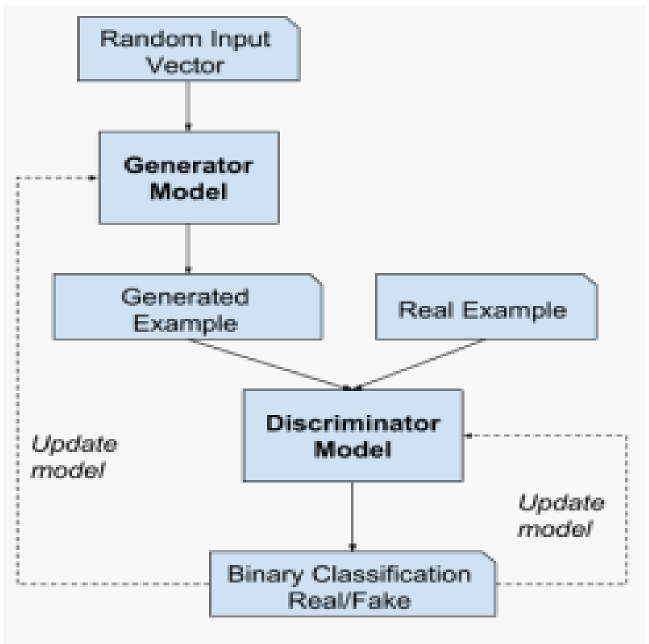
The two networks are trained together in an adversarial process, where the generator network tries to generate samples that are indistinguishable from real data samples, while the discriminator network tries to correctly distinguish between real and fake samples.

The generator network is trained to maximize the error of the discriminator network on the generated samples, while the discriminator network is trained to minimize the error between the real and generated samples.

As the two networks are trained together, the generator network learns to generate increasingly realistic samples, while the discriminator network becomes better at distinguishing between real and fake samples. The ultimate goal of the GAN is to generate new samples that are indistinguishable from the real data distribution.

GANs have been successfully applied to a wide range of tasks, including image and video synthesis, text generation, and data augmentation. Some examples of GAN applications include generating photorealistic images, creating realistic 3D models, and generating synthetic data to augment training sets for machine learning models.

GANs are a powerful tool in the deep learning arsenal, but they can also be challenging to train due to their complex architecture and the instability of the training process. However, with careful tuning and regularization techniques, GANs can produce impressive results.



Example of the Generative Adversarial Network Model Architecture

Advantages of GANs in Deep Learning:

1. **Generative Modelling:**
   GANs excel at generative modelling tasks by learning to capture complex patterns and distribution of the training data. They can generate new samples that closely resemble the original data distribution, providing a valuable tool for data augmentation, creativity, and synthesis.

2. **Realistic Outputs:**
   GANs are known for producing outputs that are highly realistic and indistinguishable from real samples. This makes them suitable for applications like image synthesis, video generation, or audio generation, where generating high-quality and visually appealing content is crucial.

3. **Unsupervised Learning:**
   GANs operate in an unsupervised learning framework, meaning they don't require labelled data during training. They learn directly from the unlabelled training data and can discover underlying patterns and structure without explicit supervision. This makes GANs useful in scenarios where labelled data is scarce or costly to obtain.

4. **Feature Learning:**
   GANs learn to extract meaningful representations and features from the data during the training process. The generator and discriminator networks develop intricate internal representations, which can be leveraged for transfer learning or fine-tuning on downstream tasks. This feature learning capability of GANs is valuable for various computer vision and natural language processing tasks.

Disadvantages of GANs in Deep Learning:

1. **Training Instability:**
   GANs can be challenging to train and prone to instability. Finding the right balance between the generator and discriminator networks is crucial. The training process often involves fine-tuning hyperparameters, such as learning rates, network architectures, and regularization techniques. In some cases, GANs can suffer from mode collapse, where the generator fails to capture the full diversity of the target distribution.

2. **Mode Dropping:**
   GANs may produce samples that only represent a subset of the target distribution, ignoring some important modes or variations present in the training data. This can limit their diversity and generate biased outputs that do not fully capture the complexity of the real data distribution.

3. **Evaluation Challenges:**
   Evaluating GANs is a non-trivial task. Traditional evaluation metrics like accuracy or loss functions may not capture the quality or diversity of the generated samples effectively. Metrics such as Inception Score or Fréchet Inception Distance (FID) have been proposed, but they have their own limitations and may not provide a comprehensive evaluation.

4. **Computationally Intensive:**
   GANs, especially deep convolutional architectures, can be computationally demanding and require significant computational resources, memory, and training time. Training large-scale GANs on high-resolution images or complex datasets can be resource-intensive and may require specialized hardware or distributed computing setups.

5. **Lack of Interpretability:**
   GANs are considered black box models, meaning it can be challenging to interpret the internal workings of the generator or discriminator networks. Understanding the specific features or representations that lead to the generation of specific outputs is a complex task, limiting the interpretability of the model.

Variational Auto Encoder (VAE):

A Variational Autoencoder (VAE) is a type of generative deep learning model that combines the concepts of autoencoders and variational inference to learn a compressed representation of the input data that can be used to generate new data samples.

Like traditional autoencoders, a VAE consists of an encoder network that maps the input data to a compressed representation, and a decoder network that maps the compressed representation back to the original data space. However, unlike traditional autoencoders, a VAE maps the input data to a probability distribution over the compressed representation, instead of a single point estimate.

The encoder network of a VAE typically maps the input data to a mean and standard deviation vector that describe the probability distribution over the compressed representation. The decoder network then takes a random sample from this distribution and maps it back to the original data space.

During training, the model is optimized to maximize the likelihood of the input data given the compressed representation, while also minimizing the divergence between the learned distribution and a prior distribution (often a standard normal distribution).

The use of a probability distribution over the compressed representation allows a VAE to generate new data samples by sampling from the learned distribution and mapping the sample back to the data space using the decoder network. VAEs have been successfully applied to a wide range of tasks, including image and text generation, data compression, and anomaly detection.

Advantages of VAEs in Deep Learning:

1. **Latent Space Representation:**
   VAEs can learn a compressed and continuous representation of the input data in the latent space. This allows for meaningful interpolation and exploration of the latent space, enabling the generation of new samples with controlled variations.

2. **Generative Modelling:**
   VAEs are generative models, capable of generating new data samples that resemble the training data distribution. By sampling from the latent space, VAEs can produce novel and diverse outputs, making them useful for tasks such as image synthesis, text generation, and anomaly detection.

3. **Probabilistic Framework:**
   VAEs are built on a probabilistic framework, allowing for uncertainty estimation. Instead of producing a single output, VAEs provide a distribution in the latent space, enabling more robust and reliable predictions. This is particularly beneficial in applications such as anomaly detection and semi-supervised learning.

4. **Regularization Effect:**
   VAEs inherently introduce a regularization effect on the latent space during training. The model learns to encode the data by maximizing the likelihood of the training samples and simultaneously minimizing the Kullback-Leibler (KL) divergence between the learned distribution and a prior distribution. This regularization helps to reduce overfitting and encourages the model to capture salient features in the latent space.

Disadvantages of VAEs in Deep Learning:

1. **Blurred Generations:**
   VAEs often produce generative samples that can be slightly blurry or less sharp compared to the original data. This is due to the trade-off between reconstruction fidelity and the regularization effect imposed by the KL divergence term. Variations introduced in the latent space during sampling may lead to less precise reconstructions.

2. **Difficulty Capturing Complex Structures:**
   VAEs may struggle to capture complex and high-dimensional structures in the data, particularly when the latent space is restricted in capacity. The assumption of a simple prior distribution and the regularizing effect of the KL divergence can limit the model's ability to represent complex variations.

3. **Mode Collapse:**
   VAEs can suffer from mode collapse, where the model fails to capture the full diversity of the training data. This can lead to over-reliance on a few dominant modes, resulting in less diverse and repetitive generations.

4. **Limited Explicit Inference:**
   VAEs do not explicitly model the posterior distribution over the latent space given an input. The model provides an approximate posterior distribution through the encoder network, but it may not capture complex or multi-modal posterior distributions accurately.

5. **Sensitivity to Hyperparameters:**
   VAEs' performance can be sensitive to hyperparameter choices, such as the dimensionality of the latent space, the choice of prior distribution, and the weight assigned to the KL divergence term. Proper tuning of these hyperparameters is crucial for achieving good performance.

Classical supervised tasks in Deep Learning:

It refer to a set of common problems that can be tackled using supervised learning techniques with deep neural networks. In these tasks, the goal is to train a model to learn a mapping between input data and their corresponding target labels. Deep learning models excel in handling large and complex datasets, capturing intricate patterns, and automatically extracting relevant features. Here, we'll explain some of the most common classical supervised tasks in detail:

1. **Image Classification:**
   Image classification involves categorizing images into predefined classes or labels. Given a dataset of labelled images, the goal is to train a deep neural network to correctly predict the class of unseen images. Convolutional Neural Networks (CNNs) are widely used for image classification tasks due to their ability to capture spatial dependencies in images. The model learns to extract hierarchical features, starting from low-level edges and textures to high-level object representations, enabling accurate classification.

2. **Object Detection:**
   Object detection involves identifying and localizing multiple objects within an image. Unlike image classification, object detection not only assigns class labels to objects but also provides bounding box coordinates for each detected object. Deep learning-based object detection typically utilizes frameworks such as Region-based CNNs (R-CNN), Faster R-CNN, or Single Shot Multibox Detector (SSD). These models leverage CNNs for feature extraction and employ additional layers to predict object classes and bounding box coordinates.

3. **Semantic Segmentation:**
   Semantic segmentation involves pixel-level labelling of an image, where each pixel is assigned a class label. It enables understanding the fine-grained details of an image, such as object boundaries and precise object localization. Fully Convolutional Networks (FCNs) are commonly used for semantic segmentation tasks.
   These networks produce dense predictions by converting traditional CNN architectures into fully convolutional architectures, enabling pixel-wise predictions.

4. **Object Instance Segmentation:**
   Object instance segmentation is an extension of semantic segmentation that aims to identify and differentiate individual instances of objects within an image. The goal is to provide not only the class label and localization but also the pixel-level mask for each object instance. Mask R-CNN is a popular deep learning model used for object instance segmentation. It combines the object detection capabilities of Faster R-CNN with the pixel-level segmentation achieved by FCNs.

5. **Text Classification:**
   Text classification involves assigning predefined categories or labels to textual data, such as documents, emails, or social media posts. Recurrent Neural Networks (RNNs), particularly Long Short-Term Memory (LSTM) or Gated Recurrent Unit (GRU), are commonly used for text classification tasks. RNNs can capture sequential dependencies in text data, enabling them to learn from the context and meaning of words in a sentence or document.

6. **Named Entity Recognition (NER):**
   NER involves identifying and classifying named entities, such as names of people, organizations, locations, or specific terms, within a text. This task is crucial for information extraction and natural language understanding. Bi-directional LSTM-CRF models are often used for NER tasks. These models combine bidirectional LSTM layers to capture contextual information and a Conditional Random Field (CRF) layer to model the dependencies between adjacent labels.