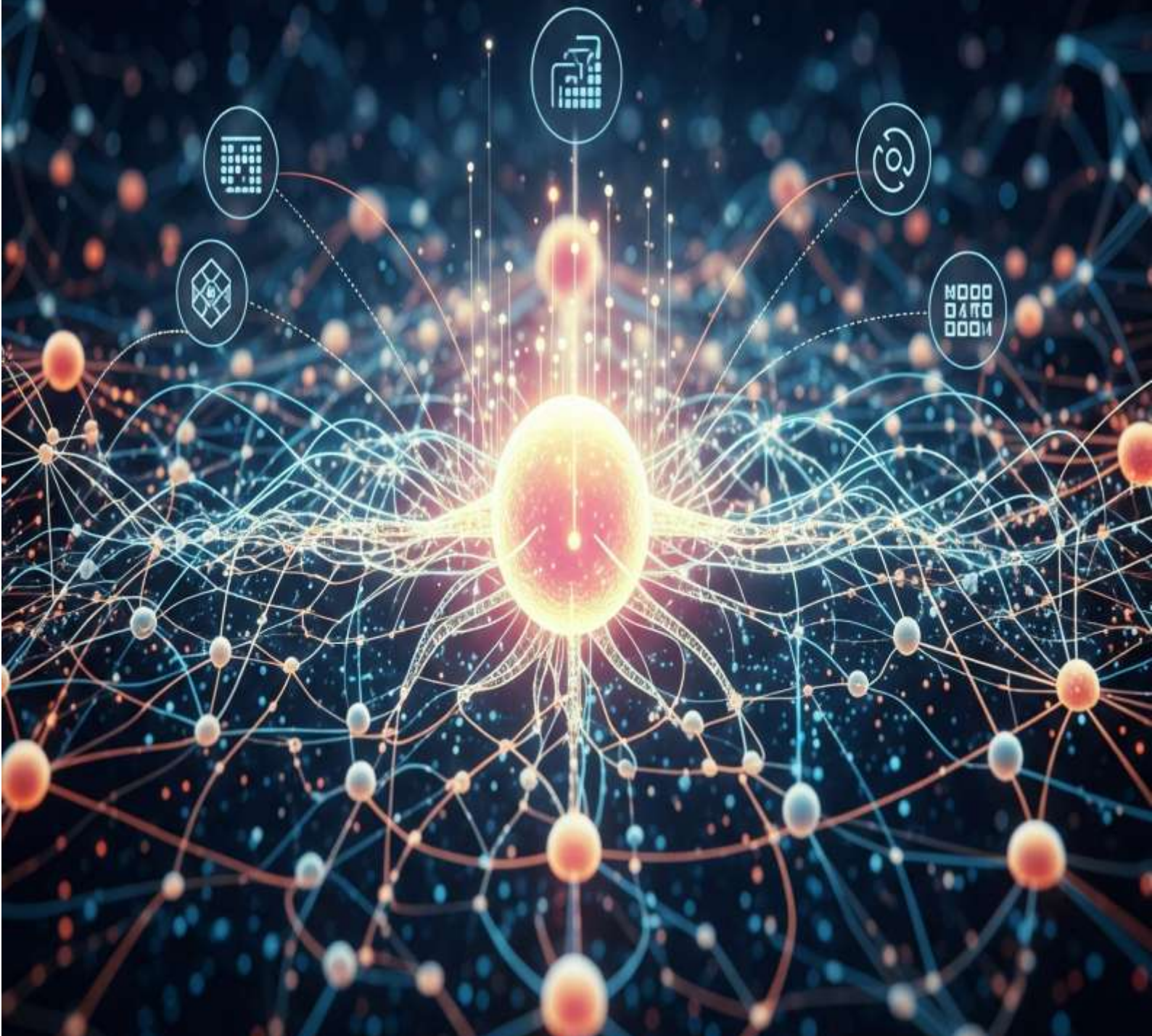
DEEP LEARNING

UNVEILING THE FUTURE TO NETWORKS



Deep Learning

Deep Learning is a subset of Artificial Intelligence (AI) that helps machines to learn from large datasets using multi-layered neural networks. It automatically finds patterns and makes predictions and eliminates the need for manual feature extraction. Deep Learning tutorial covers the basics to advanced topics making it perfect for beginners and those with experience.

Introduction to Neural Networks

Neural Networks are fundamentals of deep learning inspired by human brain. It consists of layers of interconnected nodes or "neurons" each designed to perform specific calculations. These nodes receive input data, process it through various mathematical functions and pass the output to subsequent layers.

- Neural Networks
- Biological Neurons vs Artificial Neurons
- Single Layer Perceptron
- Multi-Layer Perceptron
- Artificial Neural Networks (ANNs)
- Types of Neural Networks
- Architecture and Learning process in neural network

What is a Neural Network

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

In this article, we will explore the fundamentals of neural networks, their architecture, how they work and their applications in various fields. Understanding neural networks is essential for anyone interested in the advancements of artificial intelligence.

Understanding Neural Networks in Deep Learning

Neural networks are capable of learning and identifying patterns directly from data without pre-defined rules. These networks are built from several key components:

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

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

1. **Input Computation**: Data is fed into the network.
2. **Output Generation**: Based on the current parameters, the network generates an output.

3. **Iterative Refinement**: The network refines its output by adjusting weights and biases, gradually improving its performance on diverse tasks.

In an adaptive learning environment:

- The neural network is exposed to a simulated scenario or dataset.
- Parameters such as weights and biases are updated in response to new data or conditions.
- With each adjustment, the network's response evolves allowing it to adapt effectively to different tasks or environments.

The image illustrates the analogy between a biological neuron and an artificial neuron, showing how inputs are received and processed to produce outputs in both systems.

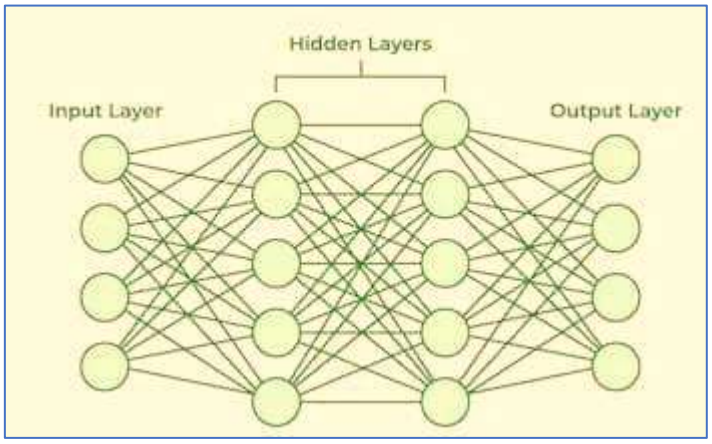
Importance of Neural Networks

Neural networks are important in identifying complex patterns, solving intricate challenges and adapting to dynamic environments. Their ability to learn from vast amounts of data is transformative, impacting technologies like **natural language processing**, **self-driving vehicles** and **automated decision-making**.

Neural networks streamline processes, increase efficiency and support decision-making across various industries. As a backbone of artificial intelligence, they continue to drive innovation, shaping the future of technology.

Layers in Neural Network Architecture

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



Working of Neural Networks

1. Forward Propagation

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

1. Linear Transformation: Each neuron in a layer receives inputs which are multiplied by the weights associated with the connections. These products are summed together and a bias is added to the sum. This can be represented mathematically as:

$$z=w1x1+w2x2+...+wnxn+bz=w1x1+w2x2+...+wnxn+b$$

where

- ww represents the weights
- xx represents the inputs
- bb is the bias

2. Activation: The result of the linear transformation (denoted as z) is then passed through an activation function. The activation function is crucial because it introduces non-linearity into the system, enabling the network to learn more complex patterns. Popular activation functions include ReLU, sigmoid and tanh.

3. Backpropagation

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

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

4. Iteration

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

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

Example of Email Classification

Let's consider a record of an email dataset:

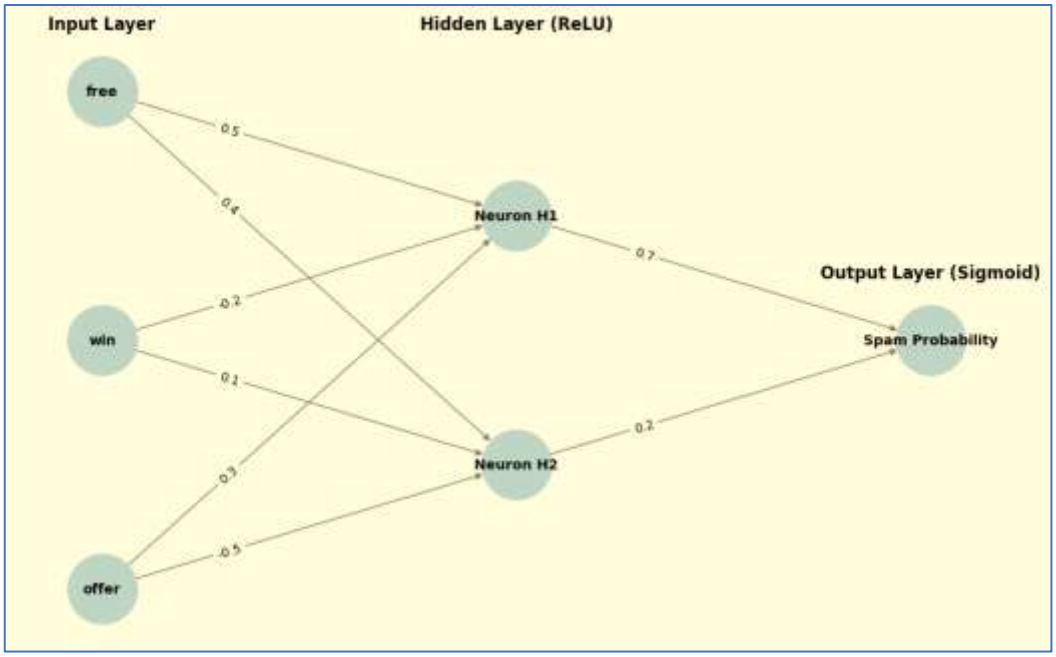
Email ID	Email Content	Sender	Subject Line	Label
1	"Get free gift cards now!"	spam@example.com	"Exclusive Offer"	1

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

The feature vector of the record can be presented as:

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

How Neurons Process Data in a Neural Network



In a neural network, input data is passed through multiple layers, including one or more hidden layers. Each neuron in these hidden layers performs several operations, transforming the input into a usable output.

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

2. Hidden Layer: The input vector is passed through the hidden layer. Each neuron in the hidden layer performs two primary operations: a weighted sum followed by an activation function.

Weights:

- Neuron H1: [0.5,–0.2,0.3]
- Neuron H2: [0.4,0.1,–0.5]

Input Vector: [1,0,1]

Weighted Sum Calculation

- **For H1: (**1×0.5)+(0×–0.2)+(1×0.3)=0.5+0+0.3=0.8
- **For H2:** (1×0.4)+(0×0.1)+(1×–0.5)=0.4+0–0.5=–0.1

Activation Function

Here we will use ReLu activation function:

- **H1 Output:** ReLU(0.8)= 0.8
- **H2 Output:** ReLu(-0.1) = 0

3. Output Layer

The activated values from the hidden neurons are sent to the output neuron where they are again processed using a weighted sum and an activation function.

- **Output Weights:** [0.7, 0.2]
- **Input from Hidden Layer:** [0.8, 0]
- **Weighted Sum:** (0.8×0.7)+(0×0.2)=0.56+0=0.56
- **Activation (Sigmoid):** $\sigma(0.56)=11+e-0.56\approx0.636\sigma(0.56)=1+e-0.561\approx0.636$

4. Final Classification

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

Neural Network for Email Classification Example

Learning of a Neural Network

1. Learning with Supervised Learning

In supervised learning, a neural network learns from labeled input-output pairs provided by a teacher. The network generates outputs based on inputs and by comparing these outputs to the known desired outputs, an error signal is created. The network iteratively adjusts its parameters to minimize errors until it reaches an acceptable performance level.

2. Learning with Unsupervised Learning

Unsupervised learning involves data without labeled output variables. The primary goal is to understand the underlying structure of the input data (X). Unlike supervised learning, there is no instructor to guide the process. Instead, the focus is on modeling data patterns and relationships, with techniques like clustering and association commonly used.

3. Learning with Reinforcement Learning

Reinforcement learning enables a neural network to learn through interaction with its environment. The network receives feedback in the form of rewards or penalties, guiding it to find an optimal policy or strategy that maximizes cumulative rewards over time. This approach is widely used in applications like gaming and decision-making.

Types of Neural Networks

There are *seven* types of neural networks that can be used.

- **Feedforward Networks:** It is a simple artificial neural network architecture in which data moves from input to output in a single direction.
- **Singlelayer Perceptron:** It has one layer and it applies weights, sums inputs and uses activation to produce output.
- **Multilayer Perceptron (MLP):** It is a type of feedforward neural network with three or more layers, including an input layer, one or more hidden layers and an output layer. It uses nonlinear activation functions.
- **Convolutional Neural Network (CNN):** It is designed for image processing. It uses convolutional layers to automatically learn features from input images, enabling effective image recognition and classification.
- **Recurrent Neural Network (RNN):** Handles sequential data using feedback loops to retain context over time.
- **Long Short-Term Memory (LSTM):** A type of RNN with memory cells and gates to handle long-term dependencies and avoid vanishing gradients.

Implementation of Neural Network using TensorFlow

Here, we implement simple feedforward neural network that trains on a sample dataset and makes predictions using following steps:

Step 1: Import Necessary Libraries

Import necessary libraries, primarily TensorFlow and Keras, along with other required packages such as NumPy and Pandas for data handling.

```
import numpy as np
import pandas as pd
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
```

Step 2: Create and Load Dataset

- Create or load a dataset. Convert the data into a format suitable for training (usually NumPy arrays).
- Define features (X) and labels (y).

```
data = {
'feature1': [0.1, 0.2, 0.3, 0.4, 0.5],
'feature2': [0.5, 0.4, 0.3, 0.2, 0.1],
```

```
'label': [0, 0, 1, 1, 1]
}

df = pd.DataFrame(data)
X = df[['feature1', 'feature2']].values
y = df['label'].values
```

Step 3: Create a Neural Network

Instantiate a Sequential model and add layers. The input layer and hidden layers are typically created using `Dense` layers, specifying the number of neurons and activation functions.

```
model = Sequential()
model.add(Dense(8, input_dim=2, activation='relu'))  # Hidden layer
model.add(Dense(1, activation='sigmoid'))  # Output layer
```

Step 4: Compiling the Model

Compile the model by specifying the loss function, optimizer and metrics to evaluate during training. Here we will use binary crossentropy and adam optimizer.

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

Step 5: Train the Model

Fit the model on the training data, specifying the number of epochs and batch size. This step trains the neural network to learn from the input data.

```
model.fit(X, y, epochs=100, batch_size=1, verbose=1)
```

Step 5: Make Predictions

Use the trained model to make predictions on new data. Process the output to interpret the predictions like converting probabilities to binary outcomes.

```
test_data = np.array([[0.2, 0.4]])
prediction = model.predict(test_data)
predicted_label = (prediction > 0.5).astype(int)
```

Output:

Predicted label: 1

Advantages of Neural Networks

Neural networks are widely used in many different applications because of their many benefits:

- **Adaptability:** Neural networks are useful for activities where the link between inputs and outputs is complex or not well defined because they can adapt to new situations and learn from data.
- **Pattern Recognition:** Their proficiency in pattern recognition renders them efficacious in tasks like as audio and image identification, natural language processing and other intricate data patterns.
- **Parallel Processing:** Because neural networks are capable of parallel processing by nature, they can process numerous jobs at once which speeds up and improves the efficiency of computations.
- **Non-Linearity:** Neural networks are able to model and comprehend complicated relationships in data by virtue of the non-linear activation functions found in neurons which overcome the drawbacks of linear models.

Disadvantages of Neural Networks

Neural networks while powerful, are not without drawbacks and difficulties:

- **Computational Intensity:** Large neural network training can be a laborious and computationally demanding process that demands a lot of computing power.
- **Black box Nature:** As "black box" models, neural networks pose a problem in important applications since it is difficult to understand how they make decisions.

- **Overfitting:** Overfitting is a phenomenon in which neural networks commit training material to memory rather than identifying patterns in the data. Although regularization approaches help to alleviate this, the problem still exists.
- **Need for Large datasets:** For efficient training, neural networks frequently need sizable, labeled datasets; otherwise, their performance may suffer from incomplete or skewed data.

Applications of Neural Networks

Neural networks have numerous applications across various fields:

1. **Image and Video Recognition**: CNNs are extensively used in applications such as facial recognition, autonomous driving and medical image analysis.
2. **Natural Language Processing (NLP)**: RNNs and transformers power language translation, chatbots and sentiment analysis.
3. **Finance**: Predicting stock prices, fraud detection and risk management.
4. **Healthcare**: Neural networks assist in diagnosing diseases, analyzing medical images and personalizing treatment plans.
5. **Gaming and Autonomous Systems**: Neural networks enable real-time decision-making, enhancing user experience in video games and enabling autonomous systems like self-driving cars.

Difference between ANN and BNN

1. Artificial Neural Networks (ANNs)

Artificial Neural Networks are computational models inspired by the human brain's neural architecture. The simplest form of ANN follows a feed-forward mechanism where data flows from input to output without looping back. These networks consist of interconnected layers: input layers that receive data, hidden layers that process it and output layers that produce the final result.

Advantages of ANNs:

- **Versatile Learning:** ANNs can handle both linear and non-linear data which makes them applicable across diverse domains.
- **Forecasting:** They are sensitive to complex patterns making them effective in time series forecasting such as predicting stock prices or economic trends.

Disadvantages of ANNs:

- **Lack of Interpretability:** Due to their black-box nature, it is difficult to understand how decisions are made within the network.
- **Hardware Dependence:** ANNs require heavy computational resources which can limit their scalability in certain environments.

2. Biological Neural Networks (BNNs)

Biological Neural Networks are the foundation of cognition in living organisms. A biological neuron comprises dendrites, a cell body and an axon. Dendrites receive signals from other neurons, the soma integrates these inputs and the axon transmits the resulting signal to subsequent neurons via synapses.

Advantages of BNNs:

- **Input Handling:** Biological synapses are capable of interpreting and integrating a wide variety of stimuli/inputs.

- **Parallel Processing:** BNNs are efficient at processing massive amounts of information simultaneously, enabling rapid responses.

Disadvantages of BNNs:

- **Lack of Central Control:** Unlike artificial systems, BNNs lack a clear central processing unit, which can make control mechanisms less structured.
- **Slower Processing:** BNNs operate at slower speeds compared to silicon-based systems due to the nature of electrochemical transmission.

Comparison Table Between ANN and BNN

Parameter	Artificial Neural Network (ANN)	Biological Neural Network (BNN)
Structure	Input -> Hidden Layers -> Output	Dendrites -> Cell Body -> Axon
Learning	Requires structured, formatted data	Can handle ambiguous, noisy inputs
Computing	Centralized, sequential, program-driven	Distributed, parallel, self-learning
Reliability	Vulnerable to damage or failure	High fault tolerance and robustness
Operating Context	Well-defined, constrained environments	Unconstrained, often unpredictable

Key Differences Between ANNs and BNNs

Neurons

- **BNNs:** Composed of biological structures like dendrites and axons, with complex behavior and signal processing abilities.
- **ANNs:** Use simplified models of neurons with a single output, focusing on numerical signal transformations through activation functions.

Learning

- **BNNs:** Adapts based on learning, experience and environmental factors.
- **ANNs:** Use fixed mathematical weights that are adjusted during training but remain static during testing.

Neural Pathways

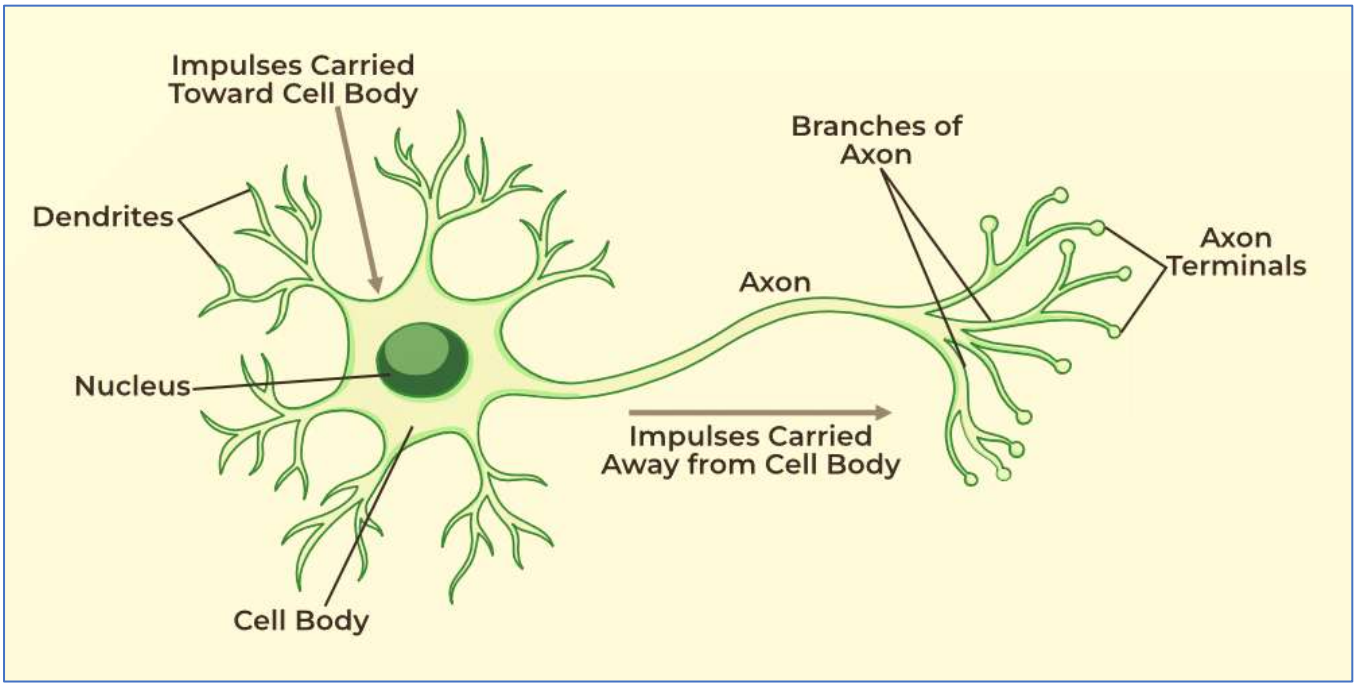
- **BNNs:** Feature a highly complex web of adaptable pathways influenced by learning and memory.
- **ANNs:** Have predefined pathways determined by network architecture and model design.

Single Layer Perceptron in TensorFlow

Single Layer Perceptron is inspired by biological neurons and their ability to process information. To understand the SLP we first need to break down the workings of a single artificial neuron which is the fundamental building block of neural networks. An **artificial neuron** is a simplified

computational model that mimics the behavior of a biological neuron. It takes inputs, processes them and produces an output. Here's how it works step by step:

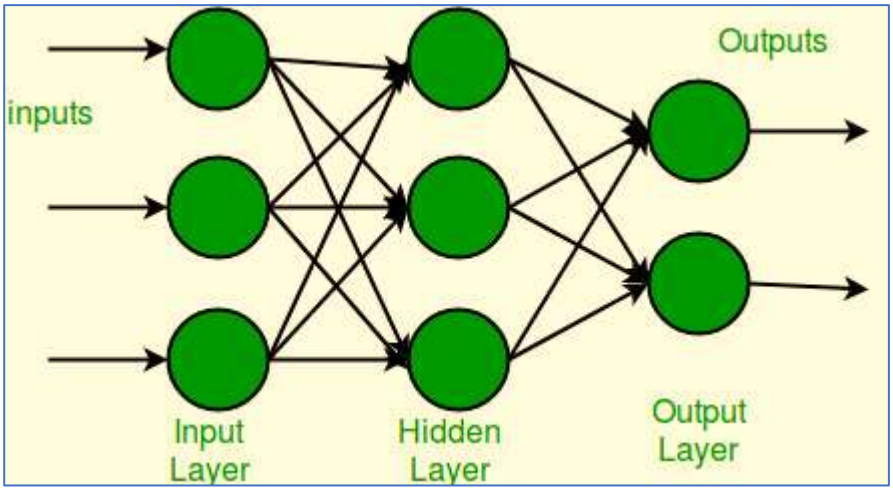
- Receive signal from outside.
- Process the signal and decide whether we need to send information or not.



- Communicate the signal to the target cell, which can be another neuron or gland.

Structure of a biological neuron

Similarly, neural networks also function in a similar manner.

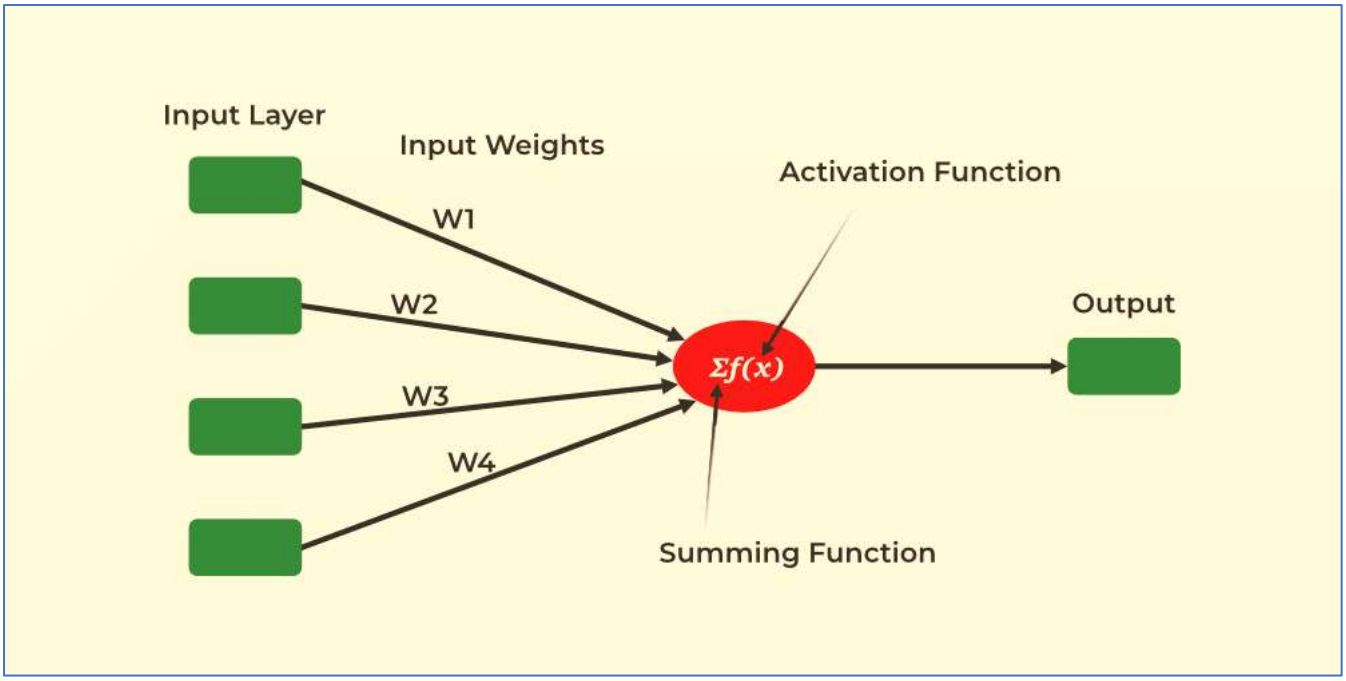


Single Layer Perceptron

It is one of the oldest and first introduced neural networks. It was proposed by **Frank Rosenblatt** in **1958**. Perceptron is also known as an artificial neural network. Perceptron is mainly used to compute the logical gate like **AND, OR and NOR** which has binary input and binary output.

The main functionality of the perceptron is:-

- Takes input from the input layer
- Weight them up and sum it up.
- Pass the sum to the nonlinear function to produce the output.



Single-layer neural network

Here activation functions can be anything like **sigmoid, tanh, relu** based on the requirement we will be choosing the most appropriate nonlinear activation function to produce the better result. Now let us implement a single-layer perceptron.

Implementation of Single-layer Perceptron

Let's build a simple **single-layer perceptron** using **TensorFlow**. This model will help you understand how neural networks work at the most basic level.

Step1: Import necessary libraries

- **Scikit-learn** – Scikit-learn provides easy-to-use and efficient tools for data mining and machine learning, enabling quick implementation of algorithms for classification, regression, clustering, and more.
- **TensorFlow** – This is an open-source library that is used for Machine Learning and Artificial intelligence and provides a range of functions to achieve complex functionalities with single lines of code.

```
import tensorflow as tf
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
```

Step 2: Create and split synthetic dataset

We will create a simple 2-feature synthetic binary-classification dataset for our demonstration and then split it into training and testing.

```
X, y = make_classification(
n_samples=1000,
n_features=2,
n_informative=2,
n_redundant=0,
n_repeated=0,
n_classes=2,
random_state=42
)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
```

Step 3: Standardize the Dataset

Now standardize the dataset to enable faster and more precise computations. **Standardization** helps the model converge more quickly and often enhances accuracy.

```
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

Step 4: Building a neural network

Next, we build the single-layer model using a Sequential architecture with one Dense layer. The **Dense(1)** indicates that this layer contains a single neuron. We apply the sigmoid activation function, which maps the output to a value between 0 and 1, suitable for binary classification. The original perceptron used a step function that only gave 0 or 1 as output and trained differently. But modern models use sigmoid because it's smooth and helps the model learn better with gradient-based methods. The **input_shape=(2,)** specifies that each input sample consists of two features.

```
model = tf.keras.Sequential([
tf.keras.layers.Dense(1, activation='sigmoid', input_shape=(2,))
])
```

Step 5: Compile the Model

Next, we compile the model using the Adam optimizer, which is a popular and efficient algorithm for optimizing neural networks. We use binary cross-entropy as the loss function, which is well-suited for binary classification tasks with sigmoid activation. Additionally, we track the model's performance using accuracy as the evaluation metric during training and testing.

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

Step 6: Train the Model

Now, we train the model by iterating over the entire training dataset a specified number of times, called epochs. During training, the data is divided into smaller batches of samples, known as the batch size, which determines how many samples are processed before updating the model's weights. We also set aside a fraction of the training data as validation data to monitor the model's performance on unseen data during training.

```
history = model.fit(X_train, y_train,
epochs=50,
batch_size=16,
validation_split=0.1,
verbose=0)
```

Step 7: Model Evaluation

After training we test the model's performance on unseen data.

```
loss, accuracy = model.evaluate(X_test, y_test, verbose=0)
print(f"Test Accuracy: {accuracy:.2f}")
```

Output:

Test Accuracy: 0.88

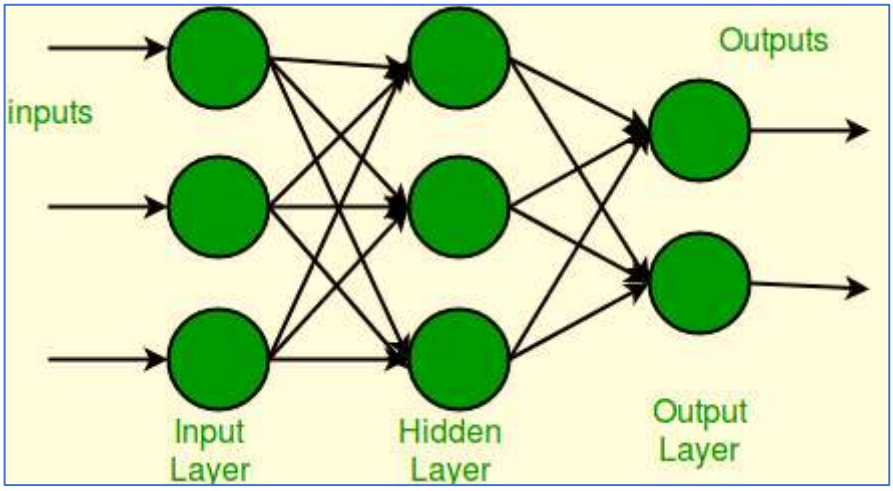
Multi-Layer Perceptron Learning in Tensorflow

Multi-Layer Perceptron (MLP) consists of fully connected dense layers that transform input data from one dimension to another. It is called **multi-layer** because it contains an input layer, one or more hidden layers and an output layer. The purpose of an MLP is to model complex relationships between inputs and outputs.

Components of Multi-Layer Perceptron (MLP)

- **Input Layer**: Each neuron or node in this layer corresponds to an input feature. For instance, if you have three input features the input layer will have three neurons.
- **Hidden Layers**: MLP can have any number of hidden layers with each layer containing any number of nodes. These layers process the information received from the input layer.
- **Output Layer**: The output layer generates the final prediction or result. If there are multiple outputs, the output layer will have a corresponding number of neurons.

Every connection in the diagram is a representation of the fully connected nature of an MLP. This



means that every node in one layer connects to every node in the next layer. As the data moves through the network each layer transforms it until the final output is generated in the output layer.

Working of Multi-Layer Perceptron

Let's see working of the multi-layer perceptron. The key mechanisms such as forward propagation, loss function, backpropagation and optimization.

1. Forward Propagation

In **forward propagation** the data flows from the input layer to the output layer, passing through any hidden layers. Each neuron in the hidden layers processes the input as follows:

1. Weighted Sum: The neuron computes the weighted sum of the inputs:

$z=\sum iwixi+bz=\sum iwixi+b$

Where:

- $xixi$ is the input feature.
- $wiwi$ is the corresponding weight.
- bb is the bias term.

2. Activation Function: The weighted sum z is passed through an activation function to introduce non-linearity. Common activation functions include:

- **Sigmoid**: $\sigma(z)=11+e-z\sigma(z)=1+e-z1$
- **ReLU (Rectified Linear Unit)**: $f(z)=\max f(0,z)f(z)=\max(0,z)$

- **Tanh (Hyperbolic Tangent)**: tanh⁡(z)=21+e−2z−1tanh(z)=1+e−2z2−1

2. Loss Function

Once the network generates an output the next step is to calculate the loss using a loss function. In supervised learning this compares the predicted output to the actual label.

For a classification problem the commonly used **binary cross-entropy** loss function is:

L=−1N∑i=1N[yilog⁡(y^i)+(1−yi)log⁡(1−y^i)]L=−N1∑i=1N[yilog(y^i)+(1−yi)log(1−y^i)]

Where:

- yiyi is the actual label.
- y^iy^i is the predicted label.
- NN is the number of samples.

For regression problems the **mean squared error (MSE)** is often used:

MSE=1N∑i=1N(yi−y^i)2MSE=N1∑i=1N(yi−y^i)2

3. Backpropagation

The goal of training an MLP is to minimize the loss function by adjusting the network's weights and biases. This is achieved through **backpropagation**:

1. **Gradient Calculation**: The gradients of the loss function with respect to each weight and bias are calculated using the chain rule of calculus.
2. **Error Propagation**: The error is propagated back through the network, layer by layer.
3. **Gradient Descent**: The network updates the weights and biases by moving in the opposite direction of the gradient to reduce the loss: w=w−η⋅∂L∂ww=w−η⋅∂w∂L

Where:

- wwis the weight.
- ηη is the learning rate.
- ∂L∂w∂w∂L is the gradient of the loss function with respect to the weight.

4. Optimization

MLPs rely on optimization algorithms to iteratively refine the weights and biases during training. Popular optimization methods include:

- **Stochastic Gradient Descent (SGD)**: Updates the weights based on a single sample or a small batch of data: w=w−η⋅∂L∂ww=w−η⋅∂w∂L
- **Adam Optimizer**: An extension of SGD that incorporates momentum and adaptive learning rates for more efficient training:
  - mt=β1mt−1+(1−β1)⋅gtmt=β1mt−1+(1−β1)⋅gt
  - vt=β2vt−1+(1−β2)⋅gt2vt=β2vt−1+(1−β2)⋅gt2
- Here gtgt represents the gradient at time tt and β1,β2β1,β2 are decay rates.

Now that we are done with the theory part of multi-layer perception, let's go ahead and implement code in python using the TensorFlow library.

Implementing Multi Layer Perceptron

In this section, we will guide through building a neural network using TensorFlow.

1. Importing Modules and Loading Dataset

First we import necessary libraries such as **TensorFlow**, **NumPy** and **Matplotlib** for visualizing the data. We also load the **MNIST dataset**.

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Flatten, Dense

(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
```

2. Loading and Normalizing Image Data

Next we normalize the image data by dividing by **255** (since pixel values range from 0 to 255) which helps in faster convergence during training.

```
gray_scale = 255

x_train = x_train.astype('float32') / gray_scale
x_test = x_test.astype('float32') / gray_scale

print("Feature matrix (x_train):", x_train.shape)
print("Target matrix (y_train):", y_train.shape)
print("Feature matrix (x_test):", x_test.shape)
print("Target matrix (y_test):", y_test.shape)
```

Output:

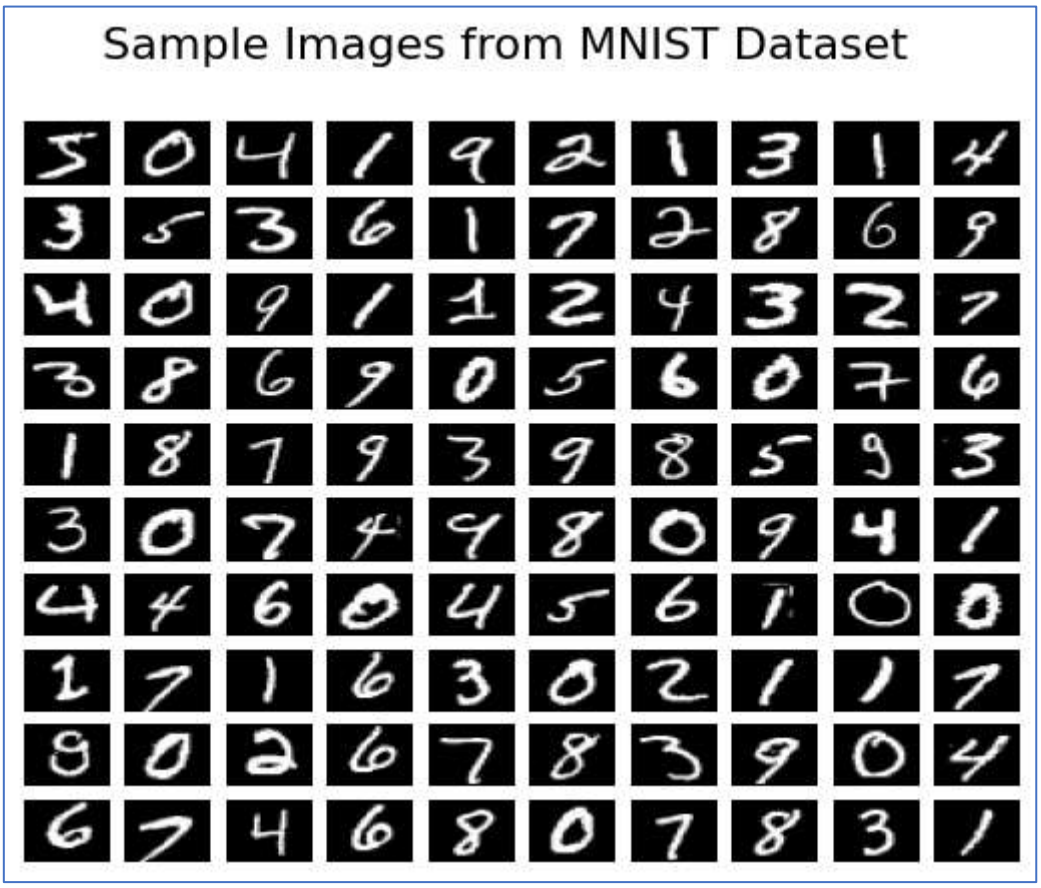
```
Feature matrix (x_train): (60000, 28, 28)
Target matrix (y_train): (60000,)
Feature matrix (x_test): (10000, 28, 28)
Target matrix (y_test): (10000,)
```

3. Visualizing Data

To understand the data better we plot the first 100 training samples each representing a digit.

```
fig, ax = plt.subplots(10, 10)
k = 0
for i in range(10):
for j in range(10):
ax[i][j].imshow(x_train[k].reshape(28, 28), aspect='auto')
k += 1
plt.show()
```

Output:



4. Building the Neural Network Model

Here we build a **Sequential neural network model**. The model consists of:

- **Flatten Layer**: Reshapes 2D input (28x28 pixels) into a 1D array of 784 elements.
- **Dense Layers**: Fully connected layers with 256 and 128 neurons, both using the relu activation function.
- **Output Layer**: The final layer with 10 neurons representing the 10 classes of digits (0-9) with **sigmoid** activation.

```
model = Sequential([
Flatten(input_shape=(28, 28)),
Dense(256, activation='sigmoid'),
Dense(128, activation='sigmoid'),
Dense(10, activation='softmax'),
])
```

5. Compiling the Model

Once the model is defined we compile it by specifying:

- **Optimizer**: Adam for efficient weight updates.
- **Loss Function**: Sparse categorical cross entropy, which is suitable for multi-class classification.
- **Metrics**: Accuracy to evaluate model performance.

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

6. Training the Model

We train the model on the training data using 10 epochs and a batch size of 2000. We also use 20% of the training data for validation to monitor the model's performance on unseen data during training.

```
mod = model.fit(x_train, y_train, epochs=10,
batch_size=2000,
validation_split=0.2)

print(mod)
```

Output:

```
Epoch 1/10
24/24 ━━━━━━━━━━━━━━━━━━━━ 2s 73ms/step - accuracy: 0.9232 - loss: 0.2767 - val_accuracy: 0.9276 - val_loss: 0.2544
Epoch 2/10
24/24 ━━━━━━━━━━━━━━━━━━━━ 2s 60ms/step - accuracy: 0.9282 - loss: 0.2558 - val_accuracy: 0.9312 - val_loss: 0.2439
Epoch 3/10
24/24 ━━━━━━━━━━━━━━━━━━━━ 1s 55ms/step - accuracy: 0.9317 - loss: 0.2422 - val_accuracy: 0.9334 - val_loss: 0.2333
Epoch 4/10
24/24 ━━━━━━━━━━━━━━━━━━━━ 2s 77ms/step - accuracy: 0.9332 - loss: 0.2334 - val_accuracy: 0.9362 - val_loss: 0.2244
Epoch 5/10
24/24 ━━━━━━━━━━━━━━━━━━━━ 2s 87ms/step - accuracy: 0.9373 - loss: 0.2242 - val_accuracy: 0.9385 - val_loss: 0.2164
Epoch 6/10
24/24 ━━━━━━━━━━━━━━━━━━━━ 2s 59ms/step - accuracy: 0.9371 - loss: 0.2170 - val_accuracy: 0.9400 - val_loss: 0.2098
Epoch 7/10
24/24 ━━━━━━━━━━━━━━━━━━━━ 1s 53ms/step - accuracy: 0.9387 - loss: 0.2093 - val_accuracy: 0.9430 - val_loss: 0.2011
Epoch 8/10
24/24 ━━━━━━━━━━━━━━━━━━━━ 3s 54ms/step - accuracy: 0.9433 - loss: 0.1974 - val_accuracy: 0.9438 - val_loss: 0.1961
Epoch 9/10
24/24 ━━━━━━━━━━━━━━━━━━━━ 1s 53ms/step - accuracy: 0.9456 - loss: 0.1909 - val_accuracy: 0.9455 - val_loss: 0.1906
Epoch 10/10
24/24 ━━━━━━━━━━━━━━━━━━━━ 1s 60ms/step - accuracy: 0.9466 - loss: 0.1858 - val_accuracy: 0.9488 - val_loss: 0.1840
```

7. Evaluating the Model

After training we evaluate the model on the test dataset to determine its performance.

results = model.evaluate(x_test, y_test, verbose=0)

```
print('Test loss, Test accuracy:', results)
```

Output:

Test loss, Test accuracy: [0.2682029604911804, 0.9257000088691711]

We got the accuracy of our model 92% by using **model.evaluate()** on the test samples.

8. Visualizing Training and Validation Loss VS Accuracy

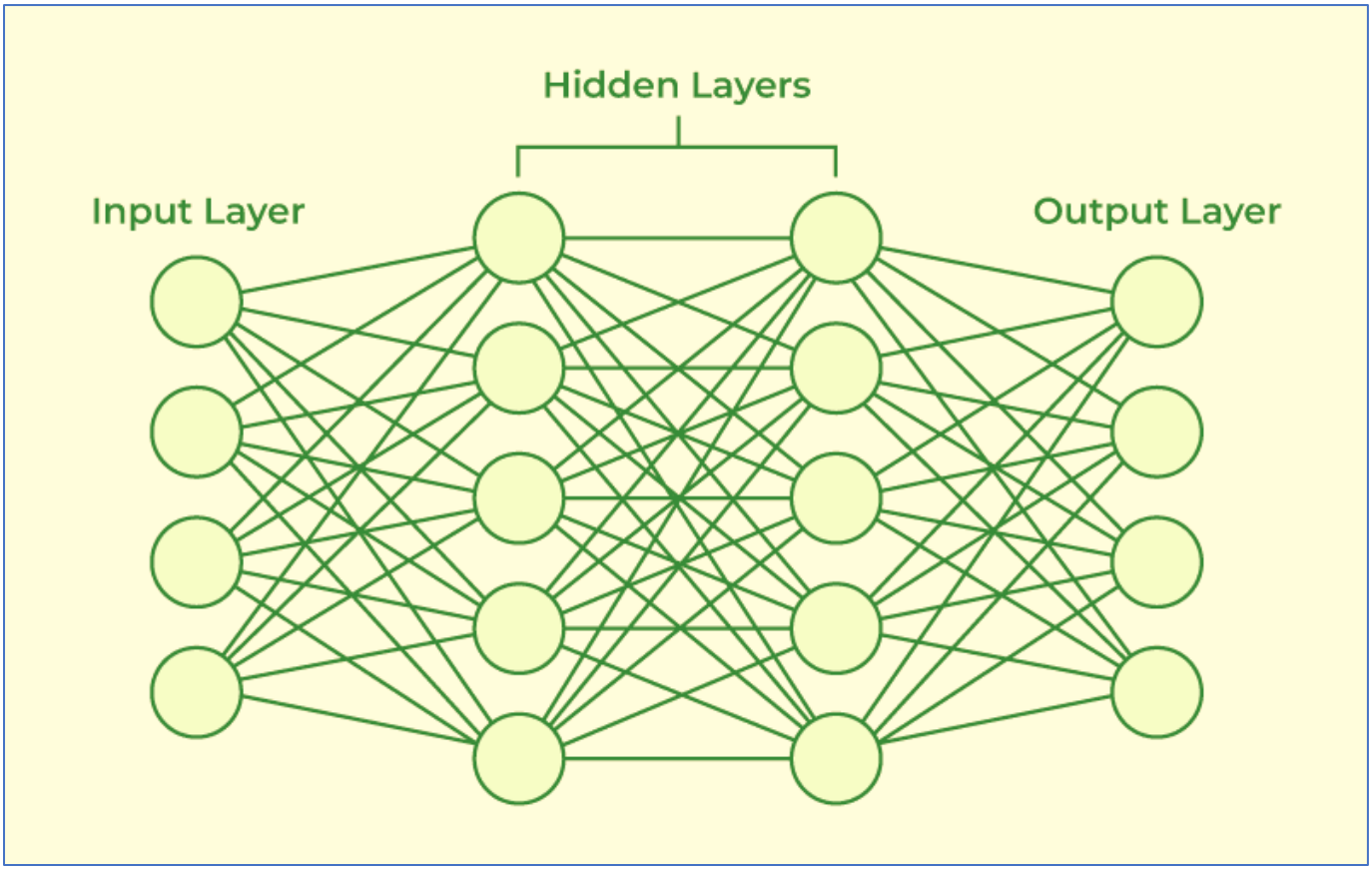
```
plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)
plt.plot(mod.history['accuracy'], label='Training Accuracy', color='blue')
plt.plot(mod.history['val_accuracy'], label='Validation Accuracy', color='orange')
plt.title('Training and Validation Accuracy', fontsize=14)
plt.xlabel('Epochs', fontsize=12)
plt.ylabel('Accuracy', fontsize=12)
plt.legend()
plt.grid(True)

plt.subplot(1, 2, 2)
plt.plot(mod.history['loss'], label='Training Loss', color='blue')
plt.plot(mod.history['val_loss'], label='Validation Loss', color='orange')
plt.title('Training and Validation Loss', fontsize=14)
plt.xlabel('Epochs', fontsize=12)
plt.ylabel('Loss', fontsize=12)
plt.legend()
plt.grid(True)

plt.suptitle("Model Training Performance", fontsize=16)
plt.tight_layout()
plt.show()
```

Artificial Neural Networks and its Applications

Artificial Neural Networks (ANNs) are computer systems designed to mimic how the human brain processes information. Just like the brain uses neurons to process data and make decisions, ANNs use artificial neurons to analyze data, identify patterns and make predictions. These networks consist of layers of interconnected neurons that work together to solve complex problems. The key idea is that ANNs can "learn" from the data they process, just as our brain learns from experience. They are used in various applications from recognizing images to making personalized recommendations. In this article, we will see more about ANNs, how they function and other core concepts.

Key Components of an ANN

- **Input Layer:** This is where the network receives information. For example, in an image recognition task, the input could be an image.
- **Hidden Layers:** These layers process the data received from the input layer. The more hidden layers there are, the more complex patterns the network can learn and understand. Each hidden layer transforms the data into more abstract information.
- **Output Layer:** This is where the final decision or prediction is made. For example, after processing an image, the output layer might decide whether it's a cat or a dog.



Working of Artificial Neural Networks

ANNs work by learning patterns in data through a process called training. During training, the network adjusts itself to improve its accuracy by comparing its predictions with the actual results. Lets see how the learning process works:

- **Input Layer**: Data such as an image, text or number is fed into the network through the input layer.
- **Hidden Layers**: Each neuron in the hidden layers performs some calculation on the input, passing the result to the next layer. The data is transformed and abstracted at each layer.
- **Output Layer**: After passing through all the layers, the network gives its final prediction like classifying an image as a cat or a dog.

The process of backpropagation is used to adjust the weights between neurons. When the network makes a mistake, the weights are updated to reduce the error and improve the next prediction.

Training and Testing:

- During training, the network is shown examples like images of cats and learns to recognize patterns in them.
- After training, the network is tested on new data to check its performance. The better the network is trained, the more accurately it will predict new data.

How do Artificial Neural Networks learn?

- Artificial Neural Networks (ANNs) learn by training on a set of data. For example, to teach an ANN to recognize a cat, we show it thousands of images of cats. The network processes these images and learns to identify the features that define a cat.
- Once the network has been trained, we test it by providing new images to see if it can correctly identify cats. The network's prediction is then compared to the actual label (whether it's a cat or not). If it makes an incorrect prediction, the network adjusts by

fine-tuning the weights of the connections between neurons using a process called backpropagation. This involves correcting the weights based on the difference between the predicted and actual result.

- This process repeats until the network can accurately recognize a cat in an image with minimal error. Essentially, through constant training and feedback, the network becomes better at identifying patterns and making predictions.

Artificial neurons vs Biological neurons

Aspect	Biological Neurons	Artificial Neurons
Structure	**Dendrites**: Receive signals from other neurons.	**Input Nodes**: Receive data and pass it on to the next layer.
	Cell Body (Soma): Processes the signals.	**Hidden Layer Nodes**: Process and transform the data.
	Axon: Transmits processed signals to other neurons.	**Output Nodes**: Produce the final result after processing.
Connections	**Synapses**: Links between neurons that transmit signals.	**Weights**: Connections between neurons that control the influence of one neuron on another.
Learning Mechanism	**Synaptic Plasticity**: Changes in synaptic strength based on activity over time.	**Backpropagation**: Adjusts the weights based on errors in predictions to improve future performance.
Activation	**Activation**: Neurons fire when signals are strong enough to reach a threshold.	**Activation Function**: Maps input to output, deciding if the neuron should fire based on the processed data.

Common Activation Functions in ANNs

Activation functions are important in neural networks because they introduce non-linearity and helps the network to learn complex patterns. Lets see some common activation functions used in ANNs:

1. **Sigmoid Function:** Outputs values between 0 and 1. It is used in binary classification tasks like deciding if an image is a cat or not.
2. **ReLU (Rectified Linear Unit):** A popular choice for hidden layers, it returns the input if positive and zero otherwise. It helps to solve the vanishing gradient problem.
3. **Tanh (Hyperbolic Tangent):** Similar to sigmoid but outputs values between -1 and 1. It is used in hidden layers when a broader range of outputs is needed.
4. **Softmax:** Converts raw outputs into probabilities used in the final layer of a network for multi-class classification tasks.
5. **Leaky ReLU:** A variant of ReLU that allows small negative values for inputs helps in preventing “dead neurons” during training.

Types of Artificial Neural Networks

1. Feedforward Neural Network (FNN)

Feedforward Neural Networks are one of the simplest types of ANNs. In this network, data flows in one direction from the input layer to the output layer, passing through one or more hidden layers. There are no loops or cycles means the data doesn’t return to any earlier layers. This type of network does not use backpropagation and is mainly used for basic classification and regression tasks.

2. Convolutional Neural Network (CNN)

Convolutional Neural Networks (CNNs) are designed to process data that has a grid-like structure such as images. It include convolutional layers that apply filters to extract important features from the data such as edges or textures. This makes CNNs effective in image and speech recognition as they can identify patterns and structures in complex data.

3. Radial Basis Function Network (RBFN)

Radial Basis Function Networks are designed to work with data that can be modeled in a radial or circular way. These networks consist of two layers: one that maps input to radial basis functions and another that finds the output. They are used for classification and regression tasks especially when the data represents an underlying pattern or trend.

4. Recurrent Neural Network (RNN)

Recurrent Neural Networks are designed to handle sequential data such as time-series or text. Unlike other networks, RNNs have feedback loops that allow information to be passed back into previous layers, giving the network memory. This feature helps RNNs to make predictions based on the context provided by previous data helps in making them ideal for tasks like speech recognition, language modeling and forecasting.

Optimization Algorithms in ANN Training

Optimization algorithms adjust the weights of a neural network during training to minimize errors. The goal is to make the network's predictions more accurate. Lets see key algorithms:

1. **Gradient Descent:** Most basic optimization algorithm that updates weights by calculating the gradient of the loss function.
2. **Adam (Adaptive Moment Estimation):** An efficient version of gradient descent that adapts learning rates for each weight used in deep learning.
3. **RMSprop:** A variation of gradient descent that adjusts the learning rate based on the average of recent gradients, it is useful in training recurrent neural networks (RNNs).
4. **Stochastic Gradient Descent (SGD):** Updates weights using one sample at a time helps in making it faster but noisier.

Types of Neural Networks

Neural networks are computational models that mimic the way biological neural networks in the human brain process information. They consist of layers of neurons that transform the input data into meaningful outputs through a series of mathematical operations.

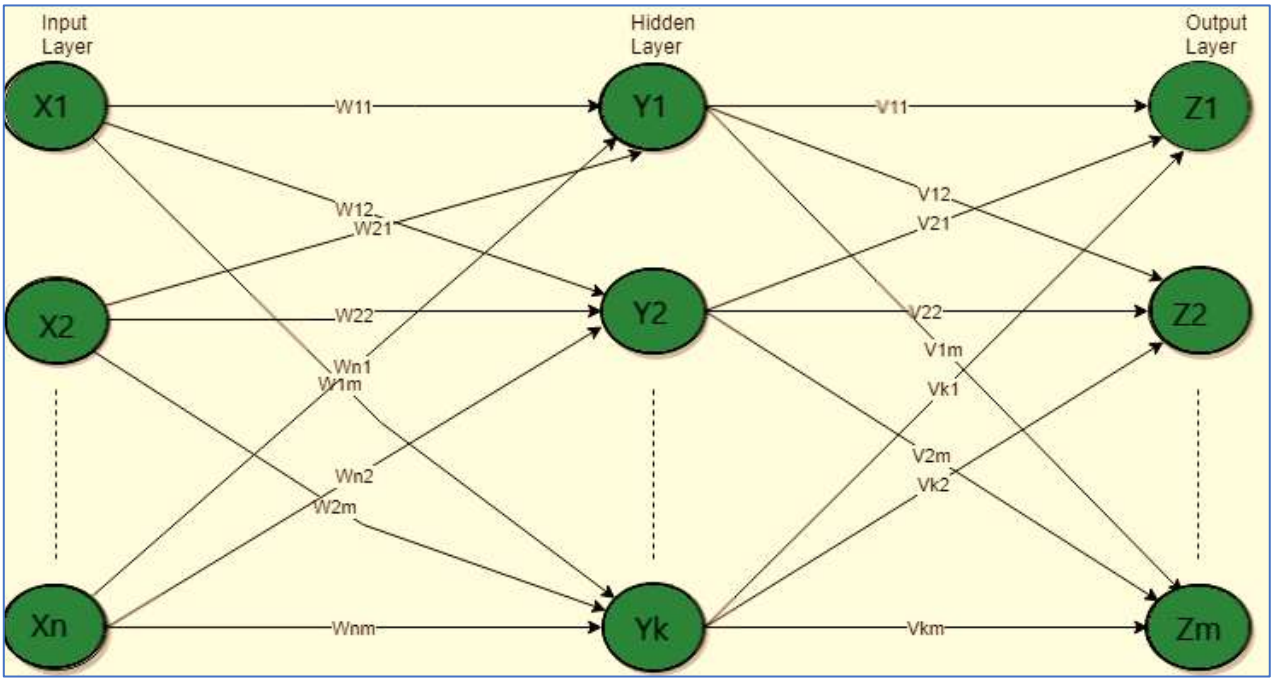
In this article, we are going to explore different types of neural networks.

1. Feedforward Neural Networks

Feedforward neural networks are a form of artificial neural network where without forming any cycles between layers or nodes means inputs can pass data through those nodes within the hidden level to the output nodes.

- **Architecture:** Made up of layers with unidirectional flow of data i.e from input through hidden and the output layer.
- **Training:** Backpropagation is often used during training for the main aim of reducing the prediction errors.
- **Applications:** In visual and voice recognition, NLP, financial forecasting and recommending system

- **When to use:** Best for general-purpose tasks like classification and regression. Ideal when data is static and has no sequential dependencies.

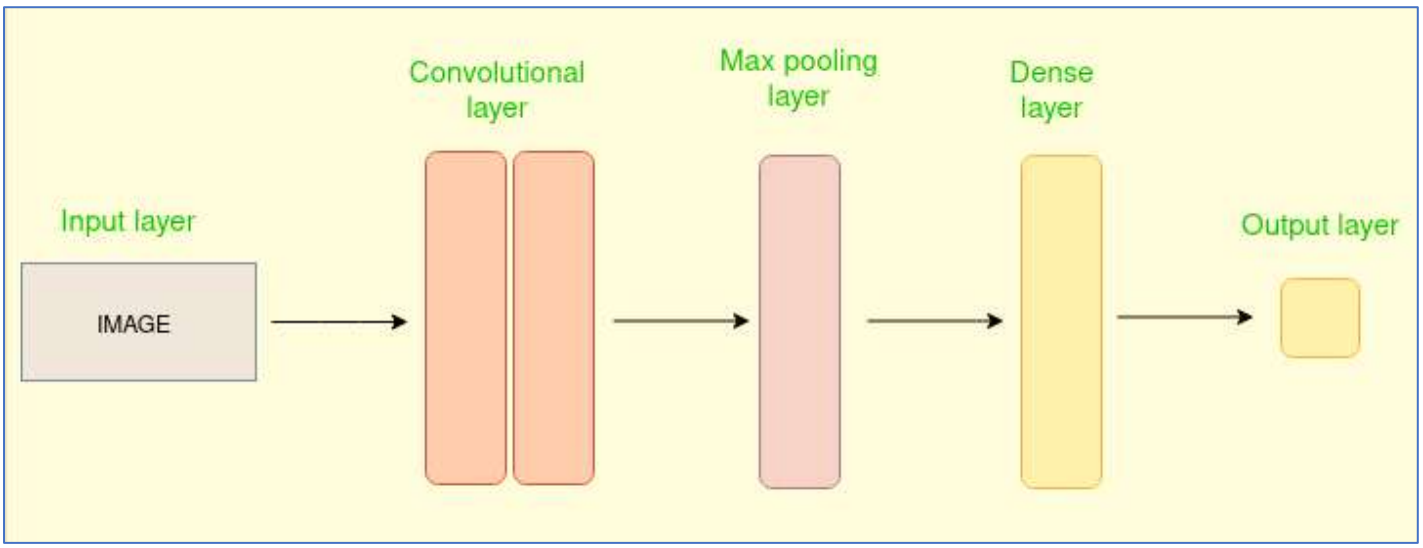


2. Convolutional Neural Networks (CNNs)

Convolutional neural networks structure is focused on processing the grid type data like images and videos by using convolutional layers filtering driving the patterns and spatial hierarchies.

- **Key Components:** Utilizing convolutional layers, pooling layers and fully connected layers.
- **Applications:** Used for classification of images, object detection, medical imaging analyzes, autonomous driving and visualization in augmented reality.
- **When to use:** Use when working with image, video or grid-structured data.

3. Recurrent Neural Networks (RNNs)



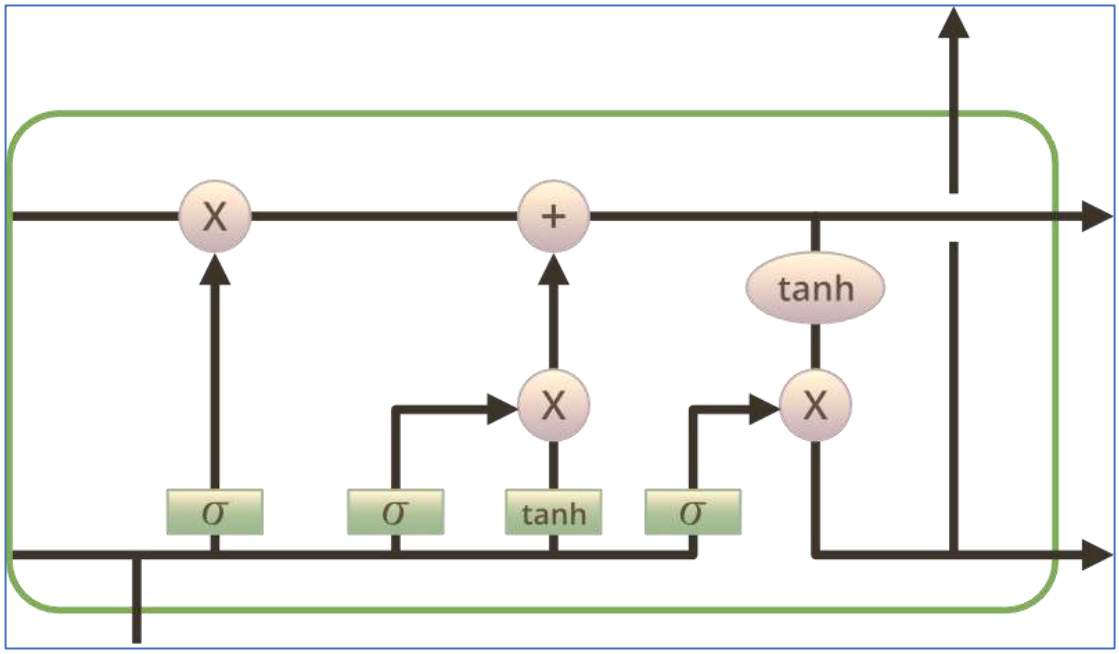
- **Architecture:** Contains recurrent connections that enable feedback loops for processing sequences.
- **Challenges:** Problems such as vanishing gradients become apparent since they limit capturing interdependence on a long scale.
- **Applications:** Language translation, open-ended text classification, ones to ones interaction and time series prediction are its applications.

- **When to use:** Use for tasks involving sequences like text, speech or time series.

4. Long Short-Term Memory Networks (LSTMs)

Long Short-Term Memory Networks (LSTMs) are a variant of RNNs. They exhibit memory cells to solve the disappearing gradient issue and keep large ranges of information in their memory.

- **Key Features:** Capture memory cells in pass information flowing and graduate greediness issue.
- **Applications**: Value of RNNs is in terms of importing long-term memory into the model like language translation and time-series forecasting.
- **When to use:** Use when you need to model long-term dependencies in sequences.



5. Gated Recurrent Units (GRUs)

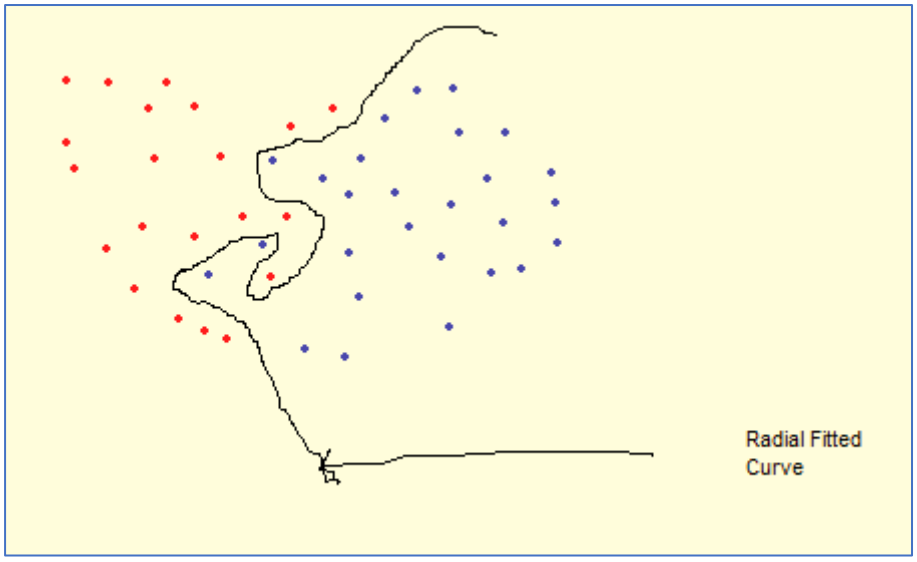
Gated Recurrent Units (GRUs) is the second usual variant of RNNs which is working on gating mechanism just like LSTM but with little parameter.

- **Advantages:** Vanishing gradient issue is addressed and it is compute-efficient than LSTM.
- **Applications:** LSTM is also involved in tasks that can be categorized as similar to speech recognition and text monitoring.
- **When to use:** Use when LSTM-like performance is needed but with lower computational cost.

6. Radial Basis Function Networks (RBFNs)

Radial basis function (RBF) networks can be regarded as models which define radial basis functions that are very useful in the function approximation and classification approaches and is used in complex input-output data modelling.

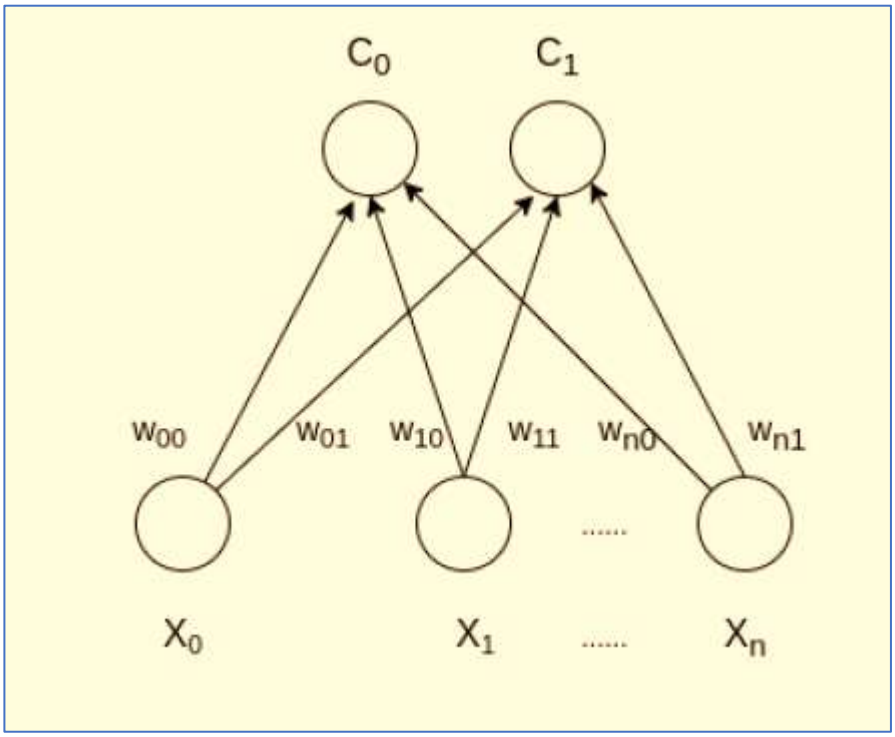
- **Applications:** It includes regression, pattern recognition and system control methods for fast-tracking.
- **When to use:** Good choice for function approximation and small to medium-scale classification tasks.



7. Self-Organizing Maps (SOMs)

Self-Organizing Maps are unsupervised neural networks, these networks are used for unsupervised cluster generation based on the retaining of topological features of the high dimensional data from an upper dimensional source, transformed into low dimensional form of output data.

- **Features:** Design methods that reduces the dimension of data from the high dimension into a low dimension without loss of the underlying geometry of the data.
- **Applications:** Visualizing data, discovering customers segments, locating anomalies and selecting needed features.
- **When to use:** Ideal for data visualization, clustering and dimensionality reduction.



8. Deep Belief Networks (DBNs)

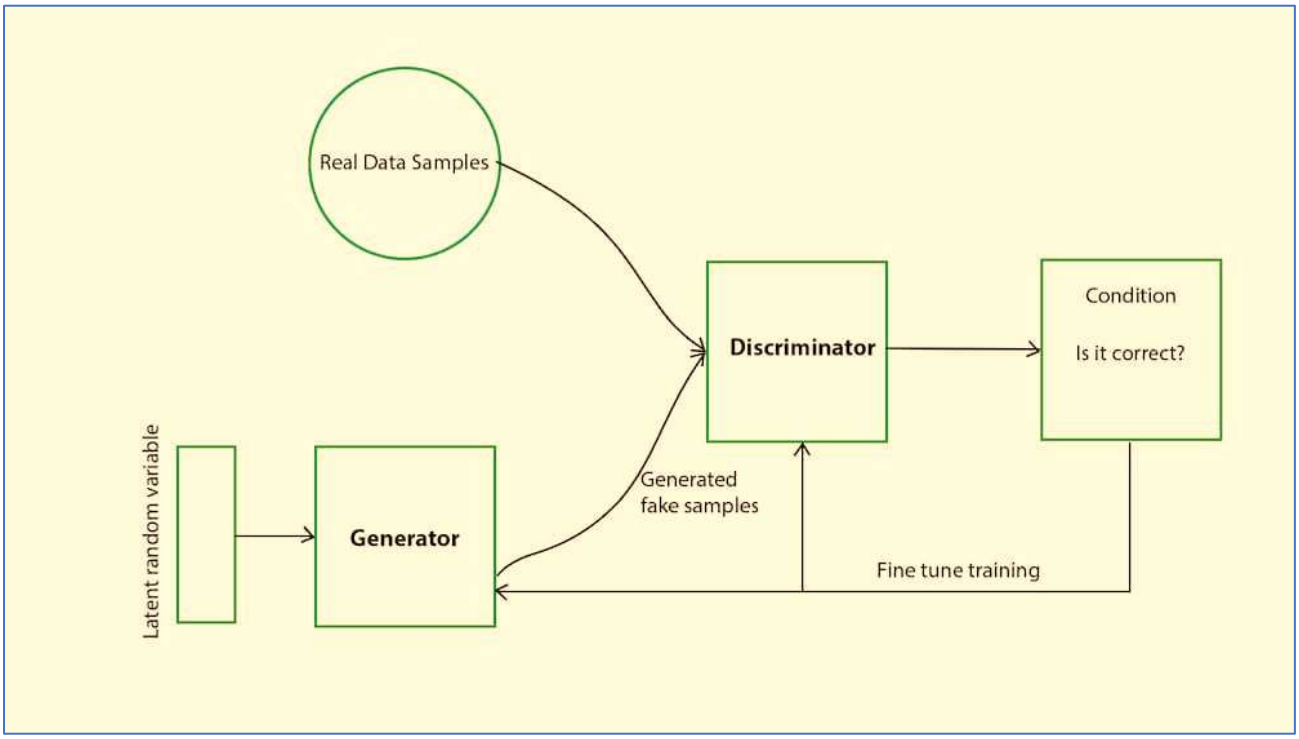
The architecture of the Deep Belief Networks is built on many stochastic, latent variables that are used for both deep supervised and unsupervised tasks such as nonlinear feature learning and mid dimensional representation.

- **Function:** If you are looking for the most effective architecture of data that can be learned via classification, this algorithm is very useful.
- **Applications:** Image and voice recognition, natural language understanding and smart devices as recommendations systems.
- **When to use:** Use when you're interested in unsupervised pre-training or deep feature extraction.

9. Generative Adversarial Networks (GANs)

Generative Adversarial Networks has made up of of two neural networks, the generator and discriminator which compete against each other. The generator creates a fake generated data and the discriminator learns to differentiate the real from and fake data.

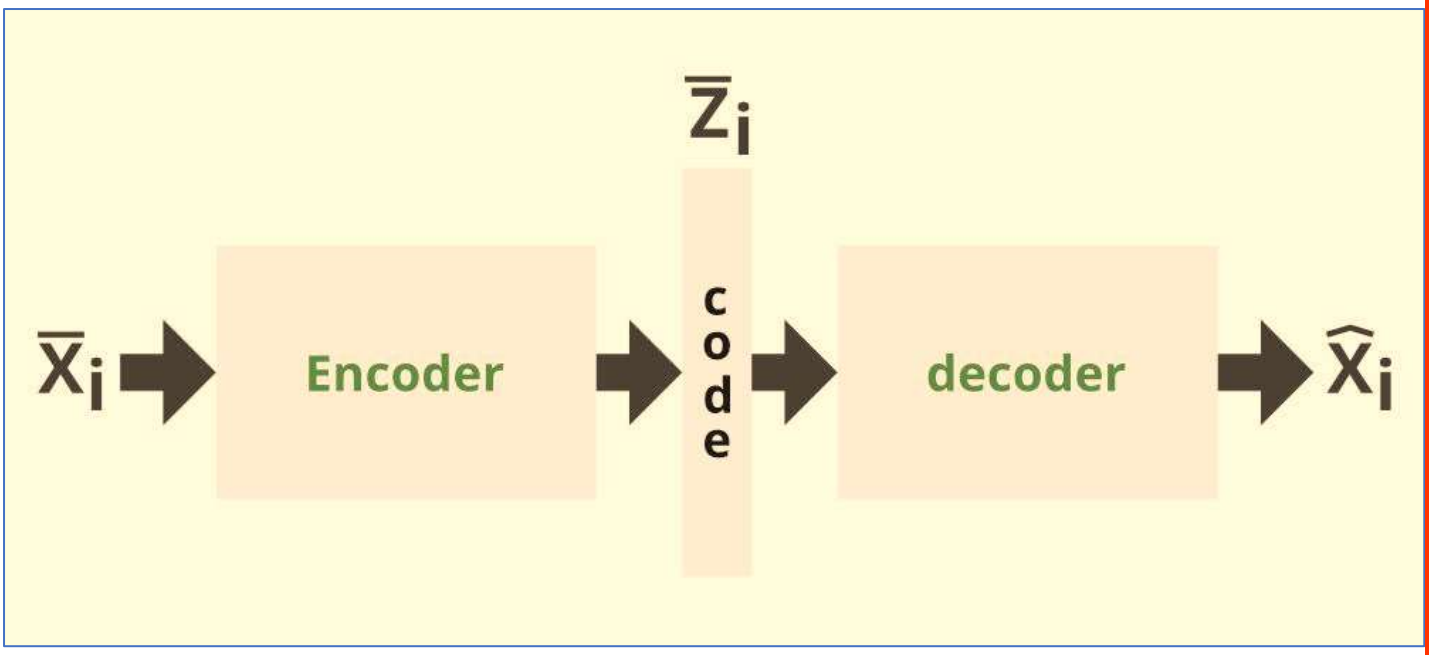
- **Working Principle:** Generator evolves after each iteration while the fake data being generated. This simultaneously makes the discriminator more discriminating as it determines whether the components are real or generated.
- **Applications:** They have proved useful not only for pattern generation but also data augmentation, style transfer and learning without any supervision.
- **When to use:** Use when you need to generate realistic synthetic data.



10. Autoencoders (AE)

Autoencoders are feedforward networks (ANNs) that are trained to acquire the most helpful presentations of the information through the process of re-coding the input data. The encoder is pinpointed to precisely map the input into the legal latent space representation while the decoder does the opposite, decoding the space from this representation.

- **Functionality:** Help in techniques like dimensionality reduction, information extraction, noise removal and generative modelling the images become comprehensible.
- **Types:** Variants include undercomplete, overcomplete and variational autoencoders.
- **When to use:** Use for unsupervised learning tasks like data compression, noise removal and anomaly detection.



11. Siamese Neural Networks

Siamese Neural Network work with networks of the same structure and an identical architecture. Comparison is being made via a similarity metric that can tell the degree of resemblance the two networks have.

- **Applications:** Face recognition as the signature, retrieval of information, image similarity comparison and category tasks.
- **When to use:** Ideal when comparing two inputs to determine similarity like face verification.

12. Capsule Networks (CapsNet)

The layers of Capsule Networks do not only incorporate localization relations of data but allows multilevel structure by passing the information from lower convolutional layers to higher. They use cyclicals to the items and their bodies too, of course, they do not do that at the same time.

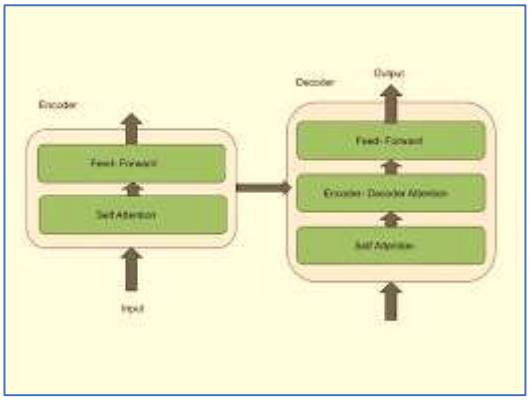
- **Applications:** Image classification, object detection and scene understanding via the immense visual data exposure.
- **When to use:** Use for image classification where part-to-whole relationships matter.

13. Transformer Networks

Transformer Networks do this by way of self-attention mechanism which results into a parallel process used for making the tokenization inputs faster and thus improved capturing of long range dependencies.

- **Key Features:** Provides better performance than any of the other models due to their capability to process natural language sufficiently and handle tasks related to machine translation, generating text and document summarization.
- **Applications:** The application of this technology had got more popular, specially in language understanding tasks and image and audio data processing applications of this time and more similar tasks.

- **When to use:** Use for NLP tasks like translation, text generation and summarization.



14. Spiking Neural Networks (SNN)

Main thing related with Spiking Neural Networks is the brain functionality which is processed by action potentials (spikes) in biological neurons in the same way. These are the key factors of "neuromorphic" technology which perform the deep learning and avoid another type of processing as well.

- **Applications:** Neuromorphic processes, learning and computation in spiro-neural computing, cognitive processes modeling and mind-related computing are also carried out with this.
- **When to use:** Use when working on neuromorphic computing and biologically inspired architectures.

Architecture and Learning process in neural network

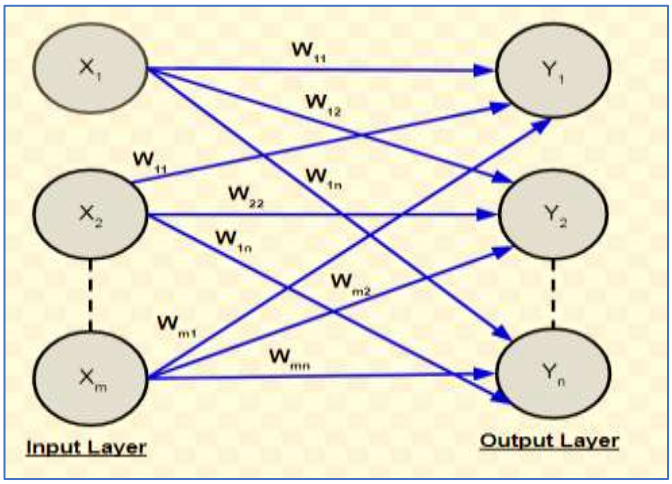
Architectures of Neural Network:

ANN is a computational system consisting of many interconnected units called **artificial neurons**. The connection between artificial neurons can transmit a signal from one neuron to another. So, there are multiple possibilities for connecting the neurons based on which the **architecture** we are going to adopt for a specific solution. Some permutations and combinations are as follows:

- There may be just two layers of neuron in the network - the input and output layer.
- There can be one or more intermediate **'hidden'** layers of a neuron.
- The neurons may be connected with all neurons in the next layer and so on .....

So let's start talking about the various possible architectures:

A. Single-layer Feed Forward Network:

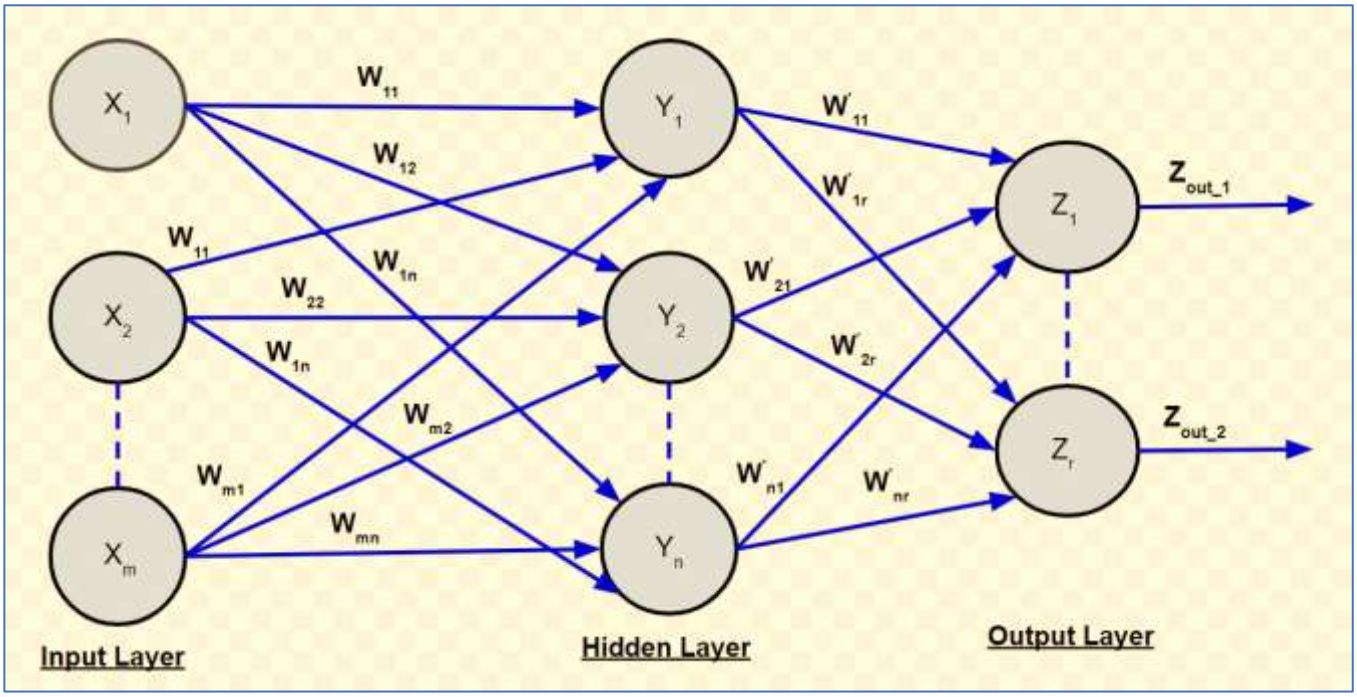


It is the simplest and most basic architecture of ANN's. It consists of only two layers- the input layer and the output layer. **The input layer** consists of 'm' input neurons connected to each of the 'n' output neurons. The

connections carry weights w_{11} and so on. The input layer of the neurons doesn't conduct any processing - they pass the i/p signals to the o/p neurons. The computations are performed in the output layer. So, though it has 2 layers of neurons, only one layer is performing the computation. This is the reason why **the network is known as SINGLE** layer. Also, the signals always flow from the input layer to the output layer. Hence, the **network is known as FEED FORWARD.**
The net signal input to the output neurons is given by:

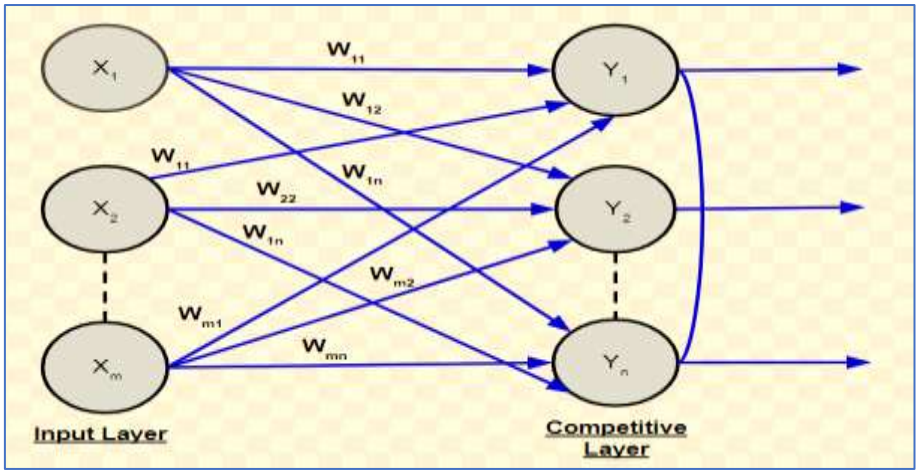
yin_k+x1w1k+x2w2k+...+xmwmk=∑i=1mxiwik*yin_k+x1w1k+x2w2k+...+xmwmk=∑i=1mxiwik*
The signal output from each output neuron will depend on the activation function used.

B. Multi-layer Feed Forward Network:



Multi-Layer Feed Forward Network
The multi-layer feed-forward network is quite similar to the single-layer feed-forward network, except for the fact that there are one or more intermediate layers of neurons between the input and output layer. Hence, the **network is termed as multi-layer.** Each of the layers may have a varying number of neurons. For example, the one shown in the above diagram has 'm' neurons in the input layer and 'r' neurons in the output layer and there is only one hidden layer with 'n' neurons.
yin_k=x1w1k+x2w2k+...+xmwmk=∑i=1mxiwik*yin_k=x1w1k+x2w2k+...+xmwmk=∑i=1mxiwik*
for the kth hidden layer neuron. The net signal input to the neuron in the output layer is given by:

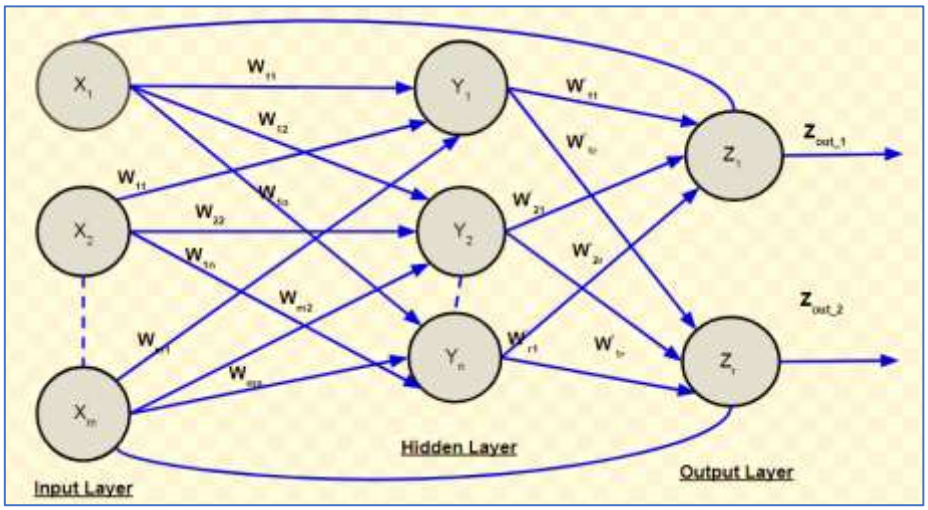
zin_k=yout_1w1k′+yout_2w2k′+...+yout_nwnk′=∑i=1nyout_iwik′*zin_k=yout_1w1k′+yout_2w2k′*
+...+yout_nwnk′=∑i=1nyout_iwik′
C. Competitive Network:
It is as same as the single-layer feed-forward network in structure. The only difference is that **the output neurons are connected with each other (either partially or fully)**. Below is the diagram for this type of network.



Competitive Network

According to the diagram, it is clear that few of the output neurons are interconnected to each other. For a given input, the output neurons compete against themselves to represent the input. It represents a form of an unsupervised learning algorithm in ANN that is suitable to find the clusters in a data set.

D. Recurrent Network:



Recurrent Network

In feed-forward networks, the signal always flows from the input layer towards the output layer (in one direction only). In the case of recurrent neural networks, there is **a feedback loop** (from the neurons in the output layer to the input layer neurons). There can be self-loops too.

Learning Process In ANN:

Learning process in ANN mainly depends on four factors, they are:

1. **The number of layers in the network (Single-layered or multi-layered)**
2. **Direction of signal flow (Feedforward or recurrent)**
3. **Number of nodes in layers:** The number of node in the input layer is equal to the number of features of the input data set. The number of output nodes will depend on possible outcomes i.e. the number of classes in case of supervised learning. But the number of layers in the hidden layer is to be chosen by the user. A larger number of nodes in the hidden layer, higher the performance but too many nodes may result in overfitting as well as increased computational expense.
4. **Weight of Interconnected Nodes:** Deciding the value of weights attached with each interconnection between each neuron so that a specific learning problem can be solved correctly is quite a difficult problem by itself. Take an example to understand the problem. Take the example of a **Multi-layered Feed-Forward Network,** we have to train an ANN model using some data, so that it can classify a new data set, say p_5(3,-2). Say we have deduced that p_1=(5,2) and p_2 = (-1,12) belonging to class C1 while p_3=(3,-5) and p_4 = (-2,-1) belonging to class C2. We assume the values of synaptic weights w_0,w_1,w_2 as -2, 1/2 and 1/4 respectively. But we will NOT get these weight values for every learning problem. For solving a learning problem with ANN, we can start with a set of values for synaptic weights and keep changing those in multiple

iterations. The stopping criterion may be the **rate of misclassification < 1% or the maximum numbers of iterations should be less than 25(a threshold value).** There may be another problem that, the rate of misclassification may not reduce progressively.

So, we can summarize the learning process in ANN as the combination of - **deciding the number of hidden layers,** the **number of nodes in each of the hidden layers,** the **direction of signal flow, deciding the connection weight.**

Multi-layer feed network is a commonly used architecture. It has been observed that a neural network with even one hidden layer can be used to reasonably approximate any continuous function. The learning methodology adopted to train a multi-layer feed-forward network is **Backpropagation**.

Backpropagation:

In the above section, we get to know that the most critical activities of training an ANN are to assign the inter-neuron connection weights. In 1986, an efficient way of training an ANN was introduced. In this method, the **difference in output values of the output layer and the expected values, are** propagated **back from the output layer to the** preceding **layers.** Hence, the algorithm implementing this method is known as BACK PROPAGATION **i.e. propagating the errors back to the** preceding **layers.**

The backpropagation algorithm is applicable for multi-layer feed-forward network. It is a supervised learning algorithm which continues adjusting the weights of the connected neurons with an objective to reduce the deviation of the output signal from the target output. This algorithm consists of multiple iterations, **known as epochs.** Each epoch consists of two phases:

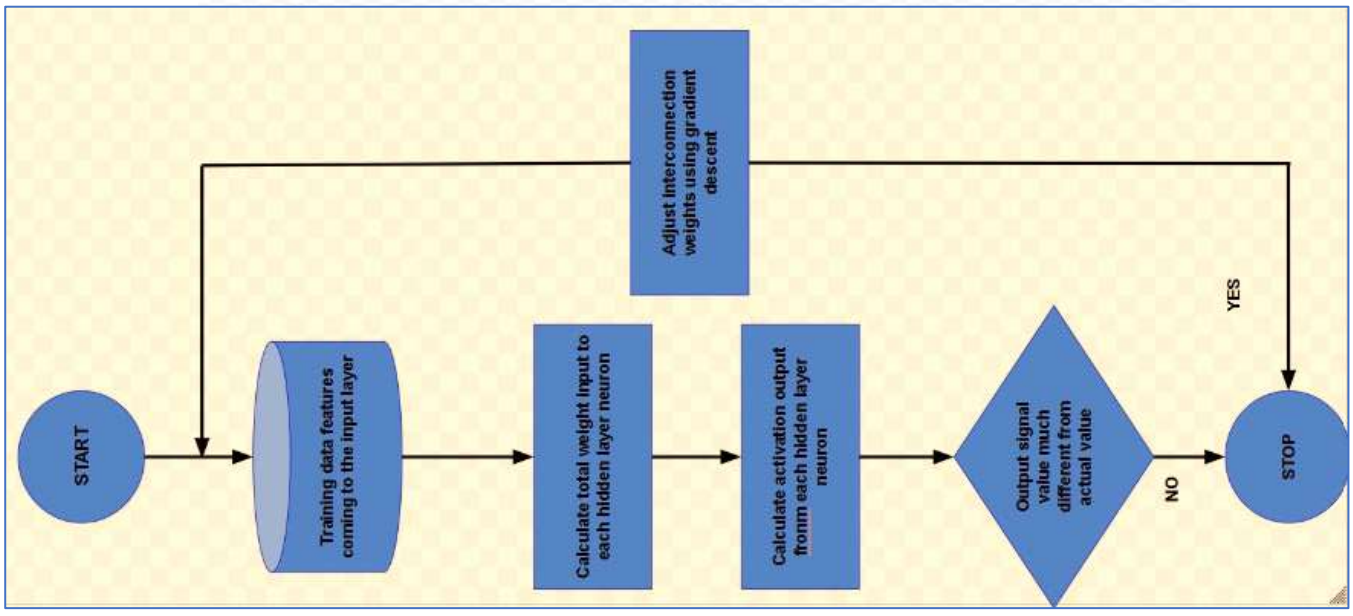
- **Forward Phase:** Signal flow from neurons in the input layer to the neurons in the output layer through the hidden layers. The weights of the interconnections and activation functions are used during the flow. In the output layer, the output signals are generated.
- **Backward Phase:** Signal is compared with the expected value. The computed errors are propagated backwards from the output to the preceding layer. The error propagated back are used to adjust the interconnection weights between the layers.

BACKPROPAGATION

The above diagram depicts a reasonably simplified version of the back propagation algorithm.

One main part of the algorithm is adjusting the interconnection weights. This is done using a technique termed as **Gradient Descent**. In simple words, the algorithm calculates the partial derivative of the activation function by each interconnection weight to identify the 'gradient' or extent of change of the weight required to minimize the cost function.

In order to understand the back propagation algorithm in detail, let us consider the **Multi-layer Feed Forward**



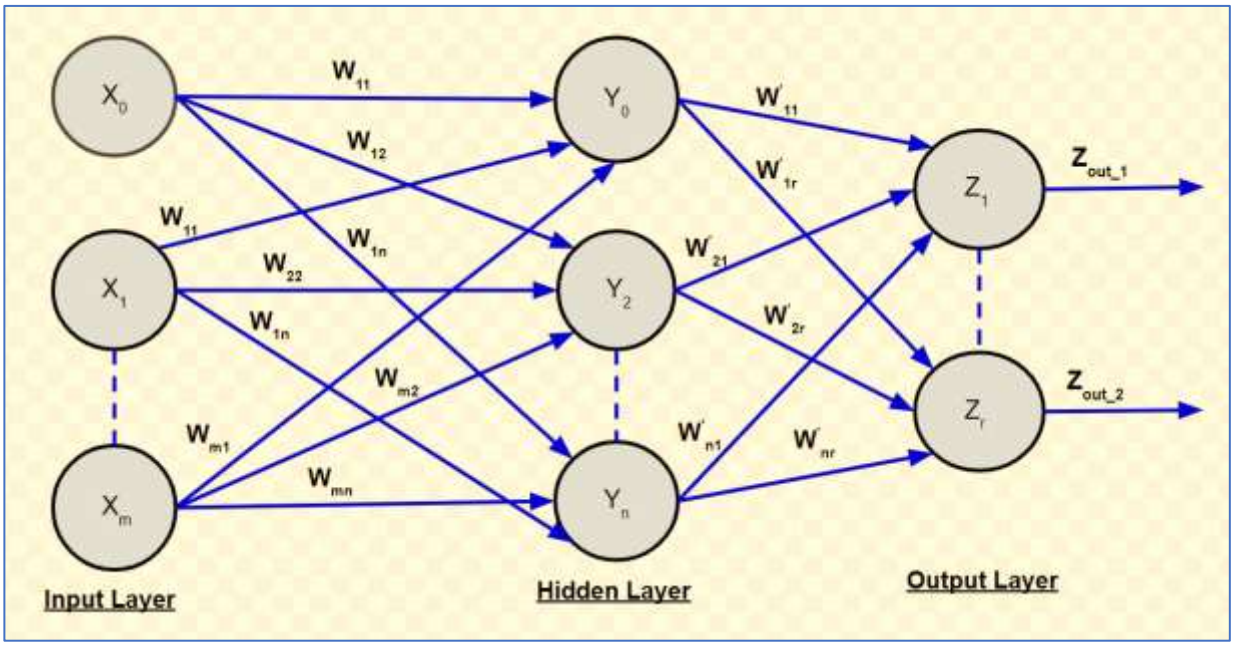
The net signal input to the hidden layer neurons is given by:

yin_k=x0w0k+x1w1k+...+xmwmk=w0k+∑i=1mxiwik*yin_k=x0w0k+x1w1k+...+xmwmk=w0k+∑i=1mxiwik*

If fy *fy* is the activation function of the hidden layer, then yout_k=fy(yin_k)*yout_k=fy(yin_k)*

The net signal input to the output layer neurons is given by:

zin_k=y0w0k′+yout_1w1k′+...+yout_nwnk′=w0k′+∑i=1nyout_iwik′*zin_k=y0w0k′+yout_1w1k′+...+yout_n*



wnk′=w0k′+∑i=1nyout_iwik′

BACKPROPAGATION NET

Note that the signals X0 *X0* and Y0 *Y0* are assumed to be 1. If fz *fz* is the activation function of the hidden layer, then zout_k=fz(zin_k)*zout_k=fz(zin_k)*

If is the target of the k-th output neuron, then the cost function defined as the squared error of the output layer is given by:

E=12∑k=1n(tk−zout_k)2*E=21∑k=1n(tk−zout_k)2*

E=12∑k=1n(tk−fz(zin_k))2*E=21∑k=1n(tk−fz(zin_k))2*

According to the descent algorithm, partial derivative of cost function E has to be taken with respect to interconnection weights. Mathematically it can be represented as:

∂E∂wjk′=*∂∂wjk′*{12∑k=1n(tk−fz(zin_k)2}*∂wjk′∂E=∂wjk′∂*{21∑*k=1n(tk−fz(zin_k)2*}

{Above expression is for the interconnection weights between the **j-th neuron in the hidden layer and the k-th neuron in the output layer**.} **This expression can be reduced to**

∂E∂wjk′=−(tk−zout_k)·fz′(zin_k)·*∂∂wjk′*{∑i=0nyout_i·wik}*∂wjk′∂E=−(tk−zout_k)·fz′(zin_k)·∂wjk′∂*{∑*i=0nyout_i·wik*}

where, fz′(zin_k)=*∂∂wjk′*(fz(zin_k)) *fz′(zin_k)=∂wjk′∂(fz(zin_k*

)) **or** ∂E∂wjk′=−(tk−zout_k)·fz′(zin_k).yout_i*∂wjk′∂E=−(tk−zout_k)·fz′(zin_k).yout_i*

If we assume δwk=−(tk−zout_k)·fz′(zin_k) *δwk=−(tk−zout_k)·fz′(zin_k)* as a component of the weight adjustment needed for weight wjk *wjk* corresponding to the k-th output neuron, then :

∂E∂wjk′=δwk′·yout_i*∂wjk′∂E=δwk′·yout_i*

On the basis of this, the weights and bias need to be updated as follows:

- **For weights:** Δwjk=−α·∂E∂wjk′=−α·δwk′·yout_iΔ*wjk=−α·∂wjk′∂E=−α·δwk′·yout_i*
- **Hence,** wjk′(new)=wjk′(old)+Δwjk′*wjk′(new)=wjk′(old)+Δwjk′*
- **For bias:** Δw0k=−α·δwk′Δ*w0k=−α·δwk′*
- **Hence,** w0k′(new)=w0k′(old)+Δw0k′*w0k′(new)=w0k′(old)+Δw0k′*

In the above expressions, alpha is the learning rate of the neural network. Learning rate is a user parameter which decreases or increases the speed with which the interconnection weights of a neural network is to be adjusted. If the learning rate is too high, the adjustment done as a part of the gradient descent process may diverge the data set rather than converging it. On the other hand, if the learning rate is too low, the optimization may consume more time because of the small steps towards the minima.

{All the above calculations are for the interconnection weight between neurons in the hidden layer and neurons in the output layer}

Like the above expressions, we can deduce the expressions for "Interconnection weights between the input and hidden layers:

- **For weights:** $\Delta wij = -\alpha \cdot \partial E \partial wij = -\alpha \cdot \delta wj \cdot xout\_i \Delta wij = -\alpha \cdot \partial wij \partial E = -\alpha \cdot \delta wj \cdot xout\_i$
- **Hence,** $wij(new) = wij(old) + \Delta wij wij(new) = wij(old) + \Delta wij$
- **For bias:** $\Delta w0j = -\alpha \cdot \delta wj \Delta w0j = -\alpha \cdot \delta wj$

Hence, $w0j(new) = w0j(old) + \Delta w0j w0j(new) = w0j(old) + \Delta w0j$

Basic Components of Neural Networks

The basic components of neural network are:

- Layers in Neural Networks
- Weights and Biases
- Forward Propagation
- Activation Functions
- Loss Functions
- Backpropagation
- Learning Rate

Layers in Artificial Neural Networks (ANN)

An Artificial Neural Networks (ANNs) consists of three primary types of layers:

- **Input Layer**
- **Hidden Layers**
- **Output Layer**

Each layer is composed of nodes (neurons) that are interconnected. The layers work together to process data through a series of transformations.

ANN Layers

Basic Layers in ANN

1. Input Layer

Input layer is the first layer in an ANN and is responsible for receiving the raw input data. This layer's neurons correspond to the features in the input data. For example, in image processing, each neuron might represent a pixel value. The input layer doesn't perform any computations but passes the data to the next layer.

Key Points:

- **Role**: Receives raw data.
- **Function**: Passes data to the hidden layers.
- **Example**: For an image, the input layer would have neurons for each pixel value.

Input Layer in ANN

2. Hidden Layers

Hidden Layers are the intermediate layers between the input and output layers. They perform most of the computations required by the network. Hidden layers can vary in number and size, depending on the complexity of the task.

Each hidden layer applies a set of weights and biases to the input data, followed by an activation function to introduce non-linearity.

3. Output Layer

Output Layer is the final layer in an ANN. It produces the output predictions. The number of neurons in this layer corresponds to the number of classes in a classification problem or the number of outputs in a regression problem.

The activation function used in the output layer depends on the type of problem:

- **Softmax** for multi-class classification
- **Sigmoid** for binary classification
- **Linear** for regression

For better understanding of the activation functions, Refer to the article - Activation functions in Neural Networks

Types of Hidden Layers in Artificial Neural Networks

Till now we have covered the basic layers: input, hidden, and output. Let's now dive into the specific types of hidden layers.

1. Dense (Fully Connected) Layer

Dense (Fully Connected) Layer is the most common type of hidden layer in an ANN. Every neuron in a dense layer is connected to every neuron in the previous and subsequent layers. This layer performs a weighted sum of inputs and applies an activation function to introduce non-linearity. The activation function (like ReLU, Sigmoid, or Tanh) helps the network learn complex patterns.

- **Role**: Learns representations from input data.
- **Function**: Performs weighted sum and activation.

2. Convolutional Layer

Convolutional layers are used in Convolutional Neural Networks (CNNs) for image processing tasks. They apply convolution operations to the input, capturing spatial hierarchies in the data. Convolutional layers use filters to scan across the input and generate feature maps. This helps in detecting edges, textures, and other visual features.

- **Role**: Extracts spatial features from images.
- **Function**: Applies convolution using filters.

Convolutional Layer

3. Recurrent Layer

Recurrent layers are used in Recurrent Neural Networks (RNNs) for sequence data like time series or natural language. They have connections that loop back, allowing information to persist across time steps. This makes them suitable for tasks where context and temporal dependencies are important.

- **Role**: Processes sequential data with temporal dependencies.
- **Function**: Maintains state across time steps.

Recurrent Layer

4. Dropout Layer

Dropout layers are a regularization technique used to prevent overfitting. They randomly drop a fraction of the neurons during training, which forces the network to learn more robust features and reduces dependency on specific neurons. During training, each neuron is retained with a probability p.

- **Role**: Prevents overfitting.
- **Function**: Randomly drops neurons during training.

Dropout Layer

5. Pooling Layer

Pooling Layer is used to reduce the spatial dimensions of the data, thereby decreasing the computational load and controlling overfitting. Common types of pooling include Max Pooling and Average Pooling.

Use Cases: Dimensionality reduction in CNNs

6. Batch Normalization Layer

A **Batch Normalization Layer** normalizes the output of a previous activation layer by subtracting the batch mean and dividing by the batch standard deviation. This helps in accelerating the training process and improving the performance of the network.

Use Cases: Stabilizing and speeding up training

Weights and Bias in Neural Networks

1. Weights

Weights are numerical values assigned to the connections between neurons. They find how much influence each input has on the network's final output.

- **Purpose**: During forward propagation, inputs are multiplied by their respective weights before being passed through an activation function. This helps decide how strongly an input will affect the output.
- **Learning Mechanism**: During training, weights are updated iteratively through optimization algorithms like gradient descent to minimize the difference between predicted and actual outcomes.

- **Generalization**: Well-tuned weights help the network not only make accurate predictions on training data but also generalize to new, unseen data.
- **Example:** In a neural network predicting house prices, the weight for the "size of the house" find how much the house size influences the price prediction. The larger the weight, the bigger the impact size will have on the final result.

2. Biases

Biases are additional parameters that adjust the output of a neuron. Unlike weights, they are not tied to any specific input but instead shift the activation function to better fit the data.

- **Purpose**: Biases help neurons activate even when the weighted sum of inputs is not enough. This allows the network to recognize patterns that don't necessarily pass through the origin.
- **Functionality**: Without biases, neurons would only activate when the input reaches a specific threshold. It makes the network more flexible by enabling activation across a wider range of conditions.
- **Training**: During training, biases are updated alongside weights through backpropagation. Together, they fine-tune the model, improving prediction accuracy.
- **Example**: In a house price prediction network, the bias might ensure that even for a house with a size of zero, the model predicts a non-zero price. This could reflect a fixed value such as land value or other baseline costs.

How Neural Networks Learn?

Neural networks learn through a process involving forward propagation and backpropagation. Let's see each step:

1. Forward Propagation

Forward propagation is the initial phase of processing input data through the neural network to produce an output or prediction. Let's see how it works:

1. **Input Layer:** The process starts with data entering the network's input layer. This could be anything from pixel values in an image to feature values in a dataset.
2. **Weighted Sum**: Each neuron calculates the weighted sum of the inputs. Each input is multiplied by its corresponding weight which shows the importance of that input.
3. **Adding Biases**: A bias is added to the weighted sum. Bias helps shift the output and provides flexibility, allowing the network to make better predictions even if all input values are zero.
4. **Activation Function**: The sum of the weighted inputs plus bias is passed through an activation function (e.g ReLU, sigmoid). The activation function decides if the neuron should activate which means it will pass information to the next layer or stay inactive.
5. **Propagation**: This process is repeated across multiple layers. The output of one layer becomes the input for the next, continuing until the network generates the final output or prediction.

2. Backpropagation

Once the network has made a prediction, it's important to evaluate how accurate that prediction is and make adjustments to improve future predictions. This is where backpropagation comes:

1. **Error Calculation:** Once the network generates an output, it's compared to the actual result (the target). The difference between the predicted and actual values is the error also called the loss.

2. **Gradient Calculation**: The error is propagated back through the network and the gradient or slope of the error with respect to the weights and biases is calculated. This tells the network how to adjust the parameters to minimize the error.
3. **Updating Weights and Biases**: Using the gradient, the network adjusts the weights and biases. The goal is to reduce the error in future predictions. This step is done through an optimization algorithm like gradient descent.
4. **Iteration**: This process of forward and backward propagation is repeated many times on different batches of data. With each iteration, the network's weights and biases get closer to the optimal values, improving the model's performance.

Real-World Applications of Neural Networks

Neural networks are increasingly used in various fields to solve complex problems. Let's see various examples of how weights and biases play an important role in below applications:

1. Image Recognition

Neural networks are efficient at tasks like object and image classification. For example, in detecting objects like cats, dogs or even specific facial features:

- **Weights**: These find which pixels are important. For example, in a picture of a cat, the weights might give more importance to features like ears, whiskers and eyes, helping the network correctly identify the object.
- **Biases**: They ensure the network remains adaptable despite changes in image conditions. For example, slight shifts in lighting, position or orientation won't stop the network from recognizing the object.

By adjusting weights and biases, the network learns to recognize patterns in data and improve its accuracy in classifying new, unseen images.

2. Natural Language Processing (NLP)

In tasks such as sentiment analysis, language translation and chatbots, neural networks analyze and generate text. For example, understanding customer reviews or translating languages:

- **Weights**: These decide how important specific words or phrases are in a given context. For example, recognizing the sentiment of the word "happy" in a review versus "sad" helps the network understand the sentiment of the sentence.
- **Biases**: They help the network to adapt to different sentence structures and tones. This helps the model recognize meaning even when the sentence might be phrased differently.

Training the network on large datasets allows it to interpret language effectively, whether it's classifying emotions in reviews or translating text between languages.

3. Autonomous Vehicles

In self-driving cars, neural networks process a range of sensor data (camera, radar, lidar) to make driving decision such as stopping at a red light or avoiding obstacles:

- **Weights**: The weights help the network focus on important input data such as recognizing pedestrians, road signs and other vehicles, adjusting their significance based on the car's current needs.
- **Biases**: Biases ensure that the car can adapt to different driving conditions like fog or night-time driving, ensuring safety and accuracy under varied circumstances.

By continuously adjusting the weights and biases, the system learns how to safely navigate complex environments and make real-time decisions.

4. Healthcare and Medical Diagnosis

Neural networks are also applied in healthcare such as in diagnosing diseases from medical images like X-rays, MRIs or CT scans:

- **Weights**: These help the network focus on important features in medical images such as specific areas indicating a tumor or anomaly. This helps the network make more accurate predictions regarding health conditions.
- **Biases**: Biases allow the network to remain flexible and adaptable to variations in imaging techniques or the patient's body type, making the system more reliable across different scenarios.

By training on thousands of medical images, the neural network learns to identify patterns and make precise diagnoses, aiding medical professionals in early disease detection.

Advantages of Weights and Biases

1. **Learning from Data**: Weights and biases help the network adjust to data patterns, enabling it to make predictions based on input significance and flexibility.
2. **Flexibility in Complex Data**: Biases allow the network to adjust outputs even when inputs are minimal, improving flexibility in tasks like image recognition or language processing.
3. **Improved Accuracy**: Through iterative updates, weights and biases help reduce prediction errors, leading to more accurate results over time.
4. **Better Generalization**: Properly tuned weights and biases help the network apply learned patterns to new, unseen data, ensuring it performs well outside the training set.
5. **Enhanced Learning Capacity**: They allow the network to capture complex patterns, improving its ability to handle complex tasks that traditional algorithms struggle with.

What is Forward Propagation in Neural Networks

Understanding Forward Propogation

In **Forward propagation** input data moves through each layer of neural network where each neuron applies weighted sum, adds bias, passes the result through an activation function and making predictions. This process is crucial before backpropagation updates the weights. It determines the output of neural network with a given set of inputs and current state of model parameters (weights and biases). Understanding this process helps in optimizing neural networks for various tasks like classification, regression and more. Below is the step by step working of forward propagation:

1. Input Layer

- The input data is fed into the network through the input layer.
- Each feature in the input dataset represents a neuron in this layer.
- The input is usually normalized or standardized to improve model performance.

2. Hidden Layers

- The input moves through one or more hidden layers where transformations occur.
- Each neuron in hidden layer computes a weighted sum of inputs and applies activation function to introduce non-linearity.
- Each neuron receives inputs, computes: $Z=WX+b$ $Z=WX+b$, where:
    - W W is the weight matrix
    - X X is the input vector
    - b b is the bias term
- The activation function such as ReLU or sigmoid is applied.

3. Output Layer

- The last layer in the network generates the final prediction.
- The activation function of this layer depends on the type of problem:
    - **Softmax** (for multi-class classification)
    - **Sigmoid** (for binary classification)
    - **Linear** (for regression tasks)

4. Prediction

- The network produces an output based on current weights and biases.
- The loss function evaluates the error by comparing predicted output with actual values.

Mathematical Explanation of Forward Propagation

Consider a neural network with one input layer, two hidden layers and one output layer.

architecture of a neural network

1. Layer 1 (First Hidden Layer)

The transformation is: $A[1]=\sigma(W[1]X+b[1])$ $A[1]=\sigma(W[1]X+b[1])$ where:

- $W[1]$ $W[1]$ is the weight matrix,
- X X is the input vector,
- $b[1]$ $b[1]$is the bias vector,
- σ σ is the activation function.

2. Layer 2 (Second Hidden Layer)

$A[2]=\sigma(W[2]A[1]+b[2])$ $A[2]=\sigma(W[2]A[1]+b[2])$

3. Output Layer

$Y=\sigma(W[3]A[2]+b[3])$ $Y=\sigma(W[3]A[2]+b[3])$ where Y Y is the final output. Thus the complete equation for forward propagation is:

$A[3]=\sigma(\sigma(\sigma(XW[1]+b[1])W[2]+b[2])W[3]+b[3])$ $A[3]=\sigma(\sigma(\sigma(XW[1]+b[1])W[2]+b[2])W[3]+b[3])$

This equation illustrates how data flows through the network:

- Weights (W W) determine the importance of each input
- Biases (b b) adjust activation thresholds
- Activation functions (σ σ) introduce non-linearity to enable complex decision boundaries.

Implementation of Forward Propagation

1. Import Required Libraries

Here we will import Numpy and pandas library.

```
import numpy as np
import pandas as pd
```

2. Create Sample Dataset

- The dataset consists of CGPA, profile score and salary in LPA.
- X X contains only input features.

```
data = {'cgpa': [8.5, 9.2, 7.8], 'profile_score': [85, 92, 78], 'lpa': [10, 12, 8]}
df = pd.DataFrame(data)
X = df[['cgpa', 'profile_score']].values
```

3. Initialize Parameters

When initilaizing parameters **Random initialization** avoids symmetry issues where neurons learn the same function.

```
def initialize_parameters():
np.random.seed(1)
W = np.random.randn(2, 1) * 0.01
b = np.zeros((1, 1))
return W, b
```

4. Define Forward Propagation

- Z=WX+BZ=*WX*+*B* computes the linear transformation.
- Sigmoid activation ensures values remain between 0 and 1.

```
def forward_propagation(X, W, b):
Z = np.dot(X, W) + b
A = 1 / (1 + np.exp(-Z))
return A
```

5. Execute Forward Propagation

Here we will execute the process of forward propagation using the above functions we created.

```
W, b = initialize_parameters()
A = forward_propagation(X, W, b)
print("Final Output:", A)
```

Output:

```
Final
[[0.40566303]
[0.39810287]
[0.41326819]]
```

Activation functions in Neural Networks

While building a neural network, one key decision is selecting the Activation Function for both the hidden layer and the output layer. It is a mathematical function applied to the output of a neuron. It introduces non-linearity into the model, allowing the network to learn and represent complex patterns in the data. Without this non-linearity feature a neural network would behave like a linear regression model no matter how many layers it has.

Activation function decides whether a neuron should be activated by calculating the weighted sum of inputs and adding a bias term. This helps the model make complex decisions and predictions by introducing non-linearities to the output of each neuron.

Before diving into the activation function, you should have prior knowledge of the following topics: ***Neural Networks***, ***Backpropagation***

Introducing Non-Linearity in Neural Network

Non-linearity means that the relationship between input and output is not a straight line. In simple terms the output does not change proportionally with the input. A common choice is the ReLU function defined as σ(x)=max⁡(0,x)*σ*(*x*)=max(0,*x*).

Imagine you want to classify apples and bananas based on their shape and color.

- If we use a linear function it can only separate them using a straight line.
- But real-world data is often more complex like overlapping colors, different lighting, etc.
- By adding a non-linear activation function like ReLU, Sigmoid or Tanh the network can create curved decision boundaries to separate them correctly.

Effect of Non-Linearity

The inclusion of the ReLU activation function σ*σ* allows h1*h*1 to introduce a non-linear decision boundary in the input space. This non-linearity enables the network to learn more complex patterns that are not possible with a purely linear model such as:

- Modeling functions that are not linearly separable.
- Increasing the capacity of the network to form multiple decision boundaries based on the combination of weights and biases.

Why is Non-Linearity Important in Neural Networks?

Neural networks consist of neurons that operate using weights, biases and activation functions.

In the learning process these weights and biases are updated based on the error produced at the output—a process known as backpropagation. Activation functions enable backpropagation by providing gradients that are essential for updating the weights and biases.

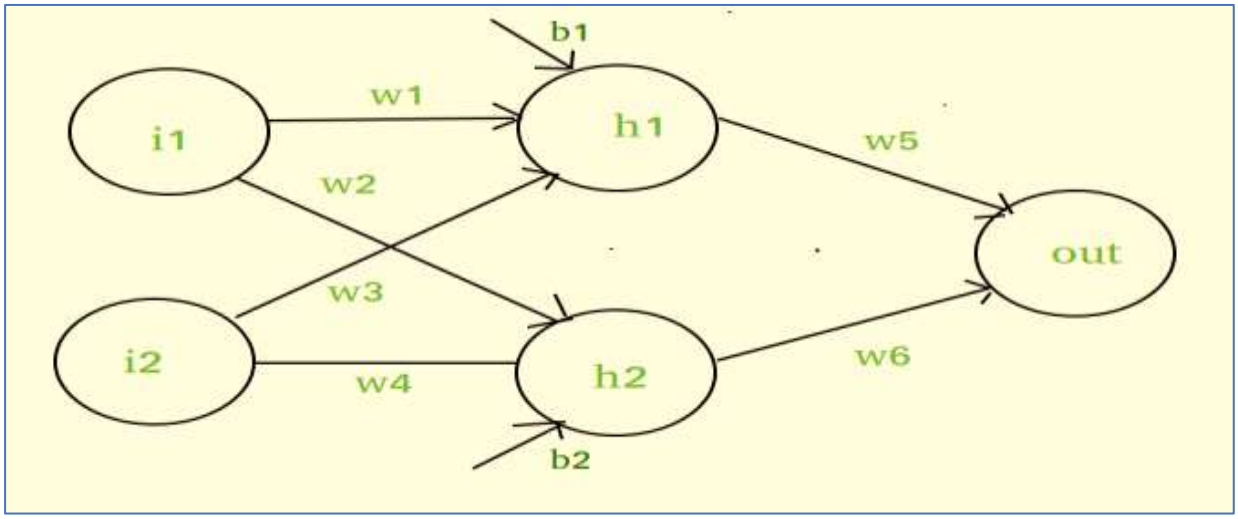
Without non-linearity even deep networks would be limited to solving only simple, linearly separable problems. Activation functions help neural networks to model highly complex data distributions and solve advanced deep learning tasks. Adding non-linear activation functions introduce flexibility and enable the network to learn more complex and abstract patterns from data.

Mathematical Proof of Need of Non-Linearity in Neural Networks

To illustrate the need for non-linearity in neural networks with a specific example let's consider a network with two input nodes (i1and i2)($i1$and $i2$), a single hidden layer containing neurons h1 and h2$h1$ and $h2$ and an output neuron (out).

We will use w1,w2$w1$,$w2$ as weights connecting the inputs to the hidden neuron and w5$w5$ as the weight connecting the hidden neuron to the output. We'll also include biases (b1$b1$ for the hidden neuron and b2$b2$ for the output neuron) to complete the model.

1. **Input Layer**: Two inputs i1$i1$ and i2$i2$.
2. **Hidden Layer**: Two neuron h1$h1$ and h2$h2$
3. **Output Layer**: One output neuron.



The input to the hidden neuron h1$h1$ is calculated as a weighted sum of the inputs plus a bias:

h1=i1.w1+i2.w3+b1$h1=i1.w1+i2.w3+b1$

h2=i1.w2+i2.w4+b2$h2=i1.w2+i2.w4+b2$

The output neuron is then a weighted sum of the hidden neuron's output plus a bias:

output=h1.w5+h2.w6+biasoutput=$h1.w5+h2.w6$+bias

Here, h_1 , h_2 \text{ and output} are linear expressions.

In order to add non-linearity, we will be using sigmoid activation function in the output layer:

σ(x)=11+e−xσ(x)=1+e−x1

final output=σ(h1.w5+h2.w6+bias)final output=σ($h1.w5+h2.w6$+bias)

final output=11+e−(h1.w5+h2.w6+bias)final output=1+e−($h1.w5+h2.w6$+bias)1

This gives the final output of the network after applying the sigmoid activation function in output layers, introducing the desired non-linearity.

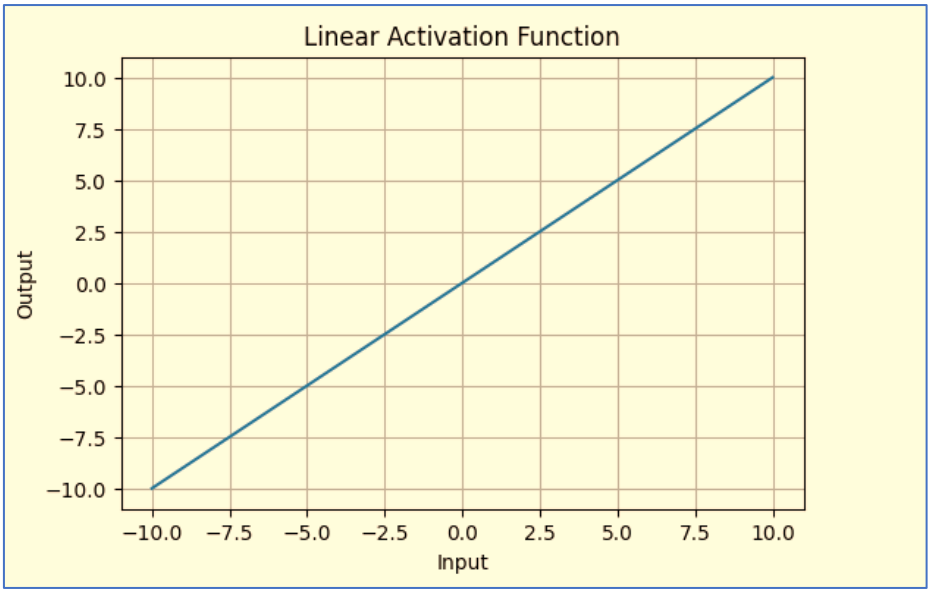
Types of Activation Functions in Deep Learning

1. Linear Activation Function

Linear Activation Function resembles straight line define by y=x. No matter how many layers the neural network contains if they all use linear activation functions the output is a linear combination of the input.

- The range of the output spans from (−∞ to +∞)(−∞ to +∞).
- Linear activation function is used at just one place i.e. output layer.
- Using linear activation across all layers makes the network's ability to learn complex patterns limited.

Linear activation functions are useful for specific tasks but must be combined with non-linear functions to enhance the neural network’s learning and predictive capabilities.



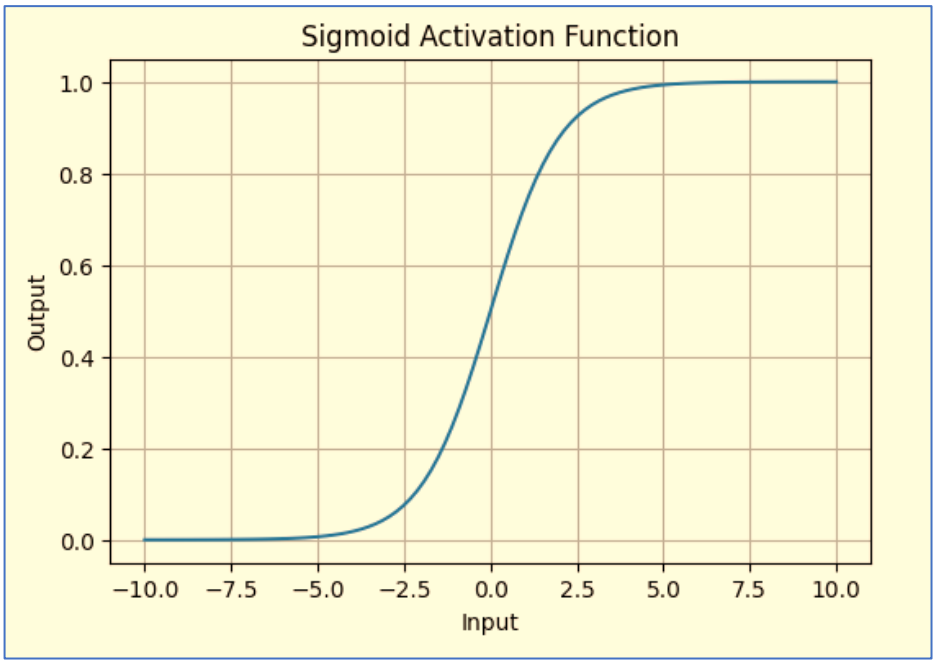
Linear Activation Function or Identity Function returns the input as the output

2. Non-Linear Activation Functions

1. Sigmoid Function

Sigmoid Activation Function is characterized by 'S' shape. It is mathematically defined asA=11+e−xA=1+e−x1. This formula ensures a smooth and continuous output that is essential for gradient-based optimization methods.

- It allows neural networks to handle and model complex patterns that linear equations cannot.
- The output ranges between 0 and 1, hence useful for binary classification.
- The function exhibits a steep gradient when x values are between -2 and 2. This sensitivity means that small changes in input x can cause significant changes in output y which is critical during the training process.



Sigmoid or Logistic Activation Function Graph

2. Tanh Activation Function

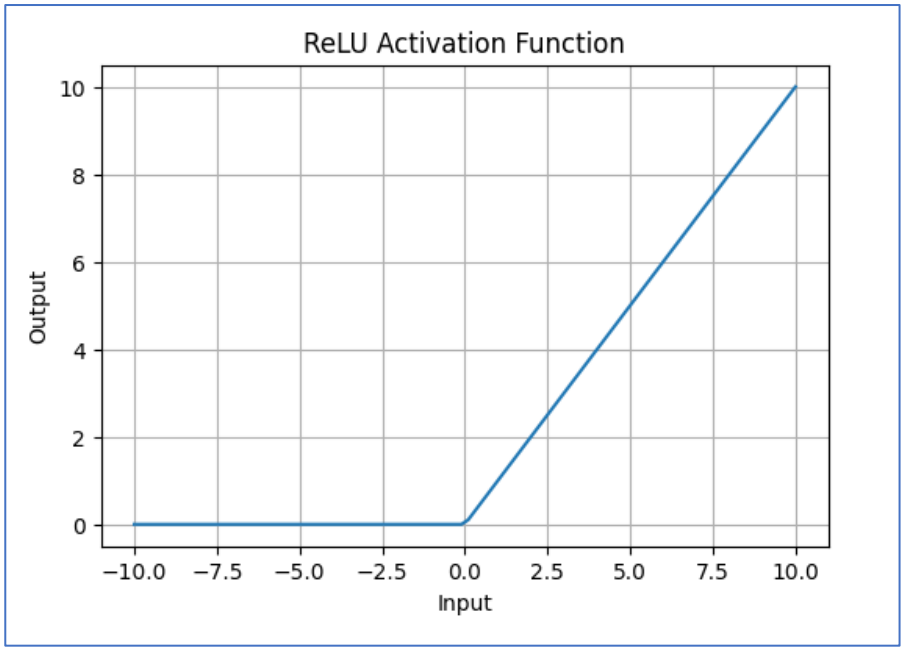
Tanh function (hyperbolic tangent function) is a shifted version of the sigmoid, allowing it to stretch across the y-axis. It is defined as:

f(x)=tanh⁡(x)=21+e−2x−1.f(x)=tanh(x)=1+e−2x2−1.

Alternatively, it can be expressed using the sigmoid function:

tanh⁡(x)=2×sigmoid(2x)−1tanh(x)=2×sigmoid(2x)−1

- **Value Range**: Outputs values from -1 to +1.
- **Non-linear**: Enables modeling of complex data patterns.



- **Use in Hidden Layers**: Commonly used in hidden layers due to its zero-centered output, facilitating easier learning for subsequent layers.

Tanh Activation Function

3. ReLU (Rectified Linear Unit) **Function**

ReLU activation is defined by A(x)=max(0,x)$A(x) = \max(0, x)$, this means that if the input x is positive, ReLU returns x, if the input is negative, it returns 0.

- **Value Range**: [0,∞)[0,∞), meaning the function only outputs non-negative values.
- **Nature**: It is a non-linear activation function, allowing neural networks to learn complex patterns and making backpropagation more efficient.
- **Advantage over other Activation:** ReLU is less computationally expensive than tanh and sigmoid because it involves simpler mathematical operations. At a time only a few neurons are activated making the network sparse making it efficient and easy for computation.

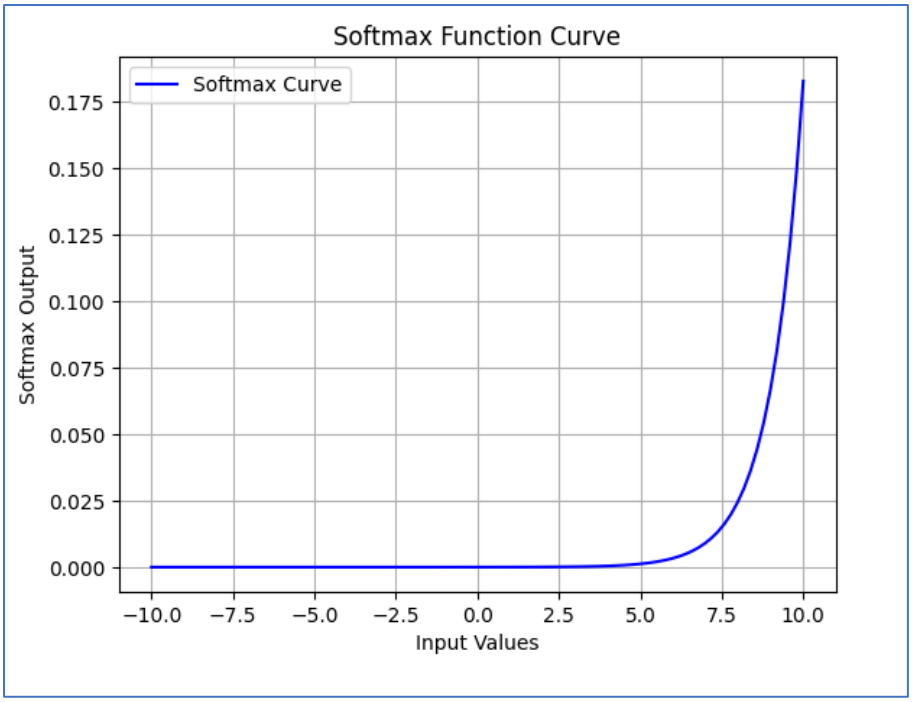
ReLU Activation Function

3. Exponential Linear Units

1. Softmax Function

Softmax function is designed to handle multi-class classification problems. It transforms raw output scores from a neural network into probabilities. It works by squashing the output values of each class into the range of 0 to 1 while ensuring that the sum of all probabilities equals 1.

- Softmax is a non-linear activation function.
- The Softmax function ensures that each class is assigned a probability, helping to identify which class the input belongs to.

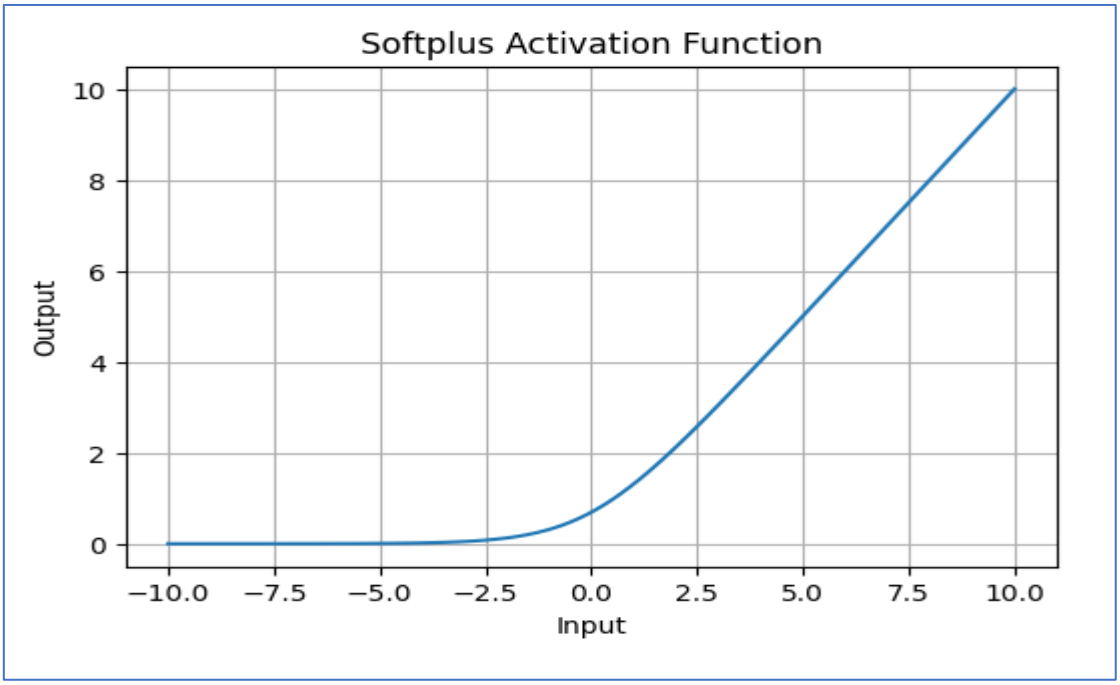


2. SoftPlus Function

Softplus function is defined mathematically as: A(x)=log⁡(1+ex)$A(x) = \log(1 + ex)$.
This equation ensures that the output is always positive and differentiable at all points which is an advantage over the traditional ReLU function.

- **Nature**: The Softplus function is non-linear.
- **Range**: The function outputs values in the range (0,∞)(0,∞), similar to ReLU, but without the hard zero threshold that ReLU has.
- **Smoothness**: Softplus is a smooth, continuous function, meaning it avoids the sharp discontinuities of ReLU which can sometimes lead to problems during optimization.

Loss Functions in Deep Learning



Loss Functions in Deep Learning

A **loss function** is a mathematical way to measure how good or bad a model's predictions are compared to the actual results. It gives a single number that tells us how far off the predictions

are. The smaller the number, the better the model is doing. Loss functions are used to train models. Loss functions are important because they:

1. **Guide Model Training:** During training, algorithms such as Gradient Descent use the loss function to adjust the model's parameters and try to reduce the error and improve the model's predictions.
2. **Measure Performance:** By finding the difference between predicted and actual values and it can be used for evaluating the model's performance.
3. **Affect learning behavior:** Different loss functions can make the model learn in different ways depending on what kind of mistakes they make.

There are many types of loss functions each suited for different tasks. Here are some common methods:

1. Regression Loss Functions

These are used when your model needs to **predict a continuous number** such as predicting the price of a product or age of a person. Popular regression loss functions are:

1. Mean Squared Error (MSE) Loss

Mean Squared Error (MSE) Loss is one of the most widely used loss functions for regression tasks. It calculates the average of the squared differences between the predicted values and the actual values. It is simple to understand and sensitive to outliers because the errors are squared which can affect the loss.

$MSE=1n\sum i=1n(yi-y\hat{}i)2MSE=n1\sum i=1n(yi-yi)2$

2. Mean Absolute Error (MAE) Loss

Mean Absolute Error (MAE) Loss is another commonly used loss function for regression. It calculates the average of the absolute differences between the predicted values and the actual values. It is less sensitive to outliers compared to MSE. But it is not differentiable at zero which can cause issues for some optimization algorithms.

$MAE=1n\sum i=1n|yi-yi\hat{}|MAE=n1\sum i=1n|yi-yi|$

3. Huber Loss

Huber Loss combines the advantages of MSE and MAE. It is less sensitive to outliers than MSE and differentiable everywhere unlike MAE. It requires tuning of the parameter $\delta\delta$. Huber Loss is defined as:

$\{12(yi-y\hat{}i)2for\ |yi-y\hat{}i|\leq\delta\delta|yi-y\hat{}i|-12\delta2for\ |yi-y\hat{}i|>\delta\{21(yi-y\hat{}i)2\delta|yi-y\hat{}i|-21\delta2for\ |yi-y\hat{}i|\leq\delta for\ |yi-y\hat{}i|>\delta$

2. Classification Loss Functions

Classification loss functions are used to evaluate how well a classification model's predictions match the actual class labels. There are different types of classification Loss functions:

1. Binary Cross-Entropy Loss (Log Loss)

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

$Binary\ Cross\text{-}Entropy=-1n\sum i=1n[yilog(y\hat{}i)+(1-yi)log(1-y\hat{}i)]Binary\ Cross\text{-}Entropy=-n1\sum i=1n[yilog(y\hat{}i)+(1-yi)log(1-y\hat{}i)]$

where:

- n is the number of data points
- yiyi is the actual binary label (0 or 1)
- y^i$y\hat{}i$ is the predicted probability.

2. Categorical Cross-Entropy Loss

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

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

where:

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

3. Sparse Categorical Cross-Entropy Loss

Sparse Categorical Cross-Entropy Loss is similar to Categorical Cross-Entropy Loss but is used when the target labels are integers instead of one-hot encoded vectors. It is efficient for large datasets with many classes.

$$\text{Sparse Categorical Cross-Entropy} = -\sum_{i=1}^{n} \log(\hat{y}_i, y_i)$$

where y_i is the integer representing the correct class for data point i.

4. Kullback-Leibler Divergence Loss (KL Divergence)

KL Divergence measures how one probability distribution diverges from a second expected probability distribution. It is often used in probabilistic models. It is sensitive to small differences in probability distributions.

$$\text{KL Divergence} = \sum_{i=1}^{n}\sum_{j=1}^{k} y_{ij} \log(\hat{y}_{ij} y_{ij})$$

5. Hinge Loss

Hinge Loss is used for training classifiers especially for support vector machines (SVMs). It is suitable for binary classification tasks as it is not differentiable at zero.

$$\text{Hinge Loss} = \frac{1}{n}\sum_{i=1}^{n} \max(0, 1 - y_i \cdot \hat{y}_i)$$

where:

- y_i is the actual label (-1 or 1)
- $\hat{y}_i$ is the predicted value.

3. Ranking Loss Functions

Ranking loss functions are used to evaluate models that predict the relative order of items. These are commonly used in tasks such as recommendation systems and information retrieval.

1. Contrastive Loss

Contrastive Loss is used to learn embeddings such that similar items are closer in the embedding space while dissimilar items are farther apart. It is often used in Siamese networks.

$$\text{Contrastive Loss} = \frac{1}{2N}\sum_{i=1}^{N} (y_i \cdot di^2 + (1 - y_i) \cdot \max(0, m - di)^2)$$

where:

- di is the distance between a pair of embeddings
- y_i is 1 for similar pairs and 0 for dissimilar pairs
- m is a margin.

2. Triplet Loss

Triplet Loss is used to learn embeddings by comparing the relative distances between triplets: anchor, positive example and negative example.

$$\text{Triplet Loss} = \frac{1}{N}\sum_{i=1}^{N} [\|f(x_{ia}) - f(x_{ip})\|_2^2 - \|f(x_{ia}) - f(x_{in})\|_2^2 + \alpha] +$$

where:

- f(x) is the embedding function
- xiaxia is the anchor
- xipxip is the positive example
- xinxin is the negative example
- αα is a margin.

3. Margin Ranking Loss

Margin Ranking Loss measures the relative distances between pairs of items and ensures that the correct ordering is maintained with a specified margin.

$$Margin\ Ranking\ Loss = \frac{1}{N}\sum_{i=1}^{N} max(0, -y_i \cdot (s_{i+} - s_{i-}) + margin)$$

where:

- si+s_{i+} and si−s_{i-} are the scores for the positive and negative samples
- yiy_i is the label indicating the correct ordering.

4. Image and Reconstruction Loss Functions

These loss functions are used to evaluate models that generate or reconstruct images ensuring that the output is as close as possible to the target images.

1. Pixel-wise Cross-Entropy Loss

Pixel-wise Cross-Entropy Loss is used for image segmentation tasks where each pixel is classified independently.

$$Pixel\text{-}wise\ Cross\text{-}Entropy = -\frac{1}{N}\sum_{i=1}^{N}\sum_{c=1}^{C} y_{i,c} log(\hat{y}_{i,c})$$

where:

- N is the number of pixels,
- C is the number of classes
- yi,c$y_{i,c}$ is the binary indicator for the correct class of pixel
- y^i,c$\hat{y}_{i,c}$ is the predicted probability for class c.

2. Dice Loss

Dice Loss is used for image segmentation tasks and is particularly effective for imbalanced datasets. It measures the overlap between the predicted segmentation and the ground truth.

$$Dice\ Loss = 1 - \frac{2\sum_{i=1}^{N} y_i \hat{y}_i}{\sum_{i=1}^{N} y_i + \sum_{i=1}^{N} \hat{y}_i}$$

where:

- yiy_i is the ground truth label
- y^i$\hat{y}_i$ is the predicted label.

3. Jaccard Loss (Intersection over Union, IoU)

Jaccard Loss is also known as IoU Loss that measures the intersection over union of the predicted segmentation and the ground truth.

$$Jaccard\ Loss = 1 - \frac{\sum_{i=1}^{N} y_i \hat{y}_i}{\sum_{i=1}^{N} y_i + \sum_{i=1}^{N} \hat{y}_i - \sum_{i=1}^{N} y_i \hat{y}_i}$$

4. Perceptual Loss

Perceptual Loss measures the difference between high-level features of images rather than pixel-wise differences. It is often used in image generation tasks.

$$Perceptual\ Loss = \sum_{i=1}^{N} \|\phi_j(y_i) - \phi_j(\hat{y}_i)\|_2^2$$

where:

- ϕjϕ_j is a layer in a pre-trained network
- yiy_i and y^i$\hat{y}_i$ are the ground truth and predicted images

5. Total Variation Loss

Total Variation Loss encourages spatial smoothness in images by penalizing differences between adjacent pixels.

$$\text{Total Variation Loss} = \sum_{i,j} ((y_{i,j+1} - y_{i,j})^2 + (y_{i+1,j} - y_{i,j})^2)$$

5. Adversarial Loss Functions

Adversarial loss functions are used in generative adversarial networks (GANs) to train the generator and discriminator networks.

1. Adversarial Loss (GAN Loss)

The standard GAN loss function involves a minimax game between the generator and the discriminator.

$$\min_G \max_D E_{x \sim p_{data}(x)}[\log D(x)] + E_{z \sim p_z(z)}[\log(1 - D(G(z)))]$$

- The discriminator tries to **maximize** the probability of correctly classifying real and fake samples.
- The generator tries to **minimize** the discriminator's ability to tell its outputs are fake.

2. Least Squares GAN Loss

LSGAN modifies the standard GAN loss by using **least squares error** instead of log loss make the training more stable:

Discriminator Loss: $\min_D \frac{1}{2} E_{x \sim p_{data}(x)}[(D(x) - 1)^2] + \frac{1}{2} E_{z \sim p_z(z)}[D(G(z))^2]$

Generator Loss: $\min_G \frac{1}{2} E_{z \sim p_z(z)}[(D(G(z)) - 1)^2]$

6. Specialized Loss Functions

Specialized loss functions are designed for specific tasks such as sequence prediction, count data and cosine similarity.

1. CTC Loss (Connectionist Temporal Classification)

CTC Loss is used for sequence prediction tasks where the alignment between input and output sequences is unknown.

$$\text{CTC Loss} = -\log(p(y|x))$$

where p(y|x) is the probability of the correct output sequence given the input sequence.

2. Poisson Loss

Poisson Loss is used for count data modeling the distribution of the predicted values as a Poisson distribution.

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

$\hat{y}_i$ is the predicted count and y_i is the actual count.

3. Cosine Proximity Loss

Cosine Proximity Loss measures the cosine similarity between the predicted and target vectors encouraging them to point in the same direction.

$$\text{Cosine Proximity Loss} = -\frac{1}{N} \sum_{i=1}^{N} \frac{y_i \cdot \hat{y}_i}{\|y_i\| \|\hat{y}_i\|}$$

4. Earth Mover's Distance (Wasserstein Loss)

Earth Mover's Distance measures the distance between two probability distributions and is used in Wasserstein GANs.

$$\text{Wasserstein Loss} = E_{x \sim p_r}[D(x)] - E_{z \sim p_z}[D(G(z))]$$

How to Choose the Right Loss Function?

Choosing the right loss function is very important for training a deep learning model that works well. Here are some guidelines to help you make the right choice:

- **Understand the Task** : The first step in choosing the right loss function is to understand what your model is trying to do. Use MSE or MAE for regression, Cross-

Entropy for classification, Contrastive or Triplet Loss for ranking and Dice or Jaccard Loss for image segmentation.

- **Consider the Output Type**: You should also think about the type of output your model produces. If the output is a continuous number use regression loss functions like MSE or MAE, classification losses for labels and CTC Loss for sequence outputs like speech or handwriting.
- **Handle Imbalanced Data**: If your dataset is imbalanced one class appears much more often than others it's important to use a loss function that can handle this. Focal Loss is useful for such cases because it focuses more on the harder-to-predict or rare examples and help the model learn better from them.
- **Robust to Outliers**: When your data has outliers it's better to use a loss function that's less sensitive to them. Huber Loss is a good option because it combines the strengths of both MSE and MAE and make it more robust and stable when outliers are present.
- **Performance and Convergence**: Choose loss functions that help your model converge faster and perform better. For example using Hinge Loss for SVMs can sometimes lead to better performance than Cross-Entropy for classification.

Backpropagation in Neural Network

Back Propagation is also known as "Backward Propagation of Errors" is a method used to train neural network . Its goal is to reduce the difference between the model's predicted output and the actual output by adjusting the weights and biases in the network.

It works iteratively to adjust weights and bias to minimize the cost function. In each epoch the model adapts these parameters by reducing loss by following the error gradient. It often uses optimization algorithms like **gradient descent** or **stochastic gradient descent**. The algorithm computes the gradient using the chain rule from calculus allowing it to effectively navigate complex layers in the neural network to minimize the cost function.

Fig(a) A simple illustration of how the backpropagation works by adjustments of weights

Back Propagation plays a critical role in how neural networks improve over time. Here's why:

1. **Efficient Weight Update**: It computes the gradient of the loss function with respect to each weight using the chain rule making it possible to update weights efficiently.
2. **Scalability**: The Back Propagation algorithm scales well to networks with multiple layers and complex architectures making deep learning feasible.
3. **Automated Learning**: With Back Propagation the learning process becomes automated and the model can adjust itself to optimize its performance.

Working of Back Propagation Algorithm

The Back Propagation algorithm involves two main steps: the **Forward Pass** and the **Backward Pass**.

1. Forward Pass Work

In **forward pass** the input data is fed into the input layer. These inputs combined with their respective weights are passed to hidden layers. For example in a network with two hidden layers (h1 and h2) the output from h1 serves as the input to h2. Before applying an activation function, a bias is added to the weighted inputs.

Each hidden layer computes the weighted sum (`a`) of the inputs then applies an activation function like **ReLU (Rectified Linear Unit)** to obtain the output (`o`). The output is passed to the

next layer where an activation function such as **softmax** converts the weighted outputs into probabilities for classification.

The forward pass using weights and biases

2. Backward Pass

In the backward pass the error (the difference between the predicted and actual output) is propagated back through the network to adjust the weights and biases. One common method for error calculation is the **Mean Squared Error (MSE)** given by:

MSE=(Predicted Output−Actual Output)2MSE=(Predicted Output−Actual Output)2

Once the error is calculated the network adjusts weights using **gradients** which are computed with the chain rule. These gradients indicate how much each weight and bias should be adjusted to minimize the error in the next iteration. The backward pass continues layer by layer ensuring that the network learns and improves its performance. The activation function through its derivative plays a crucial role in computing these gradients during Back Propagation.

Example of Back Propagation in Machine Learning

Let's walk through an example of Back Propagation in machine learning. Assume the neurons use the sigmoid activation function for the forward and backward pass. The target output is 0.5 and the learning rate is 1.

Example (1) of backpropagation sum

Forward Propagation

1. Initial Calculation

The weighted sum at each node is calculated using:

$aj=\sum(wi,j*xi)aj=\sum(wi,j*xi)$

Where,

- ajaj is the weighted sum of all the inputs and weights at each node
- wi,jwi,j represents the weights between the ithithinput and the jthjth neuron
- xixi represents the value of the ithith input

o **(output):** After applying the activation function to a, we get the output of the neuron:

ojoj = activation function(ajaj)

2. Sigmoid Function

The sigmoid function returns a value between 0 and 1, introducing non-linearity into the model.

$yj=11+e-ajyj=1+e-aj1$

To find the outputs of y3, y4 and y5

3. Computing Outputs

At h1 node

a1=(w1,1x1)+(w2,1x2)=(0.2∗0.35)+(0.2∗0.7)=0.21$a1=(w1,1x1)+(w2,1x2$
$)=(0.2*0.35)+(0.2*0.7)=0.21$

Once we calculated the a1 value, we can now proceed to find the y3 value:

yj=F(aj)=11+e−a1$yj=F(aj)=1+e-a11$

y3=F(0.21)=11+e−0.21$y3=F(0.21)=1+e-0.211$

y3=0.56$y3=0.56$

Similarly find the values of y4 at **h2** and y5 at O3

a2=(w1,2∗x1)+(w2,2∗x2)=(0.3∗0.35)+(0.3∗0.7)=0.315$a2=(w1,2*x1)+(w2,2*x2$
$)=(0.3*0.35)+(0.3*0.7)=0.315$

y4=F(0.315)=11+e−0.315$y4=F(0.315)=1+e-0.3151$

a3=(w1,3∗y3)+(w2,3∗y4)=(0.3∗0.57)+(0.9∗0.59)=0.702$a3=(w1,3*y3)+(w2,3*y4$
$)=(0.3*0.57)+(0.9*0.59)=0.702$

y5=F(0.702)=11+e−0.702=0.67$y5=F(0.702)=1+e-0.7021=0.67$

Values of y3, y4 and y5

4. Error Calculation

Our actual output is 0.5 but we obtained 0.67. To calculate the error we can use the below formula:

$Errorj=ytarget−y5Errorj=ytarget−y5$

$\Rightarrow 0.5−0.67=−0.17 \Rightarrow 0.5−0.67=−0.17$

Using this error value we will be backpropagating.

Back Propagation

1. Calculating Gradients

The change in each weight is calculated as:

$\Delta wij=\eta\times\delta j\times Oj\Delta wij=\eta\times\delta j\times Oj$

Where:

- δjδj is the error term for each unit,
- ηη is the learning rate.

2. Output Unit Error

For O3:

$\delta 5=y5(1−y5)(ytarget−y5)\delta 5=y5(1−y5)(ytarget−y5)$

$=0.67(1−0.67)(−0.17)=−0.0376=0.67(1−0.67)(−0.17)=−0.0376$

3. Hidden Unit Error

For h1:

$\delta 3=y3(1−y3)(w1,3\times\delta 5)\delta 3=y3(1−y3)(w1,3\times\delta 5)$

$=0.56(1−0.56)(0.3\times−0.0376)=−0.0027=0.56(1−0.56)(0.3\times−0.0376)=−0.0027$

For h2:

$\delta 4=y4(1−y4)(w2,3\times\delta 5)\delta 4=y4(1−y4)(w2,3\times\delta 5)$

$=0.59(1−0.59)(0.9\times−0.0376)=−0.0819=0.59(1−0.59)(0.9\times−0.0376)=−0.0819$

4. Weight Updates

For the weights from hidden to output layer:

$\Delta w2,3=1\times(−0.0376)\times0.59=−0.022184\Delta w2,3=1\times(−0.0376)\times0.59=−0.022184$

New weight:

$w2,3(new)=−0.022184+0.9=0.877816w2,3(new)=−0.022184+0.9=0.877816$

For weights from input to hidden layer:

$\Delta w1,1=1\times(−0.0027)\times0.35=0.000945\Delta w1,1=1\times(−0.0027)\times0.35=0.000945$

New weight:

$w1,1(new)=0.000945+0.2=0.200945w1,1(new)=0.000945+0.2=0.200945$

Similarly other weights are updated:

- $w1,2(new)=0.273225w1,2(new)=0.273225$
- $w1,3(new)=0.086615w1,3(new)=0.086615$
- $w2,1(new)=0.269445w2,1(new)=0.269445$
- $w2,2(new)=0.18534w2,2(new)=0.18534$

The updated weights are illustrated below

Through backward pass the weights are updated

After updating the weights the forward pass is repeated yielding:

- $y3=0.57y3=0.57$
- $y4=0.56y4=0.56$
- $y5=0.61y5=0.61$

Since $y5=0.61y5=0.61$ is still not the target output the process of calculating the error and backpropagating continues until the desired output is reached.

This process demonstrates how Back Propagation iteratively updates weights by minimizing errors until the network accurately predicts the output.

$Error=ytarget−y5Error=ytarget−y5$

$=0.5−0.61=−0.11=0.5−0.61=−0.11$

This process is said to be continued until the actual output is gained by the neural network.

Back Propagation Implementation in Python for XOR Problem

This code demonstrates how Back Propagation is used in a neural network to solve the XOR problem. The neural network consists of:

1. Defining Neural Network

We define a neural network as Input layer with 2 inputs, Hidden layer with 4 neurons, Output layer with 1 output neuron and use **Sigmoid** function as activation function.

- **self.input_size = input_size**: stores the size of the input layer
- **self.hidden_size = hidden_size:** stores the size of the hidden layer
- **self.weights_input_hidden = np.random.randn(self.input_size, self.hidden_size)**: initializes weights for input to hidden layer
- **self.weights_hidden_output = np.random.randn(self.hidden_size, self.output_size)**: initializes weights for hidden to output layer
- **self.bias_hidden = np.zeros((1, self.hidden_size)):** initializes bias for hidden layer
- **self.bias_output = np.zeros((1, self.output_size)):** initializes bias for output layer

```
import numpy as np


class NeuralNetwork:
def __init__(self, input_size, hidden_size, output_size):
self.input_size = input_size
self.hidden_size = hidden_size
self.output_size = output_size

self.weights_input_hidden = np.random.randn(
self.input_size, self.hidden_size)
self.weights_hidden_output = np.random.randn(
self.hidden_size, self.output_size)

self.bias_hidden = np.zeros((1, self.hidden_size))
self.bias_output = np.zeros((1, self.output_size))

def sigmoid(self, x):
return 1 / (1 + np.exp(-x))

def sigmoid_derivative(self, x):
return x * (1 - x)
```

2. Defining Feed Forward Network

In Forward pass inputs are passed through the network activating the hidden and output layers using the sigmoid function.

- **self.hidden_activation = np.dot(X, self.weights_input_hidden) + self.bias_hidden**: calculates activation for hidden layer
- **self.hidden_output= self.sigmoid(self.hidden_activation)**: applies activation function to hidden layer
- **self.output_activation= np.dot(self.hidden_output, self.weights_hidden_output) + self.bias_output:** calculates activation for output layer
- **self.predicted_output = self.sigmoid(self.output_activation):** applies activation function to output layer

```
def feedforward(self, X):
```

```
self.hidden_activation = np.dot(
X, self.weights_input_hidden) + self.bias_hidden
self.hidden_output = self.sigmoid(self.hidden_activation)

self.output_activation = np.dot(
self.hidden_output, self.weights_hidden_output) + self.bias_output
self.predicted_output = self.sigmoid(self.output_activation)

return self.predicted_output
```

3. Defining Backward Network

In Backward pass or Back Propagation the errors between the predicted and actual outputs are computed. The gradients are calculated using the derivative of the sigmoid function and weights and biases are updated accordingly.

- **output_error = y - self.predicted_output:** calculates the error at the output layer
- **output_delta = output_error * self.sigmoid_derivative(self.predicted_output):** calculates the delta for the output layer
- **hidden_error = np.dot(output_delta, self.weights_hidden_output.T):** calculates the error at the hidden layer
- **hidden_delta = hidden_error * self.sigmoid_derivative(self.hidden_output):** calculates the delta for the hidden layer
- **self.weights_hidden_output += np.dot(self.hidden_output.T, output_delta) * learning_rate:** updates weights between hidden and output layers
- **self.weights_input_hidden += np.dot(X.T, hidden_delta) * learning_rate:** updates weights between input and hidden layers

```
def backward(self, X, y, learning_rate):
output_error = y - self.predicted_output
output_delta = output_error * \
self.sigmoid_derivative(self.predicted_output)

hidden_error = np.dot(output_delta, self.weights_hidden_output.T)
hidden_delta = hidden_error * self.sigmoid_derivative(self.hidden_output)

self.weights_hidden_output += np.dot(self.hidden_output.T,
output_delta) * learning_rate
self.bias_output += np.sum(output_delta, axis=0,
keepdims=True) * learning_rate
self.weights_input_hidden += np.dot(X.T, hidden_delta) * learning_rate
self.bias_hidden += np.sum(hidden_delta, axis=0,
keepdims=True) * learning_rate
```

4. Training Network

The network is trained over 10,000 epochs using the Back Propagation algorithm with a learning rate of 0.1 progressively reducing the error.

- **output = self.feedforward(X):** computes the output for the current inputs
- **self.backward(X, y, learning_rate):** updates weights and biases using Back Propagation
- **loss = np.mean(np.square(y - output)):** calculates the mean squared error (MSE) loss

```
def train(self, X, y, epochs, learning_rate):
for epoch in range(epochs):
output = self.feedforward(X)
```

```
self.backward(X, y, learning_rate)
if epoch % 4000 == 0:
loss = np.mean(np.square(y - output))
print(f"Epoch {epoch}, Loss:{loss}")
```

5. Testing Neural Network

- **X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]]):** defines the input data
- **y = np.array([[0], [1], [1], [0]]):** defines the target values
- **nn = NeuralNetwork(input_size=2, hidden_size=4, output_size=1):** initializes the neural network
- **nn.train(X, y, epochs=10000, learning_rate=0.1):** trains the network
- **output = nn.feedforward(X**): gets the final predictions after training

```
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([[0], [1], [1], [0]])

nn = NeuralNetwork(input_size=2, hidden_size=4, output_size=1)
nn.train(X, y, epochs=10000, learning_rate=0.1)

output = nn.feedforward(X)
print("Predictions after training:")
print(output)
```

Output:

Trained Model

```
Epoch 0, Loss:0.271320353888644456
Epoch 4000, Loss:0.12865253014284214
Epoch 8000, Loss:0.006592119352308818
Predictions after training:
[[0.03837966]
 [0.93898139]
 [0.94188724]
 [0.07318271]]
```

- The output shows the training progress of a neural network over 10,000 epochs. Initially the loss was high (0.2713) but it gradually decreased as the network learned reaching a low value of 0.0066 by epoch 8000.
- The final predictions are close to the expected XOR outputs: approximately 0 for [0, 0] and [1, 1] and approximately 1 for [0, 1] and [1, 0] indicating that the network successfully learned to approximate the XOR function.

Advantages of Back Propagation for Neural Network Training

The key benefits of using the Back Propagation algorithm are:

1. **Ease of Implementation:** Back Propagation is beginner-friendly requiring no prior neural network knowledge and simplifies programming by adjusting weights with error derivatives.
2. **Simplicity and Flexibility:** Its straightforward design suits a range of tasks from basic feedforward to complex convolutional or recurrent networks.
3. **Efficiency**: Back Propagation accelerates learning by directly updating weights based on error especially in deep networks.
4. **Generalization:** It helps models generalize well to new data improving prediction accuracy on unseen examples.
5. **Scalability:** The algorithm scales efficiently with larger datasets and more complex networks making it ideal for large-scale tasks.

Challenges with Back Propagation

While Back Propagation is useful it does face some challenges:

1. **Vanishing Gradient Problem**: In deep networks the gradients can become very small during Back Propagation making it difficult for the network to learn. This is common when using activation functions like sigmoid or tanh.
2. **Exploding Gradients**: The gradients can also become excessively large causing the network to diverge during training.
3. **Overfitting:** If the network is too complex it might memorize the training data instead of learning general patterns.

Learning Rate in Neural Network

The learning rate is a key hyperparameter in neural networks that controls how quickly the model learns during training. It determines the size of the steps taken to minimize the loss function. It controls how much change is made in response to the error encountered, each time the model weights are updated. It determines the size of the steps taken towards a minimum of the loss function during optimization.

In mathematical terms, when using a method like **Stochastic Gradient Descent (SGD)**, the learning rate (often denoted as $\alpha\alpha$ or $\eta\eta$) is multiplied by the gradient of the loss function to update the weights:

$\mathrm{w}=\mathrm{w}-\alpha\cdot\nabla\mathrm{L}(\mathrm{w})w=w-\alpha\cdot\nabla L(w)$

Where:

- $\mathrm{w}w$ represents the weights
- $\alpha\alpha$ is the learning rate
- $\nabla\mathrm{L}(\mathrm{w})\nabla L(w)$ is the gradient of the loss function

Impact of Learning Rate on Model

The learning rate is a critical hyperparameter that directly affects how a model learns during training by controlling the magnitude of weight updates. Its value significantly affects both convergence speed and model performance.

Low Learning Rate:

- Leads to slow convergence
- Requires more training epochs
- Can improve accuracy but increases computation time

High Learning Rate:

- Speeds up training
- Risks of overshooting optimal weights
- May cause instability or divergence of the loss function

Optimal Learning Rate:

- Balances training speed and model accuracy
- Ensures stable convergence without excessive training time

Best Practices:

- Fine-tune the learning rate based on the task and model
- Use techniques like **learning rate scheduling** or **adaptive optimizers** to improve performance and stability

Identifying the ideal learning rate can be challenging but is important for improving performance without wasting resources.

Techniques for Adjusting the Learning Rate

1. Fixed Learning Rate

- A constant learning rate is maintained throughout training.
- Simple to implement and commonly used in basic models.
- Its limitation is that it lacks the ability to adapt on different training phases which may create sub optimal results.

2. Learning Rate Schedules

These techniques reduce the learning rate over time based on predefined rules to improve convergence:

- **Step Decay**: Reduces the learning rate by a fixed factor at set intervals (every few epochs).
- **Exponential Decay**: Continuously decreases the learning rate exponentially over training time.
- **Polynomial Decay**: Learning rate decays polynomially, offering smoother transitions compared to step or exponential methods.

3. Adaptive Learning Rate Methods

Adaptive methods adjust the learning rate dynamically based on gradient information, allowing better updates per parameter:

- **AdaGrad**: AdaGrad adapts the learning rate per parameter based on the squared gradients. It is effective for sparse data but may decay too quickly.
- **RMSprop:** RMSprop builds on AdaGrad by using a moving average of squared gradients to prevent aggressive decay.
- **Adam (Adaptive Moment Estimation):** Adam combines RMSprop with momentum to provide stable and fast convergence; widely used in practice.

4. Cyclic Learning Rate

- The learning rate oscillates between a minimum and maximum value in a cyclic manner throughout training.
- It increases and then decreases the learning rate linearly in each cycle.
- Benefits include better exploration of the loss surface and leading to faster convergence.

5. Decaying Learning Rate

- Gradually reduces the learning rate as training progresses.
- Helps the model take more precise steps towards the minimum. This improves stability in later epochs.

Optimization Algorithm in Deep Learning

Optimization algorithms in deep learning are used to minimize the loss function by adjusting the weights and biases of the model. The most common ones are:

- Optimization algorithms in deep learning
- Gradient Descent
- Stochastic Gradient Descent (SGD)
- Batch Normalization
- Mini-batch Gradient Descent
- Adam (Adaptive Moment Estimation)
- Momentum-based Gradient Optimizer
- Adagrad Optimizer
- RMSProp Optimizer

Optimization Algorithms in Machine Learning

First-Order Algorithms

First-order optimization algorithms are methods that rely on the first derivative (gradient) of the objective function to find the minimum or maximum. They use gradient information to decide the direction and size of updates for model parameters. These algorithms are widely used in machine learning due to their simplicity and efficiency, especially for large-scale problems. Below are some First-Order Algorithms:

1. Gradient Descent and Its Variants

Gradient Descent is an optimization algorithm used for minimizing the objective function by iteratively moving towards the minimum. It is a first-order iterative algorithm for finding a local minimum. The algorithm works by taking repeated steps in the opposite direction of the gradient of the function at the current point because it will be the direction of steepest descent.

Let's assume we want to minimize the function f(x)=x2 using gradient descent.

- The main function gradient_descent takes the gradient, a starting point, learning rate, number of iterations and a convergence tolerance.
- In each iteration, it calculates the gradient at the current point and updates the point in the opposite direction of the gradient (descent), scaled by the learning rate.
- The update continues until either the maximum number of iterations is reached or the update magnitude falls below the specified tolerance.
- The final result is printed which should be a value close to the minimum of the function.

```
import numpy as np

# Define the gradient function for f(x) = x^2
def gradient(x):
return 2 * x

# Gradient descent optimization function
def gradient_descent(gradient, start, learn_rate, n_iter=50, tolerance=1e-06):
vector = start
for _ in range(n_iter):
diff = -learn_rate * gradient(vector)
if np.all(np.abs(diff) <= tolerance):
break
vector += diff
return vector

# Initial point
start = 5.0
# Learning rate
learn_rate = 0.1
# Number of iterations
n_iter = 50
# Tolerance for convergence
tolerance = 1e-6

# Gradient descent optimization
```

```
result = gradient_descent(gradient, start, learn_rate, n_iter, tolerance)
print(result)
```

Output:

```
7.1362384635298026e-05
```

Output of Gradient

Variants of Gradient Descent

- **Stochastic Gradient Descent (SGD):** This variant suggests model update using a single training example at a time which does not require a large amount of computation and therefore is suitable for large datasets.
- **Mini-Batch Gradient Descent:** This method is designed so that it computes it for every mini-batches of data, a balance between amount of time and precision. It converges faster than SGD and is used widely in practice to train many deep learning models.
- **Momentum:** Momentum improves SGD by adding information of the previous steps of the algorithm to the next step. By adding a portion of the current update vector to the previous update, it enables the algorithm to go through flat areas and noisy gradients to minimize the time to train and find convergence.

2. Stochastic Optimization Techniques

Stochastic optimization techniques introduce randomness to the search process which can be advantageous for tackling complex optimization problems where traditional methods might struggle.

- **Simulated Annealing:** Similar to the annealing process in metallurgy this technique starts with a high temperature (high randomness) that allows exploration of the search space widely. Over time, the temperature decreases (randomness decreases) which helps the algorithm converge towards better solutions while avoiding local minima.
- **Random Search:** This simple method randomly chooses points in the search space then evaluates them. Random search is actually quite effective particularly for optimization problems that are high-dimensional. The ease of implementation and its ability to work with complex algorithms makes this approach widely used.

When using stochastic optimization algorithms, we consider the following practical aspects:

- **Repeated Evaluations**: Stochastic optimization algorithms often need repeated evaluations of the objective function which is time-consuming. Therefore, we have to balance the number of evaluations with the computational resources available.
- **Problem Structure**: The choice of stochastic optimization algorithm depends on the structure of the problem. For example, simulated annealing is suitable for problems with multiple local optima while random search is effective for high-dimensional optimization landscapes.

3. Evolutionary Algorithms

In evolutionary algorithms we take inspiration from natural selection and include techniques such as Genetic Algorithms and Differential Evolution. They are often used to solve complex optimization problems that are difficult to solve using traditional methods.

Key Components:

- **Population**: Set of candidate solutions to the optimization problem.
- **Fitness Function**: A function that evaluates the quality of each candidate solution.
- **Selection**: Mechanism for selecting the fittest candidates to reproduce.

- **Genetic Operators**: Operators that modify the selected candidates to create new offspring such as crossover and mutation.
- **Termination**: A condition for stopping the algorithm.

1. Genetic Algorithms

These algorithms use crossover and mutation operators to evolve the candidate population. It is commonly used to generate solutions to optimization/search problems by relying on biologically inspired operators such as mutation, crossover and selection. In the code example below we implement a Genetic Algorithm to minimize:

$f(x)=\sum i=1nxi2f(x)=\sum i=1nxi2$

- fitness_func returns the negative sum of squares to convert minimization into maximization.
- generate_population creates random individuals between 0 and 1.
- Each generation, the top 50% (fittest) are selected as parents.
- Offspring are created via single-point crossover between two parents.
- Mutation randomly alters one gene with a small probability.
- The process repeats for a fixed number of generations.
- Outputs the best individual and its minimized objective value.

```
import numpy as np

# Define the fitness function (negative of the objective function)
def fitness_func(individual):
return -np.sum(individual**2)

# Generate an initial population
def generate_population(size, dim):
return np.random.rand(size, dim)

# Genetic algorithm
def genetic_algorithm(population, fitness_func, n_generations=100, mutation_rate=0.01):
for _ in range(n_generations):
population = sorted(population, key=fitness_func, reverse=True)
next_generation = population[:len(population)//2].copy()
while len(next_generation) < len(population):
parents_indices = np.random.choice(len(next_generation), 2, replace=False)
parent1, parent2 = next_generation[parents_indices[0]], next_generation[parents_indices[1]]
crossover_point = np.random.randint(1, len(parent1))
child = np.concatenate((parent1[:crossover_point], parent2[crossover_point:]))
if np.random.rand() < mutation_rate:
mutate_point = np.random.randint(len(child))
child[mutate_point] = np.random.rand()
next_generation.append(child)
population = np.array(next_generation)
return population[0]

# Parameters
population_size = 10
dimension = 5
n_generations = 50
mutation_rate = 0.05

# Initialize population
```

```
population = generate_population(population_size, dimension)

# Run genetic algorithm
best_individual = genetic_algorithm(population, fitness_func, n_generations, mutation_rate)

# Output the best individual and its fitness
print("Best individual:", best_individual)
print("Best fitness:", -fitness_func(best_individual))  # Convert back to positive for the objective value
```

Output:

```
Best individual: [0.14922262 0.31997868 0.21888492 0.35663834 0.50370858]
Best fitness: 0.553477598138598
```

Output from Genetic algorithm

2. Differential Evolution (DE)

Differential Evolution seeks an optimum of a problem using improvements for a solution. It works by bringing forth new candidate solutions from the population through vector addition. DE is generally performed by mutation and crossover operations to create new vectors and replace low fitting individuals in the population.

This code implements the Differential Evolution (DE) algorithm to minimize our previously demonstrated function f(x)=∑i=1nxi2$f(x)=\sum i=1nxi2$:

- The `differential_evolution` function initializes a population of candidate solutions by sampling uniformly within the specified `bounds` for each parameter.
- For each individual (target vector) in the population, three distinct individuals `a`, `b` and `c` are selected to generate a mutant vector using the formula mutant = a + F·(b−c) where `F` is a scaling factor which controls differential variation.
- A trial vector is created by mixing the target and mutant vectors based on a **crossover rate (CR)**.
- If the fitness of the trial vector is better than that of the target, it replaces the target in the next generation.
- The process repeats for a specified number of generations (`max_generations`).
- This example uses the `sphere_function` as the objective where the goal is to minimize the sum of squares of the vector elements and the bounds define a 10-dimensional search space from −5.12 to 5.12.
- After optimization, the code prints the best solution found and its corresponding fitness value.

```
import numpy as np

def differential_evolution(objective_func, bounds, pop_size=50, max_generations=100, F=0.5, CR=0.7, seed=None):
np.random.seed(seed)
n_params = len(bounds)
population = np.random.uniform(bounds[:, 0], bounds[:, 1], size=(pop_size, n_params))
best_solution = None
best_fitness = np.inf

for generation in range(max_generations):
for i in range(pop_size):
target_vector = population[i]
indices = [idx for idx in range(pop_size) if idx != i]
a, b, c = population[np.random.choice(indices, 3, replace=False)]
```

```
mutant_vector = np.clip(a + F * (b - c), bounds[:, 0], bounds[:, 1])
crossover_mask = np.random.rand(n_params) < CR
trial_vector = np.where(crossover_mask, mutant_vector, target_vector)
trial_fitness = objective_func(trial_vector)
if trial_fitness < best_fitness:
best_fitness = trial_fitness
best_solution = trial_vector
if trial_fitness <= objective_func(target_vector):
population[i] = trial_vector

return best_solution, best_fitness

# Example objective function (minimization)
def sphere_function(x):
return np.sum(x**2)

# Define the bounds for each parameter
bounds = np.array([[-5.12, 5.12]] * 10)  # Example: 10 parameters in [-5.12, 5.12] range

# Run Differential Evolution
best_solution, best_fitness = differential_evolution(sphere_function, bounds)

# Output the best solution and its fitness
print("Best solution:", best_solution)
print("Best fitness:", best_fitness)
```

Output:

Output from Differential Evolution

4. Metaheuristic Optimization

Metaheuristic optimization algorithms are used to supply strategies at guiding lower level heuristic techniques that are used in the optimization of difficult search spaces. Tabu search and iterated local search are two techniques that are used to enhance the capabilities of local search algorithms.

1. Tabu Search

Tabu Search improves the efficiency of local search by using memory structures that prevent cycling back to recently visited solutions. This helps the algorithm escape local optima and explore new regions of the search space.

Key Components:

- **Tabu List**: A short-term memory structure that stores recently visited solutions or moves. Any move that results in a solution on this list is considered forbidden(tabu) which helps prevent revisiting the same solutions.
- **Aspiration Criteria**: An override rule that allows the algorithm to accept a tabu move if it results in a solution better than the best known so far.
- **Neighborhood Search**: At each iteration, the algorithm explores neighboring solutions and selects the best one that is not in the tabu list. If all potential moves are tabu, the best move is chosen based on the aspiration criteria.

2. Iterated Local Search (ILS)

Iterated Local Search is another strategy for enhancing local search, but unlike Tabu Search, it does not use memory structures. It relies on repeated application of local search, combined with random changes to escape local minima and continue the search.

Key Components:

- **Local Search**: Starts with an initial solution and performs local search to find a local optimum.
- **Perturbation**: Applies a small random change to the current solution, effectively getting it out of its current local optimum.
- **Restart Mechanism**: The perturbed solution is used as a new starting point for local search. If the newly found solution is better than the current best, it is accepted. If not the search continues with further perturbations.
- **Exploration vs. Exploitation**: ILS balances exploration (through perturbation) and exploitation (local search), making it simple yet effective for a wide range of optimization problems.

5. Swarm Intelligence Algorithms

Swarm intelligence algorithms resemble natural systems by using the collective, decentralized behavior observed in organisms like bird flocks and insect colonies. These systems operate through shared rules and interactions among individual agents, enabling efficient problem-solving through cooperation.

There are two of the widely applied algorithms in swarm intelligence:

1. Particle Swarm Optimization (PSO)

Particle Swarm Optimization (PSO) is a population-based optimization algorithm inspired by the social behavior of bird flocks and fish schools. Each individual in the swarm (a particle), represents a potential solution. These particles move through the search space by updating their positions based on experience and knowledge shared by neighboring particles. This cooperative mechanism helps the swarm converge toward optimal or near-optimal solutions.

Below is a simple Python implementation of PSO to minimize the **Rastrigin function**, a common benchmark in optimization problems:

- Each particle has a position, velocity and remembers its personal best position and value.
- Velocity is updated using: **Inertia** (current movement), **Cognitive component** (attraction to personal best) and **Social component** (attraction to global best).
- Position is updated by adding velocity and clipped within bounds [–5.12,5.12].
- The **PSO function** initializes particles and updates them over 100 iterations.
- In each iteration, it evaluates fitness, updates personal and global bests and moves particles.
- After all iterations, it returns and prints the **best solution** and its **fitness value**.

```
import numpy as np

def rastrigin(x):
return 10 * len(x) + sum([(xi ** 2 - 10 * np.cos(2 * np.pi * xi)) for xi in x])

class Particle:
def __init__(self, bounds):
self.position = np.random.uniform(bounds[:, 0], bounds[:, 1], len(bounds))
self.velocity = np.random.uniform(-1, 1, len(bounds))
```

```
self.pbest_position = self.position.copy()
self.pbest_value = float('inf')

def update_velocity(self, gbest_position, w=0.5, c1=1.0, c2=1.5):
r1 = np.random.rand(len(self.position))
r2 = np.random.rand(len(self.position))
cognitive_velocity = c1 * r1 * (self.pbest_position - self.position)
social_velocity = c2 * r2 * (gbest_position - self.position)
self.velocity = w * self.velocity + cognitive_velocity + social_velocity

def update_position(self, bounds):
self.position += self.velocity
self.position = np.clip(self.position, bounds[:, 0], bounds[:, 1])

def particle_swarm_optimization(objective_func, bounds, n_particles=30, max_iter=100):
particles = [Particle(bounds) for _ in range(n_particles)]
gbest_position = np.random.uniform(bounds[:, 0], bounds[:, 1], len(bounds))
gbest_value = float('inf')

for _ in range(max_iter):
for particle in particles:
fitness = objective_func(particle.position)
if fitness < particle.pbest_value:
particle.pbest_value = fitness
particle.pbest_position = particle.position.copy()

if fitness < gbest_value:
gbest_value = fitness
gbest_position = particle.position.copy()

for particle in particles:
particle.update_velocity(gbest_position)
particle.update_position(bounds)

return gbest_position, gbest_value

# Define bounds
bounds = np.array([[-5.12, 5.12]] * 10)

# Run PSO
best_solution, best_fitness = particle_swarm_optimization(rastrigin, bounds, n_particles=30,
max_iter=100)

# Output the best solution and its fitness
print("Best solution:", best_solution)
print("Best fitness:", best_fitness)
```

Output:

```
Best solution: [ 9.94959137e-01  9.94958215e-01  9.94963784e-01  4.88302863e-06
 -2.98485629e+00  1.91165848e-06 -9.94978336e-01  9.94958557e-01
  1.98991050e+00 -1.98991238e+00]
Best fitness: 21.889058996466346
```

PSO output

2. Ant Colony Optimization (ACO)

Ant Colony Optimization is inspired by the behavior of ants. Ants find the shortest path between their colony and food sources by laying down pheromones which guide other ants to the path. Here's a basic implementation of ACO for the Traveling Salesman Problem (TSP):

- Each ant constructs a complete tour by selecting unvisited cities based on pheromone intensity and inverse distance.
- The transition probability combines pheromone influence (α) and heuristic desirability (β).
- After each iteration, the best tour is updated if a shorter path is found.
- Pheromone levels are globally evaporated (rate ρ) and reinforced in proportion to the quality (1/length) of each ant's tour.
- The algorithm iterates over multiple generations to converge toward an optimal or near-optimal solution.
- Returns the shortest tour and its total length discovered during the search.

```
import numpy as np

class Ant:
def __init__(self, n_cities):
self.path = []
self.visited = [False] * n_cities
self.distance = 0.0

def visit_city(self, city, distance_matrix):
if len(self.path) > 0:
self.distance += distance_matrix[self.path[-1]][city]
self.path.append(city)
self.visited[city] = True

def path_length(self, distance_matrix):
return self.distance + distance_matrix[self.path[-1]][self.path[0]]

def ant_colony_optimization(distance_matrix, n_ants=10, n_iterations=100, alpha=1, beta=5, rho=0.1, Q=10):
n_cities = len(distance_matrix)
pheromone = np.ones((n_cities, n_cities)) / n_cities
best_path = None
best_length = float('inf')

for _ in range(n_iterations):
ants = [Ant(n_cities) for _ in range(n_ants)]
for ant in ants:
ant.visit_city(np.random.randint(n_cities), distance_matrix)

for _ in range(n_cities - 1):
current_city = ant.path[-1]
probabilities = []
for next_city in range(n_cities):
if not ant.visited[next_city]:
pheromone_level = pheromone[current_city][next_city] ** alpha
heuristic_value = (1.0 / distance_matrix[current_city][next_city]) ** beta
probabilities.append(pheromone_level * heuristic_value)
```

```
else:
probabilities.append(0)
probabilities = np.array(probabilities)
probabilities /= probabilities.sum()
next_city = np.random.choice(range(n_cities), p=probabilities)
ant.visit_city(next_city, distance_matrix)

for ant in ants:
length = ant.path_length(distance_matrix)
if length < best_length:
best_length = length
best_path = ant.path

pheromone *= (1 - rho)
for ant in ants:
contribution = Q / ant.path_length(distance_matrix)
for i in range(n_cities):
pheromone[ant.path[i]][ant.path[(i + 1) % n_cities]] += contribution

return best_path, best_length

# Example distance matrix for a TSP with 5 cities
distance_matrix = np.array([
[0, 2, 2, 5, 7],
[2, 0, 4, 8, 2],
[2, 4, 0, 1, 3],
[5, 8, 1, 0, 6],
[7, 2, 3, 6, 0]
])

# Run ACO
best_path, best_length = ant_colony_optimization(distance_matrix)

# Output the best path and its length
print("Best path:", best_path)
print("Best length:", best_length)
```

Output:

```
Best path: [3, np.int64(2), np.int64(0), np.int64(1), np.int64(4)]
Best length: 13.0
```

Output for ACO

6. Hyperparameter Optimization

Tuning of model parameters that does not directly adapt to datasets is termed as hyperparameter tuning and is a vital process in machine learning. These parameters referred to as the hyperparameters may influence the performance of a certain model. Tuning them is crucial in order to get the best out of the model, as it will theoretically work at its best.

- **Grid Search:** It is a hyperparameter optimization technique that systematically evaluates all combinations of predefined values. As it ensures the best parameters within the specified grid, it is computationally expensive and time-consuming, making it suitable only when resources are ample and the search space is relatively small.

- **Random Search**: It selects hyperparameters randomly from specified distributions. Though it may not always find the absolute best values, it often yields near-optimal results more efficiently, especially in high-dimensional or large parameter spaces.

7. Optimization Techniques in Deep Learning

Deep learning models are usually complex and some contain millions of parameters. These models are dependent on optimization techniques that enable their effective training as well as generalization on unseen data. Different optimizers can effect the speed of convergence and the quality of the result at the output of the model.

Common Techniques are:

- **Adam (Adaptive Moment Estimation):** It is a widely used optimization technique. At each time step, Adam keeps track of both the gradients and their second moments moving average. It is used to modify the learning rate for each parameter in the process. Most of them are computationally efficient, have small memory requirements and are particularly useful for large data and parameters.
- **RMSProp (Root Mean Square Propagation):** It is designed to adapt the learning rate for each parameter individually. It maintains a moving average of the squared gradients to adjust the learning rate dynamically, helping to stabilize training. By scaling the learning rate according to the magnitude of recent gradients, RMSProp ensures more balanced and efficient convergence.

Second-order algorithms

Now that we have discussed about first order algorithms lets now learn about **Second-order optimization algorithms**. They use both the first derivative (gradient) and the second derivative (Hessian) of the objective function. The Hessian provides information about the curvature, helping these methods make more informed and accurate updates. Although they often converge faster and more precisely than first-order methods, they are computationally expensive and less practical for very large datasets or deep learning models.

Below are some Second-order algorithms:

1. Newton's Method and Quasi-Newton Methods

Newton's method and quasi-Newton methods are optimization techniques used to find the minimum or maximum of a function. They are based on the idea of iteratively updating an estimate of the function's Hessian matrix to improve the search direction.

Newton's Method

Newton's method is applied on the basis of the second derivative in order to minimize or maximize Quadratic forms. It has faster rate of convergence than the first-order methods such as gradient descent but has calculation of second order derivative or Hessian matrix which is a challenge when dimensions are high.

Let's consider the function f(x)=x3−2x2+2 and find its minimum using Newton's Method:

- f_prime(x) is the first derivative $f'(x)=3x2-4xf'(x)=3x2-4x$, used to locate critical points.
- f_double_prime(x) is the second derivative $f''(x)=6x-4f''(x)=6x-4$, used to refine convergence and ensure curvature.
- The newtons_method function iteratively updates the estimate using: $xnew=x-f'(x)f''(x)xnew=x-f''(x)f'(x)$.
- Iteration stops when the step size is below a small threshold (tol) or max_iter is reached.
- Starts at $x0x0=3.0$ and returns the value of xx where a local minimum is achieved.
- Final output shows the estimated value of xx where f(x) is minimized.

Define the function and its first and second derivatives

```
def f(x):
return x**3 - 2*x**2 + 2

def f_prime(x):
return 3*x**2 - 4*x

def f_double_prime(x):
return 6*x - 4

def newtons_method(f_prime, f_double_prime, x0, tol=1e-6, max_iter=100):
x = x0
for _ in range(max_iter):
step = f_prime(x) / f_double_prime(x)
if abs(step) < tol:
break
x -= step
return x

# Initial point
x0 = 3.0
# Tolerance for convergence
tol = 1e-6
# Maximum iterations
max_iter = 100

# Apply Newton's Method
result = newtons_method(f_prime, f_double_prime, x0, tol, max_iter)
print("Minimum at x =", result)
```

Output:

```
Minimum at x = 1.3333333423743772
```

Newton's Method Output

Quasi-Newton Methods

Quasi-Newton methods are optimization algorithms that use gradient and curvature information to find local minima, but avoid computing the Hessian matrix explicitly(which Newton's Method does). It has alternatives such as the BFGS (Broyden-Fletcher-Goldfarb-Shanno) and the L-BFGS (Limited-memory BFGS) suited for large-scale optimization due to the fact that direct computation of the Hessian matrix is more challenging.

- **BFGS:** A method such as BFGS constructs an estimation of the Hessian matrix from gradients. It uses this approximation in an iterative manner where it can obtain quick rates of convergence comparable to Newton's Method without the necessity to compute the Hessian form.
- **L-BFGS:** L-BFGS is a memory efficient version of BFGS and suitable for solving problems in large scale. It maintains only a few iterations' updates which results in greater scalability without sacrificing the properties of BFGS convergence.

2. Constrained Optimization

- **Lagrange Multipliers:** Additional variables called Lagrange multipliers are introduced in this method so that a constrained problem can be turned into an unconstrained one.

It is designed for problems having equality constraints which allows finding out the points where both the objective function and constraints are satisfied optimally.

- **KKT Conditions:** These conditions generalize those of Lagrange multipliers to encompass both equality and inequality constraints. They are used to give necessary conditions of optimality for a solution incorporating primal feasibility, dual feasibility as well as complementary slackness thus extending the range of problems under consideration in constrained optimization.

3. Bayesian Optimization

Bayesian optimization is a probabilistic technique for optimizing expensive or complex objective functions. Unlike Grid or Random Search, it uses information from previous evaluations to make informed decisions about which hyperparameter values to test next. This makes it more sample-efficient, often requiring fewer iterations to find optimal solutions. It is useful when function evaluations are costly or computational resources are limited.

Optimization for Specific Machine Learning Tasks

1. Classification Task: Logistic Regression Optimization

Logistic Regression is an algorithm for classification of objects and is widely used in binary classification tasks. It estimates the likelihood of an object being in a class with the help of a logistic function. The optimization goal is the cross-entropy which is a measure of the difference between predicted probabilities and actual class labels.

Define and fit the Model

```
from sklearn.linear_model import LogisticRegression
model = LogisticRegression()
model.fit(X_train, y_train)
```

Optimization Details:

- **Optimizer**: Logistic Regression is optimized using iterative algorithms since it lacks a closed-form solution. Common solvers include Newton's Method, Gradient Descent and its variants selected based on dataset size and sparsity.
- **Loss Function:** The cost function of the Logistic Regression is the log loss or cross entropy, the calculations are made in order to optimize it.

Evaluation: After training, evaluate the model's performance using metrics like accuracy, precision, recall or ROC-AUC depending on the classification problem.

2. Regression Task: Linear Regression Optimization

Linear Regression is an essential method in the regression, as the purpose of the algorithm involves predicting the target variable. The Common goal of optimization model is generally to minimize the Mean Squared Error which represents the difference between the predicted values and the actual target values.

Define and fit the Model

```
from sklearn.linear_model import LinearRegression
model = LinearRegression()
model.fit(X_train, y_train)
```

Optimization Details:

- **Optimizer**: Linear Regression can be solved analytically using the Normal Equation or by using Gradient Descent. For regularized versions (Ridge), solvers like 'lbfgs', 'sag' and 'saga**'** are employed for efficient optimization.
- **Loss Function:** The loss function for Linear Regression is the Mean Squared Error (MSE) which is minimized during training.

Evaluation: After training, evaluate the model's performance using metrics like accuracy, precision, recall or ROC-AUC depending on the classification problem.

Challenges and Limitations of Optimization Algorithms

- **Non-Convexity**: Cost functions of many machine learning algorithms are non-convex which implies that they have a number of local minima and saddle points. Traditional optimization methods cannot guarantee to obtain the global optimum in such complex models.
- **High Dimensionality:** Finding optimal solutions in high-dimensional spaces is challenging, the algorithms and computing resources needed to do so can be expensive.
- **Overfitting:** Regularization neutralizes overfitting which leads to memorization of training data than the new data. The applied model requirements for optimization should be kept as simple as possible due to the risk of overfitting.

Gradient Descent Algorithm in Machine Learning

Gradient descent is the backbone of the learning process for various algorithms, including linear regression, logistic regression, support vector machines, and neural networks which serves as a fundamental optimization technique to minimize the cost function of a model by **iteratively adjusting the model parameters to reduce the difference between predicted and actual values, improving the model's performance**. Let's see it's role in machine learning:

Prerequisites: Understand the working and math of gradient descent.

1. Training Machine Learning Models

Neural networks are trained using Gradient Descent (or its variants) in combination with backpropagation. Backpropagation computes the gradients of the **loss function with respect to each parameter (weights and biases) in the network by applying the chain rule.** The process involves:

- **Forward Propagation**: Computes the output for a given input by passing data through the layers.
- **Backward Propagation**: Uses the chain rule to calculate gradients of the loss with respect to each parameter (weights and biases) across all layers.

Gradients are then used by Gradient Descent to update the parameters layer-by-layer, moving toward minimizing the loss function.

Neural networks often use advanced variants of Gradient Descent. If you want to read more about variants, please refer : Gradient Descent Variants.

2. Minimizing the Cost Function

The algorithm minimizes a cost function, which quantifies the error or loss of the model's predictions compared to the true labels for:

1. Linear Regression

Gradient descent minimizes the Mean Squared Error (MSE) which serves as the loss function to find the best-fit line. Gradient Descent is used to iteratively update the weights (coefficients) and bias by computing the gradient of the MSE with respect to these parameters.

Since MSE is a convex function **gradient descent guarantees convergence to the global minimum if the learning rate is appropriately chosen.** For each iteration:

The algorithm computes the gradient of the MSE with respect to the weights and biases.

It updates the weights (w) and bias (b) using the formula:

- Calculating the gradient of the log-loss with respect to the weights.
- Updating weights and biases iteratively to maximize the likelihood of the correct classification:

$w=w-\alpha\cdot\partial J(w,b)\partial w, b=b-\alpha\cdot\partial J(w,b)\partial b w=w-\alpha\cdot\partial w\partial J(w,b), b=b-\alpha\cdot\partial b\partial J(w,b)$

The formula is the **parameter update rule for gradient descent**, which adjusts the weights w and biases b to minimize a cost function. This process iteratively adjusts the line's slope and intercept to minimize the error.

2. Logistic Regression

In logistic regression, gradient descent minimizes the **Log Loss (Cross-Entropy Loss)** to optimize the decision boundary for binary classification. Since the output is probabilistic (between 0 and 1), the sigmoid function is applied. The process involves:

- Calculating the gradient of the log-loss with respect to the weights.
- Updating weights and biases iteratively to maximize the likelihood of the correct classification:

$w=w-\alpha\cdot\partial J(w)\partial w w=w-\alpha\cdot\partial w\partial J(w)$

This adjustment shifts the decision boundary to separate classes more effectively.

3. Support Vector Machines (SVMs)

For SVMs, gradient descent optimizes the **hinge loss**, which ensures a maximum-margin hyperplane. The algorithm:

- Calculates gradients for the hinge loss and the regularization term (if used, such as L2 regularization).
- Updates the weights to maximize the margin between classes while minimizing misclassification penalties with same formula provided above.

Gradient descent ensures the **optimal placement of the hyperplane to separate classes with the largest possible margin.**

Gradient Descent Python Implementation

Diving further into the concept, let's understand in depth, with practical implementation.

Import the necessary libraries

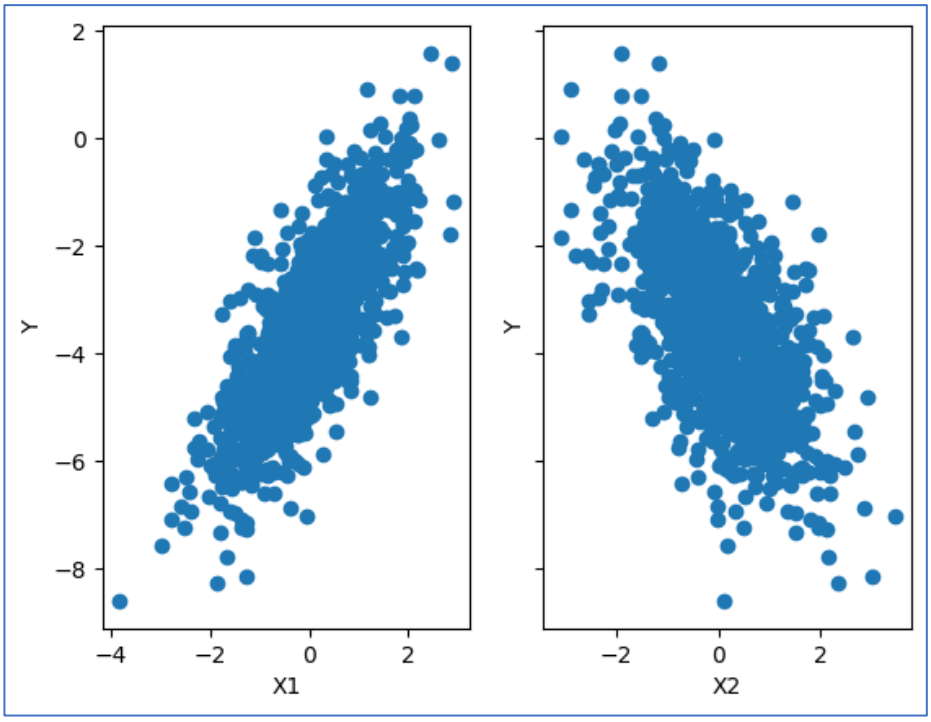
```
import torch
import torch.nn as nn
import matplotlib.pyplot as plt
```

Set the input and output data

```
# set random seed for reproducibility
torch.manual_seed(42)

# set number of samples
num_samples = 1000

# create random features with 2 dimensions
x = torch.randn(num_samples, 2)

# create random weights and bias for the linear regression model
true_weights = torch.tensor([1.3, -1])
true_bias    = torch.tensor([-3.5])

# Target variable
y = x @ true_weights.T + true_bias

# Plot the dataset
fig, ax = plt.subplots(1, 2, sharey=True)
```

```
ax[0].scatter(x[:,0],y)
ax[1].scatter(x[:,1],y)

ax[0].set_xlabel('X1')
ax[0].set_ylabel('Y')
ax[1].set_xlabel('X2')
ax[1].set_ylabel('Y')
plt.show()
```

Output:



X vs Y

Let's first try with a linear model:

$yp = xWT + b$ $yp = xWT + b$

```
# Define the model
class LinearRegression(nn.Module):
def __init__(self, input_size, output_size):
super(LinearRegression, self).__init__()
self.linear = nn.Linear(input_size, output_size)

def forward(self, x):
out = self.linear(x)
return out


# Define the input and output dimensions
input_size = x.shape[1]
output_size = 1

# Instantiate the model
model = LinearRegression(input_size, output_size)
```

Note:

The number of weight values will be equal to the input size of the model, And the input size in deep Learning is the number of independent input features i.e we are putting inside the model

In our case, input features are two so, the input size will also be two, and the corresponding weight value will also be two.

We can manually set the model parameter

```
# create a random weight & bias tensor
weight = torch.randn(1, input_size)
bias   = torch.rand(1)

# create a nn.Parameter object from the weight & bias tensor
weight_param = nn.Parameter(weight)
bias_param   = nn.Parameter(bias)

# assign the weight & bias parameter to the linear layer
model.linear.weight = weight_param
model.linear.bias   = bias_param

weight, bias = model.parameters()
print('Weight :',weight)
print('bias :',bias)
```

Output:

```
Weight                 :                 Parameter                 containing:
tensor([[-0.3239,                      0.5574]],           requires_grad=True)
bias                  :                  Parameter                 containing:
tensor([0.5710], requires_grad=True)
```

Prediction

```
y_p = model(x)
y_p[:5]
```

Output:

```
tensor([[                                                              0.7760],
        [-0.8944],
        [-0.3369],
        [-0.3095],
        [ 1.7338]], grad_fn=<SliceBackward0>)
```

Define the loss function

Loss function (J)=1n∑(actual−predicted)2Loss function (J)=$n1$∑(*actual*−*predicted*)2

Here we are calculating the Mean Squared Error by taking the square of the difference between the actual and the predicted value and then dividing it by its length (i.e n = the Total number of output or target values) which is the mean of squared errors.

```
# Define the loss function
def Mean_Squared_Error(prediction, actual):
error = (actual-prediction)**2
return error.mean()


# Find the total mean squared error
loss = Mean_Squared_Error(y_p, y)
loss
```

Output:

```
tensor(19.9126, grad_fn=<MeanBackward0>)
```

As we can see from the above right now the Mean Squared Error is 30559.4473. All the steps which are done till now are known as forward propagation.

Now our task is to find the optimal value of weight w and bias b which can fit our model well by giving very less or minimum error as possible. i.e

$$\text{minimize}\frac{1}{n}\sum(\text{actual}-\text{predicted})^2 \quad \text{minimize}\frac{1}{n}\sum(actual-predicted)^2$$

Now to update the weight and bias value and find the optimal value of weight and bias we will do backpropagation. Here the Gradient Descent comes into the role to find the optimal value weight and bias.

How the Gradient Descent Algorithm Works

For the sake of complexity, we can write our loss function for the single row as below

$$J(w,b)=\frac{1}{n}(yp-y)^2 \quad J(w,b)=\frac{1}{n}(yp-y)^2$$

In the above function x and y are our input data i.e constant. To find the optimal value of weight w and bias b. we partially differentiate with respect to w and b. This is also said that we will find the gradient of loss function J(w,b) with respect to w and b to find the optimal value of w and b.

Gradient of J(w,b) with respect to w

$$Jw'=\frac{\partial J(w,b)}{\partial w}=\frac{\partial}{\partial w}[\frac{1}{n}(yp-y)^2]=2(yp-y)n\frac{\partial}{\partial w}[(yp-y)\ ]=2(yp-y)n\frac{\partial}{\partial w}[((xWT+b)-y)\ ]=2(yp-y)n[\frac{\partial(xWT+b)}{\partial w}-\frac{\partial(y)}{\partial w}]=2(yp-y)n[x-0]=\frac{1}{n}(yp-y)[2x] \quad Jw'=\frac{\partial}{w\partial}J(w,b)=\frac{\partial}{w\partial}[\frac{n}{1}(yp-y)2]=n2(yp-y)\frac{\partial}{w\partial}[(yp-y)\ ]=n2(yp-y)\frac{\partial}{w\partial}[((xWT+b)-y)\ ]=n2(yp-y)[\frac{\partial}{w\partial}(xWT+b)-\frac{\partial}{w\partial}(y)]=n2(yp-y)[x-0]=\frac{n}{1}(yp-y)[2x]$$

i.e

$$Jw'=\frac{\partial J(w,b)}{\partial w}=J(w,b)[2x] \quad Jw'=\frac{\partial}{w\partial}J(w,b)=J(w,b)[2x]$$

Gradient of J(w,b) with respect to b

$$Jb'=\frac{\partial J(w,b)}{\partial b}=\frac{\partial}{\partial b}[\frac{1}{n}(yp-y)^2]=2(yp-y)n\frac{\partial}{\partial b}[(yp-y)\ ]=2(yp-y)n\frac{\partial}{\partial b}[((xWT+b)-y)\ ]=2(yp-y)n[\frac{\partial(xWT+b)}{\partial b}-\frac{\partial(y)}{\partial b}]=2(yp-y)n[1-0]=\frac{1}{n}(yp-y)[2] \quad Jb'=\frac{\partial}{b\partial}J(w,b)=\frac{\partial}{b\partial}[\frac{n}{1}(yp-y)2]=n2(yp-y)\frac{\partial}{b\partial}[(yp-y)\ ]=n2(yp-y)\frac{\partial}{b\partial}[((xWT+b)-y)\ ]=n2(yp-y)[\frac{\partial}{b\partial}(xWT+b)-\frac{\partial}{b\partial}(y)]=n2(yp-y)[1-0]=\frac{n}{1}(yp-y)[2]$$

i.e

$$Jb'=\frac{\partial J(w,b)}{\partial b}=J(w,b)[2] \quad Jb'=\frac{\partial}{b\partial}J(w,b)=J(w,b)[2]$$

Here we have considered the linear regression. So that here the parameters are weight and bias only. But in a fully connected neural network model there can be multiple layers and multiple parameters. but the concept will be the same everywhere. And the below-mentioned formula will work everywhere.

$$\text{Param}=\text{Param}-\gamma\nabla J \quad Param=Param-\gamma\nabla J$$

Here,

- γ γ = Learning rate
- J = Loss function
- ∇ ∇ = Gradient symbol denotes the derivative of loss function J
- Param = weight and bias There can be multiple weight and bias values depending upon the complexity of the model and features in the dataset

In our case:

$$w=w-\gamma\nabla J(w,,b) \quad b=b-\gamma\nabla J(w,,b) \quad w=w-\gamma\nabla J(w,,b) \quad b=b-\gamma\nabla J(w,,b)$$

In the current problem, two input features, So, the weight will be two.

Implementations of the Gradient Descent algorithm for the above model

Steps:

1. Find the gradient using loss.backward()
2. Get the parameter using model.linear.weight and model.linear.bias
3. Update the parameter using the above-defined equation.
4. Again assign the model parameter to our model

```
#              Find              the              gradient              using
loss.backward()
#                                 Learning                                Rate
learning_rate                             =                             0.001
#                               Model                               Parameter
w                             =                           model.linear.weight
b                              =                            model.linear.bias
```

```
#         Matually         Update         the         model         parameter
w         =         w         -         learning_rate         *         w.grad
b         =         b         -         learning_rate         *         b.grad
#   assign   the   weight   &   bias   parameter   to   the   linear   layer
model.linear.weight                    =                    nn.Parameter(w)
model.linear.bias   = nn.Parameter(b)
```

Now Repeat this process till 1000 epochs

```
# Number of epochs
num_epochs = 1000

# Learning Rate
learning_rate = 0.01

# SUBPLOT WEIGHT & BIAS VS LOSSES
fig, (ax1, ax2) = plt.subplots(1, 2, sharey=True)

for epoch in range(num_epochs):
# Forward pass
y_p = model(x)
loss = Mean_Squared_Error(y_p, y)

# Backproogation
# Find the fradient using
loss.backward()

# Learning Rate
learning_rate = 0.001

# Model Parameter
w = model.linear.weight
b = model.linear.bias

# Matually Update the model parameter
w = w - learning_rate * w.grad
b = b - learning_rate * b.grad

# assign the weight & bias parameter to the linear layer
model.linear.weight = nn.Parameter(w)
model.linear.bias   = nn.Parameter(b)

if (epoch+1) % 100 == 0:
ax1.plot(w.detach().numpy(),loss.item(),'r*-')
ax2.plot(b.detach().numpy(),loss.item(),'g+-')
print('Epoch [{}/{}], weight:{}, bias:{} Loss: {:.4f}'.format(
epoch+1,num_epochs,
w.detach().numpy(),
b.detach().numpy(),
loss.item()))

ax1.set_xlabel('weight')
ax2.set_xlabel('bias')
ax1.set_ylabel('Loss')
ax2.set_ylabel('Loss')
plt.show()
```

Output:

```
Epoch [100/1000], weight:[[-0.2618025   0.44433367]], bias:[-0.17722966] Loss:
14.1803
Epoch [200/1000], weight:[[-0.21144074  0.35393423]], bias:[-0.7892358] Loss:
10.3030
Epoch [300/1000], weight:[[-0.17063744  0.28172654]], bias:[-1.2897989] Loss:
7.7120
Epoch [400/1000], weight:[[-0.13759881   0.22408141]], bias:[-1.699218] Loss:
5.9806
Epoch [500/1000], weight:[[-0.11086453  0.17808875]], bias:[-2.0340943] Loss:
4.8235
Epoch [600/1000], weight:[[-0.08924612  0.14141548]], bias:[-2.3080034] Loss:
4.0502
Epoch [700/1000], weight:[[-0.0717768   0.11219224]], bias:[-2.5320508] Loss:
3.5333
Epoch [800/1000], weight:[[-0.0576706   0.08892148]], bias:[-2.7153134] Loss:
3.1878
Epoch [900/1000], weight:[[-0.04628877  0.07040432]], bias:[-2.8652208] Loss:
2.9569
Epoch [1000/1000], weight:[[-0.0371125   0.05568104]], bias:[-2.9878428] Loss:
2.8026
```

ML - Stochastic Gradient Descent (SGD)

Stochastic Gradient Descent (SGD) is an optimization algorithm in machine learning, particularly when dealing with large datasets. It is a variant of the traditional gradient descent algorithm but offers several advantages in terms of efficiency and scalability, making it the go-to method for many deep-learning tasks.

To understand SGD, it's essential to first comprehend the concept of gradient descent.

What is Gradient Descent?

Gradient descent is an iterative optimization algorithm used to minimize a loss function, which represents how far the model's predictions are from the actual values. The main goal is to adjust the parameters of a model (weights, biases, etc.) so that the error is minimized.

The update rule for the traditional gradient descent algorithm is:

$\theta = \theta - \eta \nabla\theta J(\theta) \theta = \theta - \eta \nabla\theta J(\theta)$

In traditional gradient descent, the gradients are computed based on the entire dataset, which can be computationally expensive for large datasets.

Need for Stochastic Gradient Descent

For large datasets, computing the gradient using all data points can be slow and memory-intensive. This is where SGD comes into play. Instead of using the full dataset to compute the gradient at each step, SGD uses only one random data point (or a small batch of data points) at each iteration. This makes the computation much faster.

Path followed by batch gradient descent vs. path followed by SGD:

Path followed by batch gradient descent vs. path followed by SGD

Working of Stochastic Gradient Descent

In Stochastic Gradient Descent, the gradient is calculated for each training example (or a small subset of training examples) rather than the entire dataset.

The update rule becomes:
$\theta=\theta-\eta \nabla\theta J(\theta;xi,yi)\theta=\theta-\eta \nabla\theta J(\theta;xi,yi)$
Where:

- xixi and yiyi represent the features and target of the i-th training example.
- The gradient $\nabla\theta J(\theta;xi,yi)\nabla\theta J(\theta;xi,yi)$ is now calculated for a single data point or a small batch.

The key difference from traditional gradient descent is that, in SGD, the parameter updates are made based on a single data point, not the entire dataset. The random selection of data points introduces stochasticity, which can be both an advantage and a challenge.

Implementing Stochastic Gradient Descent from Scratch

1. Generating the Data

In this step, we generate synthetic data for the linear regression problem. The data consists of feature X and the target y, where the relationship is linear, i.e., y = 4 + 3 * X + noise.

- X is a random array of 100 samples between 0 and 2.
- y is the target, calculated using a little random noise to make it more realistic.

```
import numpy as np

np.random.seed(42)
X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)
```

For a **linear regression** with one feature, the model is described by the equation:
$y=\theta0+(\theta1)\cdot Xy=\theta0+(\theta1)\cdot X$
Where:

- $\theta0\theta0$ is the **intercept** (the bias term),
- $\theta1\theta1$ is the **slope** or **coefficient** associated with the input feature XX.

2. Defining the SGD Function

Here we define the core function for Stochastic Gradient Descent (SGD). The function takes the input data X and y. It initializes the model parameters, performs stochastic updates for a specified number of epochs and records the cost at each step.

- theta ($\theta\theta$) is the parameter vector (intercept and slope) initialized randomly.
- **`X_bias`** is the augmented $XX$with a column of ones added for the bias term (intercept).

In each epoch, the data is shuffled and for each mini-batch (or single sample), the gradient is calculated and the parameters are updated. The cost is calculated as the mean squared error and the history of the cost is recorded to monitor convergence.

```
def sgd(X, y, learning_rate=0.1, epochs=1000, batch_size=1):
m = len(X)
theta = np.random.randn(2, 1)

X_bias = np.c_[np.ones((m, 1)), X]

cost_history = []

for epoch in range(epochs):
indices = np.random.permutation(m)
X_shuffled = X_bias[indices]
y_shuffled = y[indices]

for i in range(0, m, batch_size):
```

```
X_batch = X_shuffled[i:i+batch_size]
y_batch = y_shuffled[i:i+batch_size]

gradients = 2 / batch_size * X_batch.T.dot(X_batch.dot(theta) - y_batch)
theta -= learning_rate * gradients

predictions = X_bias.dot(theta)
cost = np.mean((predictions - y) ** 2)
cost_history.append(cost)

if epoch % 100 == 0:
print(f"Epoch {epoch}, Cost: {cost}")

return theta, cost_history
```

Step 3: Train the Model Using SGD

In this step, we call the **sgd()** function to train the model. We specify the learning rate, number of epochs and batch size for SGD.

```
theta_final, cost_history = sgd(X, y, learning_rate=0.1, epochs=1000, batch_size=1)
```

Output:

```
Epoch 0, Cost: 1.581821686856939
Epoch 100, Cost: 1.5664692700155733
Epoch 200, Cost: 1.4445422391173144
Epoch 300, Cost: 1.7037674963102662
Epoch 400, Cost: 0.9101999515899212
Epoch 500, Cost: 0.8184497904316664
Epoch 600, Cost: 0.8352333304446237
Epoch 700, Cost: 0.8542729530055074
Epoch 800, Cost: 1.0508310318687628
Epoch 900, Cost: 0.8261971232182218
```

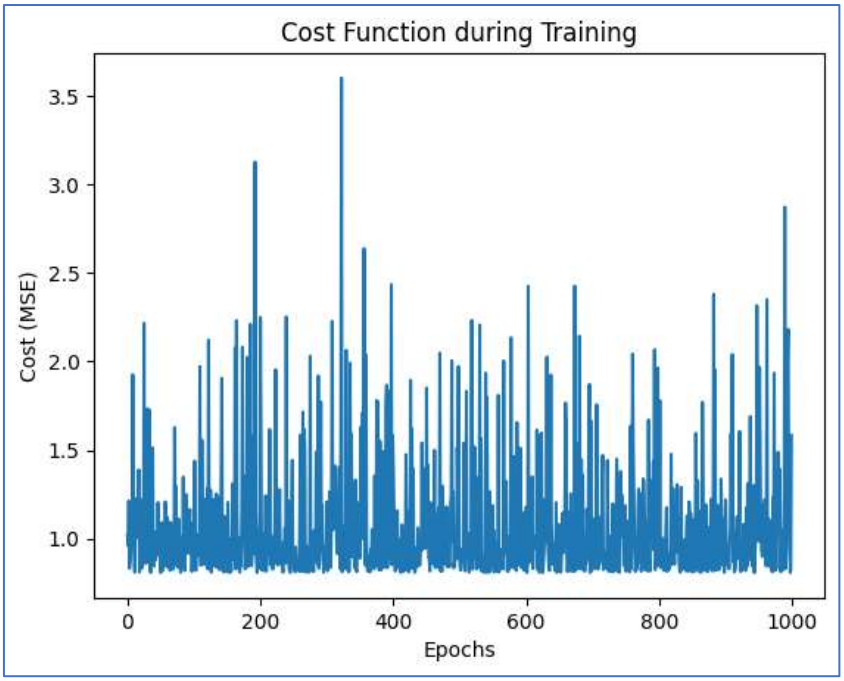
Train the Model Using SGD

4. Visualizing the Cost Function

After training, we visualize how the cost function evolves over epochs. This helps us understand if the algorithm is converging properly.

```
import matplotlib.pyplot as plt

plt.plot(cost_history)
plt.xlabel('Epochs')
plt.ylabel('Cost (MSE)')
plt.title('Cost Function during Training')
plt.show()
```

Output:



Visualize the Cost Function

5. Plotting the Data and Regression Line

We will visualize the data points and the fitted regression line after training. We plot the data points as blue dots and the predicted line (from the final $\theta\theta$) as a red line.

```
plt.scatter(X, y, color='blue', label='Data points')
plt.plot(X, np.c_[np.ones((X.shape[0], 1)), X].dot(theta_final), color='red', label='SGD fit line')
plt.xlabel('X')
plt.ylabel('y')
plt.title('Linear Regression using Stochastic Gradient Descent')
plt.legend()
plt.show()
```

Output:

Plot the Data and Regression Line

6. Printing the Final Model Parameters

After training, we print the final parameters of the model, which include the slope and intercept. These values are the result of optimizing the model using SGD.

```
print(f"Final parameters: {theta_final}")
```

Output:

Final parameters: [[4.35097872] [3.45754277]]

The final parameters returned by the model are:

$\theta 0=4.35, \theta 1=3.45 \theta 0=4.35, \theta 1=3.45$

Then the fitted linear regression model will be:

$y=4.35+(3.45) \cdot X y=4.35+(3.45) \cdot X$

This means:

- When X=0, y=4.3(the intercept or bias term).
- For each unit increase in $X, y X, y$ will increase by 3.4 units (the slope or coefficient).

Advantages of Stochastic Gradient Descent

1. **Efficiency**: Because it uses only one or a few data points to calculate the gradient, SGD can be much faster, especially for large datasets. Each step requires fewer computations, leading to quicker convergence.
2. **Memory Efficiency**: Since it does not require storing the entire dataset in memory for each iteration, SGD can handle much larger datasets than traditional gradient descent.

3. **Escaping Local Minima**: The noisy updates in SGD, caused by the stochastic nature of the algorithm, can help the model escape local minima or saddle points, potentially leading to better solutions in non-convex optimization problems (common in deep learning).
4. **Online Learning**: SGD is well-suited for online learning, where the model is trained incrementally as new data comes in, rather than on a static dataset.

Challenges of Stochastic Gradient Descent

1. **Noisy Convergence**: Since the gradient is estimated based on a single data point (or a small batch), the updates can be noisy, causing the cost function to fluctuate rather than steadily decrease. This makes convergence slower and more erratic than in batch gradient descent.
2. **Learning Rate Tuning**: SGD is highly sensitive to the choice of learning rate. A learning rate that is too large may cause the algorithm to diverge, while one that is too small can slow down convergence. Adaptive methods like Adam and RMSprop address this by adjusting the learning rate dynamically during training.
3. **Long Training Times**: While each individual update is fast, the convergence might take a longer time overall since the steps are more erratic compared to batch gradient descent.

Variants of Stochastic Gradient Descent

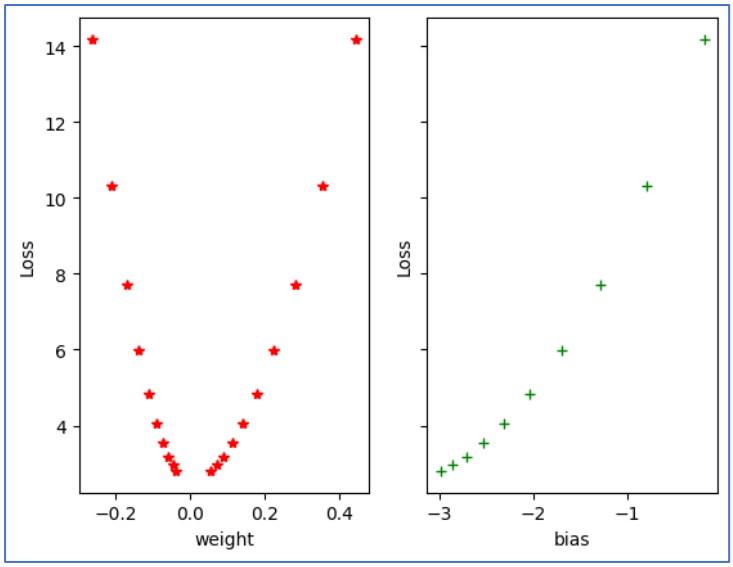
While traditional SGD is a method, there are several improvements and variants designed to improve convergence and stability:

- **Mini-batch SGD**: Instead of using a single data point, mini-batch SGD uses a small batch of data points to calculate the gradient. This strikes a balance between the efficiency of SGD and the stability of batch gradient descent. It reduces the noise in the updates while maintaining the computational efficiency.
- **Momentum**: Momentum helps accelerate SGD by adding a fraction of the previous update to the current one. This allows the algorithm to keep moving in the same direction and can help overcome oscillations in the cost function.
- **Adaptive Methods (example: Adam, RMSprop)**: These methods dynamically adjust the learning rate for each parameter. Adam, for example, uses both the average of the gradients (first moment) and the average of the squared gradients (second moment) to compute an adaptive learning rate, improving convergence and stability.

Applications of Stochastic Gradient Descent

SGD and its variants are widely used across various domains of machine learning:

- **Deep Learning**: In training deep neural networks, SGD is the default optimizer due to its efficiency with large datasets and its ability to work with large models. Deep learning frameworks like **TensorFlow** and **PyTorch** typically use variants like **Adam** or **RMSprop**, which are based on SGD.
- **Natural Language Processing (NLP)**: Models like Word2Vec and transformers are trained using SGD variants to optimize large models on vast text corpora.
- **Computer Vision**: For tasks such as image classification, object detection and segmentation, SGD has been fundamental in training convolutional neural networks (CNNs).
- **Reinforcement Learning**: SGD is also used to optimize the parameters of models used in reinforcement learning, such as deep **Q-networks (DQNs)** and policy gradient methods.



From the above graph and data, we can observe the Losses are decreasing as per the weight and bias variations.

Now we have found the optimal weight and bias values. Print the optimal weight and bias and

```
w = model.linear.weight
b = model.linear.bias

print('weight(W) = {} \n  bias(b) = {}'.format(
w.abs(),
b.abs()))
```

Output:

```
weight(W)     =     tensor([[0.0371,     0.0557]],     grad_fn=<AbsBackward0>)
  bias(b) = tensor([2.9878], grad_fn=<AbsBackward0>)
```

Prediction

```
pred =  x @ w.T + b
pred[:5]
```

Output:

```
tensor([[-2.9765],
        [-3.1385],
        [-3.0818],
        [-3.0756],
        [-2.8681]], grad_fn=<SliceBackward0>)
```

Gradient Descent Learning Rate

The learning rate is a critical hyperparameter in the context of gradient descent, influencing the size of steps taken during the optimization process to update the model parameters. Choosing an appropriate learning rate is crucial for efficient and effective model training.

When the learning rate is **too small**, the optimization process progresses very slowly. The model makes tiny updates to its parameters in each iteration, leading to sluggish convergence and potentially getting stuck in local minima.

On the other hand, an **excessively large learning rate** can cause the optimization algorithm to overshoot the optimal parameter values, leading to divergence or oscillations that hinder convergence.

Achieving the right balance is essential. A small learning rate might result in vanishing gradients and slow convergence, while a large learning rate may lead to overshooting and instability.

Vanishing and Exploding Gradients

Vanishing and exploding gradients are common problems that can occur during the training of deep neural networks. These problems can significantly slow down the training process or even prevent the network from learning altogether.

The vanishing gradient problem occurs when gradients become too small during backpropagation. The weights of the network are not considerably changed as a result, and the network is unable to discover the underlying patterns in the data. Many-layered deep neural networks are especially prone to this issue. The gradient values fall exponentially as they move backward through the layers, making it challenging to efficiently update the weights in the earlier layers.

The exploding gradient problem, on the other hand, occurs when gradients become too large during backpropagation. When this happens, the weights are updated by a large amount, which can cause the network to diverge or oscillate, making it difficult to converge to a good solution.

To address these problems the following technique can be used:

- **Weights Regularzations:** The initialization of weights can be adjusted to ensure that they are in an appropriate range. Using a different activation function, such as the Rectified Linear Unit (ReLU), can also help to mitigate the vanishing gradient problem.
- **Gradient clipping:** It involves limiting the maximum and minimum values of the gradient during backpropagation. This can prevent the gradients from becoming too large or too small and can help to stabilize the training process.
- **Batch normalization:** It can also help to address these problems by normalizing the input to each layer, which can prevent the activation function from saturating and help to reduce the vanishing and exploding gradient problems.

Different Variants of Gradient Descent

There are several variants of gradient descent that differ in the way the step size or learning rate is chosen and the way the updates are made. Here are some popular variants:

Batch Gradient Descent

In batch gradient descent, To update the model parameter values like weight and bias, the entire training dataset is used to compute the gradient and update the parameters at each iteration. This can be slow for large datasets but may lead to a more accurate model. It is effective for convex or relatively smooth error manifolds because it moves directly toward an optimal solution by taking a large step in the direction of the negative gradient of the cost function. However, it can be slow for large datasets because it computes the gradient and updates the parameters using the entire training dataset at each iteration. This can result in longer training times and higher computational costs.

Stochastic Gradient Descent (SGD)

In SGD, only one training example is used to compute the gradient and update the parameters at each iteration. This can be faster than batch gradient descent but may lead to more noise in the updates.

Mini-batch Gradient Descent

In Mini-batch gradient descent a small batch of training examples is used to compute the gradient and update the parameters at each iteration. This can be a good compromise between batch gradient descent and Stochastic Gradient Descent, as it can be faster than batch gradient descent and less noisy than Stochastic Gradient Descent.

Momentum-based Gradient Descent

In momentum-based gradient descent, Momentum is a variant of gradient descent that incorporates information from the previous weight updates to help the algorithm converge more quickly to the optimal solution. Momentum adds a term to the weight update that is proportional

to the running average of the past gradients, allowing the algorithm to move more quickly in the direction of the optimal solution. The updates to the parameters are based on the current gradient and the previous updates. This can help prevent the optimization process from getting stuck in local minima and reach the global minimum faster.

Nesterov Accelerated Gradient (NAG)

Nesterov Accelerated Gradient (NAG) is an extension of Momentum Gradient Descent. It evaluates the gradient at a hypothetical position ahead of the current position based on the current momentum vector, instead of evaluating the gradient at the current position. This can result in faster convergence and better performance.

Adagrad

In Adagrad, the learning rate is adaptively adjusted for each parameter based on the historical gradient information. This allows for larger updates for infrequent parameters and smaller updates for frequent parameters.

RMSprop

In RMSprop the learning rate is adaptively adjusted for each parameter based on the moving average of the squared gradient. This helps the algorithm to converge faster in the presence of noisy gradients.

Adam

Adam stands for adaptive moment estimation, it combines the benefits of Momentum-based Gradient Descent, Adagrad, and RMSprop the learning rate is adaptively adjusted for each parameter based on the moving average of the gradient and the squared gradient, which allows for faster convergence and better performance on non-convex optimization problems. It keeps track of two exponentially decaying averages the first-moment estimate, which is the exponentially decaying average of past gradients, and the second-moment estimate, which is the exponentially decaying average of past squared gradients. The first-moment estimate is used to calculate the momentum, and the second-moment estimate is used to scale the learning rate for each parameter. This is one of the most popular optimization algorithms for deep learning.

Advantages & Disadvantages of gradient descent

Advantages of Gradient Descent

1. **Widely used:** Gradient descent and its variants are widely used in machine learning and optimization problems because they are effective and easy to implement.
2. **Convergence**: Gradient descent and its variants can converge to a global minimum or a good local minimum of the cost function, depending on the problem and the variant used.
3. **Scalability**: Many variants of gradient descent can be parallelized and are scalable to large datasets and high-dimensional models.
4. **Flexibility**: Different variants of gradient descent offer a range of trade-offs between accuracy and speed, and can be adjusted to optimize the performance of a specific problem.

Disadvantages of gradient descent:

1. **Choice of learning rate:** The choice of learning rate is crucial for the convergence of gradient descent and its variants. Choosing a learning rate that is too large can lead to oscillations or overshooting while choosing a learning rate that is too small can lead to slow convergence or getting stuck in local minima.
2. **Sensitivity to initialization:** Gradient descent and its variants can be sensitive to the initialization of the model's parameters, which can affect the convergence and the quality of the solution.

3. **Time-consuming:** Gradient descent and its variants can be time-consuming, especially when dealing with large datasets and high-dimensional models. The convergence speed can also vary depending on the variant used and the specific problem.
4. **Local optima:** Gradient descent and its variants can converge to a local minimum instead of the global minimum of the cost function, especially in non-convex problems. This can affect the quality of the solution, and techniques like random initialization and multiple restarts may be used to mitigate this issue.

What is Batch Normalization in CNN

Batch normalization is a technique used to improve the training of deep neural networks by stabilizing the learning process. It addresses the issue of internal covariate shift where the distribution of each layer's inputs changes during training as the parameters of the previous layers change..

Batch Normalization in CNN addresses several challenges encountered during training.

1. **Addressing Internal Covariate Shift:** Internal covariate shift occurs when the distribution of network activations changes as parameters are updated during training. It addresses this by normalizing the activations in each layer. This stabilizes training and speeds up convergence.
2. **Improving Gradient Flow:** It contributes to stabilizing the gradient flow during backpropagation by reducing reliance on gradients on parameter scales. As a result, training more stable enabling effective training of deeper networks without facing issues like vanishing or exploding gradients.
3. **Regularization Effect:** During training it introduces noise to the network activations. This noise helps to handle overfitting by adding randomness to the system.

How Does Batch Normalization Work in CNN?

Batch normalization works in convolutional neural networks (CNNs) by normalizing the activations of each layer across mini-batch during training. The working is discussed below:

1. Normalization within Mini-Batch

In a CNN, each layer receives inputs from multiple channels (feature maps) and processes them through convolutional filters. Batch Normalization operates on each feature map separately, normalizing the activations across the mini-batch.

During training, batch normalization (BN) regularizes the activations of each layer by subtracting the mean and dividing by the standard deviation of each mini-batch.

- Mean Calculation: $\mu B=1m\sum i=1mxi\mu B=m1\sum i=1mxi$
- Variance Calculation: $\sigma B2=1m\sum i=1m(xi-\mu B)2\sigma B2=m1\sum i=1m(xi-\mu B)2$
- Normalization: $xi\hat{}=xi-\mu B\sigma B2+\epsilon xi=\sigma B2+\epsilon xi-\mu B$

2. Scaling and Shifting

After normalization, it adjusts the normalized activations using learned scaling and shifting parameters. These parameters enable the network to instantaneously scale and shift the activations thereby maintaining the network's ability to represent complex patterns in the data.

- Scaling: $\gamma xi\hat{}\gamma xi$
- Shifting: $zi=yi+\beta zi=yi+\beta$

3. Learnable Parameters

The parameters $\gamma\gamma$ and $\beta\beta$ are learned during training through backpropagation. This allows the network to adjust the normalization and ensure that the activations are in the appropriate range for learning.

4. Applying Batch Normalization

It is typically applied after the convolutional and activation layers in a CNN before passing the outputs to the next layer. It can also be applied before or after the activation function, depending on the network architecture.

5. Training and Inference

During training, Batch Normalization calculates the mean and variance of each mini-batch. During testing, it uses the averaged mean and variance that are calculated during training to normalize the activations. This ensures consistent normalization between training and testing.

Applying Batch Normalization in CNN model using TensorFlow

For applying batch normalization layers after the convolutional layers and before the activation functions, we use tensorflow's **'tf.keras.layers.BatchNormalization()'**.

1. Importing Required Libraries

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, BatchNormalization
```

2. Creating Sequential Model

First Convolutional Block

- **Conv2D(32):** Extracts low-level features (edges, textures).
- **BatchNormalization():** Stabilizes and speeds up training.
- **MaxPooling2D():** Reduces spatial size.

Second Convolutional Block

- **Conv2D(64):** Learns deeper patterns.
- **BatchNormalization():** Normalizes activations.
- **MaxPooling2D():** Further reduces size.

Dense Layers(Classifier)

- **`Flatten()`:** Converts 3D feature map to 1D.
- **`Dense(64)`:** Learns high-level combinations.
- **`Dense(10, softmax)`:** Outputs probabilities for 10 classes.

```
model = Sequential([
#First convolutional block
Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)),
BatchNormalization(),
MaxPooling2D((2, 2)),
#Second convolutional block
Conv2D(64, (3, 3), activation='relu'),
BatchNormalization(),
MaxPooling2D((2, 2)),
#Connecting layers
Flatten(),
Dense(64, activation='relu'),
Dense(10, activation='softmax')
])
```

Applying Batch Normalization in 1D CNN model using PyTorch

In PyTorch, we can easily apply batch normalization in a CNN model. For applying BN in 1D Convolutional Neural Network model, we use '**nn.BatchNorm1d()**'.

1. Importing Libraries

```
import torch
import torch.nn as nn
```

2. Defining the Model

In this step, we structure our model:

- **Conv1d(3, 16)**: First convolution layer that transforms 3 input channels to 16 feature maps using 3-sized filters.
- **BatchNorm1d(16)**: Normalizes the 16 output channels to improve training stability.
- **Conv1d(16, 32)**: Second convolutional layer, increasing feature channels to 32.
- **BatchNorm1d(32)**: Normalizes the output from the second conv layer.
- **Linear(32 * 28, 10)**: Fully connected layer that maps the flattened feature map to 10 output classes.

```
class CNN1D(nn.Module):
def __init__(self):
super(CNN1D, self).__init__()
self.conv1 = nn.Conv1d(3, 16, kernel_size=3, stride=1, padding=1)
self.bn1 = nn.BatchNorm1d(16)
self.conv2 = nn.Conv1d(16, 32, kernel_size=3, stride=1, padding=1)
self.bn2 = nn.BatchNorm1d(32)
self.fc = nn.Linear(32 * 28, 10)
```

3. Forward Pass

Prediction step and input flow:

- **self.conv1 -> bn1 -> ReLU:** Applies the first convolution and activates.
- **self.conv2 -> bn2 -> ReLU:** Applies the second convolution and activates.
- **view(-1, 32 * 28):** Flattens the 3D tensor into 2D for the dense layer.
- **self.fc:** Final layer that outputs a vector of size 10 (class scores).

```
def forward(self, x):
x = torch.relu(self.bn1(self.conv1(x)))
x = torch.relu(self.bn2(self.conv2(x)))
x = x.view(-1, 32 * 28)
x = self.fc(x)
return x
```

4. Initialize Model

```
model = CNN1D()
```

This code defines and creates a simple **1D Convolutional Neural Network (`CNN1D`)** in PyTorch for classification tasks.

Applying Batch Normalization in 2D CNN model using PyTorch

For applying Batch Normalization in 2D Convolutional Neural Network model, we use '**nn.BatchNorm2d()**'.

1. Importing Libraries

```
import torch
import torch.nn as nn
```

2. Structuring Model

In this step, we define the model:

- **Conv2d(3, 16, 3, 1, 1)**: Applies 16 filters to a 3-channel image using 3×3 kernels. Padding keeps the image size unchanged.
- **BatchNorm2d(16)**: Normalizes the 16 output feature maps from conv1.
- **Conv2d(16, 32, 3, 1, 1)**: Applies 32 filters, again preserving spatial dimensions.

- **BatchNorm2d(32)**: Normalizes the output of conv2.
- **Linear(32*28*28, 10)**: Fully connected layer that flattens the feature map and outputs scores for 10 classes.

```
class CNN(nn.Module):
def __init__(self):
super(CNN, self).__init__()
self.conv1 = nn.Conv2d(3, 16, kernel_size=3, stride=1, padding=1)
self.bn1 = nn.BatchNorm2d(16)
self.conv2 = nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1)
self.bn2 = nn.BatchNorm2d(32)
self.fc = nn.Linear(32 * 28 * 28, 10)
```

3. Forward Pass

Prediction step and input flow:

- **Conv1 -> BN -> ReLU**: First feature extraction block.
- **Conv2 -> BN -> ReLU**: Second feature extraction block.
- **`view(-1, 32*28*28)`**: Flattens the 3D output to 1D for the dense layer.
- **`fc`**: Maps the extracted features to 10 output classes.

```
def forward(self, x):
x = torch.relu(self.bn1(self.conv1(x)))
x = torch.relu(self.bn2(self.conv2(x)))
x = x.view(-1, 32 * 28 * 28)
x = self.fc(x)
return x
```

4. Model Initialization

```
model = CNN()
```

This code defines a **2D Convolutional Neural Network (`CNN`)** in PyTorch for **image classification** into **10 classes**.

In conclusion, batch normalization stands as a technique used in enhancing the training and performance of convolutional neural networks (CNNs).

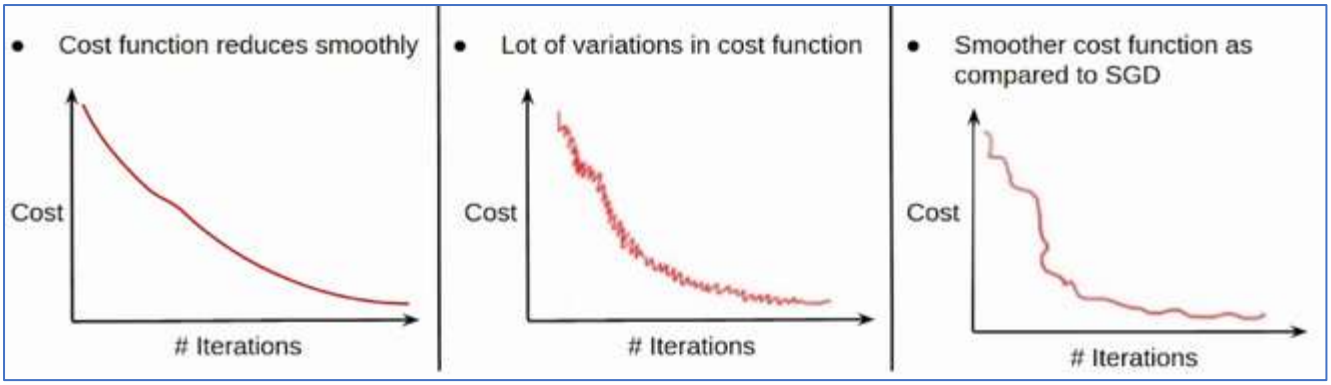
ML | Mini-Batch Gradient Descent with Python

Gradient Descent is an optimization algorithm in machine learning used to determine the optimal parameters such as weights and bias for models. The idea is to minimize the model's error by iteratively updating the parameters in the direction of the steepest descent as determined by the gradient of the loss function.

Depending on how much data is used to compute the gradient during each update, gradient descent comes in three main variants:

- **Batch Gradient Descent**
- **Stochastic Gradient Descent (SGD)**
- **Mini-Batch Gradient Descent**

Each variant has its own strengths and trade-offs in terms of speed, stability and convergence behavior.



Convergence in BGD, SGD & MBGD

Working of Mini-Batch Gradient Descent

Mini-batch gradient descent is a optimization method that updates model parameters using small subsets of the training data called mini-batches. This technique offers a middle path between the high variance of stochastic gradient descent and the high computational cost of batch gradient descent. They are used to perform each update, making training faster and more memory-efficient. It also helps stabilize convergence and introduces beneficial randomness during learning.

It is often preferred in modern machine learning applications because it combines the benefits of both batch and stochastic approaches.

Key advantages of mini-batch gradient descent:

- **Computational Efficiency:** Supports parallelism and vectorized operations on GPUs or TPUs.
- **Faster Convergence:** Provides more frequent updates than full-batch which improves speed.
- **Noise Reduction:** Less noisy than stochastic updates which leads to smoother convergence.
- **Better Generalization:** Introduces slight randomness to help escape local minima.
- **Memory Efficiency:** Doesn't require loading the entire dataset into memory.

Algorithm:

Let:

- $\theta\theta$ = model parameters
- `max_iters` = number of epochs
- $\eta\eta$ = learning rate

For itr=1,2,3,...,max_iters:

- Shuffle the training data. It is optional but often done for better randomness in mini-batch selection.
- Split the dataset into mini-batches of size bb.

For each mini-batch (Xmini$Xmini$, ymini$ymini$):

1. Forward Pass on the batch X_mini:

Make predictions on the mini-batch

$y^=f(Xmini, \theta)y^=f(Xmini, \theta)$

Compute error in predictions J(θ)$J(\theta)$ with the current values of the parameters

$J(\theta)=L(y^,ymini)J(\theta)=L(y^,ymini)$

2. Backward Pass:

Compute gradient:

$\nabla\theta J(\theta)=\partial J(\theta)\partial\theta\nabla\theta J(\theta)=\partial\theta\partial J(\theta)$

3. Update parameters:

Gradient descent rule:
$\theta=\theta-\eta\nabla\theta J(\theta)\theta=\theta-\eta\nabla\theta J(\theta)$

Python Implementation

Here we will use Mini-Batch Gradient Descent for Linear Regression.

1. Importing Libraries

We begin by importing libraries like `Numpy and Matplotlib.pyplot`

```
import numpy as np
import matplotlib.pyplot as plt
```

2. Generating Synthetic 2D Data

Here, we generate 8000 two-dimensional data points sampled from a multivariate normal distribution:

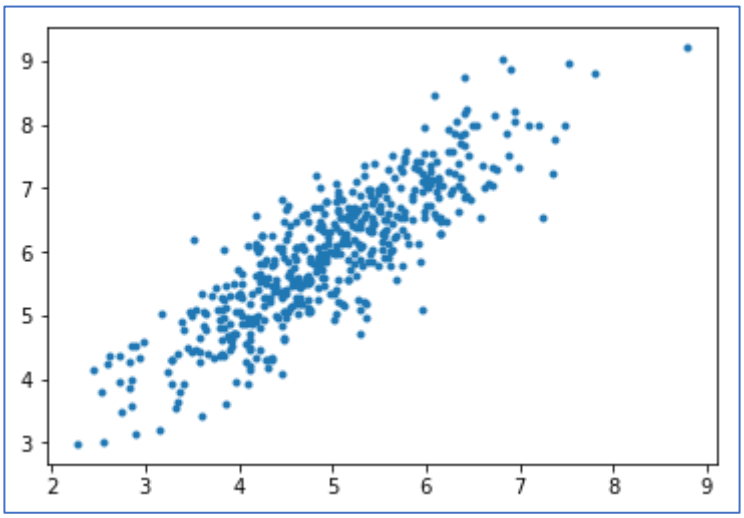
- The data is centered at the point (5.0, 6.0).
- The `cov` matrix defines the variance and correlation between the features. A value of `0.95` indicates a strong positive correlation between the two features.

```
mean = np.array([5.0, 6.0])
cov = np.array([[1.0, 0.95], [0.95, 1.2]])
data = np.random.multivariate_normal(mean, cov, 8000)
```

3. Visualizing Generated Data

```
plt.scatter(data[:500, 0], data[:500, 1], marker='.')
plt.title("Scatter Plot of First 500 Samples")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.grid(True)
plt.show()
```

Output:



4. Splitting Data

We split the data into training and testing sets:

- Original data shape: `(8000, 2)`
- New shape after adding bias: `(8000, 3)`
- 90% of the data is used for training and 10% for testing.

```
data = np.hstack((np.ones((data.shape[0], 1)), data))  # shape: (8000, 3)

split_factor = 0.90
split = int(split_factor * data.shape[0])
```

```
X_train = data[:split, :-1]
y_train = data[:split, -1].reshape((-1, 1))
X_test = data[split:, :-1]
y_test = data[split:, -1].reshape((-1, 1))
```

5. Displaying Datasets

```
print("Number of examples in training set = %d" % X_train.shape[0])
print("Number of examples in testing set = %d" % X_test.shape[0])
```

Output:

```
Number of examples in training set = 7200
Number of examples in testing set = 800
```

results

6. Defining Core Functions of Linear Regression

- **Hypothesis(X, theta)**: Computes the predicted output using the linear model $h(X)=X\cdot\theta$
- **Gradient(X, y, theta):** Calculates the gradient of the cost function which is used to update model parameters during training.
- **Cost(X, y, theta):** Computes the Mean Squared Error (MSE).

```
# Hypothesis function
def hypothesis(X, theta):
return np.dot(X, theta)

# Gradient of the cost function
def gradient(X, y, theta):
h = hypothesis(X, theta)
grad = np.dot(X.T, (h - y))
return grad

# Mean squared error cost
def cost(X, y, theta):
h = hypothesis(X, theta)
J = np.dot((h - y).T, (h - y)) / 2
return J[0]
```

7. Creating Mini-Batches for Training

This function divides the dataset into **random mini-batches** used during training:

- Combines the feature matrix X and target vector y, then shuffles the data to introduce randomness.
- Splits the shuffled data into batches of size batch_size.
- Each mini-batch is a tuple (X_mini, Y_mini) used for one update step in mini-batch gradient descent.
- Also handles the case where data isn't evenly divisible by the batch size by including the leftover samples in an extra batch.

```
# Create mini-batches from the dataset
def create_mini_batches(X, y, batch_size):
mini_batches = []
data = np.hstack((X, y))
np.random.shuffle(data)
n_minibatches = data.shape[0] // batch_size
for i in range(n_minibatches + 1):
mini_batch = data[i * batch_size:(i + 1) * batch_size, :]
```

X_mini = mini_batch[:, :-1]
Y_mini = mini_batch[:, -1].reshape((-1, 1))
mini_batches.append((X_mini, Y_mini))
if data.shape[0] % batch_size != 0:
mini_batch = data[i * batch_size:]
X_mini = mini_batch[:, :-1]
Y_mini = mini_batch[:, -1].reshape((-1, 1))
mini_batches.append((X_mini, Y_mini))
return mini_batches

8. Mini-Batch Gradient Descent Function

This function performs mini-batch gradient descent to train the linear regression model:

- **Initialization**: Weights `theta` are initialized to zeros and an empty list `error_list` tracks the cost over time.
- **Training Loop**: For a fixed number of iterations (`max_iters`), the dataset is divided into mini-batches.
- **Each mini-batch:** computes the gradient, updates `theta` to reduce cost and records the current error for tracking training progress.

# Mini-batch gradient descent
def gradientDescent(X, y, learning_rate=0.001, batch_size=32):
theta = np.zeros((X.shape[1], 1))
error_list = []
max_iters = 3

for itr in range(max_iters):
mini_batches = create_mini_batches(X, y, batch_size)
for X_mini, y_mini in mini_batches:
theta = theta - learning_rate * gradient(X_mini, y_mini, theta)
error_list.append(cost(X_mini, y_mini, theta))

return theta, error_list

9. Training and Visualization

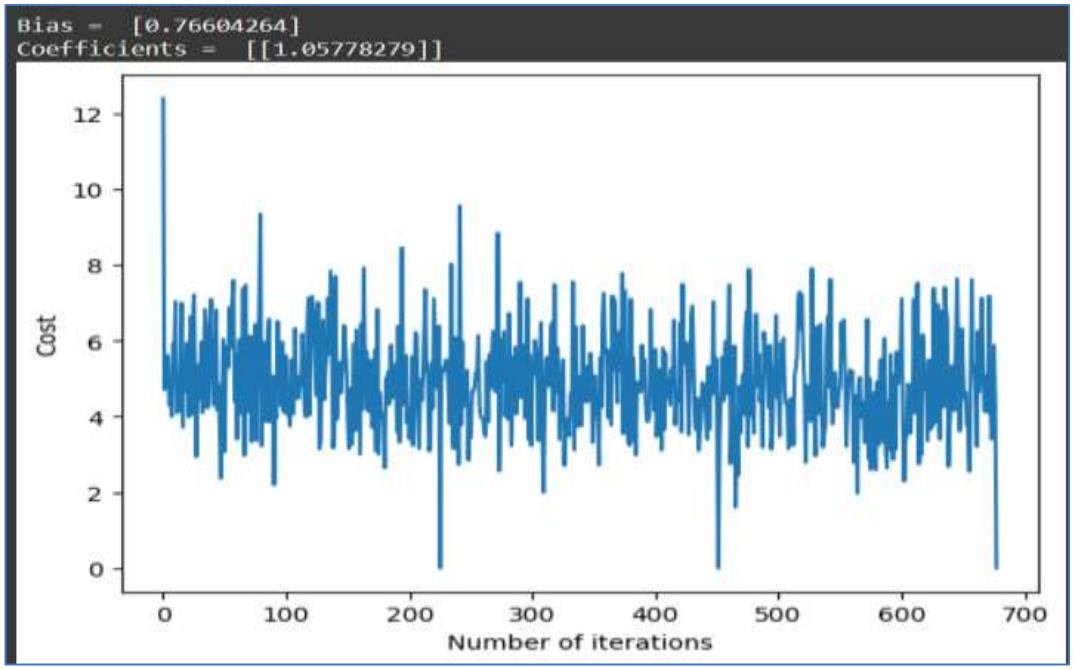
The model is trained using `gradientDescent()` on the training data. After training:

- theta[0] is the bias term (intercept).
- theta[1:] contains the feature weights (coefficients).
- The plot shows how the cost decreases as the model learns, showing convergence of the algorithm.

This provides a visual and quantitative insight into how well the mini-batch gradient descent is optimizing the regression model.

```
theta, error_list = gradientDescent(X_train, y_train)
print("Bias = ", theta[0])
print("Coefficients = ", theta[1:])

# visualising gradient descent
plt.plot(error_list)
plt.xlabel("Number of iterations")
plt.ylabel("Cost")
plt.show()
```

Output:



Mini-Batch over Regression model

10. Final Prediction and Evaluation

Prediction: The hypothesis() function is used to compute predicted values for the test set.

Visualization:

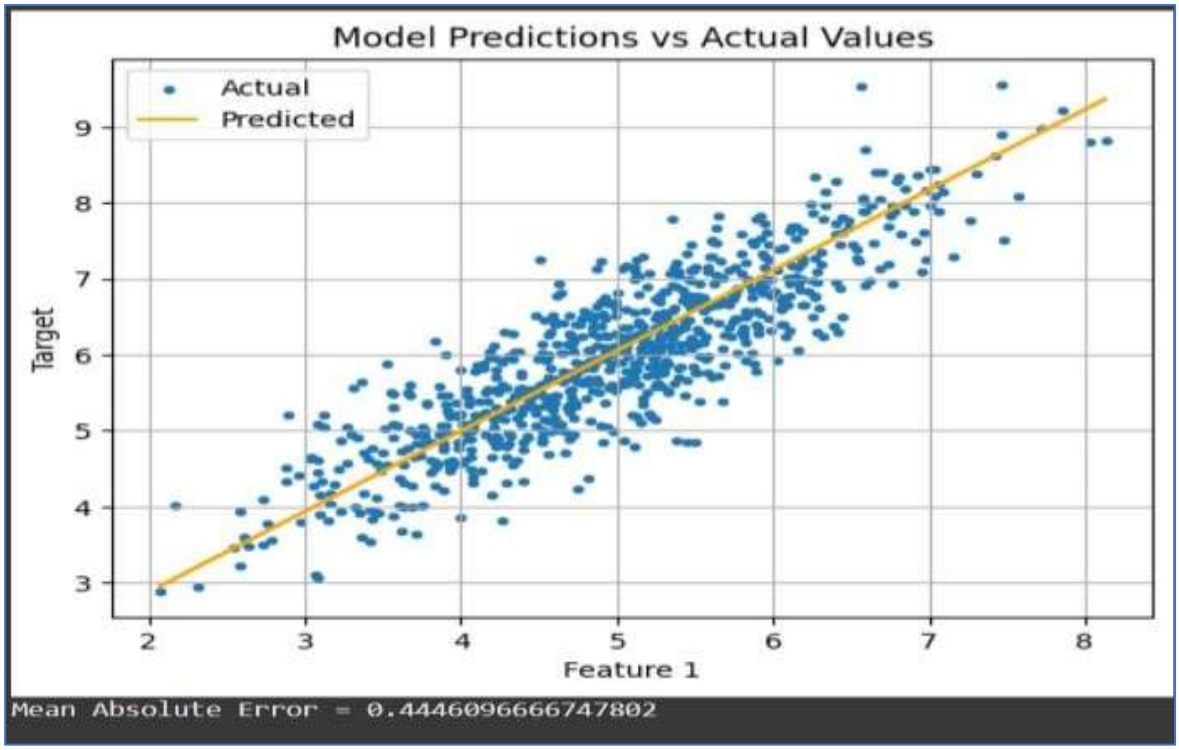
- A scatter plot shows actual test values.
- A line plot overlays the predicted values, helping to visually assess model performance.

Evaluation:

- Computes Mean Absolute Error (MAE) to measure average prediction deviation.
- A lower MAE indicates better accuracy of the model.

```
# Predicting output for X_test
y_pred = hypothesis(X_test, theta)

# Visualizing predictions vs actual values
plt.scatter(X_test[:, 1], y_test, marker='.', label='Actual')
plt.plot(X_test[:, 1], y_pred, color='orange', label='Predicted')
plt.xlabel("Feature 1")
plt.ylabel("Target")
plt.title("Model Predictions vs Actual Values")
plt.legend()
plt.grid(True)
plt.show()

# Calculating mean absolute error
error = np.sum(np.abs(y_test - y_pred)) / y_test.shape[0]
print("Mean Absolute Error =", error)
```

Output:



Model prediction and Actual values

The orange line represents the final hypothesis function i.e θ[0] + θ[1] * X_test[:, 1] + θ[2] * X_test[:, 2] = 0

This is the linear equation learned by the model where:

- `θ[0]` is the bias (intercept)
- `θ[1]` is the weight for the first feature
- `θ[2]` is the weight for the second feature

Comparison Between Gradient Descent Variants

Lets see a quick difference between Batch Gradient Descent, Stochastic Gradient Descent (SGD) and Mini-Batch Gradient Descent.

Type	Update Strategy	Speed & Efficiency	Noise in Updates
Batch Gradient Descent	Updates parameters after computing gradient using the entire training dataset	Slow, as it processes the full dataset before each update	Smooth and stable
Stochastic Gradient Descent (SGD)	Updates parameters after computing gradient using one training example	Faster updates, but cannot fully utilize vectorized computations	Highly noisy updates
Mini-Batch Gradient Descent	Updates parameters using a small batch (subset) of training examples	Efficient; leverages vectorization for faster computation	Moderate noise—dependent on batch size

What is Adam Optimizer

Adam (Adaptive Moment Estimation) optimizer combines the advantages of **Momentum** and **RMSprop** techniques to adjust learning rates during training. It works well with large datasets and complex models because it uses memory efficiently and adapts the learning rate for each parameter automatically.

How Does Adam Work?

Adam builds upon two key concepts in optimization:

1. Momentum

Momentum is used to accelerate the gradient descent process by incorporating an exponentially weighted moving average of past gradients. This helps smooth out the trajectory of the optimization allowing the algorithm to converge faster by reducing oscillations.
The update rule with momentum is:
$wt+1=wt−\alpha mt$ $wt+1=wt−\alpha mt$
where:

- mt mt is the moving average of the gradients at time t t
- α α is the learning rate
- wt wt and $wt+1$ $wt+1$ are the weights at time t t and $t+1$ $t+1$, respectively

The momentum term mt mt is updated recursively as:
$mt=\beta 1mt−1+(1−\beta 1)\partial L\partial wt$ $mt=\beta 1mt−1+(1−\beta 1)\partial wt\partial L$
where:

- $\beta 1$ $\beta 1$ is the momentum parameter (typically set to 0.9)
- $\partial L\partial wt$ $\partial wt\partial L$ is the gradient of the loss function with respect to the weights at time t t

2. RMSprop (Root Mean Square Propagation)
RMSprop is an adaptive learning rate method that improves upon AdaGrad. While AdaGrad accumulates squared gradients and RMSprop uses an exponentially weighted moving average of squared gradients, which helps overcome the problem of diminishing learning rates.
The update rule for RMSprop is:
$wt+1=wt−\alpha tvt+\epsilon \partial L\partial wt$ $wt+1=wt−vt+\epsilon \alpha t\partial wt\partial L$
where:

- vt vt is the exponentially weighted average of squared gradients:

$vt=\beta 2vt−1+(1−\beta 2)(\partial L\partial wt)2$ $vt=\beta 2vt−1+(1−\beta 2)(\partial wt\partial L)2$

- ϵ ϵ is a small constant (e.g., $10−810−8$) added to prevent division by zero

Combining Momentum and RMSprop to form Adam Optimizer
Adam optimizer combines the momentum and RMSprop techniques to provide a more balanced and efficient optimization process. The key equations governing Adam are as follows:

- **First moment (mean) estimate**:

$mt=\beta 1mt−1+(1−\beta 1)\partial L\partial wt$ $mt=\beta 1mt−1+(1−\beta 1)\partial wt\partial L$

- **Second moment (variance) estimate**:

$vt=\beta 2vt−1+(1−\beta 2)(\partial L\partial wt)2$ $vt=\beta 2vt−1+(1−\beta 2)(\partial wt\partial L)2$

- **Bias correction**: Since both mt mt and vt vt are initialized at zero, they tend to be biased toward zero, especially during the initial steps. To correct this bias, Adam computes the bias-corrected estimates:

$mt^=mt1−\beta 1t,vt^=vt1−\beta 2t$ $mt^=1−\beta 1tmt,vt^=1−\beta 2tvt$

- **Final weight update**: The weights are then updated as:

$wt+1=wt−mt^vt^+\epsilon \alpha$ $wt+1=wt−vt^+\epsilon mt^\alpha$

Key Parameters in Adam

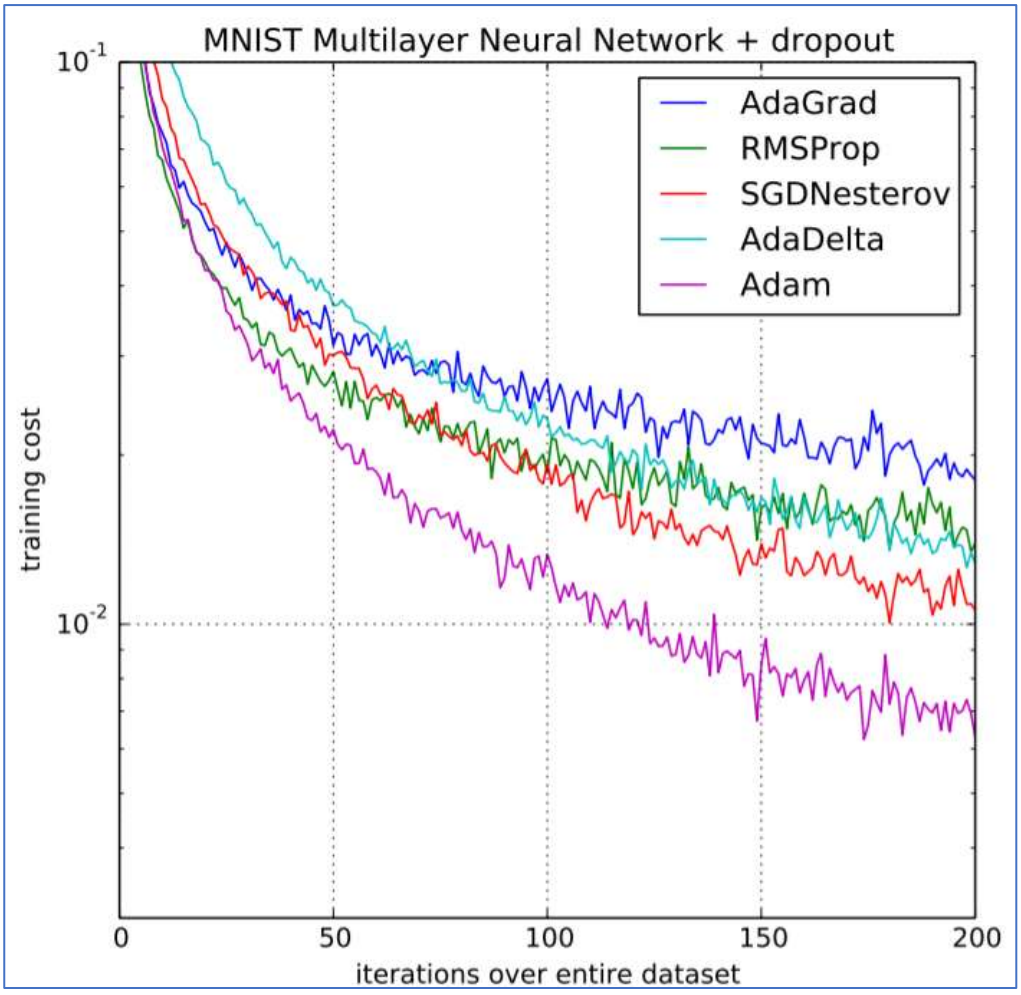
- α α: The learning rate or step size (default is 0.001)
- $\beta 1$ $\beta 1$ and $\beta 2$ $\beta 2$: Decay rates for the moving averages of the gradient and squared gradient, typically set to $\beta 1=0.9$ $\beta 1=0.9$ and $\beta 2=0.999$ $\beta 2=0.999$
- ϵ ϵ: A small positive constant (e.g., $10−810−8$) used to avoid division by zero when computing the final update

Why Adam Works So Well?
Adam addresses several challenges of gradient descent optimization:

- **Dynamic learning rates**: Each parameter has its own adaptive learning rate based on past gradients and their magnitudes. This helps the optimizer avoid oscillations and get past local minima more effectively.
- **Bias correction**: By adjusting for the initial bias when the first and second moment estimates are close to zero helping to prevent early-stage instability.
- **Efficient performance**: Adam typically requires fewer hyperparameter tuning adjustments compared to other optimization algorithms like SGD making it a more convenient choice for most problems.

Performance of Adam

In comparison to other optimizers like SGD (Stochastic Gradient Descent) and momentum-based SGD, Adam outperforms them significantly in terms of both training time and convergence accuracy. Its ability to adjust the learning rate per parameter combined with the bias-correction mechanism leading to faster convergence and more stable optimization. This makes Adam especially useful in complex models with large datasets as it avoids slow convergence and instability while reaching the global minimum.



Momentum-based gradient optimizers are used to optimize the training of machine learning models. They are more advanced than the classic gradient descent method and helps to accelerate the training process especially for large-scale datasets and deep neural networks.
By incorporating a **"momentum"** term these optimizers can navigate the loss surface more efficiently leading to faster convergence, reduced oscillations and better overall performance.

Understanding Gradient Descent

Before understanding momentum-based optimizers it's important to understand the traditional **gradient descent method**. In gradient descent the model's weights are updated by taking small steps in the direction of the negative gradient of the loss function. Mathematically its updates weight using:

$wt+1=wt−\eta\,\nabla L(wt)wt+1=wt−\eta\,\nabla L(wt)$

Where:

- $wtwt$ is the weight at time step tt
- $\eta\eta$ is the learning rate
- $\nabla L(wt)\nabla L(wt)$ is the gradient of the loss function with respect to the weights.

This method works well but it suffers from issues such as slow convergence and the tendency to get stuck in local minima especially in high-dimensional spaces. This is where momentum-based optimization can be useful.

What is Momentum?

Momentum is a concept from physics where an object's motion depends not only on the current force but also on its previous velocity. In the context of gradient optimization it refers to a method that smoothens the optimization trajectory by adding a term that helps the optimizer remember the past gradients.

In mathematical terms the momentum-based gradient descent updates can be described as:

$vt+1=\beta vt+(1−\beta)\,\nabla L(wt)vt+1=\beta vt+(1−\beta)\,\nabla L(wt)$

$wt+1=wt−\eta vt+1wt+1=wt−\eta vt+1$

Where:

- $vtvt$ is the velocity i.e a running average of gradients
- $\beta\beta$ is the momentum factor, typically a value between 0 and 1 (often around 0.9)
- $\nabla L(wt)\nabla L(wt)$ is the current gradient of the loss function
- $\eta\eta$ is the learning rate

Understanding Hyperparameters:

- **Learning Rate ($\eta\eta$)**: The learning rate determines the size of the step taken during each update. It plays a crucial role in both standard gradient descent and momentum-based optimizers.
- **Momentum Factor ($\beta\beta$)**: This controls how much of the past gradients are remembered in the current update. A value close to 1 means the optimizer will have more inertia while a value closer to 0 means less reliance on past gradients.

Working of the Algorithm:

1. **Velocity Update**: The velocity $vtvt$ is updated by considering both the previous velocity which represents the momentum and the current gradient. The momentum factor $\beta\beta$ controls the contribution of the previous velocity to the current update.
2. **Weight Update**: The weights are updated using the velocity $vt+1vt+1$ which is a weighted average of the past gradients and the current gradient.

Types of Momentum-Based Optimizers

There are several variations of momentum-based optimizers each with slight modifications to the basic momentum algorithm:

1. Nesterov Accelerated Gradient (NAG)

Nesterov momentum is an advanced form of momentum-based optimization. It modifies the update rule by calculating the gradient at the **upcoming** position rather than the current position of the weights.

The update rule becomes:

$$vt+1=\beta vt+\nabla L(wt-\eta\beta vt)vt+1=\beta vt+\nabla L(wt-\eta\beta vt)$$

$$wt+1=wt-\eta vt+1wt+1=wt-\eta vt+1$$

NAG is considered more efficient than classical momentum because it has a better understanding of the future trajectory, leading to even faster convergence and better performance in some cases.

2. AdaMomentum

AdaMomentum combines the concept of adaptive learning rates with momentum. It adjusts the momentum term based on the recent gradients making the optimizer more sensitive to the landscape of the loss function. This can help in fine-tuning the convergence process.

3. RMSProp (Root Mean Square Propagation)

Although not strictly a momentum-based optimizer in the traditional sense RMSProp incorporates a form of momentum by adapting the learning rate for each parameter. It's particularly effective when dealing with non-stationary objectives, such as in training recurrent neural networks (RNNs).

Advantages of Momentum-Based Optimizers

1. **Faster Convergence**: It helps to accelerate the convergence by considering past gradients, which helps the model navigate through flat regions more efficiently.
2. **Reduces Oscillation**: Traditional gradient descent can oscillate when there are steep gradients in some directions and flat gradients in others. Momentum reduces this oscillation by maintaining the direction of previous updates.
3. **Improved Generalization**: By smoothing the optimization process, momentum-based methods can lead to better generalization on unseen data, preventing overfitting.
4. **Helps Avoid Local Minima**: The momentum term can help the optimizer escape from local minima by maintaining a strong enough "velocity" to continue moving past these suboptimal points.

Challenges and Considerations

1. **Choosing Hyperparameters**: Selecting the appropriate values for the learning rate and momentum factor can be challenging. Typically a momentum factor of 0.9 is common but it may vary based on the specific problem or dataset.
2. **Potential for Over-Accumulation**: If the momentum term becomes too large it can lead to the optimizer overshooting the minimum, especially in the presence of noisy gradients.
3. **Initial Momentum**: When momentum is initialized it can have a significant impact on the convergence rate. Poor initialization can lead to slow or erratic optimization behavior.

Adagrad Optimizer in Deep Learning

Adagrad is an abbreviation for **Adaptive Gradient Algorithm**. It is an adaptive learning rate optimization algorithm used for training deep learning models. It is particularly effective for sparse data or scenarios where features exhibit a large variation in magnitude.

Adagrad adjusts the learning rate for each parameter individually. Unlike standard gradient descent, where a fixed learning rate is applied to all parameters Adagrad adapts the learning rate based on the historical gradients for each parameter, allowing the model to focus on more important features and learn efficiently.

How Does Adagrad Work?

The primary concept behind Adagrad is the idea of adapting the learning rate based on the historical sum of squared gradients for each parameter. Here's a step-by-step explanation of how Adagrad works:

1. Initialization

Adagrad begins by initializing the parameter values randomly, just like other optimization algorithms. Additionally, it initializes a running sum of squared gradients for each parameter, which will track the gradients over time.

2. Gradient Calculation

For each training step, the gradient of the loss function with respect to the model's parameters is calculated, just like in standard gradient descent.

3. Adaptive Learning Rate

The key difference comes next. Instead of using a fixed learning rate, Adagrad adjusts the learning rate for each parameter based on the accumulated sum of squared gradients.

The updated learning rate for each parameter is calculated as follows:

$lrt=\eta Gt+\epsilon lrt=Gt+\epsilon\eta$

Where:

- $\eta\eta$ is the global learning rate (a small constant value)
- $Gt$$Gt$ is the sum of squared gradients for a given parameter up to time step $t$$t$
- $\epsilon\epsilon$ is a small value added to avoid division by zero (often set to $1e-81e-8$)

Here, the denominator $Gt+\epsilon$$Gt+\epsilon$ grows as the squared gradients accumulate, causing the learning rate to decrease over time, which helps to stabilize the training.

4. Parameter Update

The model's parameters are updated by subtracting the product of the adaptive learning rate and the gradient at each step:

$\theta t+1=\theta t-lrt\cdot\nabla\theta\theta t+1=\theta t-lrt\cdot\nabla\theta$

Where:

- $\theta t\theta t$ is the current parameter
- $\nabla\theta J(\theta)\nabla\theta J(\theta)$is the gradient of the loss function with respect to the parameter

When to Use Adagrad?

Adagrad is ideal for:

- Problems with sparse data and features (e.g., natural language processing, recommender systems).
- Tasks where features have different levels of importance and frequency.
- Training models that do not require a very fast convergence rate but benefit from a more stable optimization process.

However, if you are dealing with problems where a more constant learning rate is preferable (e.g., in some deep learning tasks), using variants like **RMSProp** or **Adam** might be more appropriate.

Different Variants of Adagrad Optimizer

To address some of **Adagrad's** drawbacks, a few improved versions have been created like:

1. RMSProp (Root Mean Square Propagation):

RMSProp addresses the diminishing learning rate issue by introducing an exponentially decaying average of the squared gradients instead of accumulating the sum. This prevents the learning rate from decreasing too quickly, making the algorithm more effective in training deep neural networks.

The update rule for RMSProp is as follows:

$$Gt=\gamma Gt-1+(1-\gamma)(\nabla\theta J(\theta))2Gt=\gamma Gt-1+(1-\gamma)(\nabla\theta J(\theta))2$$

Where:

- $\mathrm{Gt}Gt$ is the accumulated gradient
- $\gamma\gamma$ is the decay factor (typically set to 0.9)
- $\nabla\theta\mathrm{J}(\theta)\nabla\theta J(\theta)$ is the gradient

The parameter update rule is:

$$\theta t+1=\theta t-\eta Gt+\epsilon\cdot\nabla\theta J(\theta)\theta t+1=\theta t-Gt+\epsilon\eta\cdot\nabla\theta J(\theta)$$

2. AdaDelta

AdaDelta is another modification of Adagrad that focuses on reducing the accumulation of past gradients. It updates the learning rates based on the moving average of past gradients and incorporates a more stable and bounded update rule.

The key update for AdaDelta is:

$$\Delta\theta t+1=-E[\Delta\theta]t2E[\nabla\theta J(\theta)]t2+\epsilon\cdot\nabla\theta J(\theta)\Delta\theta t+1=-E[\nabla\theta J(\theta)]t2+\epsilon E[\Delta\theta]t2\cdot\nabla\theta J(\theta)$$

Where:

- $[\Delta\theta]\mathrm{t}2[\Delta\theta]t2$ is the running average of past squared parameter updates

3. Adam (Adaptive Moment Estimation)

Adam combines the benefits of both Adagrad and momentum-based methods. It uses both the moving average of the gradients and the squared gradients to adapt the learning rate. Adam is widely used due to its robustness and superior performance in various machine learning tasks.

Adam has the following update rules:

- First moment estimate ($\mathrm{mt}mt$):

$$mt=\beta 1mt-1+(1-\beta 1)\nabla\theta J(\theta)mt=\beta 1mt-1+(1-\beta 1)\nabla\theta J(\theta)$$

- Second moment estimate ($\mathrm{vt}vt$):

$$vt=\beta 2vt-1+(1-\beta 2)(\nabla\theta J(\theta))2vt=\beta 2vt-1+(1-\beta 2)(\nabla\theta J(\theta))2$$

- Corrected moment estimates:

$$m\hat{}t=mt1-\beta 1t,v\hat{}t=vt1-\beta 2tm\hat{}t=1-\beta 1tmt,v\hat{}t=1-\beta 2tvt$$

- Parameter update:

$$\theta t+1=\theta t-\eta v\hat{}t+\epsilon\cdot m\hat{}t\theta t+1=\theta t-v\hat{}t+\epsilon\eta\cdot m\hat{}t$$

Adagrad Optimizer Implementation

Below are examples of how to implement the Adagrad optimizer in **TensorFlow** and **PyTorch**.

1. TensorFlow Implementation

In TensorFlow, implementing Adagrad is easier as it's already included in the API. Here's an example where:

- **mnist.load_data()** loads the **MNIST** dataset.
- **reshape()** flattens 28x28 images into 784-length vectors.
- Division by 255 **normalizes** pixel values to [0,1].
- **tf.keras.Sequential()** builds the neural network model.
- **tf.keras.layers.Dense()** creates fully connected layers.
- **activation='relu'** adds non-linearity in hidden layer and **softmax** outputs probabilities.
- **tf.keras.optimizers.Adagrad()** applies adaptive learning rates per parameter to improve convergence.

- **compile()** configures training with optimizer, loss function, and metrics.
- **loss='sparse_categorical_crossentropy'** computes loss for integer class labels.
- **model.fit()** trains the model for specified epochs on the training data.

```
import tensorflow as tf
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical

(x_train, y_train), (x_test, y_test) = mnist.load_data()

x_train = x_train.reshape(-1, 784).astype('float32') / 255.0
x_test = x_test.reshape(-1, 784).astype('float32') / 255.0

model = tf.keras.Sequential([
tf.keras.layers.Dense(64, activation='relu', input_shape=(784,)),
tf.keras.layers.Dense(10, activation='softmax')
])

model.compile(optimizer=tf.keras.optimizers.Adagrad(learning_rate=0.01),
loss='sparse_categorical_crossentropy',
metrics=['accuracy'])

model.fit(x_train, y_train, epochs=5)
```

Output:

```
Epoch 1/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.7966 - loss: 0.7555
Epoch 2/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9196 - loss: 0.2920
Epoch 3/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.9312 - loss: 0.2507
Epoch 4/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9395 - loss: 0.2178
Epoch 5/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.9443 - loss: 0.2056
<keras.src.callbacks.history.History at 0x7a6f5d338cd0>
```

Tensor Flow Implementation

2. PyTorch Implementation

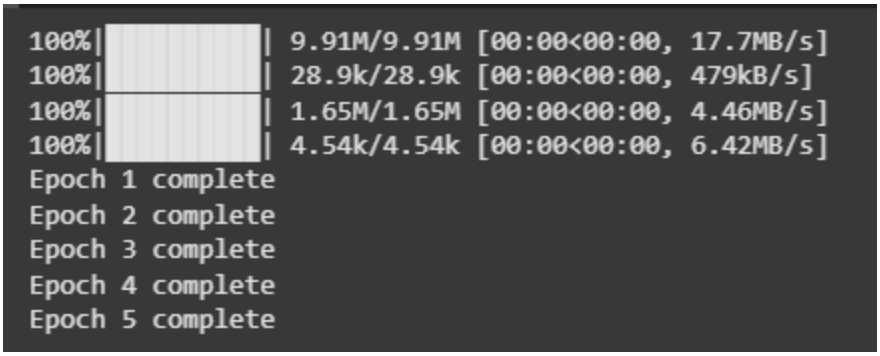
In PyTorch, Adagrad can be used with the **`torch.optim.Adagrad`** class. Here's an example where:

- **datasets.MNIST()** loads data, **ToTensor()** converts images and **Lambda()** flattens them.
- **DataLoader** batches and shuffles data.
- **SimpleModel** has two linear layers with **ReLU** in **forward()**.
- **CrossEntropyLoss** computes classification loss.
- **Adagrad optimizer** adapts learning rates per parameter based on past gradients, improving training on sparse or noisy data.
- **Training loop**: zero gradients, forward pass, compute loss, backpropagate and update weights with **Adagrad**.

```
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
```

```
from torch.utils.data import DataLoader

transform = transforms.Compose([
transforms.ToTensor(),
transforms.Lambda(lambda x: x.view(-1))  # flatten 28x28 to 784
])

train_dataset = datasets.MNIST(root='./data', train=True, download=True, transform=transform)
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)

class SimpleModel(nn.Module):
def __init__(self):
super(SimpleModel, self).__init__()
self.fc1 = nn.Linear(784, 64)
self.fc2 = nn.Linear(64, 10)

def forward(self, x):
x = torch.relu(self.fc1(x))
return self.fc2(x)

model = SimpleModel()

loss_fn = nn.CrossEntropyLoss()
optimizer = optim.Adagrad(model.parameters(), lr=0.01)

for epoch in range(5):
for data, target in train_loader:
optimizer.zero_grad()
output = model(data)
loss = loss_fn(output, target)
loss.backward()
optimizer.step()
print(f"Epoch {epoch+1} complete")
```

Output:



RMSProp Optimizer in Deep Learning

RMSProp (Root Mean Square Propagation) is an adaptive learning rate optimization algorithm designed to improve the performance and speed of training deep learning models.

- It is a variant of the **gradient descent** algorithm which adapts the learning rate for each parameter individually by considering the magnitude of recent gradients for those parameters.
- This adaptive nature helps in dealing with the challenges of non-stationary objectives and sparse gradients commonly encountered in deep learning tasks.

Need of RMSProp Optimizer

RMSProp was developed to address the limitations of previous optimization methods such as **SGD (Stochastic Gradient Descent)** and **AdaGrad** as SGD uses a constant learning rate which can be inefficient and AdaGrad reduces the learning rate too aggressively.

RMSProp balances by adapting the learning rates based on a moving average of squared gradients. This approach helps in maintaining a balance between efficient convergence and stability during the training process making RMSProp a widely used optimization algorithm in modern deep learning.

How RMSProp Works?

RMSProp keeps a moving average of the squared gradients to normalize the gradient updates. By doing so it prevents the learning rate from becoming too small which was a **drawback in AdaGrad** and ensures that the updates are appropriately scaled for each parameter. This mechanism allows RMSProp to perform well even in the presence of non-stationary objectives, making it suitable for training deep learning models.

The mathematical formulation is as follows:

1. Compute the gradient gt at time step t:

$gt=\nabla\theta gt=\nabla\theta$

2. Update the moving average of squared gradients:

$E[g2]t=\gamma E[g2]t-1+(1-\gamma)E[g2]t=\gamma E[g2]t-1+(1-\gamma)$

where γ is the decay rate.

3. Update the parameter θ using the adjusted learning rate:

$\theta t+1=\theta t-\eta E[g2]t+\epsilon\theta t+1=\theta t-E[g2]t+\epsilon\eta$

where η is the learning rate and ϵ is a small constant added for numerical stability.

Parameters Used in RMSProp

- **Learning Rate (η)**: Controls the step size during the parameter updates. RMSProp typically uses a default learning rate of 0.001, but it can be adjusted based on the specific problem.
- **Decay Rate (γ)**: Determines how quickly the moving average of squared gradients decays. A common default value is 0.9, which balances the contribution of recent and past gradients.
- **Epsilon (ϵ)**: A small constant added to the denominator to prevent division by zero and ensure numerical stability. A typical value for ϵ is 1e-8.

By carefully adjusting these parameters, RMSProp effectively adapts the learning rates during training, leading to faster and more reliable convergence in deep learning models.

Implementing RMSprop in Python using TensorFlow or Keras

We will use the following code line for initializing the RMSProp optimizer with hyperparameters:

tf.keras.optimizers.RMSprop(learning_rate=0.001, rho=0.9)

- **`learning_rate=0.001:`** `Sets the step size for weight updates. Smaller learning rates result in smaller updates, helping to fine-tune weights and prevent overshooting the minimum loss.`
- **rho=0.9:** The discounting factor for the history of gradients, controlling the influence of past gradients on the current gradient computation.

1. Importing Libraries

We are importing libraries to implement RMSprop optimizer, handle datasets, build the model and plot results.

- **tensorflow.kera**s for deep learning components.
- **matplotlib.pyplot** for visualization.

```
import tensorflow as tf
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.utils import to_categorical
import matplotlib.pyplot as plt
```

2. Loading and Preprocessing Dataset

We load the MNIST dataset, normalize pixel values to [0,1] and one-hot encode labels.

- **mnist.load_data()** loads images and labels.
- **Normalization** improves training stability.
- **to_categorical()** converts labels to one-hot vectors.

```
(x_train, y_train), (x_test, y_test) = mnist.load_data()

x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)
```

3. Building the Model

We define a neural network using Sequential with input flattening and dense layers.

- **Flatten** converts 2D images to 1D vectors.
- Dense layers learn patterns with **ReLU** and **softmax** activations.

```
model = Sequential([
Flatten(input_shape=(28, 28)),
Dense(128, activation='relu'),
Dense(64, activation='relu'),
Dense(10, activation='softmax')
])
```

4. Compiling the Model

We compile the model using the RMSprop optimizer for adaptive learning rates, categorical cross-entropy loss for multi-class classification and track accuracy metric.

- **RMSprop** adjusts learning rates based on recent gradients (parameter rho controls decay rate).
- **categorical_crossentropy** suits one-hot encoded labels.

```
model.compile(optimizer=tf.keras.optimizers.RMSprop(learning_rate=0.001, rho=0.9),
```

loss='categorical_crossentropy',
metrics=['accuracy'])

5. Training the Model

We train the model over 10 epochs with batch size 32 and validate on 20% of training data. **validation_split** monitors model performance on unseen data each epoch.

history = model.fit(x_train, y_train, epochs=10, batch_size=32, validation_split=0.2)

Output:

```
Epoch 1/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 4ms/step - accuracy: 0.8721 - loss: 0.4363 - val_accuracy: 0.9524 - val_loss: 0.1541
Epoch 2/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 9s 4ms/step - accuracy: 0.9624 - loss: 0.1212 - val_accuracy: 0.9673 - val_loss: 0.1173
Epoch 3/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.9737 - loss: 0.0864 - val_accuracy: 0.9685 - val_loss: 0.1177
Epoch 4/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.9799 - loss: 0.0664 - val_accuracy: 0.9719 - val_loss: 0.1088
Epoch 5/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.9838 - loss: 0.0526 - val_accuracy: 0.9747 - val_loss: 0.0992
Epoch 6/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 10s 4ms/step - accuracy: 0.9880 - loss: 0.0397 - val_accuracy: 0.9737 - val_loss: 0.1076
Epoch 7/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.9907 - loss: 0.0338 - val_accuracy: 0.9736 - val_loss: 0.1200
Epoch 8/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 10s 5ms/step - accuracy: 0.9920 - loss: 0.0284 - val_accuracy: 0.9749 - val_loss: 0.1208
Epoch 9/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.9928 - loss: 0.0242 - val_accuracy: 0.9707 - val_loss: 0.1409
Epoch 10/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 11s 4ms/step - accuracy: 0.9933 - loss: 0.0214 - val_accuracy: 0.9758 - val_loss: 0.1398
```

Training the Model

6. Evaluating and Visualizing Results

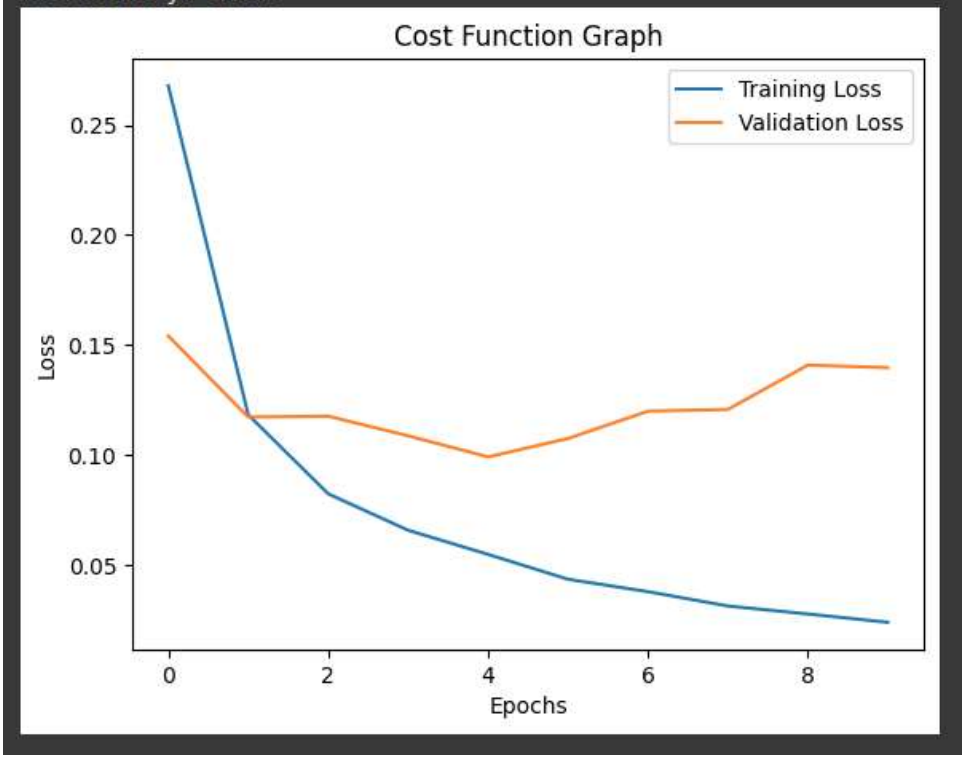
We evaluate test accuracy on unseen test data and plot training and validation loss curves to visualize learning progress.

loss, accuracy = model.evaluate(x_test, y_test)
print(f'Test accuracy: {accuracy:.4f}')

plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Cost Function Graph')
plt.legend()
plt.show()

Output:

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.9734 - loss: 0.1450
Test accuracy: 0.9773
```



Advantages of RMSProp

- **Adaptive Learning Rates:** Adjusts learning rates for each parameter individually, optimizing updates more effectively.
- **Handles Non-Stationary Objectives:** Efficiently adapts to changing optimal parameter values over time.
- **Prevents Learning Rate Decay Problem:** Maintains optimal learning rates by using a decay rate unlike AdaGrad.
- **Improved Convergence Speed:** Faster convergence due to balanced and dynamic learning rates.

Disadvantages of RMSProp

- **Sensitivity to Hyperparameters:** Performance is sensitive to settings like decay rate and epsilon meaning it requires careful tuning.
- **Poor Performance with Sparse Data:** May struggle with sparse data, leading to slower or inconsistent convergence.

7 Best Deep Learning Frameworks You Should Know in 2025 As we are now moving at this speed in technology, we're becoming more adaptive toward it. In recent years, there has been a lot of noise about ***Deep Learning***, especially in the field of technology (to be specific, ***data science***), and it's being widely used today in various industries. Deep Learning has become one of the most significant weapons in technology today including **self-driving cars, automated tasks**, **AI-based voice-overs**, and whatnot, it is widely operating in almost every domain to make work-life balance simpler and more advanced.

By the end of 2025, the deep learning market is projected to reach approximately USD 34.29 billion, with a CAGR of 38.8%.

What is a Deep Learning Framework?

A **deep learning framework** is a software library or tool that includes a set of **APIs** (**Application Programming Interfaces**), **abstractions**, and tools to assist developers in building and training deep learning models. Deep learning frameworks could help you upload data and train a **deep learning model** that would lead to accurate and intuitive predictive analysis. These frameworks simplify the process of creating and deploying neural networks, allowing researchers and engineers to focus on complicated machine-learning tasks.

Complete Machine Learning and Data Science Course*gives you access to the course explaining ML and AI concepts such as Regression, Classification, and Clustering, and you will get to learn all about NLP.*

7 Best Deep Learning Frameworks You Should Know :

Explore these **deep-learning frameworks** designed to advance your projects and boost your results. Whether you're a beginner eager to work on your first project or an experienced developer trying to find something new in **AI innovation**, this list provides you with the knowledge to choose the **perfect framework** for your needs.

Table of Content

- 1. TensorFlow
- 2. PyTorch
- 3. Keras
- 4. Theano
- 5. Deeplearning4j (DL4J)
- 6. Scikit-learn
- 7. Sonnet

1. TensorFlow

TensorFlow is one of the most popular, open-source libraries that is being heavily used for ***numerical computation in deep learning***. Google introduced it in 2015 for their internal RnD work but later when they saw the capabilities of this framework, they decided to make it open and the repository is available at TensorFlow Repository. As you'll see, learning deep learning is pretty complex but making certain implementations is far easier, and by such frameworks, it's even smoother to process the desired outcomes.

How Does it Work?

This framework allows you to create *dataflow graphs and structures* to specify **how data travels through a graph** with the help of inputs as tensors (also known as a multi-dimensional graph). TensorFlow allows users to prepare a flowchart and based on their inputs, it generates the output.

Applications of Tensor Flow:

- **Text-Based Application:** Nowadays, text-based apps are being heavily used in the market including language detection, sentimental analysis (for social media to block abusive posts)

- **Image Recognition (I-R) Based System:** Most sectors have introduced this technology in their system for motion, facial, and photo-clustering models.
- **Video Detection:** Real-time object detection is a computer vision technique to detect the motion (from both image and video) to trace back any object from the provided data.

2. PyTorch

The most famous, that even powers *"Tesla Auto-Pilot"* is none other than **Pytorch** which works on deep learning technology. It was first introduced in 2016 by a group of people (**Adam Paszke, Sam Gross, Soumith Chintala, and Gregory Chanan**), under **Facebook's AI lab**. The interesting part about PyTorch is that both **C++** & **Python** can use it but Python's interface is the most polished. Unsurprisingly, Pytorch is being backed by some of the top giants in the tech industry **(Google, Salesforce, Uber, etc.)**. It was introduced to achieve two major goals, the first is to remove the requirement of **NumPy** (so that it can power **GPU** with tensor) and the second is to offer an automatic differentiation library (that is useful to implement neural networks).

How Does it Work?

This framework uses a computational dynamic graph right after the declaration of variables. Besides this, it uses Python's basic concepts like loops, structures, etc. We have often used NLP functions in our smartphones (such as Apple's Siri or Google Assistant), and they all use deep learning algorithms known as**RNN or Recurrent Neural Network**.

Applications of PyTorch:

- **Weather Forecast:** To predict and highlight the pattern of a particular set of data, PyTorch is being used (not only for forecast but also for real-time analysis).
- **Text Auto Detection:** You might have noticed that sometimes when we try to search for something on Google or any other search engine, it starts showing some "auto-suggestions", that's where the algorithm works and PyTorch is being used.
- **Fraud Detection:** To prevent any unauthorized activities on credit/debit cards, this algorithm is being used to detect anomalous behavior and outliers.

3. Keras

Keras is another highly productive library that focuses on solving deep learning problems. Besides this, Keras also helps engineers to take full advantage of the scalability and cross-platform capabilities to apply within their projects. It was first introduced in 2015 under the **ONEIROS (Open-ended Neuro-Electronic Intelligent Robot Operating System)** project. Keras is an open-source platform and is being actively used as a part of Python's interface in machine learning and deep neural learning. Today, big tech giants like **Netflix, Uber**, etc. are using Keras actively to improve their scalability.

How Does it Work?

The architecture of **Keras** has been designed in such a way that it acts as a **high-level neural network (written in Python**). Besides this, It works as a wrapper for **low-level librarie**s *(*such as **TensorFlow** or **Theano***)* and high-level neural network libraries. It was introduced with the concept of performing fast testing and experiments before going on the full scale.

Applications of Keras:

- Today, companies are using Keras to develop smartphones powered by **machine learning and deep learning** in their system. **Apple** is one of the biggest giants that has incorporated this technology in past few years.
- In the healthcare industry, developers have built a predictive technology where the machine can predict the **patient's diagnosis** and can also alert pre-heart attack issues.
- Face Mask Detection: During the pandemic, many companies have offered various contributions, and companies have built a system using deep learning mechanisms for using facial recognition to detect whether the person is wearing a facial mask or not. (Nokia was among the companies to initiate this using the Keras library)

4. Theano

To ***define any mathematical expressions in deep learning,*** we use Python's library Theano. It was named after a great Greek mathematician "**Theano**". It was released in 2007 by **MILA (Montreal Institute for Learning Algorithms)** and Theano uses a host of clever code optimizations to deliver as much performance at maximum caliber from your hardware. Besides this, there are two salient features at the core of any deep-learning library:

- The tensor operations, and

- The capability to run the code on **CPU** or Graphical Computation Unit (**GPU**).

These two features enable us to work with a big bucket of data. Moreover, Theano proposes automatic differentiation which is a very useful feature and can also solve numeric optimization on a big picture than deep learning complex issues.

How Does it Work?

If you talk about its working algorithm, *Theano itself is effectively dead, but the deep learning frameworks built on top of Theano, are still functioning* which also include the more user-friendly frameworks- ***Keras***, ***Lasagne***, *and* ***Blocks***that offer a high-level framework for fast prototyping and model testing in deep learning and machine learning algorithms.

Applications of Theano:

- **Implementation Cycle:** Theanos works in 3 different steps where it starts by defining the objects/variables then moves into different stages to define the mathematical expressions (in the form of functions) and at last it helps in evaluating expressions by passing values to it.
- Companies like IBM are using Theanos for implementing neural networks and to enhance their efficiency
- For using Theanos, make sure you have pre-installed some of the following dependencies: **Python, NumPy, SciPy, and BLAS** (for matrix operations).

5. Deeplearning4j (DL4J)

Deeplearning4j (DL4J) is a free tool, a deep learning framework for building applications using **Java and Scala**. It was created by **Skymind. T**he reason behind its popularity is that it works well with existing Java-based systems, thanks to its compatibility with the **Java Virtual Machine (JVM)**. DL4J lets developers make and use strong models for things like recognizing images and speech, understanding language, and making predictions.

How Does it Work?

Deeplearning4j (DL4J) helps developers by providing a set of libraries for **Java and Scala** programmers to build and deploy deep learning models. It takes advantage of the **Java Virtual Machine (JVM)** for compatibility and supports various neural network architectures. Its emphasis on distributed computing enables efficient training of large-scale models across multiple machines.

Application of Deeplearning4j (DL4J)

- DL4J is suited for integration with existing systems due to its compatibility with the **JVM**.
- It is good at **training large deep-learning models** because it can work on many machines at once.
- It is widely used in various domains such as **image and speech recognition, natural language processing, and predictive analytics**, making it a versatile choice for different tasks.

6. Scikit-learn

Originating from the notion **SciPy Toolkit** was designed ***to operate and handle high-performance linear algebra.*** Firstly, it was introduced back in 2007 during the **Google Summer of Code** project by **David Cournapeau**. This model is designed on various frameworks such as NumPy, **SciPy**, and **Matplotlib** and has been written in **Python**. The objective of **sci-kit-learn** is to offer some of efficient tools for Deep learning, Machine learning, and statistical modeling that enlist:

- Regression (Linear and Logistic)
- Classification (K-Nearest Neighbors)
- Clustering (K-means and K-means++)
- Model Selection,
- Preprocessing (min to max normalization), and
- Dimensionality reduction (used for visualization, summarization, and feature selection)

Moreover, it offers two different varieties of algorithms (supervised and unsupervised).

How Does it Work?

The sole purpose of introducing this library is **to achieve the level of robustness and support required for use in production systems,** which means a deep focus on concerns (that include ease of use, code quality, collaboration, documentation, and performance). Although the interface is Python, c-libraries are an advantage for performance (such as **NumPy**) for arrays and matrix operations.

Application of Scikit-learn

- Companies like **Spotify, Inria, and J.P. Morgan** are actively using this framework to improve linear algebra and statistical analysis.
- It works on the user's behavior and displays the outputs based on their activity

- It helps in **collecting data, analyzing those stats, and providing satisfactory outputs** of what users would want to see.

7. Sonnet

Sonnet, crafted by **DeepMind,** is a high-level toolkit for creating sophisticated neural network architectures in **TensorFlow**. This deep learning framework is built on top of TensorFlow. It seeks to construct and generate Python objects that correspond to certain parts of a neural network. These objects are then individually linked to the computational TensorFlow graph. This approach of independently building Python objects and attaching them to a graph simplifies the construction of high-level structures. This is one of the greatest deep-learning frameworks available.

How Does it Work?

It simplifies the creation of models through **high-level abstractions, modular design, and efficient parameter management**. Sonnet also works well with TensorFlow, making it easier for scientists and developers to create and train smart systems for different jobs. It's like a friendly assistant that makes constructing and optimizing sophisticated models straightforward.

Application of Sonnet

- Sonnet plays a crucial role in advanced neural network research, offering a flexible framework for quickly trying out new ideas in model architectures and optimization methods.
- In NLP, it is used to build language models like **transformers**, making it great for tasks such as understanding and generating text.
- For computer vision tasks such as recognizing images and finding objects, Sonnet is very valuable. It smoothly works with TensorFlow and supports GPU acceleration, making designing and training models efficient.

Types of Deep Learning Models

Lets see various types of Deep Learning Models:

1. Convolutional Neural Networks (CNNs)

Convolutional Neural Networks (CNNs) are a class of deep neural networks that are designed for processing grid-like data such as images. They use convolutional layers to automatically detect patterns like edges, textures and shapes in the data.

- Deep Learning Algorithms
- Convolutional Neural Networks (CNNs)
- Basics of Digital Image Processing
- Importance for CNN
- Padding
- Convolutional Layers
- Pooling Layers
- Fully Connected Layers
- Backpropagation in CNNs
- CNN based Image Classification using PyTorch
- CNN based Images Classification using TensorFlow

Top 10 Deep Learning Algorithms

Deep learning is going to further transform the world from as we know it to something different in the future and lead the way in most industries across the globe. And the most important part of this technology are the **algorithms** that are used to create and train those models. These algorithms are starting to dominate sectors as diverse as **healthcare**, **autonomous vehicles** and **finance** by **analyzing and learning from huge datasets**. The availability of advanced algorithms, powerful computing technologies and a wealth of data has made **deep**

learning the leading subfield of AI that is paving the way to the development of new and better solutions and, thus, to technological progress.

Top 10 Deep Learning Algorithms

In this article, we highlight the **top 10 deep learning algorithms in 2025**. From **Convolutional Neural Networks (CNNs) to Generative Adversarial Networks (GANs)**, these algorithms are driving innovations in various industries. We will also take a look at their key mechanisms which define them and their key functionalities. But before we deep-dive into those algorithms, let us familiarize ourselves with the concept of deep learning.

Table of Content

- What is Deep Learning?
- What are Deep Learning Algorithms?
- 1. Convolutional Neural Network (CNN)
- 2. Recurrent Neural Network (RNN)
- 3. Long Short-Term Memory (LSTM)
- 4. Auto-Encoders
- 5. Deep Belief Network (DBN)
- 6. Generative Adversarial Network (GAN)
- 7. Self-Organizing Map (SOM)
- 8. Variational Autoencoders (VAEs)
- 9. Graph Neural Networks (GNNs)
- 10. Transformers

What is Deep Learning?

Deep learning is a subfield of **machine learning**, which is itself a part of **artificial intelligence**, that focuses on the use of many layered **neural networks** to train themselves on large amounts of data. Developed based on the idea of biological brains, these networks are able to learn from data without being programmed explicitly, which makes deep learning particularly effective for tasks that involve **images, speech, natural language**, and many other kinds of input data. Traditional machine learning is not as efficient at dealing with complex and unorganized data, and the effectiveness only improves with the size of the dataset and computational resources available, which is where deep learning models excel.

Learn more: Deep Learning Tutorial

Emergence of Deep Learning: A Quick Look Back

The fascinating field of Deep Learning has been around longer than you might think. It was first introduced in the **1940s**, with the development of the **perceptron** in the late **1950s** acting as a cornerstone of modern deep learning. The evolution of deep learning has been marked by remarkable breakthroughs, often spurred by progress in computer processing power, the availability of vast amounts of data, and algorithmic refinements.

What are Deep Learning Algorithms?

The **deep learning algorithms** are a type of specific **machine learning models** based on the principles of the human brain. These algorithms apply the **artificial neural networks** in the processing of data, where each network is consisted of connected nodes or neurons. Deep learning algorithms are different from regular machine learning models because they are able to learn complex patterns from the data sets without needing manual extraction. Because of this, they are very successful in their application areas, which include **image classification, speech recognition, and natural language processing**.

Top 10 Deep Learning Algorithms in 2025

1. Convolutional Neural Network (CNN)

Convolutional Neural Networks are advanced forms of neural networks which are primarily employed in various tasks that involve **images and videos**. They are designed to learn features directly from the data, automatically detecting patterns such as **edges, textures and shapes**, thus making them very useful for applications like **object detection**, **medical imaging** and **facial recognition**.

Key Mechanisms:

- **Convolution Layer:** It applies **filters (kernels)** on the **input data (e.g. an image)** to identify basic features like **edges or corners**. Each filter slides over the image to capture local patterns.
- **Pooling Layer:** After detecting the features, **the pooling layer** down samples the data, retaining only the most significant features, thereby enhancing the computational efficiency of the model.
- **Fully Connected Layer:** After the convolution and pooling operations, the extracted features are passed through a **fully connected layer** to make the prediction about the class of the input.
- **Activation Function:** An **activation function** is a mathematical function that is used in neural networks to introduce non-linearity, and thereby enables the model to learn complex patterns and make better predictions.

2. Recurrent Neural Network (RNN)

RNNs are designed for sequential data such as **time series or natural language**. Traditional neural networks differ from RNNs as RNNs have a memory that **keeps information from the previous steps**, making them suitable for applications like **speech recognition, language translation, and stock price prediction**.

Key Mechanisms:

- **Sequential Processing:** RNNs process data one step at a time, and **output at each step depends on the current input and the previous step's output**, effectively capturing temporal patterns.
- **Hidden States:** The states of the RNNs are hidden, being updated after each step, to enable the network to remember past information. These states are also fed into the next step in the sequence.
- **Weight Sharing:** RNNs use the same weights across time steps, which is useful when dealing with sequences of varying length, and make the models more efficient.
- **Backpropagation Through Time (BPTT):** In the training phase, RNNs learn to minimize the error from future steps, learning to better predict each part of the sequence by adjusting their weights.

3. Long Short-Term Memory (LSTM)

To overcome the **vanishing gradient problem**, there is a particular kind of RNN, i.e., **LSTM**. It can learn many dependencies in data, and therefore, find its application in **language modeling, text generation, and video analysis**.

Key Mechanisms:

- **Cell State:** The LSTMs keep a state called **cell state which is the long term memory of the network**. It can store, update or forget information over time, helping the network keep track of important information.
- **Forget Gate:** This gate decides **what information from the previous cell state should be discarded**, allowing the network to forget some information.

- **Input Gate:** It **controls the input of new information to the cell state**, and hence what is added to the memory.
- **Output Gate:** This gate controls **what information from the cell state is outputted to the next layer** or time step.

4. Auto-Encoders

Auto-encoders are unsupervised learning models used to reduce the dimensionality of data. They learn to compress input data into a lower-dimensional representation and then reconstruct it back to its original form, making them useful for tasks like data compression and anomaly detection.

Key Mechanisms:

- **Encoder:** The **network encoder** part of the network is to compress the input data to a lower dimensional representation. It learns the most important characteristics of the input data.
- **Bottleneck:** The bottleneck layer is implemented to make the network **learn a compact representation of the input**, identifying crucial characteristics.
- **Decoder:** The **decoder** attempts to synthesize the original input from the encoded data, trying to make the output match the original input as much as possible.
- **Loss Function:** The model uses a **loss function**, such as **Mean Squared Error**, for defining the error between the input and output of the model.

5. Deep Belief Network (DBN)

Deep Belief Networks are composed of multiple layers of **Restricted Boltzmann Machines (RBMs)** stacked together. They are often used for **feature learning**, **image recognition,** and **unsupervised pretraining.**

Key Mechanisms:

- **Layered Structure: DBNs** are a kind of **deep neural networks (DNNs)** which are constructed by stacking several layers of **Restricted Boltzmann machines (RBMs)**. Each RBM is responsible for learning features from the data and increasing the level of complexity with each subsequent layer.
- **Unsupervised Pretraining:** The layers are pretrained in an **unsupervised manner**, and each RBM tries to learn the distribution of the data.
- **Fine-Tuning:** After that, the network is fine-tuned for actual labeled data in order to enhance the performance on certain tasks, like **classification**.
- **Stochastic Units:** The RBMs utilize **stochastic (probabilistic) units**, which determine the activation of each unit by probability, enabling the network to learn complicated, non-linear relationships.

6. Generative Adversarial Network (GAN)

GANs use two models: **a Generator** and **a Discriminator**. The **Generator produces the fake data** (for ex. images), and the **Discriminator checks if the data is real or fake**. GANs are probably the most popular model for **creating realistic images, videos and even deepfakes**.

Key Mechanisms:

- **Generator:** The Generator is **trained on random noise** and learns to create synthetic data that looks similar to real data, e.g. images or text.
- **Discriminator:** The Discriminator evaluates the generated data, compares it to real data and provides feedback to the Generator.
- **Adversarial Training:** The Generator and Discriminator are trained together in an **adversarial training process** where each is attempting to fool the other. The

Generator wants to create more plausible data while the Discriminator tries to get better at telling real data from fake.

- **Loss Function:** The models are trained with a **specific type of loss function** that determines the discrepancy between the Discriminator's output and the actual class labels to further enhance both networks' training process.

7. Self-Organizing Map (SOM)

Self-Organizing Maps are a type of **unsupervised learning model** used to **map high-dimensional data to a lower-dimensional grid.** They are particularly useful for clustering and visualizing complex data.

Key Mechanisms:

- **Neuron Grid:** The network has a grid of neurons, each neuron being a representation of a cluster of similar data points.
- **Competitive Learning:** Neurons respond to input data by competing for it, updating the weights of the 'winner' neuron with the input.
- **Neighborhood Function:** Other neurons nearby the winner also learn their weights, helping the network to learn the similarities in the data, and preserve its structure.
- **Topological Preservation:** SOMs maintain the topological relationships of the data, so that the similar data points end up near each other on the map.

8. Variational Autoencoders (VAEs)

Variational Autoencoders are a **probabilistic version of autoencoders** used for generative tasks. VAEs learn a distribution of the data and generate new data by sampling from that distribution.

Key Mechanisms:

- **Encoder:** The encoder is in charge of learning a compressed representation of the input in the form of a probabilistic distribution, typically in terms of the mean and variance.
- **Latent Space:** This distribution is then used for sampling new data points from the latent space to enable the model to create new data that is completely new.
- **Decoder:** The decoder learns to reconstruct the data from the sampled latent variables, **creating synthetic output**.
- **KL Divergence:** The model learns to minimize the **Kullback-Leibler divergence**, so that the learned distribution is close to a standard prior distribution, like a normal distribution.

9. Graph Neural Networks (GNNs)

Graph Neural Networks are designed to work with **graph-structured data**, such as **social networks, molecular structures,** and **recommendation systems**. They capture relationships between nodes and edges in the graph to make predictions or understand the structure.

Key Mechanisms:

- **Node Aggregation:** They collect information from their neighbor nodes and give a better representation of their context.
- **Message Passing:** Information is passed between adjacent nodes in the graph to capture dependencies and relationships between entities and helping the model learn them.
- **Graph Pooling:** This mechanism creates a global representation of the graph by learning the information from all of the nodes.

- **Backpropagation:** The optimization of the node features is achieved through the standard **backpropagation** so as to enhance the learning process and the prediction of graph-based tasks.

10. Transformers

Transformers are widely used in **Natural Language Processing (NLP)** tasks like **machine translation**, **text generation**, and **sentiment analysis**. They are based on **self-attention mechanisms** that help models capture long-range dependencies in data.

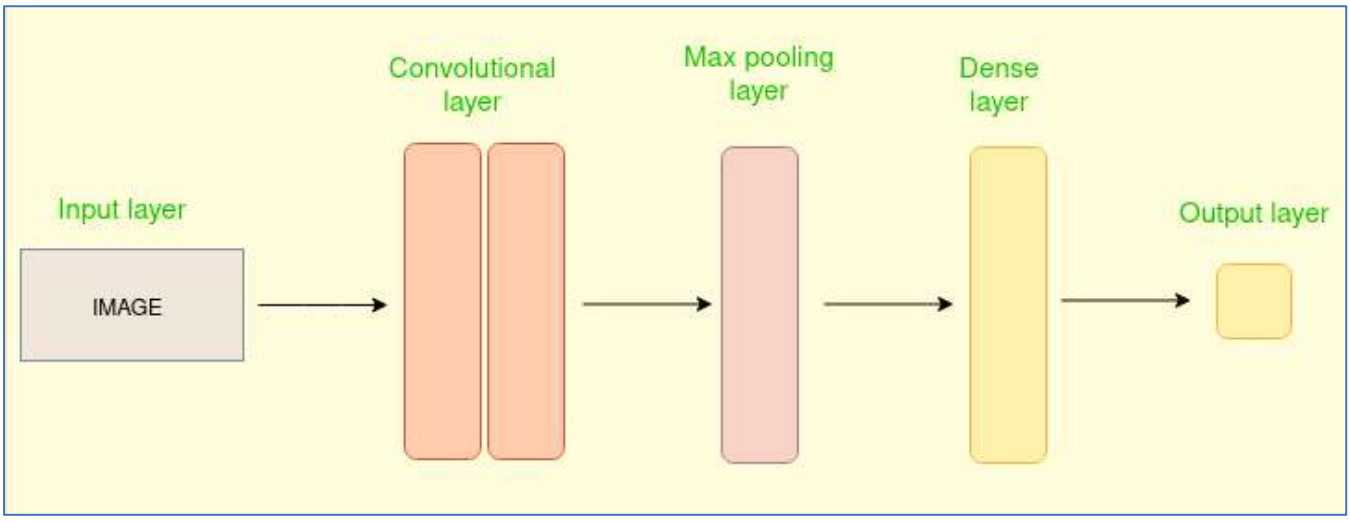
Key Mechanisms:

- **Self-Attention:** Every token in the input sequence is able to learn the relationship of every other token, thus **learning long range dependencies** without needing any sequence order.
- **Multi-Head Attention:** Multiple attention mechanisms work in parallel, capturing different types of relationships between tokens.
- **Positional Encoding:** Since transformers are not sequential in processing the data, positional encodings are employed to provide information about the position of the tokens in the sequence.
- **Feedforward Layers:** After the attention mechanisms, the **information is then passed through fully connected layers**, which are able to process and transform the data for further tasks like classification or generation.

Introduction to Convolution Neural Network

Convolutional Neural Network (CNN) is an advanced version of **artificial neural networks (ANNs)**, primarily designed to extract features from grid-like matrix datasets. This is particularly useful for visual datasets such as images or videos, where data patterns play a crucial role. CNNs are widely used in **computer vision** applications due to their effectiveness in processing visual data.

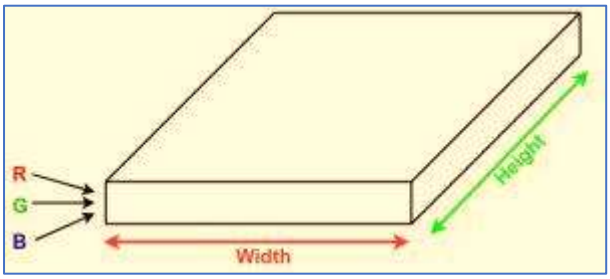
CNNs consist of multiple layers like the input layer, Convolutional layer, pooling layer, and fully connected layers. Let's learn more about CNNs in detail.



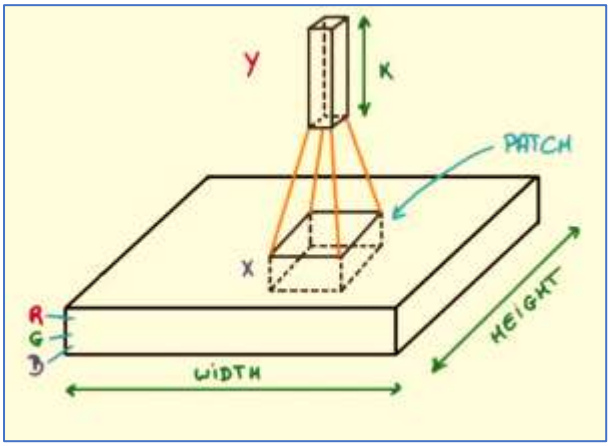
How Convolutional Layers Works?

Convolution Neural Networks are neural networks that share their parameters.

Imagine you have an image. It can be represented as a cuboid having its length, width (dimension of the image), and height (i.e the channel as images generally have red, green, and blue channels).



Now imagine taking a small patch of this image and running a small neural network, called a filter or kernel on it, with say, K outputs and representing them vertically.
Now slide that neural network across the whole image, as a result, we will get another image with different widths, heights, and depths. Instead of just R, G, and B channels now we have more channels but lesser width and height. This operation is called **Convolution**. If the patch size is the same as that of the image it will be a regular neural network. Because of this small patch, we have fewer weights.



Mathematical Overview of Convolution

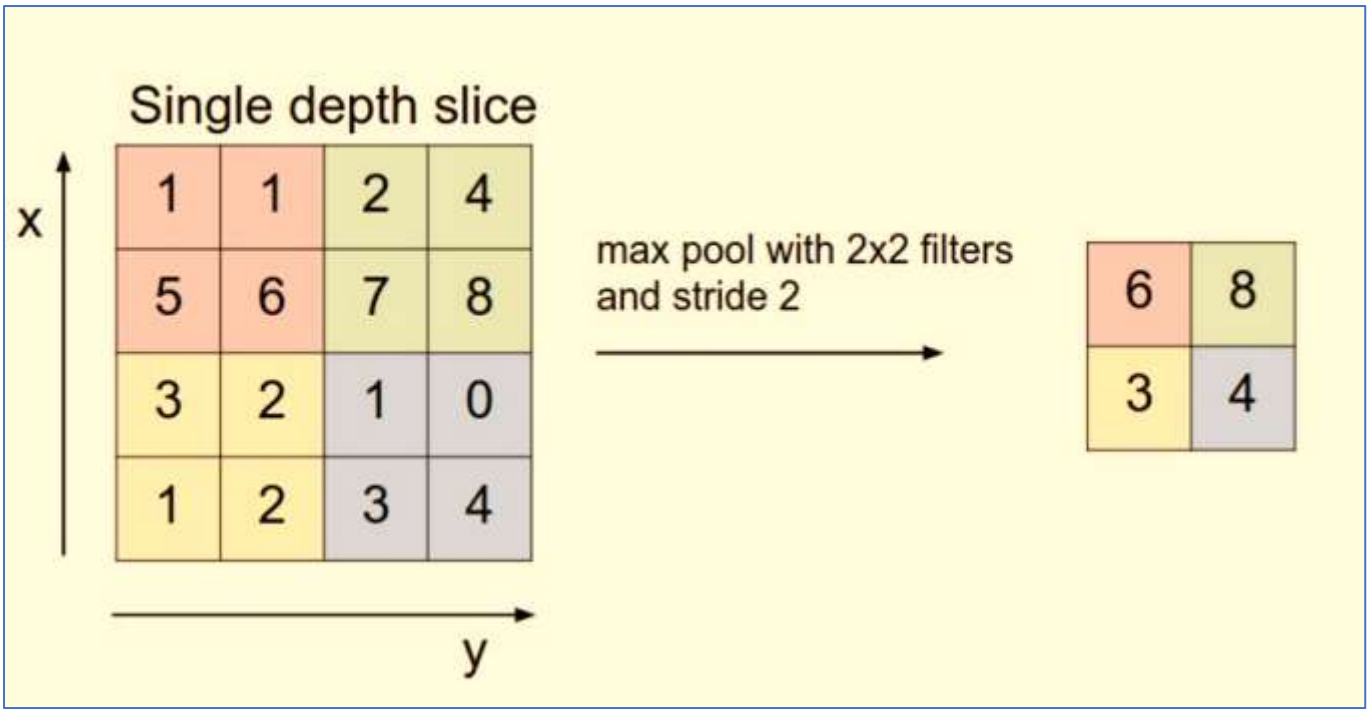
Now let's talk about a bit of mathematics that is involved in the whole convolution process.

- Convolution layers consist of a set of learnable filters (or kernels) having small widths and heights and the same depth as that of input volume (3 if the input layer is image input).
- For example, if we have to run convolution on an image with dimensions 34x34x3. The possible size of filters can be axax3, where 'a' can be anything like 3, 5, or 7 but smaller as compared to the image dimension.
- During the forward pass, we slide each filter across the whole input volume step by step where each step is called **stride** (which can have a value of 2, 3, or even 4 for high-dimensional images) and compute the dot product between the kernel weights and patch from input volume.
- As we slide our filters we'll get a 2-D output for each filter and we'll stack them together as a result, we'll get output volume having a depth equal to the number of filters. The network will learn all the filters.

Layers Used to Build ConvNets

A complete Convolution Neural Networks architecture is also known as covnets. A covnets is a sequence of layers, and every layer transforms one volume to another through a differentiable function.
Let's take an example by running a covnets on of image of dimension 32 x 32 x 3.

- **Input Layers:** It's the layer in which we give input to our model. In CNN, Generally, the input will be an image or a sequence of images. This layer holds the raw input of the image with width 32, height 32, and depth 3.
- **Convolutional Layers:** This is the layer, which is used to extract the feature from the input dataset. It applies a set of learnable filters known as the kernels to the input images. The filters/kernels are smaller matrices usually 2x2, 3x3, or 5x5 shape. it slides over the input image data and computes the dot product between kernel weight and the corresponding input image patch. The output of this layer is referred as feature maps. Suppose we use a total of 12 filters for this layer we'll get an output volume of dimension 32 x 32 x 12.
- **Activation Layer**: By adding an activation function to the output of the preceding layer, activation layers add nonlinearity to the network. it will apply an element-wise activation function to the output of the convolution layer. Some common activation functions are **RELU**: max(0, x), **Tanh**, **Leaky RELU**, etc. The volume remains unchanged hence output volume will have dimensions 32 x 32 x 12.
- **Pooling layer**: This layer is periodically inserted in the covnets and its main function is to reduce the size of volume which makes the computation fast reduces memory and also prevents overfitting. Two common types of pooling layers are **max pooling** and **average pooling**. If we use a max pool with 2 x 2 filters and stride 2, the resultant volume will be of dimension 16x16x12.



- **Flattening:** The resulting feature maps are flattened into a one-dimensional vector after the convolution and pooling layers so they can be passed into a completely linked layer for categorization or regression.
- **Fully Connected Layers:** It takes the input from the previous layer and computes the final classification or regression task.
- **Output Layer:** The output from the fully connected layers is then fed into a logistic function for classification tasks like sigmoid or softmax which converts the output of each class into the probability score of each class.

Example: Applying CNN to an Image

Step:

- import the necessary libraries
- set the parameter
- define the kernel
- Load the image and plot it.
- Reformat the image
- Apply convolution layer operation and plot the output image.
- Apply activation layer operation and plot the output image.
- Apply pooling layer operation and plot the output image.

```
# import the necessary libraries
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
from itertools import product

# set the param
plt.rc('figure', autolayout=True)
plt.rc('image', cmap='magma')

# define the kernel
kernel = tf.constant([[-1, -1, -1],
[-1,  8, -1],
[-1, -1, -1],
])

# load the image
image = tf.io.read_file('Ganesh.jpg')
image = tf.io.decode_jpeg(image, channels=1)
image = tf.image.resize(image, size=[300, 300])

# plot the image
img = tf.squeeze(image).numpy()
plt.figure(figsize=(5, 5))
plt.imshow(img, cmap='gray')
plt.axis('off')
plt.title('Original Gray Scale image')
plt.show();


# Reformat
image = tf.image.convert_image_dtype(image, dtype=tf.float32)
image = tf.expand_dims(image, axis=0)
kernel = tf.reshape(kernel, [*kernel.shape, 1, 1])
kernel = tf.cast(kernel, dtype=tf.float32)

# convolution layer
conv_fn = tf.nn.conv2d

image_filter = conv_fn(
input=image,
filters=kernel,
strides=1, # or (1, 1)
padding='SAME',
```

```
)

plt.figure(figsize=(15, 5))

# Plot the convolved image
plt.subplot(1, 3, 1)

plt.imshow(
tf.squeeze(image_filter)
)
plt.axis('off')
plt.title('Convolution')

# activation layer
relu_fn = tf.nn.relu
# Image detection
image_detect = relu_fn(image_filter)

plt.subplot(1, 3, 2)
plt.imshow(
# Reformat for plotting
tf.squeeze(image_detect)
)

plt.axis('off')
plt.title('Activation')

# Pooling layer
pool = tf.nn.pool
image_condense = pool(input=image_detect,
window_shape=(2, 2),
pooling_type='MAX',
strides=(2, 2),
padding='SAME',
)

plt.subplot(1, 3, 3)
plt.imshow(tf.squeeze(image_condense))
plt.axis('off')
plt.title('Pooling')
plt.show()
```

Digital Image Processing Basics

Digital Image Processing means processing digital image by means of a digital computer. We can also say that it is a use of computer algorithms, in order to get enhanced image either to extract some useful information.

Digital image processing is the use of algorithms and mathematical models to process and analyze digital images. The goal of digital image processing is to enhance the quality of images, extract meaningful information from images, and automate image-based tasks.

The basic steps involved in digital image processing are:

1. Image acquisition: This involves capturing an image using a digital camera or scanner, or importing an existing image into a computer.
2. Image enhancement: This involves improving the visual quality of an image, such as increasing contrast, reducing noise, and removing artifacts.
3. Image restoration: This involves removing degradation from an image, such as blurring, noise, and distortion.
4. Image segmentation: This involves dividing an image into regions or segments, each of which corresponds to a specific object or feature in the image.
5. Image representation and description: This involves representing an image in a way that can be analyzed and manipulated by a computer, and describing the features of an image in a compact and meaningful way.
6. Image analysis: This involves using algorithms and mathematical models to extract information from an image, such as recognizing objects, detecting patterns, and quantifying features.
7. Image synthesis and compression: This involves generating new images or compressing existing images to reduce storage and transmission requirements.
8. Digital image processing is widely used in a variety of applications, including medical imaging, remote sensing, computer vision, and multimedia.

Image processing mainly include the following steps:

1.Importing the image via image acquisition tools;
2.Analysing and manipulating the image;
3.Output in which result can be altered image or a report which is based on analysing that image.

What is an image?

An image is defined as a two-dimensional function,**F(x,y)**, where x and y are spatial coordinates, and the amplitude of **F** at any pair of coordinates (x,y) is called the **intensity** of that image at that point. When x,y, and amplitude values of **F** are finite, we call it a **digital image**. In other words, an image can be defined by a two-dimensional array specifically arranged in rows and columns.

Digital Image is composed of a finite number of elements, each of which elements have a particular value at a particular location.These elements are referred to as *picture elements,image elements,and pixels*.A *Pixel* is most widely used to denote the elements of a Digital Image.

Types of an image

1. **BINARY IMAGE**- The binary image as its name suggests, contain only two pixel elements i.e 0 & 1,where 0 refers to black and 1 refers to white. This image is also known as Monochrome.
2. **BLACK AND WHITE IMAGE**- The image which consist of only black and white color is called BLACK AND WHITE IMAGE.
3. **8 bit COLOR FORMAT**- It is the most famous image format.It has 256 different shades of colors in it and commonly known as Grayscale Image. In this format, 0 stands for Black, and 255 stands for white, and 127 stands for gray.

4. **16 bit COLOR FORMAT**- It is a color image format. It has 65,536 different colors in it.It is also known as High Color Format. In this format the distribution of color is not as same as Grayscale image.

A 16 bit format is actually divided into three further formats which are Red, Green and Blue. That famous RGB format.

Image as a Matrix

As we know, images are represented in rows and columns we have the following syntax in which images are represented:

$$f(x,y) = \begin{bmatrix} f(0,0) & f(0,1) & f(0,2) & \dots & f(0,N-1) \\ f(1,0) & f(1,1) & f(1,2) & \dots & f(1,N-1) \\ \vdots & \vdots & \vdots & & \vdots \\ f(M-1,0) & f(M-1,1) & f(M-1,2) & \dots & f(M-1,N-1) \end{bmatrix}$$

The right side of this equation is digital image by definition. Every element of this matrix is called image element , picture element , or pixel.

DIGITAL IMAGE REPRESENTATION IN MATLAB:

$$f = \begin{bmatrix} f(1,1) & f(1,2) & \cdots & f(1,N) \\ f(2,1) & f(2,2) & \cdots & f(2,N) \\ \vdots & \vdots & & \vdots \\ f(M,1) & f(M,2) & \cdots & f(M,N) \end{bmatrix}$$

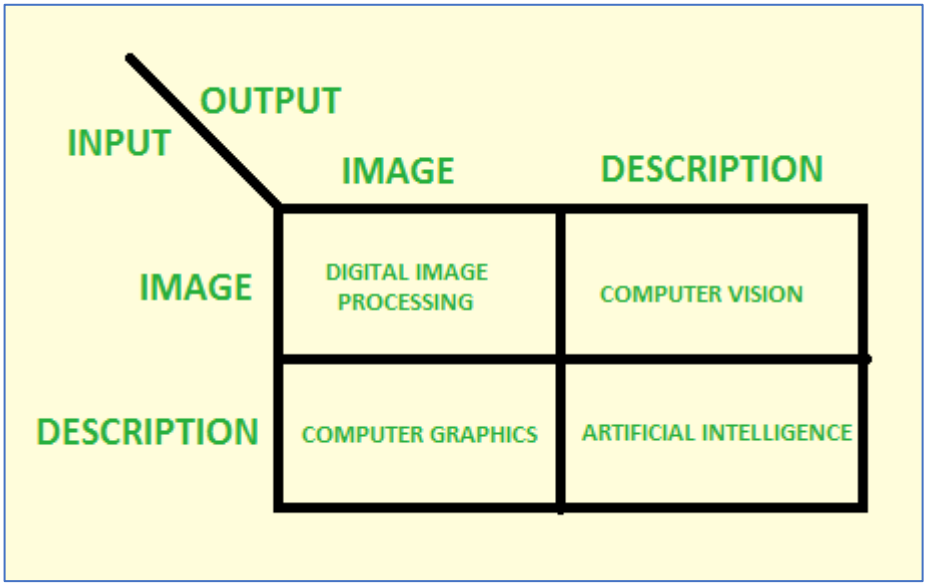
In MATLAB the start index is from 1 instead of 0. Therefore, f(1,1) = f(0,0). henceforth the two representation of image are identical, except for the shift in origin. In MATLAB, matrices are stored in a variable i.e X,x,input_image , and so on. The variables must be a letter as same as other programming languages.

PHASES OF IMAGE PROCESSING:

1.**ACQUISITION**- It could be as simple as being given an image which is in digital form. The main

work involves:
a) Scaling
b) Color conversion(RGB to Gray or vice-versa)
2.**IMAGE ENHANCEMENT**- It is amongst the simplest and most appealing in areas of Image Processing it is also used to extract some hidden details from an image and is subjective.
3.**IMAGE RESTORATION**- It also deals with appealing of an image but it is objective(Restoration is based on mathematical or probabilistic model or image degradation).
4.**COLOR IMAGE PROCESSING**- It deals with pseudocolor and full color image processing color models are applicable to digital image processing.
5.**WAVELETS AND MULTI-RESOLUTION PROCESSING**- It is foundation of representing images in various degrees.
6.**IMAGE COMPRESSION**-It involves in developing some functions to perform this operation. It mainly deals with image size or resolution.
7.**MORPHOLOGICAL PROCESSING**-It deals with tools for extracting image components that are useful in the representation & description of shape.
8.**SEGMENTATION PROCEDURE**-It includes partitioning an image into its constituent parts or objects. Autonomous segmentation is the most difficult task in Image Processing.
9.**REPRESENTATION & DESCRIPTION**-It follows output of segmentation stage, choosing a representation is only the part of solution for transforming raw data into processed data.
10.**OBJECT DETECTION AND RECOGNITION**-It is a process that assigns a label to an object based on its descriptor.

OVERLAPPING FIELDS WITH IMAGE PROCESSING



According to block 1,if input is an image and we get out image as a output, then it is termed as Digital Image Processing.
According to block 2,if input is an image and we get some kind of information or description as a output, then it is termed as Computer Vision.
According to block 3,if input is some description or code and we get image as an output, then it

is termed as Computer Graphics.
According to block 4,if input is description or some keywords or some code and we get description or some keywords as a output,then it is termed as Artificial Intelligence

Advantages of Digital Image Processing:

1. Improved image quality: Digital image processing algorithms can improve the visual quality of images, making them clearer, sharper, and more informative.
2. Automated image-based tasks: Digital image processing can automate many image-based tasks, such as object recognition, pattern detection, and measurement.
3. Increased efficiency: Digital image processing algorithms can process images much faster than humans, making it possible to analyze large amounts of data in a short amount of time.
4. Increased accuracy: Digital image processing algorithms can provide more accurate results than humans, especially for tasks that require precise measurements or quantitative analysis.

Disadvantages of Digital Image Processing:

1. High computational cost: Some digital image processing algorithms are computationally intensive and require significant computational resources.
2. Limited interpretability: Some digital image processing algorithms may produce results that are difficult for humans to interpret, especially for complex or sophisticated algorithms.
3. Dependence on quality of input: The quality of the output of digital image processing algorithms is highly dependent on the quality of the input images. Poor quality input images can result in poor quality output.
4. Limitations of algorithms: Digital image processing algorithms have limitations, such as the difficulty of recognizing objects in cluttered or poorly lit scenes, or the inability to recognize objects with significant deformations or occlusions.
5. Dependence on good training data: The performance of many digital image processing algorithms is dependent on the quality of the training data used to develop the algorithms. Poor quality training data can result in poor performance of the algorithm.

Importance of Convolutional Neural Network | ML

1. Weight Sharing and Local Spatial Coherence

CNNs take advantage of the spatial structure in data. Instead of learning separate parameters for each input feature, convolutional layers use a set of shared weights (kernels) that slide across the input image. This means the same feature detector is used at every position, drastically reducing the total number of parameters.

- **Benefit**: Significant reduction in computational cost, making CNNs efficient even on low-power GPUs or machines without dedicated graphics cards.

2. Memory Efficiency

Because of weight sharing and the nature of convolutions, CNNs generally require far fewer parameters than fully connected neural networks.

- **Example**: On the MNIST dataset, a simple CNN with one hidden layer and 10 output nodes might use only a few hundred parameters, whereas a fully connected neural network with similar capacity could require upwards of 19,000 parameters for the same task.
- **Benefit**: Lower memory requirements make CNNs suitable for devices with limited resources and also reduce overfitting risk.

3. Robustness to Local Variations

CNNs are specifically designed to extract features from different regions of an image, making them robust to small shifts and variations in the input. Example: If a fully connected network is trained for face recognition using only head-shot images, it may fail when presented with full-body images. However, a CNN can adapt to such changes and still recognize faces because it focuses on spatial features rather than exact positions.

4. Equivariance to Transformations

Equivariance refers to the property where a transformation applied to the input results in the same transformation in the output. In CNNs, convolution operations are equivariant to translations: if an object shifts in the image, its feature map also shifts correspondingly.

- **Mathematical Formulation**: If f is a convolution operation and g is a transformation (like translation), then $f(g(x))=g(f(x))$.
- **Benefit**: This property helps the model reliably detect features even when they move within the input, increasing reliability and consistency in predictions.

5. Independence from Image Transformations

CNNs exhibit relative invariance to common geometric transformations such as translation, rotation and scaling. This means they can recognize objects regardless of their orientation or size within the image.

- **Translation Invariance Example:** A CNN can detect the same object even if it's shifted to different parts of the image.
- **Rotation Invariance Example**: A CNN can still correctly identify an object

Properties of CNN and their Importance

Property	Why it Matters	CNN Advantage over Traditional Networks
Weight Sharing	Reduces parameters	Faster computation, less memory usage
Local Feature Extraction	Leverages spatial coherence	Robust to local variations
Equivariance	Predictable output changes	Reliable under input transformations
Memory Efficiency	Fewer model parameters	Lower risk of overfitting

CNN | Introduction to Padding

What Is Padding

padding is a technique used to preserve the spatial dimensions of the input image after convolution operations on a feature map. Padding involves adding extra pixels around the border of the input feature map before convolution.

This can be done in two ways:

- **Valid Padding**: In the valid padding, no padding is added to the input feature map, and the output feature map is smaller than the input feature map. This is useful when we want to reduce the spatial dimensions of the feature maps.
- **Same Padding**: In the same padding, padding is added to the input feature map such that the size of the output feature map is the same as the input feature map. This is useful when we want to preserve the spatial dimensions of the feature maps.

The number of pixels to be added for padding can be calculated based on the size of the kernel and the desired output of the feature map size. The most common padding value is zero-padding, which involves adding zeros to the borders of the input feature map.

Padding can help in reducing the loss of information at the borders of the input feature map and can improve the performance of the model. However, it also increases the computational cost of the convolution operation. Overall, padding is an important technique in CNNs that helps in preserving the spatial dimensions of the feature maps and can improve the performance of the model.

Problem With Convolution Layers Without Padding

- For a grayscale (n x n) image and (f x f) filter/kernel, the dimensions of the image resulting from a convolution operation is **(n - f + 1) x (n - f + 1)**. For example, for an (8 x 8) image and (3 x 3) filter, the output resulting after the convolution operation would be of size (6 x 6). Thus, the image shrinks every time a convolution operation is performed. This places an upper limit to the number of times such an operation could be performed before the image reduces to nothing thereby precluding us from building deeper networks.
- Also, the pixels on the corners and the edges are used much less than those in the middle.

  For example,
- Clearly, pixel A is touched in just one convolution operation and pixel B is touched in 3 convolution operations, while pixel C is touched in 9 convolution operations. In general, pixels in the middle are used more often than pixels on corners and edges. Consequently, the information on the borders of images is not preserved as well as the information in the middle.

Effect Of Padding On Input Images

Padding is simply a process of adding layers of zeros to our input images so as to avoid the problems mentioned above through the following changes to the input image.

padding in convolutional network

Padding prevents the shrinking of the input image.

***p** = number of layers of zeros added to the border of the image,*

*then **(n x n)** image **---> (n + 2p) x (n + 2p)** image after padding.*

***(n + 2p) x (n + 2p) * (f x f) -----> outputs (n + 2p - f + 1) x (n + 2p - f + 1)** images*

For example, by adding one layer of padding to an (8 x 8) image and using a (3 x 3) filter we would get an (8 x 8) output after performing a convolution operation.

This increases the contribution of the pixels at the border of the original image by bringing them into the middle of the padded image. Thus, information on the borders is preserved as well as the information in the middle of the image.

Types of Padding

Valid Padding: It implies no padding at all. The input image is left in its valid/unaltered shape. So

$$(n \times n) * (f \times f) \longrightarrow (n-f+1) \times (n-f+1) \quad (n \times n) * (f \times f) \longrightarrow (n-f+1) \times (n-f+1)$$

where, nxn is the dimension of input image

fxf is kernel size

n-f+1 is output image size

** represents a convolution operation.*

Same Padding: In this case, we add 'p' padding layers such that the output image has the same dimensions as the input image. So,

[(n + 2p) x (n + 2p) image] * [(f x f) filter] ---> [(n x n) image]

which gives **p = (f - 1) / 2** (because n + 2p - f + 1 = n).

What are Convolution Layers

Convolution layers are key building blocks of convolutional neural networks (CNNs) which are used in computer vision and image processing. They apply convolution operation to the input data which involves a filter (or kernel) that slides over the input data, performing element-wise multiplications and summing the results to produce a feature map. This process allows the network to detect patterns such as edges, textures and shapes in the input images.

Key Components of a Convolution Layer

1. Filters(Kernels):

- Small matrices that extract specific features from the input.
- For example, one filter might detect horizontal edges while another detects vertical edges.
- The values of filters are learned and updated during training.

2. Stride:

- Refers to the step size with which the filter moves across the input data.
- Larger strides result in smaller output feature maps and faster computation.

3. Padding:

- Zeros or other values may be added around the input to control the spatial dimensions of the output.
- Common types: "valid" (no padding) and "same" (pads output so feature map dimensions match input).

4. Activation Function:

- After convolution, a non-linear function like ReLU (Rectified Linear Unit) is often applied allowing the network to learn complex relationships in data.
- Common activations: ReLU, Tanh, Leaky ReLU.

Types of Convolution Layers

- **2D Convolution (Conv2D):** Most common for image data where filters slide in two dimensions (height and width) across the image.
- **Depthwise Separable Convolution:** Used for computational efficiency, applying depthwise and pointwise convolutions separately to reduce parameters and speed up computation.
- **Dilated (Atrous) Convolution:** Inserts spaces (zeros) between kernel elements to increase the receptive field without increasing computation, useful for tasks requiring context aggregation over larger areas.

Steps in a Convolution Layer

1. **Initialize Filters:** Randomly initialize a set of filters with learnable parameters.
2. **Convolve Filters with Input:** Slide the filters across the width and height of the input data, computing the dot product between the filter and the input sub-region.
3. **Apply Activation Function:** Apply a non-linear activation function to the convolved output to introduce non-linearity.
4. **Pooling (Optional):** Often followed by a pooling layer (like max pooling) to reduce the spatial dimensions of the feature map and retain the most important information.

Example Of Convolution Layer

Consider an input image of size 32x32x3 (32x32 pixels with 3 color channels). A convolution layer with ten 5x5 filters, a stride of 1 and 'same' padding will produce an output feature map of size 32x32x10. Each of the 10 filters detects different features in the input image.

Applications of Convolutional Layers

- **Image and Video Recognition:** Identifying objects, faces and scenes in images and videos.
- **Medical Imaging:** Detecting diseases in X-rays and MRIs.
- **Autonomous Vehicles:** Recognizing lanes, signs and obstacles.
- **NLP and Speech:** Sentiment analysis, text classification and speech recognition using 1D convolutions.
- **Industry and Business:** Quality control, fraud detection and product recommendations.

Convolutional Layers vs. Fully Connected Layers

Let's see the differences between Convolutional Layers vs. Fully Connected Layers,

Aspect	Convolutional Layers	Fully Connected Layers
Connectivity	Local (each neuron connects to local regions)	Global (each neuron connects to all inputs)
Parameter Count	Lower (weight sharing)	Higher
Spatial Information	Preserved (via convolution operations)	Lost (flattening removes spatial structure)
Typical Use	Feature extraction	Classification, regression

Benefits of Convolution Layers

- **Parameter Sharing:** The same filter is used repeatedly across the input, greatly reducing the number of parameters in the model compared to fully connected layers.
- **Local Connectivity:** Each filter focuses on a small local region, capturing fine-grained features and patterns.
- **Hierarchical Feature Learning:** Stacking multiple convolution layers enables the network to learn increasingly complex features—from low-level edges in early layers to entire objects in deeper layers.

- **Computational Efficiency:** Fewer parameters make convolution layers more efficient both in storage and computation allowing deep architectures suitable for large-scale visual tasks.

CNN | Introduction to Pooling Layer

Pooling layer is used in CNNs to reduce the spatial dimensions (width and height) of the input feature maps while retaining the most important information. It involves sliding a two-dimensional filter over each channel of a feature map and summarizing the features within the region covered by the filter.

For a feature map with dimensions $\mathrm{nh}\times\mathrm{nw}\times\mathrm{nc}$$nh\times nw\times nc$, the dimensions of the output after a pooling layer are:

$$(\mathrm{nh}-\mathrm{f}+1\mathrm{s})\times(\mathrm{nw}-\mathrm{f}+1\mathrm{s})\times\mathrm{nc}$$$$(snh-f+1)\times(snw-f+1)\times nc$$

where:

- $\mathrm{nh}$$nh \rightarrow$ height of the feature map
- $\mathrm{nw}$$nw \rightarrow$ width of the feature map
- $\mathrm{nc}$$nc \rightarrow$ number of channels in the feature map
- $\mathrm{f}$$f \rightarrow$ size of the pooling filter
- $\mathrm{s}$$s \rightarrow$ stride length

A typical CNN model architecture consists of multiple convolution and pooling layers stacked together.

Why are Pooling Layers Important?

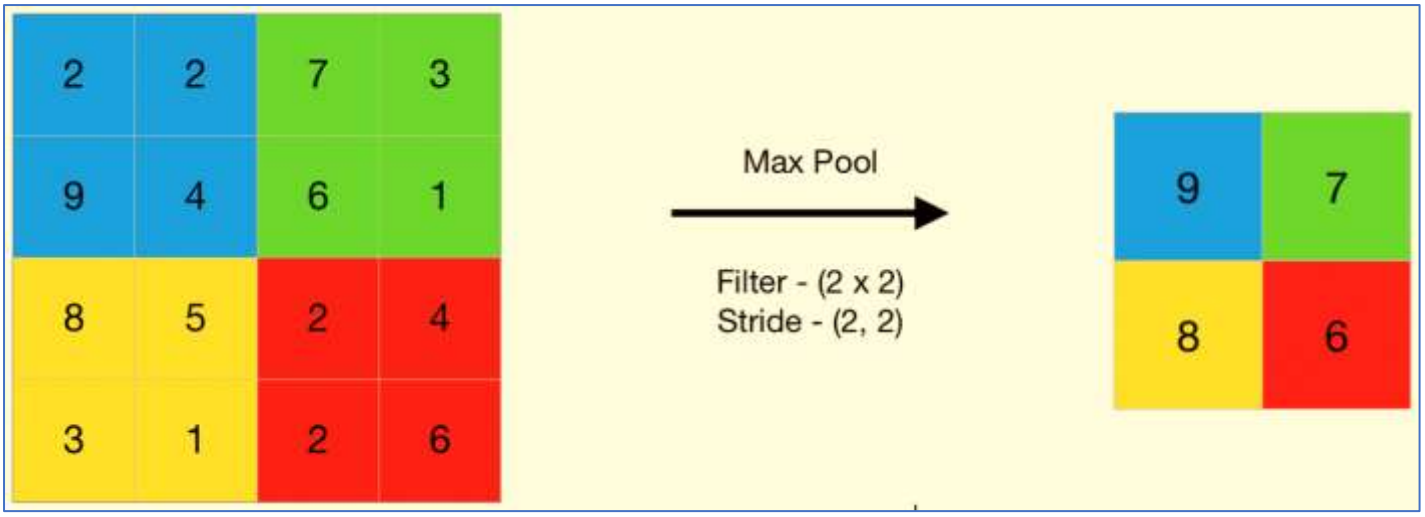
1. **Dimensionality Reduction**: Pooling layers reduce the spatial size of the feature maps, which decreases the number of parameters and computations in the network. This makes the model faster and more efficient.
2. **Translation Invariance**: Pooling helps the network become invariant to small translations or distortions in the input image. For example, even if an object in an image is slightly shifted, the pooled output will remain relatively unchanged.
3. **Overfitting Prevention**: By reducing the spatial dimensions, pooling layers help prevent overfitting by providing a form of regularization.
4. **Feature Hierarchy**: Pooling layers help build a hierarchical representation of features, where lower layers capture fine details and higher layers capture more abstract and global features.

Types of Pooling Layers

1. Max Pooling

Max pooling selects the maximum element from the region of the feature map covered by the filter. Thus, the output after max-pooling layer would be a feature map containing the most prominent features of the previous feature map.

Max pooling layer preserves the most important features (edges, textures, etc.) and provides better performance in most cases.



Max Pooling in Keras:

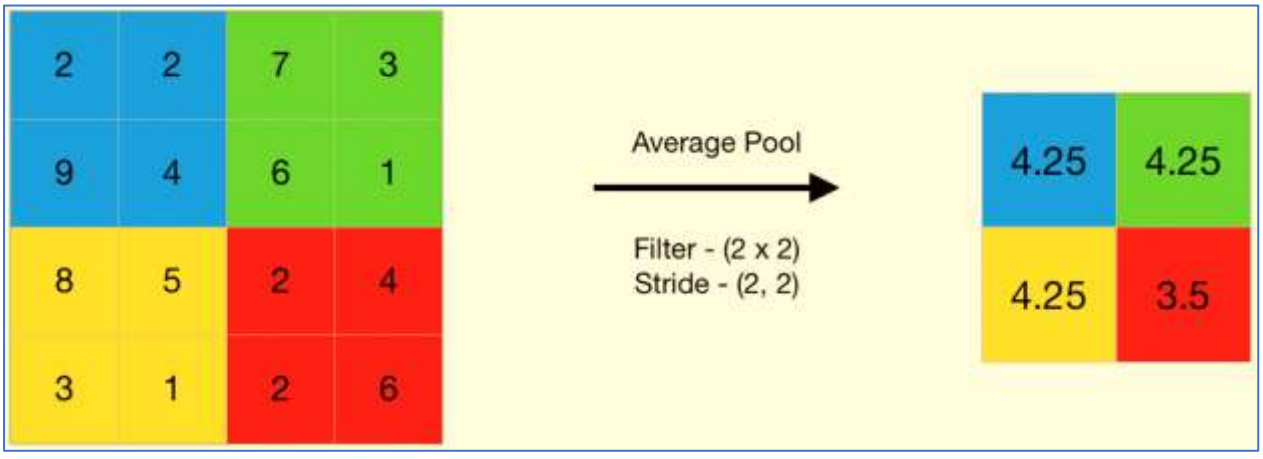
```
from tensorflow.keras.layers import MaxPooling2D
import numpy as np

# Example input feature map
feature_map = np.array([
[1, 3, 2, 9],
[5, 6, 1, 7],
[4, 2, 8, 6],
[3, 5, 7, 2]
]).reshape(1, 4, 4, 1)

# Applying max pooling
max_pool = MaxPooling2D(pool_size=(2, 2), strides=2)
output = max_pool(feature_map)

print(output.numpy().reshape(2, 2))
```

Output:

[[6 9]
[5 8]]

2. Average Pooling

Average pooling computes the average of the elements present in the region of feature map covered by the filter. Thus, while max pooling gives the most prominent feature in a particular patch of the feature map, average pooling gives the average of features present in a patch. Average pooling provides a more generalized representation of the input. It is useful in the cases where preserving the overall context is important.



Average Pooling using Keras:

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.layers import AveragePooling2D

feature_map = np.array([
[1, 3, 2, 9],
[5, 6, 1, 7],
[4, 2, 8, 6],
[3, 5, 7, 2]
], dtype=np.float32).reshape(1, 4, 4, 1)  # Convert to float32

# Applying average pooling
avg_pool = AveragePooling2D(pool_size=(2, 2), strides=2)
output = avg_pool(feature_map)
print(output.numpy().reshape(2, 2))
```

Output:

[[3.75 4.75]
[3.5 5.75]]

3. Global Pooling

Global pooling reduces each channel in the feature map to a single value, producing a $1\times1\times nc$ output. This is equivalent to applying a filter of size $nh\times nw$.

There are two types of global pooling:

- **Global Max Pooling**: Takes the maximum value across the entire feature map.
- **Global Average Pooling**: Computes the average of all values in the feature map.

Global Pooling using Keras:

```
from tensorflow.keras.layers import GlobalMaxPooling2D, GlobalAveragePooling2D

feature_map = np.array([
[1, 3, 2, 9],
[5, 6, 1, 7],
[4, 2, 8, 6],
[3, 5, 7, 2]
], dtype=np.float32).reshape(1, 4, 4, 1)

# Applying global max pooling
gm_pool = GlobalMaxPooling2D()
gm_output = gm_pool(feature_map)

# Applying global average pooling
ga_pool = GlobalAveragePooling2D()
ga_output = ga_pool(feature_map)

print("Global Max Pooling Output:", gm_output.numpy())
print("Global Average Pooling Output:", ga_output.numpy())
```

Output:

Global Max Pooling Output: [[9]]
Global Average Pooling Output: [[4.4375]]

How Pooling Layers Work?

1. **Define a Pooling Window (Filter)**: The size of the pooling window (e.g., 2x2) is chosen, along with a stride (the step size by which the window moves). A common choice is a 2x2 window with a stride of 2, which reduces the feature map size by half.
2. **Slide the Window Over the Input**: The pooling operation is applied to each region of the input feature map covered by the window.
3. **Apply the Pooling Operation**: Depending on the type of pooling (max, average, etc.), the operation extracts the required value from each window.
4. **Output the Downsampled Feature Map**: The result is a smaller feature map that retains the most important information.

Key Factors to Consider for Optimizing Pooling Layer

- **Pooling Window Size**: The size of the pooling window affects the degree of downsampling. A larger window results in more aggressive downsampling but may lose important details.
- **Stride**: The stride determines how much the pooling window moves at each step. A larger stride results in greater dimensionality reduction.
- **Padding**: In some cases, padding is used to ensure that the pooling operation covers the entire input feature map.

Advantages of Pooling Layer

1. **Dimensionality reduction:** Pooling layer helps in reducing the spatial dimensions of the feature maps. This reduces the computational cost and also helps in avoiding overfitting by reducing the number of parameters in the model.
2. **Translation invariance:** Pooling layers are useful in achieving translation invariance in the feature maps. This means that the position of an object in the image does not affect the classification result, as the same features are detected regardless of the position of the object.
3. **Feature selection:** Pooling layers help in selecting the most important features from the input, as max pooling selects the most salient features and average pooling provides a balanced representation.

Disadvantages of Pooling Layers

1. **Information Loss**: Pooling reduces spatial resolution, which can lead to a loss of important fine details.
2. **Over-smoothing**: Excessive pooling may blur out crucial features.
3. **Hyperparameter Tuning**: The choice of pooling size and stride affects performance and requires careful tuning.

What is Fully Connected Layer in Deep Learning

Fully Connected (FC) layers are also known as dense layers which are used in neural networks especially in of deep learning. They are a type of neural network layer where every neuron in the layer is connected to every neuron in the previous and subsequent layers. The "fully connected" descriptor comes from the fact that each of the neurons in these layers is connected to every activation in the previous layer creating a highly interconnected network.

- In CNNs fully connected layers often follow convolutional and pooling layers used to interpret the feature maps generated by these layers into the final output categories or predictions.
- In fully connected feedforward networks these layers are the main building blocks that directly process the input data into outputs.

Structure of Fully Connected Layers

The structure of FC layers is one of the most significant factors that define how it works in a neural network. This structure involves the fact that every neuron in one layer will interconnect with every neuron in the subsequent layer.

Dense (Fully Connected) Layer

Key Components of Fully Connected Layers

A Fully Connected layer is characterized by its dense interconnectivity. Here's a breakdown of its key components:

- **Neurons**: Basic units that receive inputs from all neurons of the previous layer and send outputs to all neurons of the subsequent layer.
- **Weights**: Each connection between neurons has an associated weight indicating the strength and influence of one neuron on another.
- **Biases**: A bias term for each neuron helps adjust the output along with the weighted sum of inputs.
- **Activation Function**: Functions like ReLU, Sigmoid or Tanh introduce non-linearity to the model helping it to learn complex patterns and behaviors.

Working and Structure of Fully Connected Layers in Neural Networks

The extensive connectivity allows for comprehensive information processing and feature integration making FC layers essential for tasks requiring complex pattern recognition.

Key Operations in Fully Connected Layers

1. Input Processing

Each neuron in an FC layer receives inputs from all neurons of the previous layer with each connection having a specific weight and each neuron incorporating a bias. The input to each neuron is a weighted sum of these inputs plus a bias:

$zj=\sum i(wij.xi)+bj$ $zj=\sum i(wij.xi)+bj$

Here wij wij is the weight from neuron i of the previous layer to neuron j, xi xi is the input from neuron i and bj bj is the bias for neuron j

2. Activation

The weighted sum is then processed through a non-linear activation function such as ReLU, Sigmoid or Tanh. This step introduces non-linearity enabling the network to learn complex functions:

$aj=f(zj)$ $aj=f(zj)$

f denotes the activation function transforming the linear combination of inputs into a non-linear output.

Example Configuration

Consider a neural network transition from a layer with 4 neurons to an FC layer with 3 neurons:

- **Previous Layer** (4 neurons) → **Fully Connected Layer** (3 neurons)

Each neuron in the FC layer receives inputs from all four neurons of the previous layer resulting in a configuration that involves 12 weights and 3 biases. This design of FC layer helps in transforming and combining features from the input layer hence helping in network's ability to perform complex decision-making tasks.

Key Role of Fully Connected Layers in Neural Networks

The key roles of fully connected layers in neural network are discussed below:

1. Feature Integration and Abstraction

FC layers consolidate features extracted by earlier layers (e.g., convolutional or recurrent), transforming them into a form suitable for accurate prediction by capturing complex patterns and relationships.

2. Decision Making and Output Generation

Typically used as the final layer in classification or regression tasks, FC layers convert high-level features into output scores. For classification, these scores are passed through Softmax to yield class probabilities.

3. Introduction of Non-Linearity

Activation functions like ReLU, Sigmoid, or Tanh applied in FC layers introduce non-linearity, allowing the network to learn complex, non-linear patterns and generalize effectively.

4. Universal Approximation

According to the Universal Approximation Theorem, an FC layer with enough neurons can approximate any continuous function, showcasing its power in modeling diverse problems.

5. Flexibility across Domains

FC layers are input-agnostic and versatile, applicable to various domains like vision, speech, and NLP, supporting both shallow and deep architectures.

6. Regularization and Overfitting Control

Techniques like Dropout and L2 regularization are crucial in FC layers to prevent overfitting, promoting generalization by reducing reliance on specific neurons or large weights.

Advantages of Fully Connected Layers

- **Integration of Features**: They are capable of combining all features before making predictions, essential for complex pattern recognition.
- **Flexibility**: FC layers can be incorporated into various network architectures and handle any form of input data provided it is suitably reshaped.
- **Simplicity**: These layers are straightforward to implement and are supported by all major deep learning frameworks.

Limitations of Fully Connected Layers

Despite their benefits FC layers have several drawbacks:

- **High Computational Cost**: The dense connections can lead to a large number of parameters, increasing both computational complexity and memory usage.
- **Prone to Overfitting**: Due to the high number of parameters they can easily overfit on smaller datasets unless techniques like dropout or regularization are used.
- **Inefficiency with Spatial Data**: Unlike convolutional layers, FC layers do not exploit the spatial hierarchy of images or other structured data, which can lead to less effective learning.

Backpropagation in Convolutional Neural Networks

Understanding Backpropagation

Backpropagation, short for "*backward propagation of errors*," is an algorithm used to calculate the gradient of the loss function of a neural network with respect to its weights. It is essentially a method to update the weights to minimize the loss. Backpropagation is crucial because it tells us how to change our weights to improve our network's performance.

Role of Backpropagation in CNNs

In a CNN, backpropagation plays a crucial role in fine-tuning the filters and weights during training, allowing the network to better differentiate features in the input data. CNNs typically consist of multiple layers, including convolutional layers, pooling layers, and fully connected layers. Each of these layers has weights and biases that are adjusted via backpropagation.

Fundamentals of Backpropagation

Backpropagation, in essence, is an application of the chain rule from calculus used to compute the gradients (partial derivatives) of a loss function with respect to the weights of the network.

The process involves three main steps: the forward pass, loss calculation, and the backward pass.

The Forward Pass

During the forward pass, input data (e.g., an image) is passed through the network to compute the output. For a CNN, this involves several key operations:

1. **Convolutional Layers**: Each convolutional layer applies numerous filters to the input. For a given layer ll with filters denoted by FF, input II, and bias bb, the output OO is given by: O=(I∗F)+b$O=(I*F)+b$ Here, ∗∗ denotes the convolution operation.
2. **Activation Functions**: After convolution, an activation function σ\sigmaσ (e.g., ReLU) is applied element-wise to introduce non-linearity: O=σ((I∗F)+b)$O=\sigma((I*F)+b)$
3. **Pooling Layers**: Pooling (e.g., max pooling) reduces dimensionality, summarizing the features extracted by the convolutional layers.

Loss Calculation

After computing the output, a loss function LL is calculated to assess the error in prediction. Common loss functions include mean squared error for regression tasks or cross-entropy loss for classification:

L=−∑ylog⁡(y^)$L=-\sum y\log(y^)$

Here, yy is the true label, and y^$y^$ is the predicted label.

The Backward Pass (Backpropagation)

The backward pass computes the gradient of the loss function with respect to each weight in the network by applying the chain rule:

1. **Gradient with respect to output**: First, calculate the gradient of the loss function with respect to the output of the final layer: ∂L∂O$\partial L\partial O$
2. **Gradient through activation function**: Apply the chain rule through the activation function: ∂L∂I=∂L∂O∂O∂I$\partial L\partial I=\partial L\partial O\partial O\partial I$ For ReLU, ∂O∂I$\partial O\partial I$ is 1 for I>0$I>0$ and 0 otherwise.
3. **Gradient with respect to filters in convolutional layers**: Continue applying the chain rule to find the gradients with respect to the filters:∂L∂F=∂L∂O∗rot180(I)$\partial L\partial F=\partial L\partial O*rot180(I)$Here, rot180(I)$rot180(I)$ rotates the input by 180 degrees, aligning it for the convolution operation used to calculate the gradient with respect to the filter.

Weight Update

Using the gradients calculated, the weights are updated using an optimization algorithm such as SGD:

Fnew=Fold−η∂L∂F$Fnew=Fold-\eta\partial L\partial F$

Here, ηη is the learning rate, which controls the step size during the weight update.

Challenges in Backpropagation

Vanishing Gradients

In deep networks, backpropagation can suffer from the vanishing gradient problem, where gradients become too small to make significant changes in weights, stalling the training. Advanced activation functions like ReLU and optimization techniques such as batch normalization are used to mitigate this issue.

Exploding Gradients

Conversely, gradients can become excessively large; this is known as exploding gradients. This can be controlled by techniques such as gradient clipping.

Implementation of a CNN based Image Classifier using PyTorch

Deep learning has revolutionized computer vision applications making it possible to classify and interpret images with good accuracy. We will perform a practical step-by-step implementation of a convolutional neural network (CNN) for image classification using PyTorch on CIFAR-10 dataset.

Step 1: Importing Libraries and Setting Up

To build our model, we first import PyTorch libraries and prepare the environment for visualization and data handling.

- **torch (PyTorch):** Enables building, training and running deep learning models using tensors.
- **torchvision:** Supplies standard vision datasets, image transforms and visualization utilities.
- **matplotlib.pyplot:** Plots images, graphs and visual representations of data and results.
- **numpy:** Provides efficient array operations and mathematical utilities for data processing.
- **ssl:** Adjusts security settings to bypass certificate errors during dataset downloads.
- Set up global plot parameters and SSL context to prevent download errors.

```
import torch
import torchvision
import matplotlib.pyplot as plt
import numpy as np
import ssl

ssl._create_default_https_context = ssl._create_unverified_context
plt.rcParams['figure.figsize'] = 14, 6
```

Step 2: Defining Data Transformations and Loading CIFAR-10

We define a normalization transformation, scaling pixel values to have mean 0.5 and standard deviation 0.5 per channel. We then download and load the CIFAR-10 dataset for both training and testing, applying the transform.

```
normalize_transform = torchvision.transforms.Compose([
torchvision.transforms.ToTensor(),
torchvision.transforms.Normalize(mean=(0.5, 0.5, 0.5), std=(0.5, 0.5, 0.5))
])

train_dataset = torchvision.datasets.CIFAR10(
root="./CIFAR10/train",        train=True,        transform=normalize_transform,
download=True)
test_dataset = torchvision.datasets.CIFAR10(
root="./CIFAR10/test",        train=False,        transform=normalize_transform,
download=True)
```

Step 3: Creating Data Loaders

- Set batch size to 128 for efficiency.
- Create data loaders for both train and test sets to manage batching and easy iteration.

```
batch_size = 128
train_loader = torch.utils.data.DataLoader(
train_dataset, batch_size=batch_size)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=batch_size)
```

Step 4: Visualizing Sample Images

- Obtain a batch of images and labels from the train loader.
- Display a grid of 25 training images for visual confirmation of the data pipeline.

```
dataiter = iter(train_loader)
```

```
images, labels = next(dataiter)
plt.imshow(np.transpose(torchvision.utils.make_grid(
images[:25], normalize=True, padding=1, nrow=5).numpy(), (1, 2, 0)))
plt.axis('off')
plt.show()
```

Step 5: Analyzing Dataset Class Distribution

- Collect all class labels from training data.
- Count occurrences for every class and visualize with a bar chart, revealing class balance.

```
classes = []
for batch_idx, data in enumerate(train_loader):
x, y = data
classes.extend(y.tolist())

unique, counts = np.unique(classes, return_counts=True)
names = list(test_dataset.class_to_idx.keys())
plt.bar(names, counts)
plt.xlabel("Target Classes")
plt.ylabel("Number of training instances")
plt.show()
```

Step 6: Building the CNN Architecture

Build a convolutional neural network (CNN) using PyTorch modules:

- Three sets of convolution, activation (ReLU) and max pooling layers.
- Flatten the features and add two fully connected layers.
- Output layer predicts class scores for 10 classes.

```
class CNN(torch.nn.Module):
def __init__(self):
super().__init__()
self.model = torch.nn.Sequential(
torch.nn.Conv2d(in_channels=3, out_channels=32,
kernel_size=3, padding=1),
torch.nn.ReLU(),
torch.nn.MaxPool2d(kernel_size=2),
torch.nn.Conv2d(in_channels=32, out_channels=64,
kernel_size=3, padding=1),
torch.nn.ReLU(),
torch.nn.MaxPool2d(kernel_size=2),
torch.nn.Conv2d(in_channels=64, out_channels=64,
kernel_size=3, padding=1),
torch.nn.ReLU(),
torch.nn.MaxPool2d(kernel_size=2),
torch.nn.Flatten(),
torch.nn.Linear(64 * 4 * 4, 512),
torch.nn.ReLU(),
torch.nn.Linear(512, 10)
)

def forward(self, x):
return self.model(x)
```

Step 7: Configuring the Training Process

- Select computation device: GPU if available, otherwise CPU.

- Instantiate the model and move it to the selected device.
- Number of training epochs (50)

```
device = 'cuda' if torch.cuda.is_available() else 'cpu'
model = CNN().to(device)

num_epochs = 50
learning_rate = 0.001
weight_decay = 0.01
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(
model.parameters(), lr=learning_rate, weight_decay=weight_decay)
```

Step 8: Training the Model

- Train the CNN through all epochs.
- Set model to training mode.
- For each batch, move data to device, compute predictions and loss, backpropagate and update parameters.
- Accumulate and record mean loss per epoch.

```
train_loss_list = []
for epoch in range(num_epochs):
print(f'Epoch {epoch+1}/{num_epochs}:', end=' ')
train_loss = 0
model.train()
for images, labels in train_loader:
images = images.to(device)
labels = labels.to(device)
outputs = model(images)
loss = criterion(outputs, labels)
optimizer.zero_grad()
loss.backward()
optimizer.step()
train_loss += loss.item()
train_loss_list.append(train_loss / len(train_loader))
print(f"Training loss = {train_loss_list[-1]}")
```

Step 9: Plotting Training Loss

Visualizing the learning curve by plotting average loss against every epoch.

```
plt.plot(range(1, num_epochs + 1), train_loss_list)
plt.xlabel("Number of epochs")
plt.ylabel("Training loss")
plt.show()
```

Step 10: Evaluating Model Accuracy

- Switch model to evaluation mode and disable gradient calculations.
- For each test batch, compute predictions and accumulate number of correct classifications.
- Calculate and print total accuracy as percentage of correctly classified test images.

```
test_acc = 0
model.eval()
with torch.no_grad():
for images, labels in test_loader:
images = images.to(device)
```

```
y_true = labels.to(device)
outputs = model(images)
_, y_pred = torch.max(outputs.data, 1)
test_acc += (y_pred == y_true).sum().item()

print(f"Test set accuracy = {100 * test_acc / len(test_dataset)} %")
```
Output: Test set accuracy = 71.94 %

Step 11: Visualizing Model Predictions

- From a test batch, select a few images and gather their actual and predicted class names.
- Show these images using a grid, with a title indicating both actual and predicted labels.

```
num_images = 5
y_true_name = [names[y_true[idx]] for idx in range(num_images)]
y_pred_name = [names[y_pred[idx]] for idx in range(num_images)]
title = f"Actual labels: {y_true_name}, Predicted labels: {y_pred_name}"

plt.imshow(np.transpose(torchvision.utils.make_grid(
images[:num_images].cpu(), normalize=True, padding=1).numpy(), (1, 2, 0)))
plt.title(title)
plt.axis("off")
plt.show()
```

2. Recurrent Neural Networks (RNNs)

Recurrent Neural Networks (RNNs) are a class of neural networks that are used for modeling sequence data such as time series or natural language.

- Recurrent Neural Networks (RNNs)
- How RNN Differs from Feedforward Neural Networks
- Backpropagation Through Time (BPTT)
- Vanishing Gradient and Exploding Gradient Problem
- Training of RNN in TensorFlow
- Sentiment Analysis with RNN

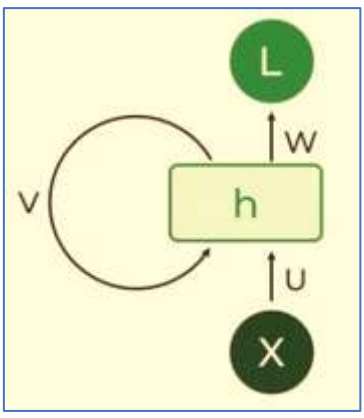
Introduction to Recurrent Neural Networks

Key Components of RNNs

There are mainly two components of RNNs that we will discuss.

1. Recurrent Neurons

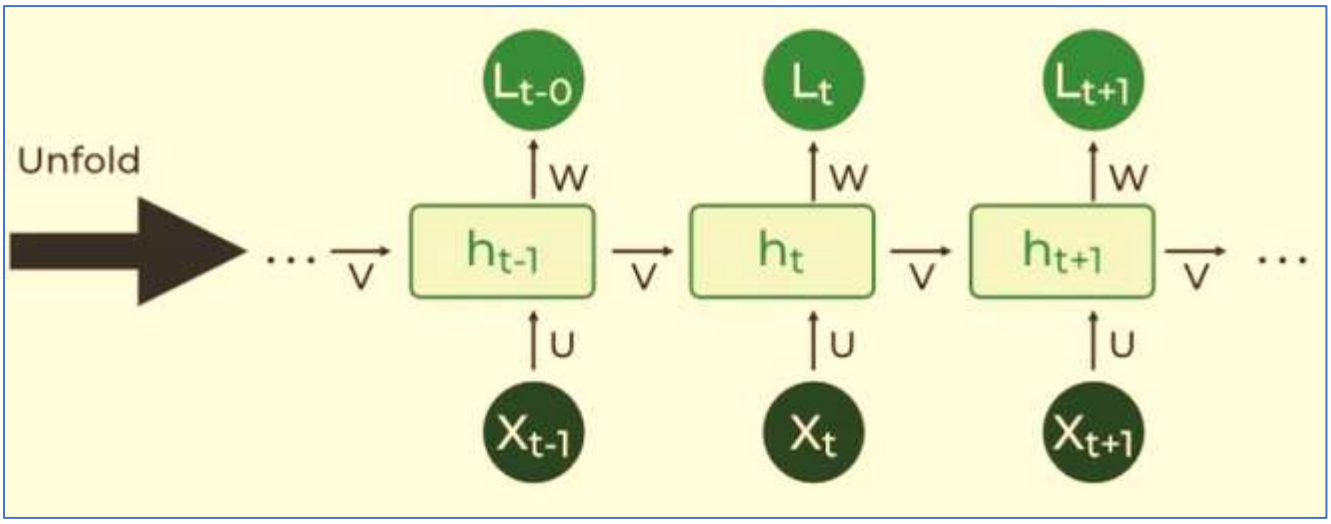
The fundamental processing unit in RNN is a **Recurrent Unit.** They hold a hidden state that maintains information about previous inputs in a sequence. Recurrent units can "remember" information from prior steps by feeding back their hidden state, allowing them to capture dependencies across time.



2. RNN Unfolding

RNN unfolding or unrolling is the process of expanding the recurrent structure over time steps. During unfolding each step of the sequence is represented as a separate layer in a series illustrating how information flows across each time step.

This unrolling enables **backpropagation through time (BPTT)** a learning process where errors are propagated across time steps to adjust the network's weights enhancing the RNN's ability to learn dependencies within sequential data.



Recurrent Neural Network Architecture

RNNs share similarities in input and output structures with other deep learning architectures but differ significantly in how information flows from input to output. Unlike traditional deep neural networks where each dense layer has distinct weight matrices. RNNs use shared weights across time steps, allowing them to remember information over sequences.

In RNNs the hidden state H_i is calculated for every input X_i to retain sequential dependencies. The computations follow these core formulas:

1. Hidden State Calculation:

$h = \sigma(U \cdot X + W \cdot h_{t-1} + B)$

Here:

- h represents the current hidden state.
- U and W are weight matrices.
- B is the bias.

2. Output Calculation:

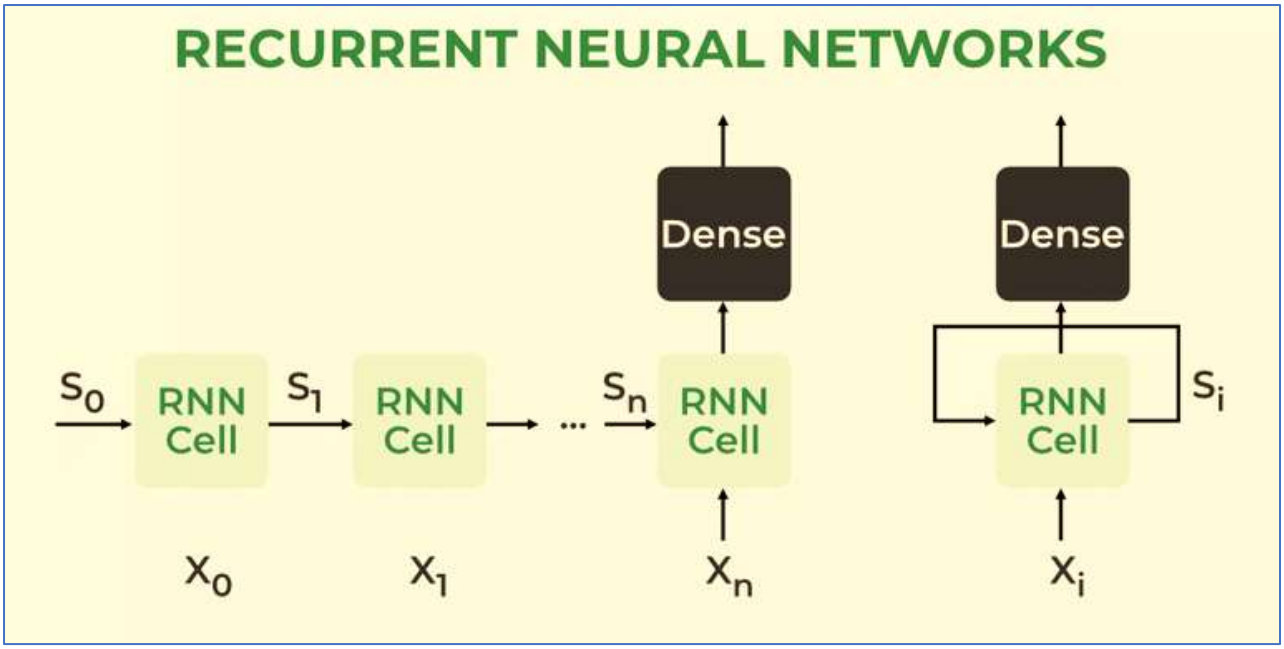
$Y = O(V \cdot h + C)$

The output Y is calculated by applying O an activation function to the weighted hidden state where V and C represent weights and bias.

3. Overall Function:

$Y = f(X, h, W, U, V, B, C)$

This function defines the entire RNN operation where the state matrix S holds each element s_i representing the network's state at each time step i.



Recurrent Neural Architecture

How does RNN work?

At each time step RNNs process units with a fixed activation function. These units have an internal hidden state that acts as memory that retains information from previous time steps. This memory allows the network to store past knowledge and adapt based on new inputs.

Updating the Hidden State in RNNs

The current hidden state h_t depends on the previous state h_{t-1} and the current input x_t and is calculated using the following relations:

1. State Update:

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

where:

- h_t is the current state
- h_{t-1} is the previous state
- x_t is the input at the current time step

2. Activation Function Application:

$$h_t = \tanh(W_{hh} \cdot h_{t-1} + W_{xh} \cdot x_t)$$

Here, W_{hh} is the weight matrix for the recurrent neuron and W_{xh} is the weight matrix for the input neuron.

3. Output Calculation:

y_t = W_{hy} \cdot h_t

where y_t is the output and W_{hy} is the weight at the output layer.

These parameters are updated using backpropagation. However, since RNN works on sequential data here we use an updated backpropagation which is known as **backpropagation through time**.

Backpropagation Through Time (BPTT) in RNNs

Since RNNs process sequential data **Backpropagation Through Time (BPTT)** is used to update the network's parameters. The loss function L(θ) depends on the final hidden state h_3 and each hidden state relies on preceding ones forming a sequential dependency chain:

h_3 depends on \text{ depends on } h_2, \, h_2 \text{ depends on } h_1, \, \dots, \, h_1 \text{ depends on } h_0.

Backpropagation Through Time (BPTT) In RNN

In BPTT, gradients are backpropagated through each time step. This is essential for updating network parameters based on temporal dependencies.

1. Simplified Gradient Calculation:

$$\frac{\partial L(\theta)}{\partial W} = \frac{\partial L (\theta)}{\partial h_3} \cdot \frac{\partial h_3}{\partial W}$$

2. Handling Dependencies in Layers: Each hidden state is updated based on its dependencies:

$$h_3 = \sigma(W \cdot h_2 + b)$$

The gradient is then calculated for each state, considering dependencies from previous hidden states.

3. Gradient Calculation with Explicit and Implicit Parts: The gradient is broken down into explicit and implicit parts summing up the indirect paths from each hidden state to the weights.

$$\frac{\partial h_3}{\partial W} = \frac{\partial h_3^{+}}{\partial W} + \frac{\partial h_3}{\partial h_2} \cdot \frac{\partial h_2^{+}}{\partial W}$$

4. Final Gradient Expression: The final derivative of the loss function with respect to the weight matrix W is computed:

$$\frac{\partial L(\theta)}{\partial W} = \frac{\partial L(\theta)}{\partial h_3} \cdot \sum_{k=1}^{3} \frac{\partial h_3}{\partial h_k} \cdot \frac{\partial h_k}{\partial W}$$

This iterative process is the essence of backpropagation through time.

Types Of Recurrent Neural Networks

There are four types of RNNs based on the number of inputs and outputs in the network:

1. One-to-One RNN

This is the simplest type of neural network architecture where there is a single input and a single output. It is used for straightforward classification tasks such as binary classification where no sequential data is involved.

2. One-to-Many RNN

In a One-to-Many RNN the network processes a single input to produce multiple outputs over time. This is useful in tasks where one input triggers a sequence of predictions (outputs). For example in image captioning a single image can be used as input to generate a sequence of words as a caption.

3. Many-to-One RNN

The **Many-to-One RNN** receives a sequence of inputs and generates a single output. This type is useful when the overall context of the input sequence is needed to make one prediction. In sentiment analysis the model receives a sequence of words (like a sentence) and produces a single output like positive, negative or neutral.

4. Many-to-Many RNN

The **Many-to-Many RNN** type processes a sequence of inputs and generates a sequence of outputs. In language translation task a sequence of words in one language is given as input and a corresponding sequence in another language is generated as output.

Variants of Recurrent Neural Networks (RNNs)

There are several variations of RNNs, each designed to address specific challenges or optimize for certain tasks:

1. Vanilla RNN

This simplest form of RNN consists of a single hidden layer where weights are shared across time steps. Vanilla RNNs are suitable for learning short-term dependencies but are limited by the vanishing gradient problem, which hampers long-sequence learning.

2. Bidirectional RNNs

Bidirectional RNNs process inputs in both forward and backward directions, capturing both past and future context for each time step. This architecture is ideal for tasks where the entire sequence is available, such as named entity recognition and question answering.

3. Long Short-Term Memory Networks (LSTMs)

Long Short-Term Memory Networks (LSTMs) introduce a memory mechanism to overcome the vanishing gradient problem. Each LSTM cell has three gates:

- **Input Gate**: Controls how much new information should be added to the cell state.
- **Forget Gate**: Decides what past information should be discarded.
- **Output Gate**: Regulates what information should be output at the current step. This selective memory enables LSTMs to handle long-term dependencies, making them ideal for tasks where earlier context is critical.

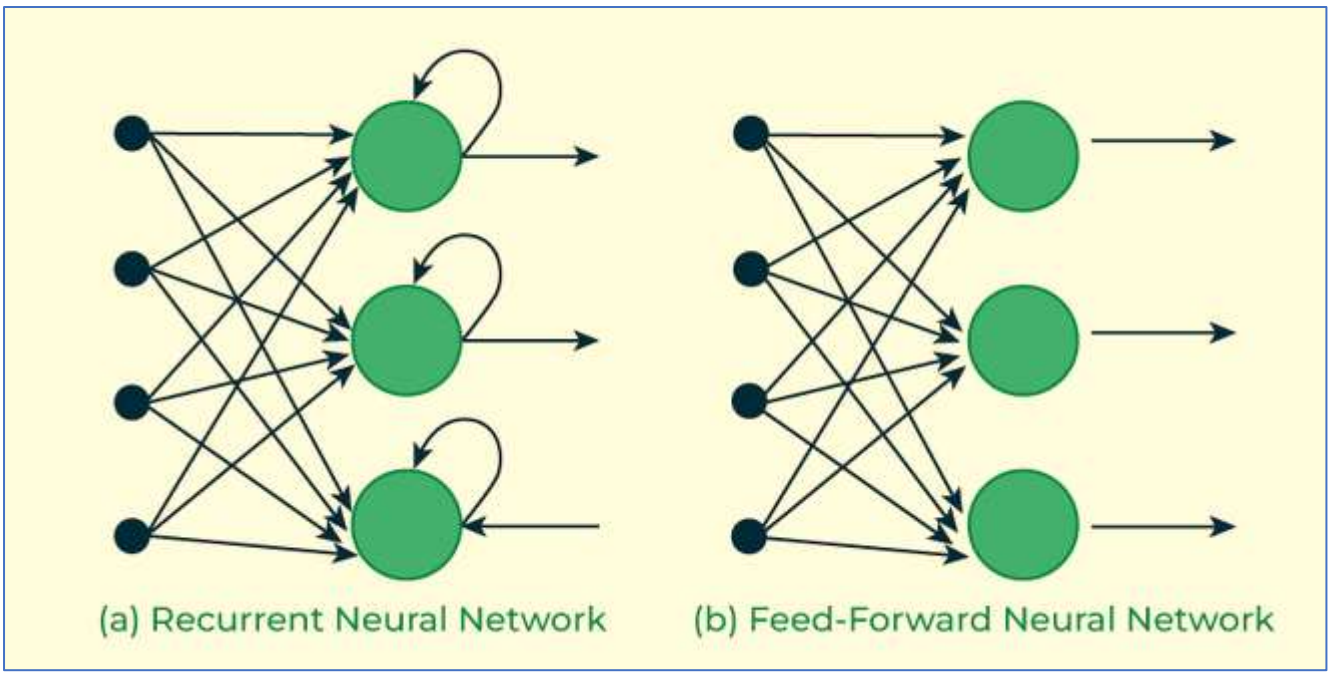
4. Gated Recurrent Units (GRUs)

Gated Recurrent Units (GRUs) simplify LSTMs by combining the input and forget gates into a single update gate and streamlining the output mechanism. This design is computationally efficient, often performing similarly to LSTMs and is useful in tasks where simplicity and faster training are beneficial.

How RNN Differs from Feedforward Neural Networks?

Feedforward Neural Networks (FNNs) process data in one direction from input to output without retaining information from previous inputs. This makes them suitable for tasks with independent inputs like image classification. However FNNs struggle with sequential data since they lack memory.

Recurrent Neural Networks (RNNs) solve this **by incorporating loops that allow information from previous steps to be fed back into the network**. This feedback enables RNNs to remember prior inputs making them ideal for tasks where context is important.



Recurrent Vs Feedforward networks

Implementing a Text Generator Using Recurrent Neural Networks (RNNs)

In this section, we create a character-based text generator using Recurrent Neural Network (RNN) in TensorFlow and Keras. We'll implement an RNN that learns patterns from a text sequence to generate new text character-by-character.

1. Importing Necessary Libraries

We start by importing essential libraries for data handling and building the neural network.

```
import numpy as np
```

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import SimpleRNN, Dense
```

2. Defining the Input Text and Prepare Character Set

We define the input text and identify unique characters in the text which we'll encode for our model.

```
text = "This is Tech a software training institute"
chars = sorted(list(set(text)))
char_to_index = {char: i for i, char in enumerate(chars)}
index_to_char = {i: char for i, char in enumerate(chars)}
```

3. Creating Sequences and Labels

To train the RNN, we need sequences of fixed length (`seq_length`) and the character following each sequence as the label.

```
seq_length = 3
sequences = []
labels = []

for i in range(len(text) - seq_length):
seq = text[i:i + seq_length]
label = text[i + seq_length]
sequences.append([char_to_index[char] for char in seq])
labels.append(char_to_index[label])

X = np.array(sequences)
y = np.array(labels)
```

4. Converting Sequences and Labels to One-Hot Encoding

For training we convert `X` and `y` into one-hot encoded tensors.

```
X_one_hot = tf.one_hot(X, len(chars))
y_one_hot = tf.one_hot(y, len(chars))
```

5. Building the RNN Model

We create a simple RNN model with a hidden layer of 50 units and a Dense output layer with softmax activation.

```
model = Sequential()
model.add(SimpleRNN(50, input_shape=(seq_length, len(chars)),
activation='relu'))
model.add(Dense(len(chars), activation='softmax'))
```

6. Compiling and Training the Model

We compile the model using the `categorical_crossentropy` loss and train it for 100 epochs.

```
model.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])
model.fit(X_one_hot, y_one_hot, epochs=100)
```

Output:

Training the RNN model

```
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 29ms/step - accuracy: 0.9618 - loss: 0.0664
Epoch 94/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 26ms/step - accuracy: 0.9514 - loss: 0.0729
Epoch 95/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 26ms/step - accuracy: 0.9514 - loss: 0.0661
Epoch 96/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 26ms/step - accuracy: 0.9514 - loss: 0.0655
Epoch 97/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 28ms/step - accuracy: 0.9514 - loss: 0.0652
Epoch 98/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 26ms/step - accuracy: 0.9618 - loss: 0.0648
Epoch 99/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 29ms/step - accuracy: 0.9618 - loss: 0.0590
Epoch 100/100
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 26ms/step - accuracy: 0.9618 - loss: 0.0583
<keras.src.callbacks.history.History at 0x7a5e40a739d0>
```

7. Generating New Text Using the Trained Model

After training we use a starting sequence to generate new text character by character.

```
start_seq = "This is G"
generated_text = start_seq

for i in range(50):
x = np.array([[char_to_index[char] for char in generated_text[-seq_length:]]])
x_one_hot = tf.one_hot(x, len(chars))
prediction = model.predict(x_one_hot)
next_index = np.argmax(prediction)
next_char = index_to_char[next_index]
generated_text += next_char

print("Generated Text:")
print(generated_text)
```

Output:

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 37ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 36ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 42ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 39ms/step
Generated Text:
This is Geeks a software training institutetraining institu
```

Predicting the next word

Advantages of Recurrent Neural Networks

- **Sequential Memory**: RNNs retain information from previous inputs making them ideal for time-series predictions where past data is crucial.
- **Enhanced Pixel Neighborhoods**: RNNs can be combined with convolutional layers to capture extended pixel neighborhoods improving performance in image and video data processing.

Limitations of Recurrent Neural Networks (RNNs)

While RNNs excel at handling sequential data they face two main training challenges i.e vanishing gradient and exploding gradient problem:

1. **Vanishing Gradient**: During backpropagation gradients diminish as they pass through each time step leading to minimal weight updates. This limits the RNN's ability to learn long-term dependencies which is crucial for tasks like language translation.
2. **Exploding Gradient**: Sometimes gradients grow uncontrollably causing excessively large weight updates that de-stabilize training.

These challenges can hinder the performance of standard RNNs on complex, long-sequence tasks.

Applications of Recurrent Neural Networks

RNNs are used in various applications where data is sequential or time-based:

- **Time-Series Prediction**: RNNs excel in forecasting tasks, such as stock market predictions and weather forecasting.
- **Natural Language Processing (NLP)**: RNNs are fundamental in NLP tasks like language modeling, sentiment analysis and machine translation.
- **Speech Recognition**: RNNs capture temporal patterns in speech data, aiding in speech-to-text and other audio-related applications.
- **Image and Video Processing**: When combined with convolutional layers, RNNs help analyze video sequences, facial expressions and gesture recognition.

Difference Between Feed-Forward Neural Networks and Recurrent Neural Networks

Neural networks have become essential tools in solving complex machine learning tasks. Among them most widely used architectures are Feed-Forward Neural Networks (FNNs) and Recurrent Neural Networks (RNNs). While both are capable of learning patterns from data, they are structurally and functionally different.

Feed-Forward Neural Networks

Feed-forward neural networks is a type of neural network where the connections between nodes do not form cycles. It processes input data in one direction i.e from input to output, without any feedback loops.

- No memory of previous inputs.
- Best suited for static data (e.g., images).
- Simple and fast to train.
- Cannot handle sequences or time dependencies.

Basic Example:

Used in classification tasks like identifying handwritten digits using the MNIST dataset.

Recurrent Neural Networks

Recurrent neural networks add a missing element from feed-forward networks i.e memory. They can remember information from previous steps, making them ideal for sequential data where context matters.

- Has memory of previous inputs using hidden states.
- Ideal for sequential data like text, speech, time series.
- Can suffer from vanishing gradient problems.
- More complex and slower to train.

Basic Example:

Used in language modeling such as predicting the next word in a sentence.

Key Differences

Feature	Feed-Forward Neural Network (FNN)	Recurrent Neural Network (RNN)

Data Flow	One-way (input → output)	Cyclic (can loop over previous states)
Memory	No memory	Has memory via hidden states
Best For	Static input (images, tabular data)	Sequential input (text, audio, time series)
Complexity	Lower	Higher
Training Time	Faster	Slower due to time dependencies
Gradient Issues	Less prone	Can suffer from vanishing/exploding gradients
Example Use Cases	Image classification, object detection	Sentiment analysis, speech recognition

When to Use Each Architecture

Feed-Forward Networks are ideal for:

- Image classification where each image is independent
- Medical diagnosis where patient symptoms don't depend on previous patients
- Credit scoring as current application doesn't depend on previous applications
- Any problem where inputs are independent

RNNs are ideal for:

- Language translation where word order matters
- Stock price prediction as today's price depends on yesterday's
- Weather forecasting as tomorrow's weather depends on today's
- Speech recognition

Computational Considerations

Feed-Forward Networks

- **Simple Structure**: Feed-forward networks follow a straight path from input to output. This makes them easier to implement and tune.
- **Parallel Computation**: Inputs can be processed in batches, enabling fast training using modern hardware.
- **Efficient Backpropagation**: They use standard backpropagation which is stable and well-supported across frameworks.
- **Lower Resource Use**: No memory of past inputs means less overhead during training and inference.

Recurrent Neural Networks

- **Sequential Nature**: RNNs process data step-by-step, this limits parallelism and slows down training.
- **Harder to Train**: Training uses Backpropagation Through Time (BPTT) which can be unstable and slower.
- **Captures Temporal Patterns**: They are suited for sequential data but require careful tuning to learn long-term dependencies.
- **Higher Compute Demand**: Maintaining hidden states and learning over time steps makes RNNs more resource-intensive.

Limitations and Challenges

Limitation	Feed-Forward Neural Network	Recurrent Neural Network (RNN)
Input Handling	Cannot handle variable-length input sequences	Supports sequences but struggles with long ones
Memory	No memory of previous inputs	Limited memory; forgets long-term context

Temporal Modeling	Ineffective at capturing time-based patterns	Can model temporal patterns but with difficulty
Performance Issues	Good parallelism, but lacks temporal context	Sequential nature slows training and inference
Training Challenges	Relatively stable	Prone to vanishing gradient and unstable training

Back Propagation through time - RNN

RNN Architecture

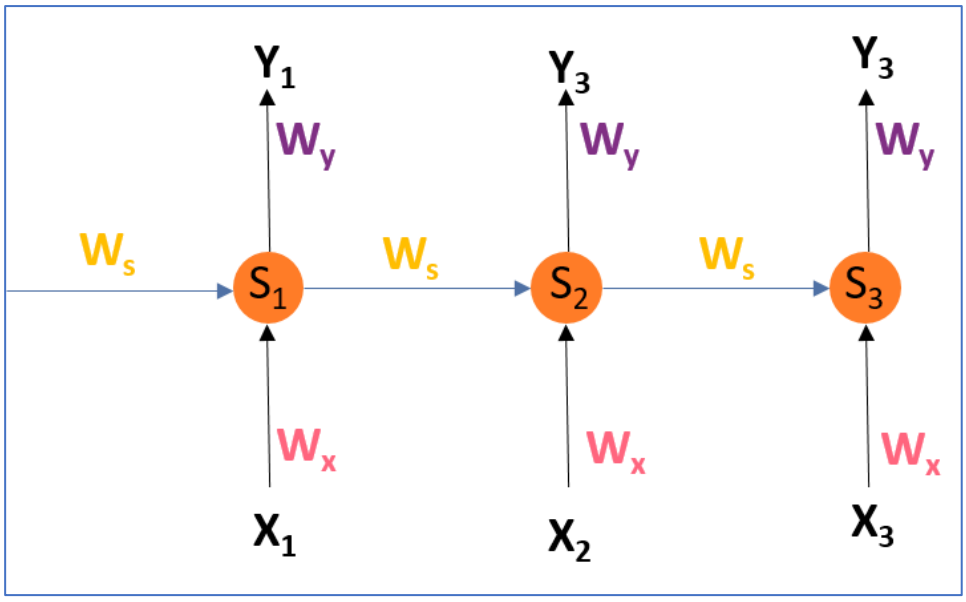
At each timestep t, the RNN maintains a hidden state St, which acts as the network's memory summarizing information from previous inputs. The hidden state St updates by combining the current input Xt and the previous hidden state$St-1$, applying an activation function to introduce non-linearity. Then the output $Yt$is generated by transforming this hidden state.

$St=g1(WxXt+WsSt-1)$

- St represents the hidden state (memory) at time t.
- Xt is the input at time t.
- Yt is the output at time t.
- Ws,Wx,Wy are weight matrices for hidden states, inputs and outputs, respectively.

$Yt=g2(WySt)$

where $g1$ and $g2$ are activation functions.



Error Function at Time $t=3$

To train the network, we measure how far the predicted output Yt is from the desired output dt using an error function. We use the squared error to measure the difference between the desired output dt and actual output Yt:

$Et=(dt-Yt)2$

At $t=3$:

$E3=(d3-Y3)2$

This error quantifies the difference between the predicted output and the actual output at time 3.

Updating Weights Using BPTT

BPTT updates the weights Wy,Ws,WxWy,Ws,Wx to minimize the error by computing gradients. Unlike standard backpropagation, BPTT unfolds the network across time steps, considering how errors at time tt depend on all previous states.

We want to adjust the weights WyWy, WsWs and WxWx to minimize the error E3E3.

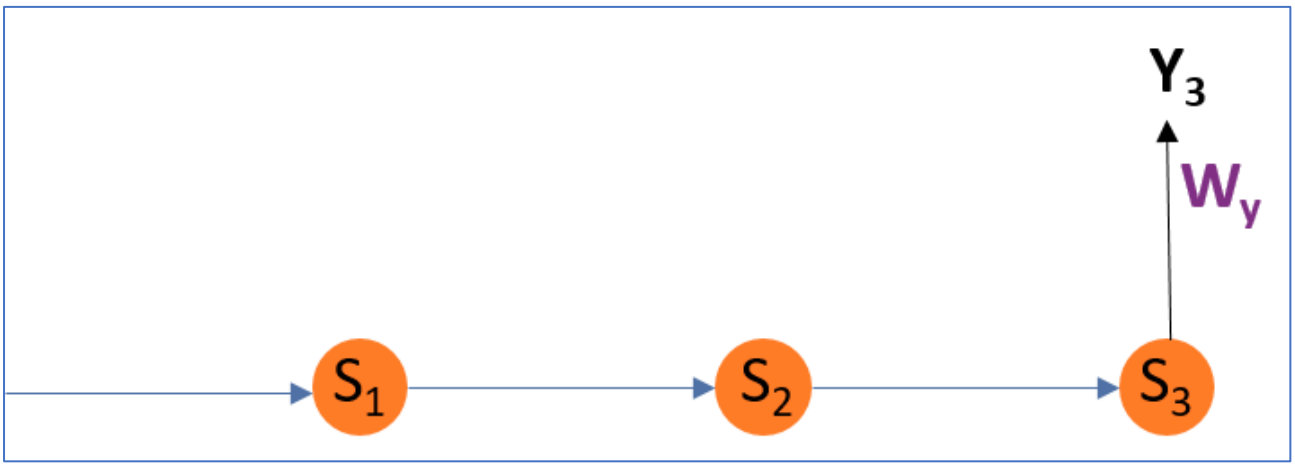
1. Adjusting Output Weight WyWy

The output weight WyWy affects the output directly at time 3. This means we calculate how the error changes as Y3Y3 changes, then how Y3Y3 changes with respect to WyWy. Updating WyWy is straightforward because it only influences the current output.

Using the chain rule:

∂E3∂Wy=∂E3∂Y3×∂Y3∂Wy∂Wy∂E3=∂Y3∂E3×∂Wy∂Y3

- E3E3 depends on Y3Y3, so we differentiate E3E3 w.r.t. Y3Y3.
- Y3Y3 depends on WyWy, so we differentiate Y3Y3 w.r.t. WyWy.



Adjusting Wy

2. Adjusting Hidden State Weight WsWs

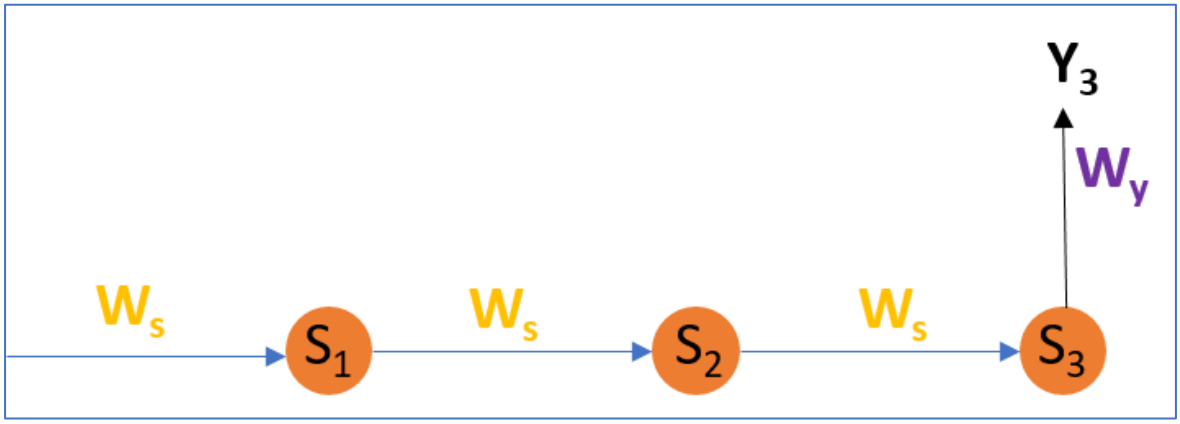
The hidden state weight WsWs influences not just the current hidden state but all previous ones because each hidden state depends on the previous one. To update WsWs, we must consider how changes to WsWs affect all hidden statesS1,S2,S3S1,S2,S3 and consequently the output at time 3.

The gradient for WsWs considers all previous hidden states because each hidden state depends on the previous one:

∂E3∂Ws=∑i=13∂E3∂Y3×∂Y3∂Si×∂Si∂Ws∂Ws∂E3=∑i=13∂Y3∂E3×∂Si∂Y3×∂Ws∂Si

Breaking down:

- Start with the error gradient at output Y3Y3.
- Propagate gradients back through all hidden states S3,S2,S1S3,S2,S1 since they affect Y3Y3.
- Each SiSi depends on WsWs, so we differentiate accordingly.



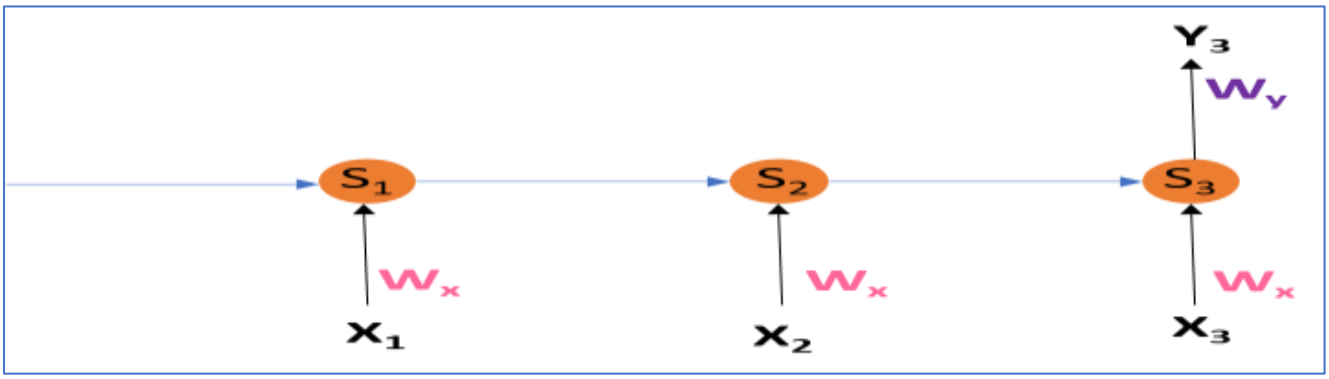
3. Adjusting Input Weight WxWx

Similar to WsWs, the input weight WxWx affects all hidden states because the input at each timestep shapes the hidden state. The process considers how every input in the sequence impacts the hidden states leading to the output at time 3.

$\partial E3\partial Wx=\sum i=13\partial E3\partial Y3\times\partial Y3\partial Si\times\partial Si\partial Wx\partial Wx\partial E3=\sum i=13\partial Y3\partial E3\times\partial Si\partial Y3\times\partial Wx\partial Si$

The process is similar to WsWs, accounting for all previous hidden states because inputs at each timestep affect the hidden states.

Adjusting Wx



Advantages of Backpropagation Through Time (BPTT)

- **Captures Temporal Dependencies:** BPTT allows RNNs to learn relationships across time steps, crucial for sequential data like speech, text and time series.
- **Unfolding over Time:** By considering all previous states during training, BPTT helps the model understand how past inputs influence future outputs.
- **Foundation for Modern RNNs:** BPTT forms the basis for training advanced architectures such as LSTMs and GRUs, enabling effective learning of long sequences.
- **Flexible for Variable Length Sequences**: It can handle input sequences of varying lengths, adapting gradient calculations accordingly.

Vanishing and Exploding Gradients Problems in Deep Learning

What is Vanishing Gradient?

The vanishing gradient problem is a challenge that emerges during backpropagation when the derivatives or slopes of the activation functions become progressively smaller as we move backward through the layers of a neural network. This phenomenon is particularly prominent in deep networks with many layers, hindering the effective training of the model. The weight updates becomes extremely tiny, or even exponentially small, it can significantly prolong the training time, and in the worst-case scenario, it can halt the training process altogether.

Why the Problem Occurs?

During backpropagation, the gradients propagate back through the layers of the network, they decrease significantly. This means that as they leave the output layer and return to the input layer, the gradients become progressively smaller. As a result, the weights associated with the initial levels, which accommodate these small gradients, are updated little or not at each iteration of the optimization process.

The vanishing gradient problem is particularly associated with the sigmoid and hyperbolic tangent (tanh) activation functions because their derivatives fall within the range of 0 to 0.25 and 0 to 1, respectively. Consequently, extreme weights becomes very small, causing the updated weights to closely resemble the original ones. This persistence of small updates contributes to the vanishing gradient issue.

The sigmoid and tanh functions limit the input values to the ranges [0,1] and [-1,1], so that they saturate at 0 or 1 for sigmoid and -1 or 1 for Tanh. The derivatives at points becomes zero as

they are moving. In these regions, especially when inputs are very small or large, the gradients are very close to zero. While this may not be a major concern in shallow networks with a few layers, it is a more pronounced issue in deep networks. When the inputs fall in saturated regions, the gradients approach zero, resulting in little update to the weights of the previous layer. In simple networks this does not pose much of a problem, but as more layers are added, these small gradients, which multiply between layers, decay significantly and consequently the first layer tears very slowly , and hinders overall model performance and can lead to convergence failure.

How can we identify?

Identifying the vanishing gradient problem typically involves monitoring the training dynamics of a deep neural network.

- One key indicator is observing model weights **converging to 0** or stagnation in the improvement of the model's performance metrics over training epochs.
- During training, if the **loss function fails to decrease** significantly, or if there is erratic behavior in the learning curves, it suggests that the gradients may be vanishing.
- Additionally, examining the gradients themselves during backpropagation can provide insights. **Visualization techniques**, such as gradient histograms or norms, can aid in assessing the distribution of gradients throughout the network.

How can we solve the issue?

- **Batch Normalization** : Batch normalization normalizes the inputs of each layer, reducing internal covariate shift. This can help stabilize and accelerate the training process, allowing for more consistent gradient flow.
- **Activation function**: Activation function like **Rectified Linear Unit (ReLU)** can be used. With **ReLU,** the gradient is 0 for negative and zero input, and it is 1 for positive input, which helps alleviate the vanishing gradient issue. Therefore, ReLU operates by replacing poor enter values with 0, and 1 for fine enter values, it preserves the input unchanged.
- **Skip Connections and Residual Networks (ResNets)**: Skip connections, as seen in ResNets, allow the gradient to bypass certain layers during backpropagation. This facilitates the flow of information through the network, preventing gradients from vanishing.
- **Long Short-Term Memory Networks (LSTMs) and Gated Recurrent Units (GRUs)**: In the context of recurrent neural networks (RNNs), architectures like LSTMs and GRUs are designed to address the vanishing gradient problem in sequences by incorporating gating mechanisms .
- **Gradient Clipping**: Gradient clipping involves imposing a threshold on the gradients during backpropagation. Limit the magnitude of gradients during backpropagation, this can prevent them from becoming too small or exploding, which can also hinder learning.

Build and train a model for Vanishing Gradient Problem

let's see how the problems occur , and way to handle them.

Step 1: Import Libraries

First, import the necessary libraries for model

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import tensorflow as tf
import keras
```

```
from sklearn.model_selection import train_test_split
from keras.layers import Dense
from keras.models import Sequential
import seaborn as sns
from imblearn.over_sampling import RandomOverSampler
from keras.layers import Dense, Dropout
from keras.optimizers import Adam
from keras.callbacks import EarlyStopping
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import classification_report
from keras.initializers import random_normal
from keras.constraints import max_norm
from keras.optimizers import SGD
```

Step 2: Loading dataset

The code loads two CSV files (Credit_card.csv and Credit_card_label.csv) into Pandas DataFrames, df and labels.

Link to dataset: Credit card details Binary classification.

```
df = pd.read_csv('/content/Credit_card.csv')
labels = pd.read_csv('/content/Credit_card_label.csv')
```

Step 3: Data Preprocessing

We create a new column 'Approved' in the DataFrame by converting the 'label' column from the 'labels' DataFrame to integers.

```
dep = 'Approved'

df[dep] = labels.label.astype(int)

df.loc[df[dep] == 1, 'Status'] = 'Approved'
df.loc[df[dep] == 0, 'Status'] = 'Declined'
```

Step 4: Feature Engineering

We perform some feature engineering on the data, creating new columns 'Age', 'EmployedDaysOnly', and 'UnemployedDaysOnly' based on existing columns.

It converts categorical variables in the 'cats' list to numerical codes using pd.Categorical and fills missing values with the mode of each column.

```
cats = [
'GENDER', 'Car_Owner', 'Propert_Owner', 'Type_Income',
'EDUCATION', 'Marital_status', 'Housing_type', 'Mobile_phone',
'Work_Phone', 'Phone', 'Type_Occupation', 'EMAIL_ID'
]

conts = [
'CHILDREN', 'Family_Members', 'Annual_income',
'Age', 'EmployedDaysOnly', 'UnemployedDaysOnly'
]
def proc_data():
df['Age'] = -df.Birthday_count // 365
df['EmployedDaysOnly'] = df.Employed_days.apply(lambda x: x if x > 0 else 0)
df['UnemployedDaysOnly'] = df.Employed_days.apply(lambda x: abs(x) if x < 0 else 0)

for cat in cats:
df[cat] = pd.Categorical(df[cat])
```

```
modes = df.mode().iloc[0]
df.fillna(modes, inplace=True)

proc_data()
```

Step 5: Oversampling due to heavily skewed data and Data Splitting

```
X = df[cats + conts]
y = df[dep]
X_over, y_over = RandomOverSampler().fit_resample(X, y)

X_train, X_val, y_train, y_val = train_test_split(X_over, y_over, test_size=0.25)
```

Step 6: Encoding

The code applies the cat.codes method to each column specified in the cats list. The method is applicable to Pandas categorical data types and assigns a unique numerical code to each unique category in the categorical variable. The result is that the categorical variables are replaced with their corresponding numerical codes.

```
X_train[cats] = X_train[cats].apply(lambda x: x.cat.codes)
X_val[cats] = X_val[cats].apply(lambda x: x.cat.codes)
```

Step 7: Model Creation

Create a Sequential model using Keras. A Sequential model allows you to build a neural network by stacking layers one after another.

```
model = Sequential()
```

Step 8: Adding layers

Adding 10 dense layers to the model. Each dense layer has 10 units (neurons) and uses the sigmoid activation function. The first layer specifies input_dim=18, indicating that the input data has 18 features. This is the input layer. The last layer has a single neuron and uses sigmoid activation, making it suitable for binary classification tasks.

```
model.add(Dense(10, activation='sigmoid', input_dim=18))
model.add(Dense(10, activation='sigmoid'))
model.add(Dense(10, activation='sigmoid'))
model.add(Dense(10, activation='sigmoid'))
model.add(Dense(10, activation='sigmoid'))
model.add(Dense(10, activation='sigmoid'))
model.add(Dense(10, activation='sigmoid'))
model.add(Dense(10, activation='sigmoid'))
model.add(Dense(10, activation='sigmoid'))
model.add(Dense(1, activation='sigmoid'))
```

Step 9: Model Compilation

This step specifies the loss function, optimizer, and evaluation metrics.

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

Step 10: Model training

Train the model using the training data (X_train and y_train) for 100 epochs.The training history is stored in the history object, which contains information about the training process, including loss and accuracy at each epoch.

```
history = model.fit(X_train, y_train, epochs=100)
```

Output:

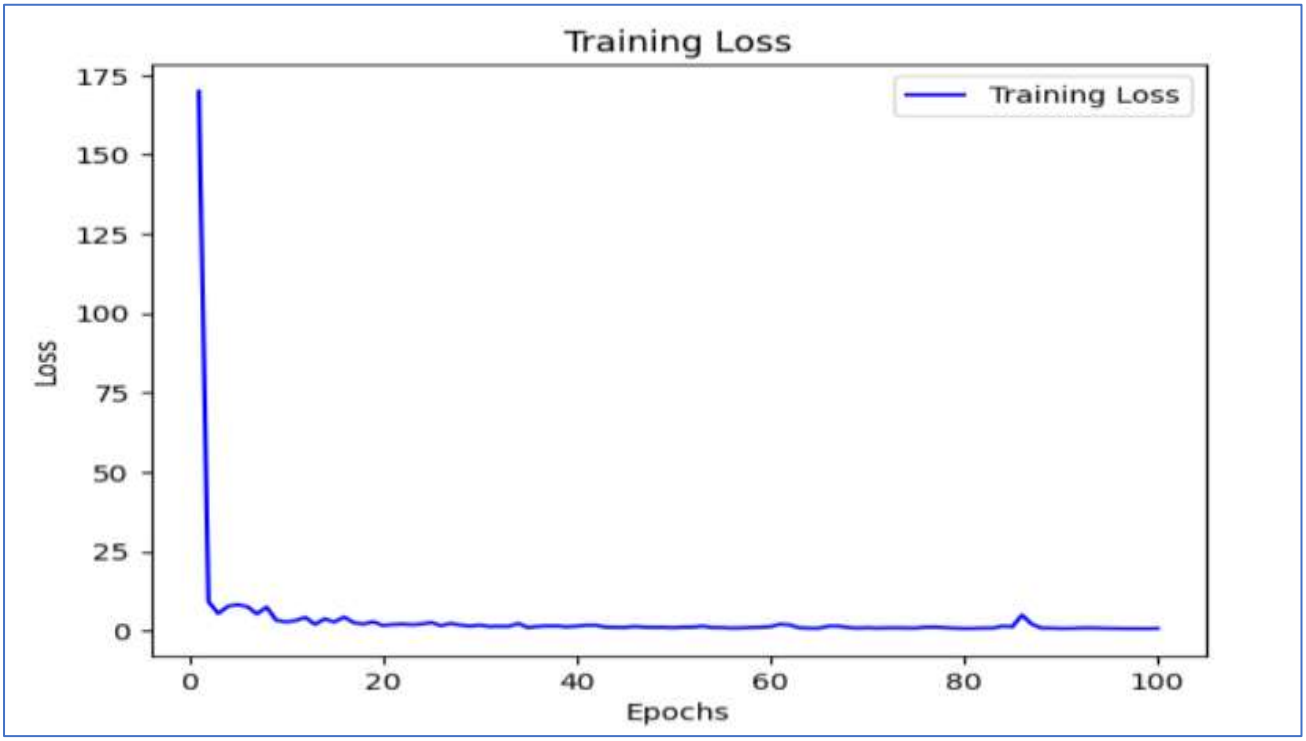
```
Epoch                                                                  1/100
65/65 [==============================] - 3s 3ms/step - loss: 0.7027 - accuracy:
0.5119
Epoch                                                                  2/100
65/65 [==============================] - 0s 3ms/step - loss: 0.6936 - accuracy:
```

```
0.5119
Epoch                                                                3/100
65/65 [==============================] - 0s 3ms/step - loss: 0.6933 - accuracy:
0.5119
.
.
Epoch                                                               97/100
65/65 [==============================] - 0s 3ms/step - loss: 0.6930 - accuracy:
0.5119
Epoch                                                               98/100
65/65 [==============================] - 0s 3ms/step - loss: 0.6930 - accuracy:
0.5119
Epoch                                                               99/100
65/65 [==============================] - 0s 3ms/step - loss: 0.6932 - accuracy:
0.5119
Epoch                                                              100/100
65/65 [==============================] - 0s 3ms/step - loss: 0.6929 - accuracy:
0.5119
```

Step 11: Plotting the training loss

```
loss = history.history['loss']
epochs = range(1, len(loss) + 1)
plt.plot(epochs, loss, 'b', label='Training Loss')
plt.title('Training Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

Output:



Loss does not change much as gradient becomes too small

Solution for Vanishing Gradient Problem

Step 1: Scaling

```
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_val_scaled = scaler.transform(X_val)
```

Step 2: Modify the Model

1. Deeper Architecture: Augment model with more layers with increased numbers of neurons in each layer. Deeper architectures can capture more complex relationships in the data.
2. Early Stopping: Early stopping is implemented to monitor the validation loss. Training will stop if the validation loss does not improve for a certain number of epochs (defined by patience).
3. Increased Dropout: Dropout layers are added after each dense layer to help prevent overfitting.
4. Adjusting Learning Rate: The learning rate is set to 0.001. You can experiment with different learning rates.

```
model2 = Sequential()

model2.add(Dense(128, activation='relu', input_dim=18))
model2.add(Dropout(0.5))
model2.add(Dense(256, activation='relu'))
model2.add(Dropout(0.5))
model2.add(Dense(128, activation='relu'))
model2.add(Dropout(0.5))
model2.add(Dense(64, activation='relu'))
model2.add(Dropout(0.5))
model2.add(Dense(1, activation='sigmoid'))

early_stopping = EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True)

model2.compile(loss='binary_crossentropy', optimizer=Adam(learning_rate=0.001), metrics=['accuracy'])

history2 = model2.fit(X_train_scaled, y_train, epochs=100, validation_data=(X_val_scaled, y_val), batch_size=32, callbacks=[early_stopping])
```

Output:

```
Epoch 1/100
65/65 [==============================] - 3s 8ms/step - loss: 0.7167 - accuracy: 0.5308 - val_loss: 0.6851 - val_accuracy: 0.5590
Epoch 2/100
65/65 [==============================] - 0s 5ms/step - loss: 0.6967 - accuracy: 0.5367 - val_loss: 0.6771 - val_accuracy: 0.6259
Epoch 3/100
65/65 [==============================] - 0s 5ms/step - loss: 0.6879 - accuracy: 0.5488 - val_loss: 0.6767 - val_accuracy: 0.5721
Epoch 4/100
65/65 [==============================] - 0s 5ms/step - loss: 0.6840 - accuracy: 0.5673 - val_loss: 0.6628 - val_accuracy: 0.6114
.
.
Epoch 96/100
65/65 [==============================] - 0s 7ms/step - loss: 0.1763 - accuracy: 0.9349 - val_loss: 0.1909 - val_accuracy: 0.9301
Epoch 97/100
65/65 [==============================] - 0s 7ms/step - loss: 0.1653 - accuracy: 0.9325 - val_loss: 0.1909 - val_accuracy: 0.9345
Epoch 98/100
```

```
65/65 [==============================] - 1s 8ms/step - loss: 0.1929 - accuracy:
0.9237      -      val_loss:      0.1975      -      val_accuracy:      0.9229
Epoch                                                                  99/100
65/65 [==============================] - 1s 9ms/step - loss: 0.1846 - accuracy:
0.9281      -      val_loss:      0.1904      -      val_accuracy:      0.9330
Epoch                                                                 100/100
65/65 [==============================] - 0s 7ms/step - loss: 0.1885 - accuracy:
0.9228 - val_loss: 0.1981 - val_accuracy: 0.9330
```

Evaluation metrics

```
predictions = model2.predict(X_val_scaled)
rounded_predictions = np.round(predictions)
report = classification_report(y_val, rounded_predictions)
print(f'Classification Report:\n{report}')
```

Output:

```
22/22       [==============================]       -       0s       2ms/step
Classification                                                        Report:
              precision                recall       f1-score            support
           0              1.00            0.87            0.93              352
           1              0.88            1.00            0.94              335
    accuracy                                                 0.93              687
   macro  avg              0.94            0.93            0.93              687
weighted avg       0.94      0.93      0.93       687
```

What is Exploding Gradient?

The exploding gradient problem is a challenge encountered during training deep neural networks. It occurs when the gradients of the network's loss function with respect to the weights (parameters) become excessively large.

Why Exploding Gradient Occurs?

The issue of exploding gradients arises when, during backpropagation, the derivatives or slopes of the neural network's layers grow progressively larger as we move backward. This is essentially the opposite of the vanishing gradient problem.

The root cause of this problem lies in the weights of the network, rather than the choice of activation function. High weight values lead to correspondingly high derivatives, causing significant deviations in new weight values from the previous ones. As a result, the gradient fails to converge and can lead to the network oscillating around local minima, making it challenging to reach the global minimum point.

In summary, exploding gradients occur when weight values lead to excessively large derivatives, making convergence difficult and potentially preventing the neural network from effectively learning and optimizing its parameters.

As we discussed earlier, the update for the weights during backpropagation in a neural network is given by:

$\Delta Wi = -\alpha \cdot \partial L / \partial Wi$ $\Delta Wi = -\alpha \cdot \partial L / \partial Wi$

where,

- ΔWi ΔWi : The change in the weight Wi Wi
- **α**: The learning rate, a hyperparameter that controls the step size of the update.
- **L**: The loss function that measures the error of the model.
- $\partial L / \partial Wi$ $\partial L / \partial Wi$: The partial derivative of the loss function with respect to the weight Wi Wi, which indicates the gradient of the loss function with respect to that weight.

The exploding gradient problem occurs when the gradients become very large during backpropagation. This is often the result of gradients greater than 1, leading to a rapid increase in values as you propagate them backward through the layers.
Mathematically, the update rule becomes problematic when$|\nabla \mathrm{Wi}|>1|\nabla Wi|>1$, causing the weights to increase exponentially during training.

How can we identify the problem?

Identifying the presence of exploding gradients in deep neural network requires careful observation and analysis during training. Here are some key indicators:

- The loss function exhibits erratic behavior, oscillating wildly instead of steadily decreasing suggesting that the network weights are being updated excessively by large gradients, preventing smooth convergence.
- The training process encounters "NaN" (Not a Number) values in the loss function or other intermediate calculations..
- If network weights, during training exhibit significant and rapid increases in their values, it suggests the presence of exploding gradients.
- Tools like TensorBoard can be used to visualize the gradients flowing through the network.

How can we solve the issue?

- **Gradient Clipping**: It sets a maximum threshold for the magnitude of gradients during backpropagation. Any gradient exceeding the threshold is clipped to the threshold value, preventing it from growing unbounded.
- **Batch Normalization:** This technique normalizes the activations within each mini-batch, effectively scaling the gradients and reducing their variance. This helps prevent both vanishing and exploding gradients, improving stability and efficiency.

Build and train a model for Exploding Gradient Problem

We work on the same preprocessed data from the Vanishing gradient example but define a different neural network.

Step 1: Model creation and adding layers

```
model = Sequential()

model.add(Dense(10, activation='tanh', kernel_initializer=random_normal(mean=0.0, stddev=1.0), input_dim=18))
model.add(Dense(10, activation='tanh', kernel_initializer=random_normal(mean=0.0, stddev=1.0)))
model.add(Dense(10, activation='tanh', kernel_initializer=random_normal(mean=0.0, stddev=1.0)))
model.add(Dense(10, activation='tanh', kernel_initializer=random_normal(mean=0.0, stddev=1.0)))
model.add(Dense(10, activation='tanh', kernel_initializer=random_normal(mean=0.0, stddev=1.0)))
model.add(Dense(10, activation='tanh', kernel_initializer=random_normal(mean=0.0, stddev=1.0)))
model.add(Dense(10, activation='tanh', kernel_initializer=random_normal(mean=0.0, stddev=1.0)))
model.add(Dense(10, activation='tanh', kernel_initializer=random_normal(mean=0.0, stddev=1.0)))
model.add(Dense(1, activation='sigmoid'))
```

Step 2: Model compiling

```
optimizer = SGD(learning_rate=1.0)
model.compile(loss='binary_crossentropy', optimizer=optimizer, metrics=['accuracy'])
```

Step 3: Model training

```
history = model.fit(X_train, y_train, epochs=100)
```

Output:

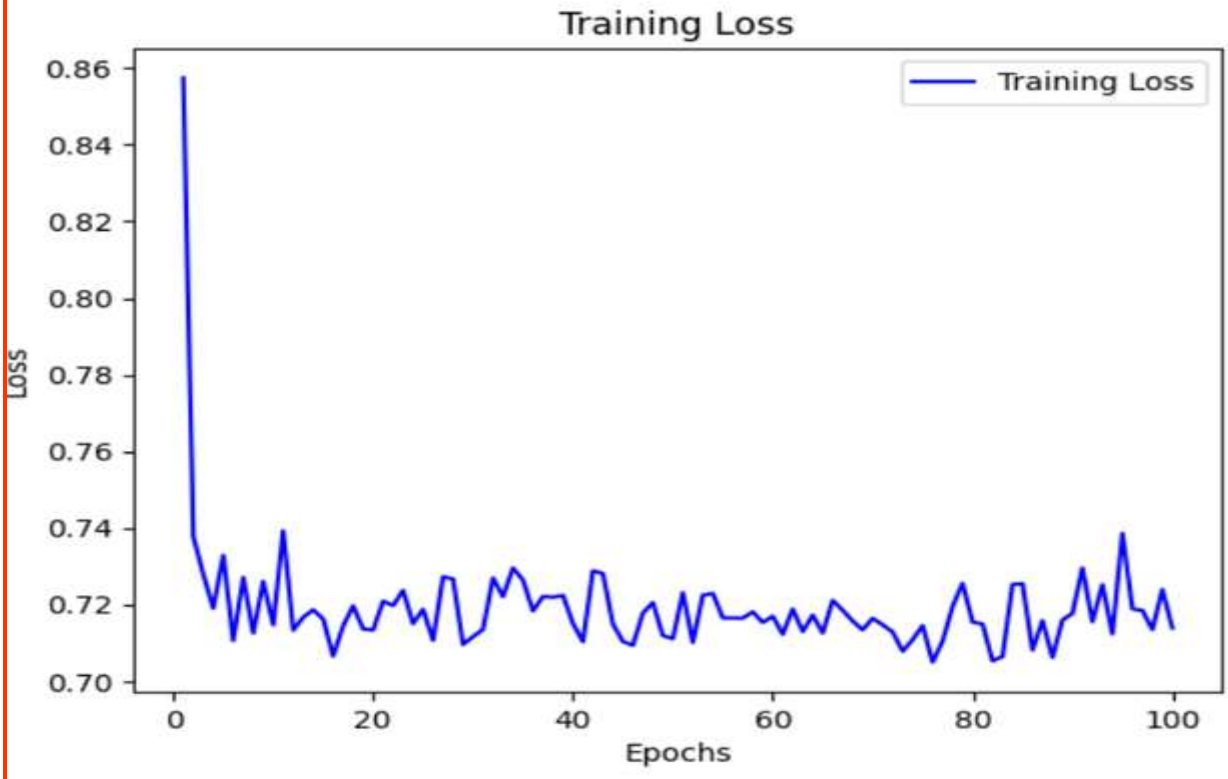
```
Epoch 1/100
65/65 [==============================] - 2s 5ms/step - loss: 0.7919 - accuracy: 0.5032
```

```
Epoch                                                                    2/100
65/65 [==============================] - 0s 4ms/step - loss: 0.7440 - accuracy:
0.5017
.
.
Epoch                                                                   99/100
65/65 [==============================] - 0s 4ms/step - loss: 0.7022 - accuracy:
0.5085
Epoch                                                                  100/100
65/65 [==============================] - 0s 5ms/step - loss: 0.7037 - accuracy:
0.5061
```

Step 4: Plotting training loss

```
loss = history.history['loss']
epochs = range(1, len(loss) + 1)

plt.plot(epochs, loss, 'b', label='Training Loss')
plt.title('Training Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

Output:



It is observed that the loss does not converge and keeps fluctuating which shows we have encountered an exploding gradient problem.

Solution for Exploding Gradient Problem

Below methods can be used to modify the model:

1. Weight Initialization: The weight initialization is changed to 'glorot_uniform,' which is a commonly used initialization for neural networks.

2. Gradient Clipping: The clipnorm parameter in the Adam optimizer is set to 1.0, which performs gradient clipping. This helps prevent exploding gradients.
3. Kernel Constraint: The max_norm constraint is applied to the kernel weights of each layer with a maximum norm of 2.0. This further helps in preventing exploding gradients.

```
model = Sequential()

model.add(Dense(10, activation='tanh', kernel_initializer='glorot_uniform', kernel_constraint=max_norm(2.0), input_dim=18))
model.add(Dense(10, activation='tanh', kernel_initializer='glorot_uniform', kernel_constraint=max_norm(2.0)))
model.add(Dense(10, activation='tanh', kernel_initializer='glorot_uniform', kernel_constraint=max_norm(2.0)))
model.add(Dense(10, activation='tanh', kernel_initializer='glorot_uniform', kernel_constraint=max_norm(2.0)))
model.add(Dense(10, activation='tanh', kernel_initializer='glorot_uniform', kernel_constraint=max_norm(2.0)))
model.add(Dense(10, activation='tanh', kernel_initializer='glorot_uniform', kernel_constraint=max_norm(2.0)))
model.add(Dense(10, activation='tanh', kernel_initializer='glorot_uniform', kernel_constraint=max_norm(2.0)))
model.add(Dense(10, activation='tanh', kernel_initializer='glorot_uniform', kernel_constraint=max_norm(2.0)))
model.add(Dense(1, activation='sigmoid'))

early_stopping = EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True)

model.compile(loss='binary_crossentropy', optimizer=Adam(learning_rate=0.001, clipnorm=1.0), metrics=['accuracy'])

history = model.fit(X_train_scaled, y_train, epochs=100, validation_data=(X_val_scaled, y_val), batch_size=32, callbacks=[early_stopping])
```

Output:

```
Epoch 1/100
65/65 [==============================] - 6s 11ms/step - loss: 0.6865 - accuracy: 0.5537 - val_loss: 0.6818 - val_accuracy: 0.5764
Epoch 2/100
65/65 [==============================] - 1s 8ms/step - loss: 0.6608 - accuracy: 0.6202 - val_loss: 0.6746 - val_accuracy: 0.6070
Epoch 3/100
65/65 [==============================] - 1s 8ms/step - loss: 0.6440 - accuracy: 0.6357 - val_loss: 0.6624 - val_accuracy: 0.6099
.
.
Epoch 68/100
65/65 [==============================] - 1s 11ms/step - loss: 0.1909 - accuracy: 0.9257 - val_loss: 0.3819 - val_accuracy: 0.8486
Epoch 69/100
65/65 [==============================] - 1s 11ms/step - loss: 0.1811 - accuracy: 0.9286 - val_loss: 0.3533 - val_accuracy: 0.8574
Epoch 70/100
65/65 [==============================] - 1s 10ms/step - loss: 0.1836 - accuracy: 0.9276 - val_loss: 0.3641 - val_accuracy: 0.8515
```

Evaluation metrics

```
predictions = model.predict(X_val)
rounded_predictions = np.round(predictions)
report = classification_report(y_val, rounded_predictions)
print(f'Classification Report:\n{report}')
```

Output:

```
22/22       [==============================]       -       0s       2ms/step
Classification                                                       Report:
              precision                recall       f1-score            support
           0              0.98            0.74            0.85              352
           1              0.78            0.99            0.87              335
    accuracy                                                0.86              687
   macro  avg              0.88            0.86            0.86              687
weighted avg       0.89      0.86      0.86       687
```

Training of Recurrent Neural Networks (RNN) in TensorFlow

1. Importing Libraries

We will be importing **Pandas**, **NumPy**, **Matplotlib**, **Seaborn**, **TensorFlow**, **Keras**, **NLTK** and **Scikit-learn** for implemntation.

```
import warnings
from tensorflow.keras.utils import pad_sequences
from tensorflow.keras.preprocessing.text import Tokenizer
from sklearn.model_selection import train_test_split
import tensorflow as tf
from tensorflow import keras
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import plotly.express as px
import numpy as np

import re
import nltk
nltk.download('all')
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.stem import WordNetLemmatizer
lemm = WordNetLemmatizer()

warnings.filterwarnings("ignore")
```

2. Loading the Dataset

```
data = pd.read_csv("Clothing Review.csv")
data.head(7)

data = data[data['Class Name'].isnull() == False]
```

Output:

	Clothing ID	Age	Title	Review Text	Rating	Recommended IND	Positive Feedback Count	Division Name	Department Name	Class Name
0	767	33	NaN	Absolutely wonderful - silky and sexy and comf...	4	1	0	Initmates	Intimate	Intimates
1	1080	34	NaN	Love this dress! it's sooo pretty. i happene...	5	1	4	General	Dresses	Dresses
2	1077	60	Some major design flaws	I had such high hopes for this dress and reall...	3	0	0	General	Dresses	Dresses
3	1049	50	My favorite buy!	I love, love, love this jumpsuit. it's fun, fl...	5	1	0	General Petite	Bottoms	Pants
4	847	47	Flattering shirt	This shirt is very flattering to all due to th...	5	1	6	General	Tops	Blouses
5	1080	49	Not for the very petite	I love tracy reese dresses, but this one is no...	2	0	4	General	Dresses	Dresses
6	858	39	Cagrcoal shimmer fun	I aded this in my basket at hte last mintue to...	5	1	1	General Petite	Tops	Knits

First five rows of the dataset

We can use **data.shape** to give us dimensions of the dataset.

print(data.shape)

(23472, 10)

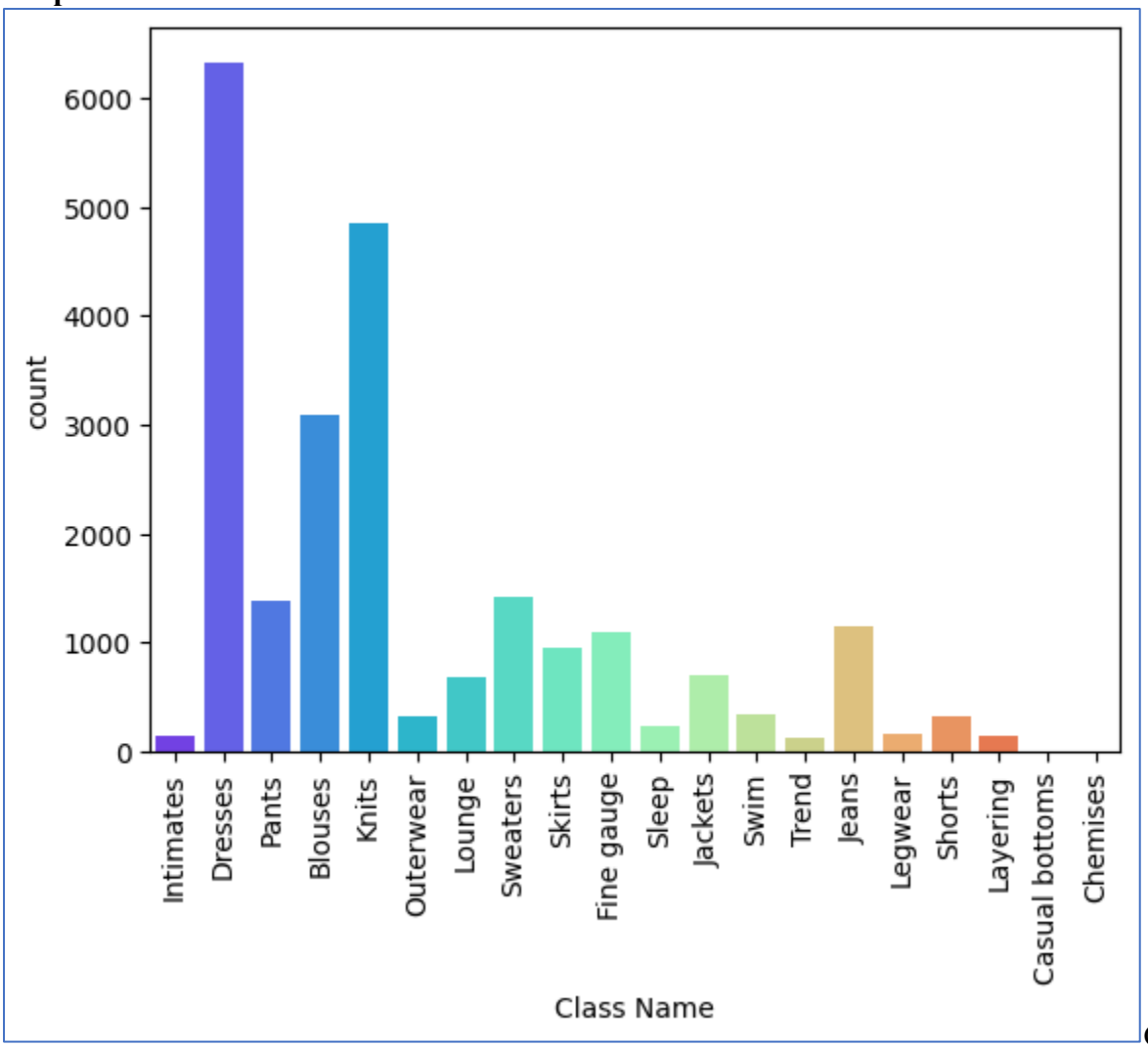
3. Performing Exploratory Data Analysis

EDA helps one understand how the data is distributed. To perform EDA one must perform various visualizing techniques so that one can understand the data before building a model.

We use **sns.countplot()** to visualize the count of each category in the '**Class Name**' column. The **plt.xticks(rotation=90)** rotates the x-axis labels for better readability.

```
sns.countplot(data=data, x='Class Name', palette='rainbow')
plt.xticks(rotation=90)
plt.show()
```

Output:



Countplot

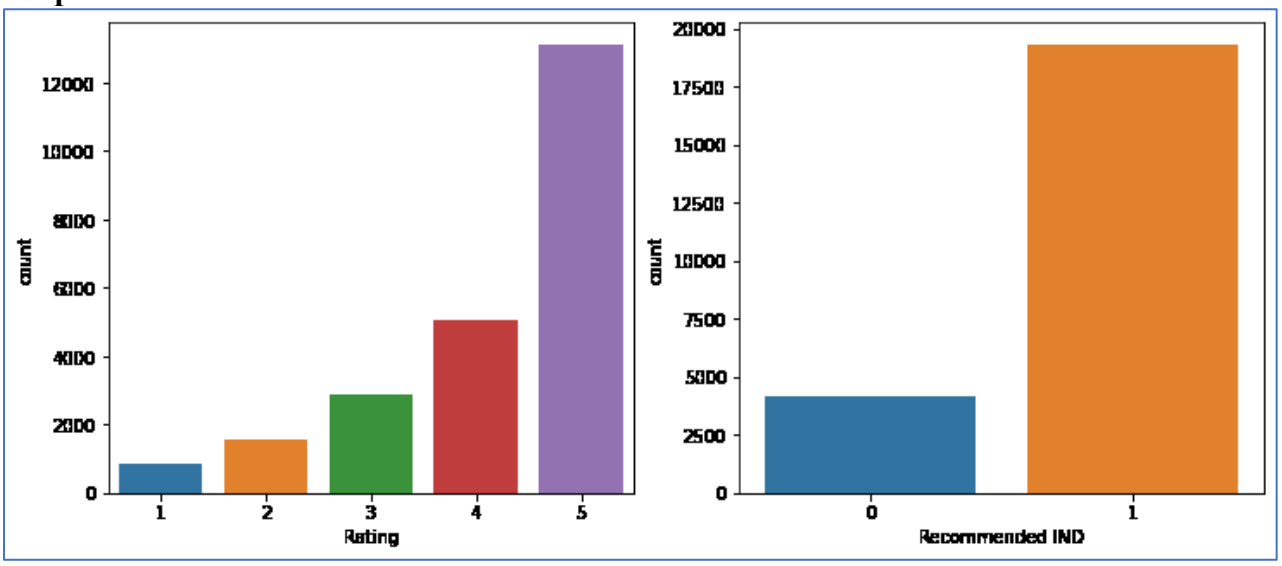
We create a figure with size 12x5 inches using **plt.subplots()**:

- Using **plt.subplot(1, 2, 1)** we plot a countplot of the Rating column
- Using **plt.subplot(1, 2, 2)** we plot a countplot of the Recommended IND column.

```
plt.subplots(figsize=(12, 5))
plt.subplot(1, 2, 1)
```

```
sns.countplot(data=data, x='Rating',palette="deep")

plt.subplot(1, 2, 2)
sns.countplot(data=data, x="Recommended IND", palette="deep")
plt.show()
```

Output:

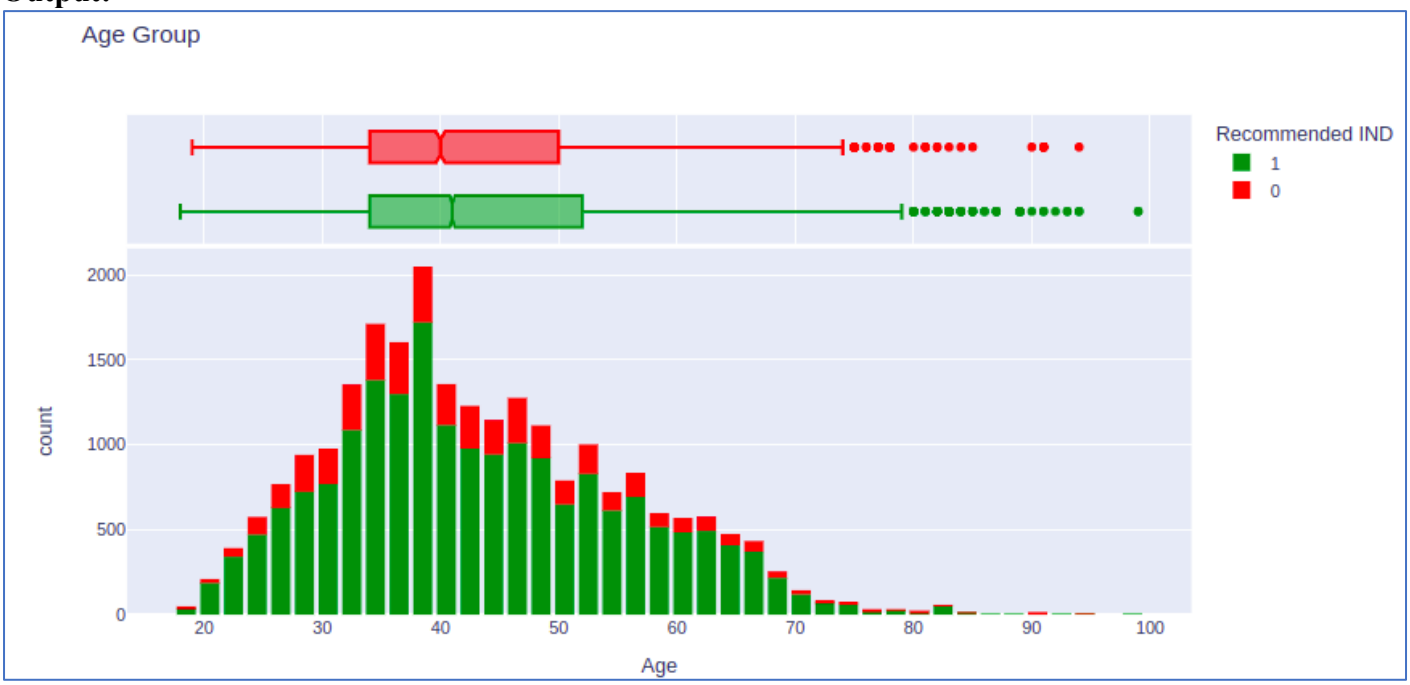


Countplot for the Rating and Recommended IND category

We create a histogram using `px.histogram()` to show the frequency distribution of Age. The histogram includes a box plot as a marginal plot. Data is colored based on Recommended IND values using green and red colors. Number of bins is set to the range from 18 to 65. **fig.update_layout()** adjusts the gap between bars.

```
fig = px.histogram(data, marginal='box',
x="Age", title="Age Group",
color="Recommended IND",
nbins=65-18,
color_discrete_sequence=['green', 'red'])
fig.update_layout(bargap=0.2)
```

Output:



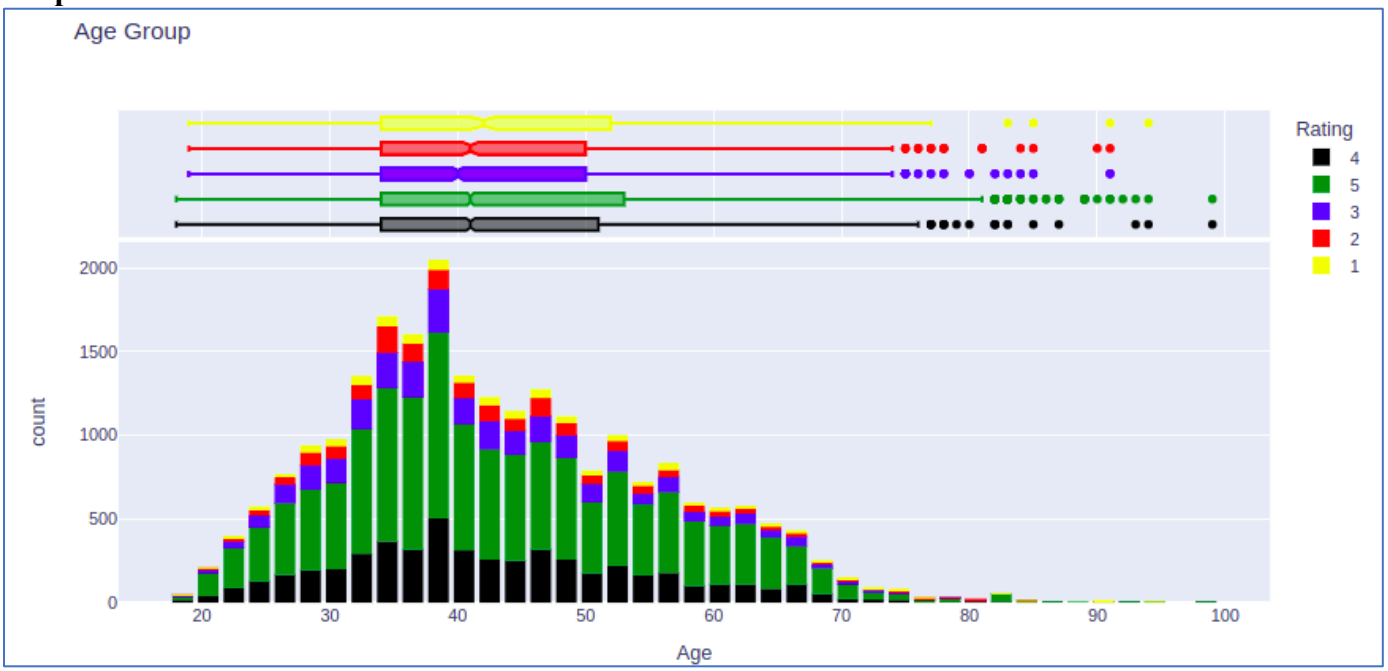
The histogram on the bottom shows age distribution with green bars for recommended individuals and red bars for non-recommended ones. The box plots at the top display the spread and outliers of ages for each recommendation group helping to visualize differences in age distribution between the two groups.

We can visualize the distribution of the age columns data along with the **Rating.**

```
fig = px.histogram(data,
```

```
x="Age",
marginal='box',
title="Age Group",
color="Rating",
nbins=65-18,
color_discrete_sequence
=['black', 'green', 'blue', 'red', 'yellow'])
fig.update_layout(bargap=0.2)
```

Output:



The histogram at the bottom represents the count of individuals in each age group with bars color coded by rating from 1 to 5. The boxplots at the top provide a summary of age distribution for each rating showing the median, interquartile range and outliers. It helps to analyze how ratings vary with age groups.

4. Prepare the Data to build Model

Since we are working on the NLP-based dataset it could be valid to use Text columns as the feature. So we select the features that are text and the Rating column is used for Sentiment Analysis. By the above Rating counterplot we can observe that there is too much of an imbalance between the rating. So all the rating above 3 is made as 1 and below 3 as 0.

```
def filter_score(rating):
return int(rating > 3)

features = ['Class Name', 'Title', 'Review Text']

X = data[features]
y = data['Rating']
y = y.apply(filter_score)
```

5. Text Preprocessing

The text data we have comes with too much noise. This noise can be in form of repeated words or commonly used sentences. In text preprocessing we need the text in the same format so we first convert the entire text into lowercase and then perform Lemmatization to remove the superposition of the words. Since we need clean text we also remove common words such as Stopwords and punctuation.

Note: Lemmatization and stop words are some concepts used in NLP and they are apart from RNN.

```
def toLower(data):
if isinstance(data, float):
return '<UNK>'
else:
return data.lower()
```

```
stop_words = stopwords.words("english")

def remove_stopwords(text):
no_stop = []
for word in text.split(' '):
if word not in stop_words:
no_stop.append(word)
return " ".join(no_stop)

def remove_punctuation_func(text):
return re.sub(r'[^a-zA-Z0-9]', ' ', text)

X['Title'] = X['Title'].apply(toLower)
X['Review Text'] = X['Review Text'].apply(toLower)

X['Title'] = X['Title'].apply(remove_stopwords)
X['Review Text'] = X['Review Text'].apply(remove_stopwords)

X['Title'] = X['Title'].apply(lambda x: lemm.lemmatize(x))
X['Review Text'] = X['Review Text'].apply(lambda x: lemm.lemmatize(x))

X['Title'] = X['Title'].apply(remove_punctuation_func)
X['Review Text'] = X['Review Text'].apply(remove_punctuation_func)

X['Text'] = list(X['Title']+X['Review Text']+X['Class Name'])


X_train, X_test, y_train, y_test = train_test_split(
X['Text'], y, test_size=0.25, random_state=42)
```

If you notice at the end of the code, we have created a new column "**Text**" which is of type list. The reason we did this is that we need to perform Tokenization on the entire feature taken to train the model.

6. Tokenization

In Tokenization we convert the text into Vectors. Keras API supports text pre-processing. This API consists of Tokenizer that takes in the total num_words to create the Word index. OOV stands for out of vocabulary this is triggered when new text is encountered. Also remember that we **fit_on_texts** only on training data and not testing.

Vectors are numeric representation of data so that machine can easily be understand and use it for training.

```
tokenizer = Tokenizer(num_words=10000, oov_token='<OOV>')
tokenizer.fit_on_texts(X_train)
```

7. Padding the Text Data

Keras preprocessing helps in organizing the text. Padding helps in building models of the same size that further becomes easy to train neural network models. The padding adds extra zeros to satisfy the maximum length to feed a neural network. If the text length exceeds then it can be truncated from either the beginning or end. By default it is pre, we can set it to post or leave it as it is.

Both padding and tokenization are a part of NLP processing not specific for RNN task.

```
train_seq = tokenizer.texts_to_sequences(X_train)
test_seq = tokenizer.texts_to_sequences(X_test)

train_pad = pad_sequences(train_seq,
maxlen=40,
truncating="post",
padding="post")
test_pad = pad_sequences(test_seq,
maxlen=40,
truncating="post",
```

```
padding="post")
```

8. Building a Recurrent Neural Network (RNN) in TensorFlow

Now that the data is ready, the next step is building a Simple Recurrent Neural network. Before training with SImpleRNN, the data is passed through the Embedding layer to perform the equal size of Word Vectors.

Note: *We use* ***return_sequences = True*** *only when we need another layer to stack.*

```
from tensorflow import keras

model = keras.models.Sequential()
model.add(keras.layers.Embedding(input_dim=10000,                output_dim=128,
input_length=40))
model.add(keras.layers.SimpleRNN(64, return_sequences=True))
model.add(keras.layers.SimpleRNN(64))
model.add(keras.layers.Dense(128, activation="relu"))
model.add(keras.layers.Dropout(0.4))
model.add(keras.layers.Dense(1, activation="sigmoid"))

model.build(input_shape=(None, 40))

model.summary()
```

Output:

```
Model: "sequential_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_4 (Embedding)     (None, None, 128)         1280000

 simple_rnn_1 (SimpleRNN)    (None, None, 64)          12352

 simple_rnn_2 (SimpleRNN)    (None, 64)                8256

 dense_6 (Dense)             (None, 128)               8320

 dropout_3 (Dropout)         (None, 128)               0

 dense_7 (Dense)             (None, 1)                 129

=================================================================
Total params: 1,309,057
Trainable params: 1,309,057
Non-trainable params: 0
```

Summary of the architecture of the model

9. Training the Model

In Tensorflow after developing a model, it needs to be compiled using the three important parameters i.e Optimizer, Loss Function and Evaluation metrics. We will be training the model on the **train_pad** which we preprocessed and use **5 epochs** to calculate the accuracy on **y_train** dataset.

```
model.compile(loss="binary_crossentropy",
optimizer="adam",
metrics=["accuracy"])


history = model.fit(train_pad,
y_train,
epochs=5)
```

Output:

```
Epoch 1/5
551/551 ━━━━━━━━━━━━━━━━━━━━ 10s 11ms/step - accuracy: 0.7683 - loss: 0.5413
Epoch 2/5
551/551 ━━━━━━━━━━━━━━━━━━━━ 7s 8ms/step - accuracy: 0.7860 - loss: 0.5153
Epoch 3/5
551/551 ━━━━━━━━━━━━━━━━━━━━ 5s 8ms/step - accuracy: 0.7778 - loss: 0.5193
Epoch 4/5
551/551 ━━━━━━━━━━━━━━━━━━━━ 4s 7ms/step - accuracy: 0.7795 - loss: 0.4999
Epoch 5/5
551/551 ━━━━━━━━━━━━━━━━━━━━ 5s 7ms/step - accuracy: 0.7800 - loss: 0.4942
```

Types of Recurrent Neural Networks: There are various types of RNN which are as follows:

- Types of Recurrent Neural Networks
- Bidirectional RNNs
- Long Short-Term Memory (LSTM)
- Bidirectional Long Short-Term Memory (Bi-LSTM)
- Gated Recurrent Units (GRU)

Types of Recurrent Neural Networks (RNN) in Tensorflow

Architecture Variants Based on Input-Output Relationships

The classification of RNNs based on their input-output structure reveals four distinct architectural patterns, each suited for different types of sequential learning tasks.

1. One-to-One RNN

The one-to-one architecture represents the simplest form, equivalent to a standard feedforward neural network with a single input producing a single output. While technically not using the sequential processing capabilities of RNNs, this configuration serves as a building block for understanding more complex architectures.

One-to-One RNN

Code Implementation:

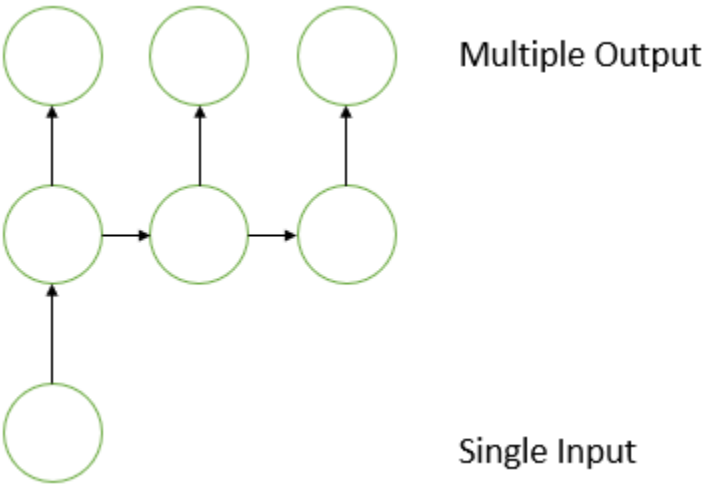
- Suitable for tasks where one input leads to one output like binary classification.
- SimpleRNN processes a single time step (shape: (1, input_dim)).
- A Dense layer with sigmoid activation outputs a binary probability.
- Uses binary crossentropy for training on binary labels.

```
import tensorflow as tf
import numpy as np

# Build and compile a one-to-one RNN model
model = tf.keras.Sequential([
tf.keras.layers.SimpleRNN(32, input_shape=(1, 10)),
```

```
tf.keras.layers.Dense(1, activation='sigmoid')
])
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])


X = np.random.rand(1000, 1, 10)
y = np.random.randint(0, 2, (1000, 1))

# Train the model
model.fit(X, y, epochs=10, batch_size=32, verbose=0)
```

2. One-to-Many RNN

One-to-many architectures accept a single input and generate a sequence of outputs. This pattern proves invaluable for generative tasks where a single piece of information must be expanded into a structured sequence. The network processes the initial input and then uses its internal state to generate subsequent outputs.



One-to-Many RNN

Code Implementation:

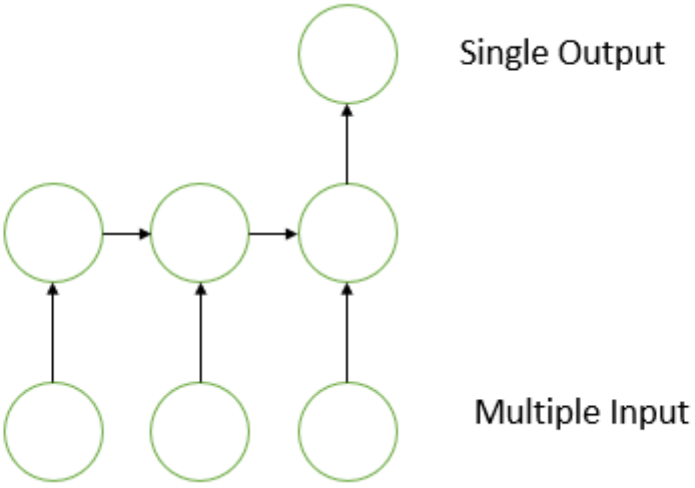
- Used in image captioning where a single input vector generates a word sequence.
- Dense transforms image features before repetition.
- RepeatVector duplicates input across time steps.
- SimpleRNN decodes the repeated vector into a sequence.
- Final layer predicts word probabilities at each step (vocab_size outputs).

```
import tensorflow as tf
import numpy as np

# Build a One-to-Many RNN model
model = tf.keras.Sequential([
tf.keras.layers.Dense(64, input_shape=(2048,)),
tf.keras.layers.RepeatVector(20),
tf.keras.layers.SimpleRNN(128, return_sequences=True),
tf.keras.layers.Dense(10000, activation='softmax')
])

model.compile(optimizer='adam', loss='categorical_crossentropy')

# Simulate image features and caption labels
X = np.random.rand(500, 2048)
y = np.random.rand(500, 20, 10000)
```

```
# Train the model
model.fit(X, y, epochs=5, batch_size=16, verbose=0)
```

3. Many-to-One RNN

Many-to-one networks process entire sequences to produce single outputs, making them ideal for classification and regression tasks on sequential data. The network accumulates information across all time steps before generating a final decision.



Many-to-One RNN

Code Implementation:

- Designed for sequence classification (e.g., sentiment analysis).
- Inputs: word embeddings of a sentence (shape: (sequence_length, input_dim)).
- SimpleRNN encodes the sequence into a single hidden state.
- Dense layers decode that state to predict one of the num_classes.
- Categorical crossentropy used for multiclass classification.

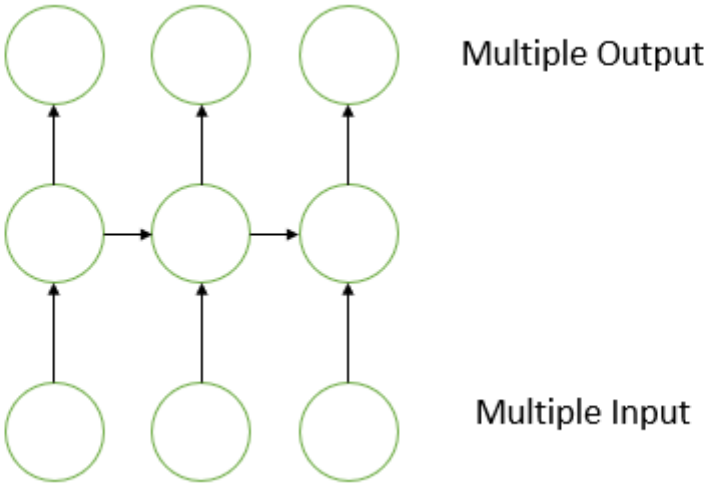
```
import tensorflow as tf
import numpy as np

# Build a Many-to-One RNN model for sentiment classification
model = tf.keras.Sequential([
tf.keras.layers.SimpleRNN(64, input_shape=(100, 128)),
tf.keras.layers.Dense(32, activation='relu'),
tf.keras.layers.Dense(3, activation='softmax')
])

model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])


X = np.random.rand(2000, 100, 128)
y = tf.keras.utils.to_categorical(np.random.randint(0, 3, 2000))

# Train the model
model.fit(X, y, epochs=10, batch_size=32, validation_split=0.2, verbose=0)
```

4. Many-to-Many RNN

Many-to-many architectures represent the most complex variant, processing input of sequences to generate output of sequences.

It has two sub-variants:

- Synchronized where we have equal input and output lengths
- Asynchronous where we have different lengths with encoder-decoder structure



Many-to-Many RNN

Code Implementation:

- Implements an encoder-decoder architecture for tasks like machine translation.
- Encoder processes the source sequence like English sentence.
- Decoder generates the target sequence step by step like French sentence.
- return_state=True is set for capturing the encoder's context.
- Output at each step is predicted from a dense layer with softmax.

```
import tensorflow as tf
import numpy as np

# Encoder-Decoder RNN for sequence-to-sequence translation
def build_seq2seq_rnn(input_len, output_len, in_vocab, out_vocab, units=256):
encoder_input = tf.keras.Input(shape=(input_len, in_vocab))
_, state = tf.keras.layers.SimpleRNN(units, return_state=True)(encoder_input)

decoder_input = tf.keras.Input(shape=(output_len, out_vocab))
decoder_output, _ = tf.keras.layers.SimpleRNN(units, return_sequences=True, return_state=True)(decoder_input, initial_state=state)

output = tf.keras.layers.Dense(out_vocab, activation='softmax')(decoder_output)
model = tf.keras.Model([encoder_input, decoder_input], output)
model.compile(optimizer='adam', loss='categorical_crossentropy')
return model

# Setup for translation
in_vocab, out_vocab = 10000, 12000
input_len, output_len = 50, 60
model = build_seq2seq_rnn(input_len, output_len, in_vocab, out_vocab)

# Simulated training data
X_enc = np.random.rand(1000, input_len, in_vocab)
X_dec = np.random.rand(1000, output_len, out_vocab)
y_dec = np.random.rand(1000, output_len, out_vocab)

model.fit([X_enc, X_dec], y_dec, epochs=5, batch_size=64, verbose=0)
```

Practical Limitations

- **Vanishing Gradients**: Basic RNNs fail to learn long-term dependencies beyond (approx. 30 steps).

- **Training Instability**: Batching variable-length sequences leads to inefficient padding and potential learning bias.
- **High Memory Usage**: Backpropagation through time stores hidden states at each step, consuming large amounts of memory.

Advanced RNN in TensorFlow

TensorFlow supports advanced RNN variants like LSTM and GRU which offer significant improvements over basic RNNs, especially for long sequences and complex patterns.

LSTM Networks

Long Short-Term Memory (LSTM) networks overcome the vanishing gradient problem using gating mechanisms (input, forget and output gates) that control how information flows through time. This enables them to retain dependencies, making them ideal for applications like time-series forecasting, speech recognition and language modeling.

TensorFlow allows flexible LSTM architectures:

- **Many-to-One** for sequence classification or regression.
- **Many-to-Many** for output at each time step, such as in translation.

LSTMs are often paired with dense layers, trained using optimizers like Adam and losses like cross-entropy or MSE.

GRU Networks

Gated Recurrent Units (GRUs) simplify LSTM design by merging gates, reducing parameters while maintaining similar performance. Their efficiency makes them suitable for real-time and resource-constrained tasks. TensorFlow supports Bidirectional GRUs which read sequences both forward and backward and are useful where context from both directions matters.

Bidirectional Recurrent Neural Network

Overview of Bidirectional Recurrent Neural Networks (BRNNs)

A Bidirectional Recurrent Neural Network (BRNN) is an extension of the traditional RNN that processes sequential data in both forward and backward directions. This allows the network to utilize both past and future context when making predictions providing a more comprehensive understanding of the sequence.

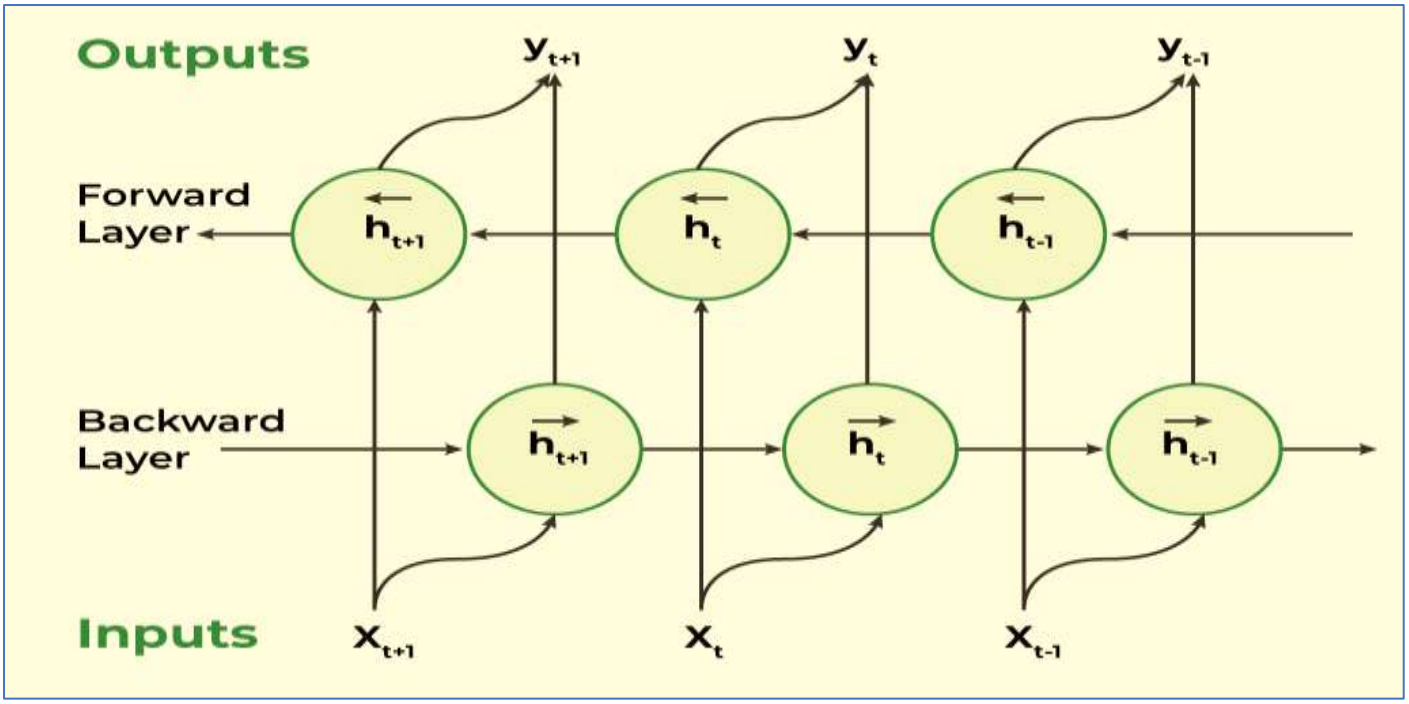
Like a traditional RNN, a BRNN moves forward through the sequence, updating the hidden state based on the current input and the prior hidden state at each time step. The key difference is that a BRNN also has a backward hidden layer which processes the sequence in reverse, updating the hidden state based on the current input and the hidden state of the next time step.

Compared to unidirectional RNNs BRNNs improve accuracy by considering both the past and future context. This is because the two hidden layers i.e forward and backward complement each other and predictions are made using the combined outputs of both layers.

Example:

Consider the sentence: "I like apple. It is very healthy."

In a traditional unidirectional RNN the network might struggle to understand whether "apple" refers to the fruit or the company based on the first sentence. However a BRNN would have no such issue. By processing the sentence in both directions, it can easily understand that "apple" refers to the fruit, thanks to the future context provided by the second sentence ("It is very healthy.").



Bi-directional Recurrent Neural Network

Working of Bidirectional Recurrent Neural Networks (BRNNs)

1. Inputting a Sequence: A sequence of data points each represented as a vector with the same dimensionality is fed into the BRNN. The sequence may have varying lengths.

2. Dual Processing: BRNNs process data in two directions:

- **Forward direction**: The hidden state at each time step is determined by the current input and the previous hidden state.
- **Backward direction**: The hidden state at each time step is influenced by the current input and the next hidden state.

3. Computing the Hidden State: A non-linear activation function is applied to the weighted sum of the input and the previous hidden state creating a memory mechanism that allows the network to retain information from earlier steps.

4. Determining the Output: A non-linear activation function is applied to the weighted sum of the hidden state and output weights to compute the output at each step. This output can either be:

- The final output of the network.
- An input to another layer for further processing.

Implementation of Bi-directional Recurrent Neural Network

Here's a simple implementation of a Bidirectional RNN using Keras and TensorFlow for sentiment analysis on the **IMDb dataset** available in keras:

1. Loading and Preprocessing Data

We first load the IMDb dataset and preprocess it by padding the sequences to ensure uniform length.

- **warnings.filterwarnings('ignore')** suppresses any warnings during execution.
- **imdb.load_data(num_words=features)** loads the IMDb dataset, considering only the top 2000 most frequent words.
- **pad_sequences(X_train, maxlen=max_len)** and **pad_sequences(X_test, maxlen=max_len)** pad the training and test sequences to a maximum length of 50 words ensuring consistent input size.

```
import warnings
warnings.filterwarnings('ignore')
```

```
from keras.datasets import imdb
from keras.preprocessing.sequence import pad_sequences

features = 2000  # Number of most frequent words to consider
max_len = 50     # Maximum length of each sequence

(X_train, y_train), (X_test, y_test) = imdb.load_data(num_words=features)

X_train = pad_sequences(X_train, maxlen=max_len)
X_test = pad_sequences(X_test, maxlen=max_len)
```

2. Defining the Model Architecture

We define a Bidirectional Recurrent Neural Network model using Keras. The model uses an embedding layer with 128 dimensions, a Bidirectional SimpleRNN layer with 64 hidden units and a dense output layer with a sigmoid activation for binary classification.

- **Embedding()** layer maps input features to dense vectors of size embedding (128), with an input length of len.
- **Bidirectional(SimpleRNN(hidden))** adds a bidirectional RNN layer with hidden (64) units.
- **Dense(1, activation='sigmoid')** adds a dense output layer with 1 unit and a sigmoid activation for binary classification.
- **model.compile()** configures the model with Adam optimizer, binary cross-entropy loss and accuracy as the evaluation metric.

```
from keras.models import Sequential
from keras.layers import Embedding, Bidirectional, SimpleRNN, Dense

embedding_dim = 128
hidden_units = 64

model = Sequential()

model.add(Embedding(features, embedding_dim, input_length=max_len))

model.add(Bidirectional(SimpleRNN(hidden_units)))

model.add(Dense(1, activation='sigmoid'))

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

3. Training the Model

As we have compiled our model successfully and the data pipeline is also ready so, we can move forward toward the process of training our BRNN.

- **batch_size=32** defines how many samples are processed together in one iteration.
- **epochs=5** sets the number of times the model will train on the entire dataset.
- **model.fit()** trains the model on the training data and evaluates it using the provided validation data.

```
batch_size = 32
epochs = 5

model.fit(X_train, y_train,
batch_size=batch_size,
epochs=epochs,
```

```
validation_data=(X_test, y_test))
```

Output:

```
Epoch 1/5
782/782 ━━━━━━━━━━━━━━━━━━━━ 42s 47ms/step - accuracy: 0.6312 - loss: 0.6193 - val_accuracy: 0.7739 - val_loss: 0.4794
Epoch 2/5
782/782 ━━━━━━━━━━━━━━━━━━━━ 35s 40ms/step - accuracy: 0.8174 - loss: 0.4092 - val_accuracy: 0.7872 - val_loss: 0.4572
Epoch 3/5
782/782 ━━━━━━━━━━━━━━━━━━━━ 45s 45ms/step - accuracy: 0.8681 - loss: 0.3163 - val_accuracy: 0.7772 - val_loss: 0.5238
Epoch 4/5
782/782 ━━━━━━━━━━━━━━━━━━━━ 40s 44ms/step - accuracy: 0.9191 - loss: 0.2073 - val_accuracy: 0.7586 - val_loss: 0.5886
Epoch 5/5
782/782 ━━━━━━━━━━━━━━━━━━━━ 41s 45ms/step - accuracy: 0.9540 - loss: 0.1308 - val_accuracy: 0.7429 - val_loss: 0.7648
<keras.src.callbacks.history.History at 0x79c26f4a4410>
```

Training the Model

4. Evaluating the Model

Now as we have our model ready let's evaluate its performance on the validation data using different evaluation metrics. For this purpose we will first predict the class for the validation data using this model and then compare the output with the true labels.

- **model.evaluate(X_test, y_test)** evaluates the model's performance on the test data (X_test, y_test), returning the loss and accuracy.

```
loss, accuracy = model.evaluate(X_test, y_test)

print('Test accuracy:', accuracy)
```

Output :

Test accuracy: 0.7429199814796448

Here we achieved a accuracy of 74% and we can increase it accuracy by more fine tuning.

5. Predict on Test Data

We will use the model to predict on the test data and compare the predictions with the true labels.

- **model.predict(X_test)** generates predictions for the test data.
- **y_pred = (y_pred > 0.5)** converts the predicted probabilities into binary values (0 or 1) based on a threshold of 0.5.
- **classification_report(y_test, y_pred, target_names=['Negative', 'Positive'])** generates and prints a classification report including precision, recall, f1-score and support for the negative and positive classes.

```
from sklearn.metrics import classification_report

y_pred = model.predict(X_test)

y_pred = (y_pred > 0.5)

print(classification_report(y_test, y_pred, target_names=['Negative', 'Positive']))
```

Output:

```
782/782 ━━━━━━━━━━━━━━━━━━━━ 6s 8ms/step
              precision    recall  f1-score   support

    Negative       0.74      0.75      0.75     12500
    Positive       0.75      0.73      0.74     12500

    accuracy                           0.74     25000
   macro avg       0.74      0.74      0.74     25000
weighted avg       0.74      0.74      0.74     25000
```

Predict on Test Data

Advantages of BRNNs

- **Enhanced Context Understanding**: Considers both past and future data for improved predictions.
- **Improved Accuracy**: Particularly effective for NLP and speech processing tasks.
- **Better Handling of Variable-Length Sequences**: More flexible than traditional RNNs making it suitable for varying sequence lengths.
- **Increased Robustness**: Forward and backward processing help filter out noise and irrelevant information, improving robustness.

Challenges of BRNNs

- **High Computational Cost**: Requires twice the processing time compared to unidirectional RNNs.
- **Longer Training Time**: More parameters to optimize result in slower convergence.
- **Limited Real-Time Applicability**: Since predictions depend on the entire sequence hence they are not ideal for real-time applications like live speech recognition.
- **Less Interpretability**: The bidirectional nature of BRNNs makes it more difficult to interpret predictions compared to standard RNNs.

What is LSTM - Long Short Term Memory

Problem with Long-Term Dependencies in RNN

Recurrent Neural Networks (RNNs) are designed to handle sequential data by maintaining a hidden state that captures information from previous time steps. However they often face challenges in learning long-term dependencies where information from distant time steps becomes crucial for making accurate predictions for current state. This problem is known as the vanishing gradient or exploding gradient problem.

- **Vanishing Gradient**: When training a model over time, the gradients which help the model learn can shrink as they pass through many steps. This makes it hard for the model to learn long-term patterns since earlier information becomes almost irrelevant.
- **Exploding Gradient**: Sometimes gradients can grow too large causing instability. This makes it difficult for the model to learn properly as the updates to the model become erratic and unpredictable.

Both of these issues make it challenging for standard RNNs to effectively capture long-term dependencies in sequential data.

LSTM Architecture

LSTM architectures involves the memory cell which is controlled by three gates:

1. **Input gate**: Controls what information is added to the memory cell.
2. **Forget gate**: Determines what information is removed from the memory cell.
3. **Output gate**: Controls what information is output from the memory cell.

This allows LSTM networks to selectively retain or discard information as it flows through the network which allows them to learn long-term dependencies. The network has a hidden state which is like its short-term memory. This memory is updated using the current input, the previous hidden state and the current state of the memory cell.

Working of LSTM

LSTM architecture has a chain structure that contains four neural networks and different memory blocks called cells.

LSTM Model

Information is retained by the cells and the memory manipulations are done by the gates. There are three gates -

1. Forget Gate

The information that is no longer useful in the cell state is removed with the forget gate. Two inputs xtxt (input at the particular time) and ht−1$ht-1$ (previous cell output) are fed to the gate and multiplied with weight matrices followed by the addition of bias. The resultant is passed through sigmoid activation function which gives output in range of [0,1]. If for a particular cell state the output is 0 or near to 0, the piece of information is forgotten and for output of 1 or near to 1, the information is retained for future use.

The equation for the forget gate is:

ft=σ(Wf·[ht−1,xt]+bf)$ft=\sigma(Wf\cdot[ht-1,xt]+bf)$

Where:

- WfWfrepresents the weight matrix associated with the forget gate.
- [ht−1,xt]$[ht-1,xt]$ denotes the concatenation of the current input and the previous hidden state.
- bf$bf$is the bias with the forget gate.
- σσ is the sigmoid activation function.

Forget Gate

2. Input gate

The addition of useful information to the cell state is done by the input gate. First the information is regulated using the sigmoid function and filter the values to be remembered similar to the forget gate using inputs ht−1$ht-1$and xtxt. Then, a vector is created using *tanh* function that gives an output from -1 to +1 which contains all the possible values from ht−1$ht-1$ and xtxt. At last the values of the vector and the regulated values are multiplied to obtain the useful information. The equation for the input gate is:

it=σ(Wi·[ht−1,xt]+bi)$it=\sigma(Wi\cdot[ht-1,xt]+bi)$

C^t=tanh(Wc·[ht−1,xt]+bc)$C\hat{}t=\tanh(Wc\cdot[ht-1,xt]+bc)$

We multiply the previous state by ftft effectively filtering out the information we had decided to ignore earlier. Then we add it⊙Ct$it\odot Ct$ which represents the new candidate values scaled by how much we decided to update each state value.

Ct=ft⊙Ct−1+it⊙C^t$Ct=ft\odot Ct-1+it\odot C\hat{}t$

where

- ⊙⊙ denotes element-wise multiplication
- tanh is activation function

Input Gate

3. Output gate

The output gate is responsible for deciding what part of the current cell state should be sent as the hidden state (output) for this time step.First, the gate uses a sigmoid function to determine which information from the current cell state will be output. This is done using the previous hidden state ht−1$ht-1$ and the current input xtxt:

ot=σ(Wo·[ht−1,xt]+bo)$ot=\sigma(Wo\cdot[ht-1,xt]+bo)$

Next, the current cell state CtCt is passed through a tanh activation to scale its values between −1-1 and +1$+1$. Finally, this transformed cell state is multiplied element-wise with otot to produce the hidden state htht:

ht=ot⊙tanh(Ct)$ht=ot\odot\tanh(Ct)$

Here:

- $o t$ is the output gate activation.
- $C t$ is the current cell state.
- $\odot$ represents element-wise multiplication.
- σ is the sigmoid activation function.

This hidden state $h t$ is then passed to the next time step and can also be used for generating the output of the network.

Bidirectional LSTM in NLP

Understanding Bidirectional LSTM (BiLSTM)

A Bidirectional LSTM (BiLSTM) consists of two separate LSTM layers:

- **Forward LSTM**: Processes the sequence from start to end
- **Backward LSTM**: Processes the sequence from end to start

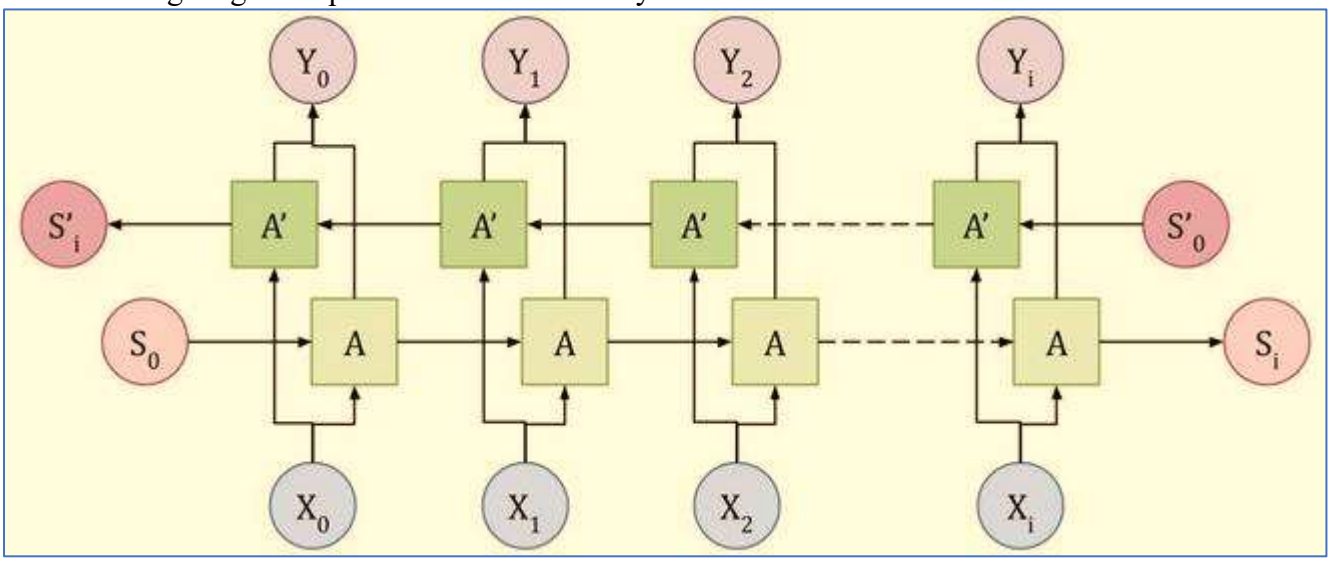
The outputs of both LSTMs are then combined to form the final output. Mathematically, the final output at time **t** is computed as:

$p t = p t f + p t b$

Where:

- $p t$: Final probability vector of the network.
- $p t f$: Probability vector from the forward LSTM network.
- $p t b$: Probability vector from the backward LSTM network.

The following diagram represents the BiLSTM layer:



Bidirectional LSTM layer Architecture

Here:

- $X i$ is the input token
- $Y i$ is the output token
- A and A' are Forward and backward LSTM units
- The final output of $Y i$ is the combination of A and A' LSTM nodes.

Implementation: Sentiment Analysis Using BiLSTM

Now let us look into an implementation of a review system using BiLSTM layers in Python using Tensorflow. We would be performing **sentiment analysis on the IMDB movie review dataset.** We would implement the network from scratch and train it to identify if the review is positive or negative.

1. Importing Libraries

We will be using python libraries like numpy, pandas , matplotlib and tensorflow libraries for building our model.

```
import tensorflow as tf
import tensorflow_datasets as tfds
import numpy as np
import matplotlib.pyplot as plt
```

2. Loading and Preparing the IMDB Dataset

We will load IMDB dataset from tensorflow which contains 25,000 labeled movie reviews for training and testing. Shuffling ensures that the model does not learn patterns based on the order of reviews.

```
dataset = tfds.load('imdb_reviews', as_supervised=True)
train_dataset, test_dataset = dataset['train'], dataset['test']

batch_size = 32

train_dataset = train_dataset.shuffle(10000).batch(batch_size)
test_dataset = test_dataset.batch(batch_size)
```

Printing a sample review and its label from the training set.

```
example, label = next(iter(train_dataset))
print('Text:\n', example.numpy()[0])
print('\nLabel: ', label.numpy()[0])
```

Output:

Text: b "Having seen men Behind the Sun ... 1 as a treatment of the subject)." Label: 0

3. Performing Text Vectorization

We will first perform text vectorization and let the encoder map all the words in the training dataset to a token. We can also see in the example below how we can encode and decode the sample review into a vector of integers.

- **vectorize_layer :** tokenizes and normalizes the text. It converts words into numeric values for the neural network to process easily.

```
vectorize_layer = tf.keras.layers.TextVectorization(output_mode='int', output_sequence_length=100)

vectorize_layer.adapt(train_dataset.map(lambda x, y: x))
```

4. Defining Model Architecture (BiLSTM Layers)

We define the model for sentiment analysis. The first layer, Text Vectorization, converts input text into token indices. These tokens go through an embedding layer that maps words into trainable 32-dimensional vectors. During training, these vectors adjust so that words with similar meanings have similar representations.

The Bidirectional LSTM layers process these sequences from both directions to capture context:

- The first Bidirectional LSTM has 32 units and outputs sequences.
- A dropout layer with rate 0.4 helps prevent overfitting.
- The second Bidirectional LSTM has 16 units and refines the learned features.
- Another dropout layer with rate 0.4 follows.

The Dense layers then perform classification:

- A dense layer with 16 neurons and ReLU activation learns patterns from LSTM output.
- The final dense layer with a single neuron outputs the sentiment prediction.

```
model = tf.keras.Sequential([
vectorize_layer,
tf.keras.layers.Embedding(len(vectorize_layer.get_vocabulary()), 64, mask_zero=True),
tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(64, return_sequences=True)),
tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(32)),
tf.keras.layers.Dense(64, activation='relu'),
tf.keras.layers.Dense(1)
])

model.build(input_shape=(None,))
```

```
model.compile(
loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
optimizer=tf.keras.optimizers.Adam(),
metrics=['accuracy']
)

model.summary()
```

Output:

Model: "sequential_13"

Layer (type)	Output Shape	Param #
text_vectorization_2 (TextVectorization)	(None, 100)	0
embedding_13 (Embedding)	(None, 100, 32)	3,900,608
bidirectional_26 (Bidirectional)	(None, 100, 64)	16,640
dropout_16 (Dropout)	(None, 100, 64)	0
bidirectional_27 (Bidirectional)	(None, 32)	10,368
dropout_17 (Dropout)	(None, 32)	0
dense_26 (Dense)	(None, 16)	528
dense_27 (Dense)	(None, 1)	17

Total params: 3,928,161 (14.98 MB)
Trainable params: 3,928,161 (14.98 MB)
Non-trainable params: 0 (0.00 B)

5. Training the Model

Now we will train the model we defined in the previous step for three epochs.

```
history = model.fit(
train_dataset,
epochs=3,
validation_data=test_dataset,
)
```

Output:

```
Epoch 1/3
782/782 ━━━━━━━━━━━━━━━━━━━━ 32s 35ms/step - accuracy: 0.6626 - loss: 0.5676 - val_accuracy: 0.7864 - val_loss: 0.4097
Epoch 2/3
782/782 ━━━━━━━━━━━━━━━━━━━━ 26s 33ms/step - accuracy: 0.9022 - loss: 0.2429 - val_accuracy: 0.7705 - val_loss: 0.5500
Epoch 3/3
782/782 ━━━━━━━━━━━━━━━━━━━━ 26s 33ms/step - accuracy: 0.9592 - loss: 0.1156 - val_accuracy: 0.7829 - val_loss: 0.7064
```

3. Generative Models in Deep Learning

Generative models generate new data that resembles the training data. The key types of generative models include:

- Generative Adversarial Networks (GANs)
- Autoencoders

- GAN vs. Transformer Models

Generative Adversarial Network (GAN)

Generative Adversarial Networks (GAN) help machines to create new, realistic data by learning from existing examples. It is introduced by Ian Goodfellow and his team in 2014 and they have transformed how computers generate images, videos, music and more. Unlike traditional models that only recognize or classify data, they take a creative way by generating entirely new content that closely resembles real-world data. This ability helped various fields such as art, gaming, healthcare and data science. In this article, we will see more about GANs and its core concepts.

Architecture of GAN

GAN consist of two main models that work together to create realistic synthetic data which are as follows:

1. Generator Model

The generator is a deep neural network that takes random noise as input to generate realistic data samples like images or text. It learns the underlying data patterns by adjusting its internal parameters during training through backpropagation. Its objective is to produce samples that the discriminator classifies as real.

Generator Loss Function: The generator tries to minimize this loss:

$$JG=-1m\Sigma i=1mlogD(G(zi))JG=-m1\Sigma i=1mlogD(G(zi))$$

where

- JGJGmeasure how well the generator is fooling the discriminator.
- G(zi)$G(zi)$ is the generated sample from random noise zizi
- D(G(zi))$D(G(zi))$ is the discriminator's estimated probability that the generated sample is real.

The generator aims to maximize D(G(zi))$D(G(zi))$ meaning it wants the discriminator to classify its fake data as real (probability close to 1).

2. Discriminator Model

The discriminator acts as a binary classifier helps in distinguishing between real and generated data. It learns to improve its classification ability through training, refining its parameters to detect fake samples more accurately. When dealing with image data, the discriminator uses convolutional layers or other relevant architectures which help to extract features and enhance the model's ability.

Discriminator Loss Function: The discriminator tries to minimize this loss:

$$JD=-1m\Sigma i=1mlogD(xi)-1m\Sigma i=1mlog(1-D(G(zi))JD=-m1\Sigma i=1mlogD(xi)-m1\Sigma i=1mlog(1-D(G(zi))$$

- JDJD measures how well the discriminator classifies real and fake samples.
- xixi is a real data sample.
- G(zi)$G(zi)$ is a fake sample from the generator.
- D(xi)$D(xi)$ is the discriminator's probability that xixi is real.
- D(G(zi))$D(G(zi))$ is the discriminator's probability that the fake sample is real.

The discriminator wants to correctly classify real data as real (maximize logD(xi)$logD(xi)$ and fake data as fake (maximize log(1−D(G(zi))$log(1-D(G(zi))$)

MinMax Loss

GANs are trained using a MinMax Loss between the generator and discriminator:

$$minGmaxD(G,D)=[Ex\sim pdata[logD(x)]+Ez\sim pz(z)[log(1-D(g(z)))]minGmaxD(G,D)=[Ex\sim pdata[logD(x)]+Ez\sim pz(z)[log(1-D(g(z)))]$$

where,

- G$G$is generator network and is DD is the discriminator network
- pdata(x)$pdata(x)$ = true data distribution
- pz(z)$pz(z)$= distribution of random noise (usually normal or uniform)
- D(x)$D(x)$ = discriminator's estimate of real data
- D(G(z))$D(G(z))$= discriminator's estimate of generated data

The generator tries to minimize this loss (to fool the discriminator) and the discriminator tries to maximize it (to detect fakes accurately).

How does a GAN work?

GAN train by having two networks the Generator (G) and the Discriminator (D) compete and improve together. Here's the step-by-step process

1. Generator's First Move

The generator starts with a random noise vector like random numbers. It uses this noise as a starting point to create a fake data sample such as a generated image. The generator's internal layers transform this noise into something that looks like real data.

2. Discriminator's Turn

The discriminator receives two types of data:

- Real samples from the actual training dataset.
- Fake samples created by the generator.

D's job is to analyze each input and find whether it's real data or something G cooked up. It outputs a probability score between 0 and 1. A score of 1 shows the data is likely real and 0 suggests it's fake.

3. Adversarial Learning

- If the discriminator correctly classifies real and fake data it gets better at its job.
- If the generator fools the discriminator by creating realistic fake data, it receives a positive update and the discriminator is penalized for making a wrong decision.

4. Generator's Improvement

- Each time the discriminator mistakes fake data for real, the generator learns from this success.
- Through many iterations, the generator improves and creates more convincing fake samples.

5. Discriminator's Adaptation

- The discriminator also learns continuously by updating itself to better spot fake data.
- This constant back-and-forth makes both networks stronger over time.

6. Training Progression

- As training continues, the generator becomes highly proficient at producing realistic data.
- Eventually the discriminator struggles to distinguish real from fake shows that the GAN has reached a well-trained state.
- At this point, the generator can produce high-quality synthetic data that can be used for different applications.

Types of GAN

There are several types of GANs each designed for different purposes. Here are some important types:

1. Vanilla GAN

Vanilla GAN is the simplest type of GAN. It consists of:

- A generator and a discriminator both are built using multi-layer perceptrons (MLPs).

- The model optimizes its mathematical formulation using stochastic gradient descent (SGD).

While foundational, Vanilla GAN can face problems like:

- **Mode collapse**: The generator produces limited types of outputs repeatedly.
- **Unstable training**: The generator and discriminator may not improve smoothly.

2. Conditional GAN (CGAN)

Conditional GAN (CGAN) adds an additional conditional parameter to guide the generation process. Instead of generating data randomly they allow the model to produce specific types of outputs.

Working of CGANs:

- A conditional variable (y) is fed into both the generator and the discriminator.
- This ensures that the generator creates data corresponding to the given condition (e.g generating images of specific objects).
- The discriminator also receives the labels to help distinguish between real and fake data.

Example: Instead of generating any random image, CGAN can generate a specific object like a dog or a cat based on the label.

3. Deep Convolutional GAN (DCGAN)

Deep Convolutional GAN (DCGAN) are among the most popular types of GANs used for image generation.

They are important because they:

- Uses Convolutional Neural Networks (CNNs) instead of simple multi-layer perceptrons (MLPs).
- Max pooling layers are replaced with convolutional stride helps in making the model more efficient.
- Fully connected layers are removed, which allows for better spatial understanding of images.

DCGANs are successful because they generate high-quality, realistic images.

4. Laplacian Pyramid GAN (LAPGAN)

Laplacian Pyramid GAN (LAPGAN) is designed to generate ultra-high-quality images by using a multi-resolution approach.

Working of LAPGAN:

- Uses multiple generator-discriminator pairs at different levels of the Laplacian pyramid.
- Images are first down sampled at each layer of the pyramid and upscaled again using Conditional GAN (CGAN).
- This process allows the image to gradually refine details and helps in reducing noise and improving clarity.

Due to its ability to generate highly detailed images, LAPGAN is considered a superior approach for photorealistic image generation.

5. Super Resolution GAN (SRGAN)

Super-Resolution GAN (SRGAN) is designed to increase the resolution of low-quality images while preserving details.

Working of SRGAN:

- Uses a deep neural network combined with an adversarial loss function.
- Enhances low-resolution images by adding finer details helps in making them appear sharper and more realistic.

- Helps to reduce common image upscaling errors such as blurriness and pixelation.

Implementation of Generative Adversarial Network (GAN) using PyTorch

Generative Adversarial Networks (GAN) can generate realistic images by learning from existing image datasets. Here we will be implementing a GAN trained on the CIFAR-10 dataset using PyTorch.

Step 1: Importing Required Libraries

We will be using Pytorch, Torchvision, Matplotlib and Numpy libraries for this. Set the device to GPU if available otherwise use CPU.

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
from torchvision import datasets, transforms
import matplotlib.pyplot as plt
import numpy as np

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

Step 2: Defining Image Transformations

We use PyTorch's transforms to convert images to tensors and normalize pixel values between -1 and 1 for better training stability.

```
transform = transforms.Compose([
transforms.ToTensor(),
transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])
```

Step 3: Loading the CIFAR-10 Dataset

```
train_dataset = datasets.CIFAR10(root='./data',\
train=True, download=True, transform=transform)
dataloader = torch.utils.data.DataLoader(train_dataset, \
batch_size=32, shuffle=True)
```

Step 4: Defining GAN Hyperparameters

Set important training parameters:

- **latent_dim**: Dimensionality of the noise vector.
- **lr**: Learning rate of the optimizer.
- **beta1, beta2**: Beta parameters for Adam optimizer (e.g 0.5, 0.999)
- **num_epochs**: Number of times the entire dataset will be processed (e.g 10)

```
latent_dim = 100
lr = 0.0002
beta1 = 0.5
beta2 = 0.999
num_epochs = 10
```

Step 5: Building the Generator

Create a neural network that converts random noise into images. Use transpose convolutional layers, batch normalization and ReLU activations. The final layer uses Tanh activation to scale outputs to the range [-1, 1].

- **nn.Linear(latent_dim, 128 * 8 * 8)**: Defines a fully connected layer that projects the noise vector into a higher dimensional feature space.
- **nn.Upsample(scale_factor=2)**: Doubles the spatial resolution of the feature maps by upsampling.
- **nn.Conv2d(128, 128, kernel_size=3, padding=1)**: Applies a convolutional layer keeping the number of channels the same to refine features.

```
class Generator(nn.Module):
def __init__(self, latent_dim):
super(Generator, self).__init__()

self.model = nn.Sequential(
nn.Linear(latent_dim, 128 * 8 * 8),
nn.ReLU(),
nn.Unflatten(1, (128, 8, 8)),
nn.Upsample(scale_factor=2),
nn.Conv2d(128, 128, kernel_size=3, padding=1),
nn.BatchNorm2d(128, momentum=0.78),
nn.ReLU(),
nn.Upsample(scale_factor=2),
nn.Conv2d(128, 64, kernel_size=3, padding=1),
nn.BatchNorm2d(64, momentum=0.78),
nn.ReLU(),
nn.Conv2d(64, 3, kernel_size=3, padding=1),
nn.Tanh()
)

def forward(self, z):
img = self.model(z)
return img
```

Step 6: Building the Discriminator

Create a binary classifier network that distinguishes real from fake images. Use convolutional layers, batch normalization, dropout, LeakyReLU activation and a Sigmoid output layer to give a probability between 0 and 1.

- **nn.Conv2d(32, 64, kernel_size=3, stride=2, padding=1)**: Second convolutional layer increasing channels to 64, downsampling further.
- **nn.BatchNorm2d(256, momentum=0.8)**: Batch normalization for 256 feature maps with momentum 0.8.

```
class Discriminator(nn.Module):
def __init__(self):
super(Discriminator, self).__init__()

self.model = nn.Sequential(
nn.Conv2d(3, 32, kernel_size=3, stride=2, padding=1),
nn.LeakyReLU(0.2),
nn.Dropout(0.25),
nn.Conv2d(32, 64, kernel_size=3, stride=2, padding=1),
nn.ZeroPad2d((0, 1, 0, 1)),
nn.BatchNorm2d(64, momentum=0.82),
nn.LeakyReLU(0.25),
nn.Dropout(0.25),
nn.Conv2d(64, 128, kernel_size=3, stride=2, padding=1),
nn.BatchNorm2d(128, momentum=0.82),
nn.LeakyReLU(0.2),
nn.Dropout(0.25),
nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1),
nn.BatchNorm2d(256, momentum=0.8),
nn.LeakyReLU(0.25),
nn.Dropout(0.25),
```

```
nn.Flatten(),
nn.Linear(256 * 5 * 5, 1),
nn.Sigmoid()
)

def forward(self, img):
validity = self.model(img)
return validity
```

Step 7: Initializing GAN Components

- **Generator and Discriminator** are initialized on the available device (GPU or CPU).
- **Binary Cross-Entropy (BCE) Loss** is chosen as the loss function.
- **Adam optimizers** are defined separately for the generator and discriminator with specified learning rates and betas.

```
generator = Generator(latent_dim).to(device)
discriminator = Discriminator().to(device)

adversarial_loss = nn.BCELoss()

optimizer_G = optim.Adam(generator.parameters()\
, lr=lr, betas=(beta1, beta2))
optimizer_D = optim.Adam(discriminator.parameters()\
, lr=lr, betas=(beta1, beta2))
```

Step 8: Training the GAN

Train the discriminator on real and fake images, then update the generator to improve its fake image quality. Track losses and visualize generated images after each epoch.

- **valid = torch.ones(real_images.size(0), 1, device=device):** Create a tensor of ones representing real labels for the discriminator.
- **fake = torch.zeros(real_images.size(0), 1, device=device):** Create a tensor of zeros representing fake labels for the discriminator.
- **z = torch.randn(real_images.size(0), latent_dim, device=device):** Generate random noise vectors as input for the generator.
- **g_loss = adversarial_loss(discriminator(gen_images), valid):** Calculate generator loss based on the discriminator classifying fake images as real.
- **grid = torchvision.utils.make_grid(generated, nrow=4, normalize=True):** Arrange generated images into a grid for display, normalizing pixel values.

```
for epoch in range(num_epochs):
for i, batch in enumerate(dataloader):

real_images = batch[0].to(device)

valid = torch.ones(real_images.size(0), 1, device=device)
fake = torch.zeros(real_images.size(0), 1, device=device)

real_images = real_images.to(device)

optimizer_D.zero_grad()

z = torch.randn(real_images.size(0), latent_dim, device=device)

fake_images = generator(z)
```

```
real_loss = adversarial_loss(discriminator\
(real_images), valid)
fake_loss = adversarial_loss(discriminator\
(fake_images.detach()), fake)
d_loss = (real_loss + fake_loss) / 2

d_loss.backward()
optimizer_D.step()

optimizer_G.zero_grad()

gen_images = generator(z)

g_loss = adversarial_loss(discriminator(gen_images), valid)
g_loss.backward()
optimizer_G.step()

if (i + 1) % 100 == 0:
print(
f"Epoch [{epoch+1}/{num_epochs}]\
Batch {i+1}/{len(dataloader)} "
f"Discriminator Loss: {d_loss.item():.4f} "
f"Generator Loss: {g_loss.item():.4f}"
)
if (epoch + 1) % 10 == 0:
with torch.no_grad():
z = torch.randn(16, latent_dim, device=device)
generated = generator(z).detach().cpu()
grid = torchvision.utils.make_grid(generated,\
nrow=4, normalize=True)
plt.imshow(np.transpose(grid, (1, 2, 0)))
plt.axis("off")
plt.show()
```

Autoencoders in Machine Learning

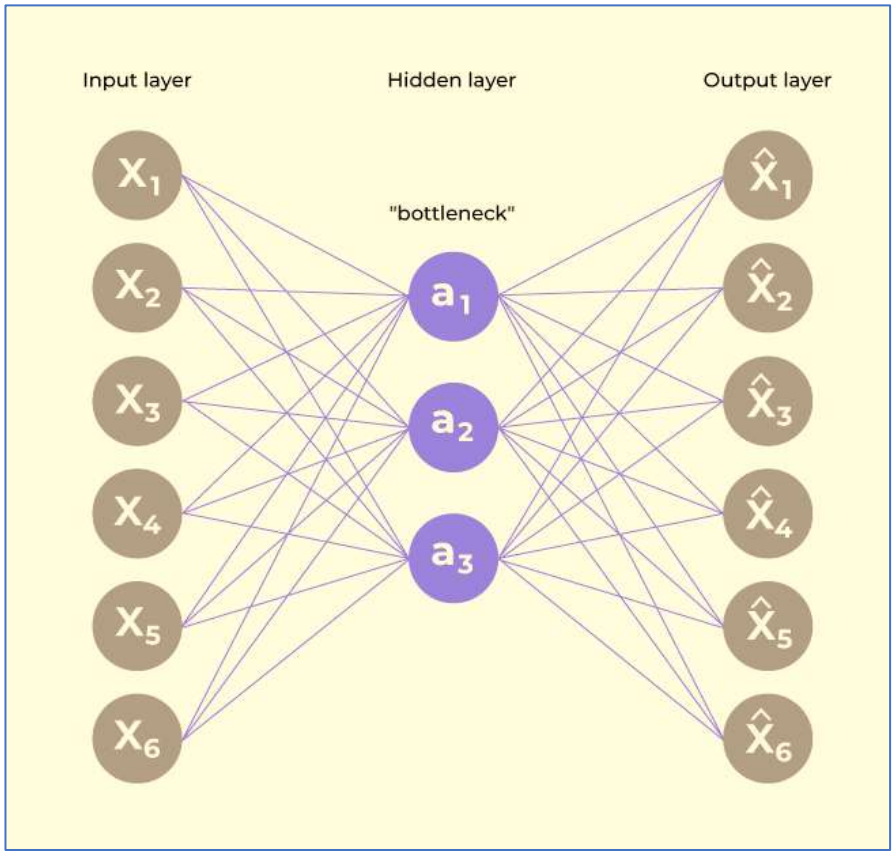
Autoencoders are a special type of **neural networks** that learn to compress data into a compact form and then reconstruct it to closely match the original input. They consist of an:

- **Encoder** that captures important features by reducing dimensionality.
- **Decoder** that rebuilds the data from this compressed representation.

The model trains by minimizing reconstruction error using loss functions like **Mean Squared Error** or **Binary Cross-Entropy**. These are applied in tasks such as noise removal, error detection and feature extraction where capturing efficient data representations is important.

Architecture of Autoencoder

An autoencoder's architecture consists of three main components that work together to compress and then reconstruct data which are as follows:



1. Encoder

It compress the input data into a smaller, more manageable form by reducing its dimensionality while preserving important information. It has three layers which are:

- **Input Layer**: This is where the original data enters the network. It can be images, text features or any other structured data.
- **Hidden Layers**: These layers perform a series of transformations on the input data. Each hidden layer applies weights and activation functions to capture important patterns, progressively reducing the data's size and complexity.
- **Output(Latent Space)**: The encoder outputs a compressed vector known as the **latent representation** or **encoding**. This vector captures the important features of the input data in a condensed form helps in filtering out noise and redundancies.

2. Bottleneck (Latent Space)

It is the smallest layer of the network which represents the most compressed version of the input data. It serves as the information bottleneck which force the network to prioritize the most significant features. This compact representation helps the model learn the underlying structure and key patterns of the input helps in enabling better generalization and efficient data encoding.

3. Decoder

It is responsible for taking the compressed representation from the latent space and reconstructing it back into the original data form.

- **Hidden Layers**: These layers progressively expand the latent vector back into a higher-dimensional space. Through successive transformations decoder attempts to restore the original data shape and details
- **Output Layer**: The final layer produces the reconstructed output which aims to closely resemble the original input. The quality of reconstruction depends on how well the encoder-decoder pair can minimize the difference between the input and output during training.

Loss Function in Autoencoder Training

During training an autoencoder's goal is to minimize the reconstruction loss which measures how different the reconstructed output is from the original input. The choice of loss function depends on the type of data being processed:

- **Mean Squared Error (MSE)**: This is commonly used for continuous data. It measures the average squared differences between the input and the reconstructed data.
- **Binary Cross-Entropy**: Used for binary data (0 or 1 values). It calculates the difference in probability between the original and reconstructed output.

During training the network updates its weights using backpropagation to minimize this reconstruction loss. By doing this it learns to extract and retain the most important features of the input data which are encoded in the latent space.

Efficient Representations in Autoencoders

Constraining an autoencoder helps it learn meaningful and compact features from the input data which leads to more efficient representations. After training only the encoder part is used to encode similar data for future tasks. Various techniques are used to achieve this are as follows:

- **Keep Small Hidden Layers**: Limiting the size of each hidden layer forces the network to focus on the most important features. Smaller layers reduce redundancy and allows efficient encoding.
- **Regularization**: Techniques like L1 or L2 regularization add penalty terms to the loss function. This prevents overfitting by removing excessively large weights which helps in ensuring the model to learns general and useful representations.
- **Denoising:** In denoising autoencoders **random noise** is added to the input during training. It learns to remove this noise during reconstruction which helps it focus on core, noise-free features and helps in improving robustness.
- **Tuning the Activation Functions**: Adjusting activation functions can promote sparsity by activating only a few neurons at a time. This sparsity reduces model complexity and forces the network to capture only the most relevant features.

Types of Autoencoders

Lets see different types of Autoencoders which are designed for specific tasks with unique features:

1. Denoising Autoencoder

Denoising Autoencoder is trained to handle corrupted or noisy inputs, it learns to remove noise and helps in reconstructing clean data. It prevent the network from simply memorizing the input and encourages learning the core features.

2. Sparse Autoencoder

Sparse Autoencoder contains more hidden units than input features but only allows a few neurons to be active simultaneously. This sparsity is controlled by zeroing some hidden units, adjusting activation functions or adding a sparsity penalty to the loss function.

3. Variational Autoencoder

Variational autoencoder (VAE) makes assumptions about the probability distribution of the data and tries to learn a better approximation of it. It uses stochastic gradient descent to optimize and learn the distribution of latent variables. They used for generating new data such as creating realistic images or text.

It assumes that the data is generated by a Directed Graphical Model and tries to learn an approximation to $q\phi(z|x) q\phi(z|x)$ to the conditional property $q\theta(z|x) q\theta(z|x)$ where ϕ ϕ and θ θ are the parameters of the encoder and the decoder respectively.

4. Convolutional Autoencoder

Convolutional autoencoder uses convolutional neural networks (CNNs) which are designed for processing images. The encoder extracts features using convolutional layers and the decoder reconstructs the image through **deconvolution** also called as upsampling.

Implementation of Autoencoders

We will create a simple autoencoder with two Dense layers: an **encoder** that compresses images into a 64-dimensional latent vector and a **decoder** that reconstructs the original image from this compressed form.

Step 1: Import necessary libraries

We will be using Matplotlib, NumPy, TensorFlow and the MNIST dataset loader for this.

```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers, losses
from tensorflow.keras.models import Model
from keras.datasets import mnist
```

Step 2: Load the MNIST dataset

We will be loading the **MNIST** dataset which is inbuilt dataset and **normalize pixel values** to [0,1] also reshape the data to fit the model.

```
(x_train, _), (x_test, _) = mnist.load_data()

x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.

x_train = np.reshape(x_train, (len(x_train), 28, 28, 1))
x_test = np.reshape(x_test, (len(x_test), 28, 28, 1))
```

Output:

Shape of the training data: (60000, 28, 28)
Shape of the testing data: (10000, 28, 28)

Step 3: Define a basic Autoencoder

Creating a simple autoencoder class with an encoder and decoder using Keras Sequential model.

- **layers.Input(shape=(28, 28, 1))**: Input layer expecting grayscale images of size 28x28.
- **layers.Dense(latent_dimensions, activation='relu'):** Dense layer that compresses the input to the latent space using ReLU activation.
- **layers.Dense(28 * 28, activation='sigmoid'):** Dense layer that expands the latent vector back to the original image size with sigmoid activation.

```
class SimpleAutoencoder(Model):
def __init__(self, latent_dimensions):
super(SimpleAutoencoder, self).__init__()
self.encoder = tf.keras.Sequential([
layers.Input(shape=(28, 28, 1)),
layers.Flatten(),
layers.Dense(latent_dimensions, activation='relu'),
])

self.decoder = tf.keras.Sequential([
layers.Dense(28 * 28, activation='sigmoid'),
layers.Reshape((28, 28, 1))
])

def call(self, input_data):
```

```
encoded = self.encoder(input_data)
decoded = self.decoder(encoded)
return decoded
```

Step 4: Compiling and Fitting Autoencoder

Here we compile the model using **Adam optimizer** and **Mean Squared Error loss** also we train for 10 epochs with batch size 256.

- **latent_dimensions = 64**: Sets the size of the compressed latent space to 64.

```
latent_dimensions = 64
autoencoder = SimpleAutoencoder(latent_dimensions)
autoencoder.compile(optimizer='adam', loss=losses.MeanSquaredError())

autoencoder.fit(x_train, x_train,
epochs=10,
batch_size=256,
shuffle=True,
validation_data=(x_test, x_test))
```

Output:

```
Epoch 1/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 3s 10ms/step - loss: 0.0973 - val_loss: 0.0321
Epoch 2/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 2s 9ms/step - loss: 0.0289 - val_loss: 0.0209
Epoch 3/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 2s 8ms/step - loss: 0.0197 - val_loss: 0.0151
Epoch 4/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 3s 10ms/step - loss: 0.0144 - val_loss: 0.0115
Epoch 5/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 3s 10ms/step - loss: 0.0112 - val_loss: 0.0093
Epoch 6/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 2s 9ms/step - loss: 0.0091 - val_loss: 0.0077
Epoch 7/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 2s 8ms/step - loss: 0.0076 - val_loss: 0.0067
Epoch 8/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 2s 8ms/step - loss: 0.0066 - val_loss: 0.0059
Epoch 9/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 3s 9ms/step - loss: 0.0060 - val_loss: 0.0055
Epoch 10/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 3s 10ms/step - loss: 0.0056 - val_loss: 0.0051
```

Training

Step 5: Visualize original and reconstructed data

Now compare original images and their reconstructions from the autoencoder.

- **encoded_imgs = autoencoder.encoder(x_test).numpy()**: Passes test images through the encoder to get their compressed latent representations as NumPy arrays.
- **decoded_imgs = autoencoder.decoder(encoded_imgs).numpy()**: Reconstructs images by passing the latent representations through the decoder and converts them to NumPy arrays.

```
encoded_imgs = autoencoder.encoder(x_test).numpy()
decoded_imgs = autoencoder.decoder(encoded_imgs).numpy()

n = 6
plt.figure(figsize=(12, 6))
for i in range(n):
ax = plt.subplot(2, n, i + 1)
plt.imshow(x_test[i].reshape(28, 28), cmap='gray')
plt.title("Original")
```

```
plt.axis('off')

ax = plt.subplot(2, n, i + 1 + n)
plt.imshow(decoded_imgs[i].reshape(28, 28), cmap='gray')
plt.title("Reconstructed")
plt.axis('off')

plt.show()
```

GAN vs Transformer Models

Understanding GANs

Generative Adversarial Networks (GANs) are a framework consisting of two competing neural networks: a generator that creates fake data and a discriminator that tries to differentiate between real and fake data. The generator learns to produce increasingly realistic data by trying to fool the discriminator, while the discriminator becomes better at detecting fake data. This adversarial training process continues until the generator produces data so realistic that the discriminator can barely tell the difference from real data.

GAN architecture

GANs consist of two neural networks trained in opposition to one another:

- **Generator**: Produces synthetic data that mimics the distribution of real training data.
- **Discriminator**: Attempts to distinguish between real and generated (fake) samples.

The underlying training objective is modeled as a minimax optimization problem, where the Generator seeks to minimize the Discriminator's accuracy and the Discriminator itself aims to maximize it. This dynamic leads to a equilibrium in which the generated data becomes statistically indifferentiable from the real data.

Understanding Transformers

Transformers are neural networks that use self-attention mechanisms to process data sequences in parallel. They can focus on all parts of an input simultaneously, which makes them effective at capturing relationships between elements in sequential data. This architecture powers modern models like GPT, BERT and ChatGPT, enabling unforeseen performance in language understanding, generation and various other tasks.

Transformer Architecture

Key components of transformers include:

- **Self-Attention**: Allows each position to attend to all other positions in the sequence
- **Encoder-Decoder Architecture**: Processes input and generates output sequences
- **Positional Encoding**: Provides sequence order information since attention is position-agnostic

Attention mechanism computes relationships between all pairs of positions in a sequence, enabling the model to focus on relevant parts. This parallel processing capability makes Transformers highly efficient for training on modern hardware.

Differences between GAN and Transformers

Aspect	GANs	Transformers
Training Paradigm	Unsupervised adversarial training with competing networks	Supervised learning with next-token prediction
Data Processing	Fixed-size inputs/outputs	Variable-length sequences processed in parallel

Architecture	Generator vs Discriminator competition	Encoder-decoder with attention mechanisms
Training Challenges	Training instability, delicate balance	High computational requirements, quadratic complexity
Pretrained Models	Rarely used, train from scratch	Commonly used (BERT, GPT, T5)
Best Applications	Image/video generation, creative tasks, data augmentation	NLP, sequential data, language modeling
Dependency Modeling	Short-range, local patterns	Long-range, contextual relationships

Real-World Applications

GANs (Generative Adversarial Networks) are ideal when the goal is to create realistic synthetic data, particularly in visual domains. They perform well in tasks like:

- High-quality image and video generation
- Style transfer and creative applications
- Data augmentation when labeled samples are limited
- Synthetic dataset creation for training deep models
- Deepfakes and media synthesis, where realism is important

Transformers are best suited for tasks involving sequential or structured input. They work well in:

- Natural language processing such as translation, summarization and sentiment analysis
- Conversational AI and question answering
- Code generation and programming assistance
- Document understanding and information retrieval

Choosing the Right Architecture

Choose GANs if:	Choose Transformers if:
Creating visual content is priority	Processing text/sequential data
Unsupervised generation needed	Understanding context is crucial
Fixed output size acceptable	Variable input/output lengths required
Visual quality over interpretability	Leveraging pretrained models preferred

Types of Generative Adversarial Networks (GANs): GANs consist of two neural networks, the generator and the discriminator that compete with each other. Variants of GANs include:

- Deep Convolutional GAN (DCGAN)
- Conditional GAN (cGAN)
- Cycle-Consistent GAN (CycleGAN)

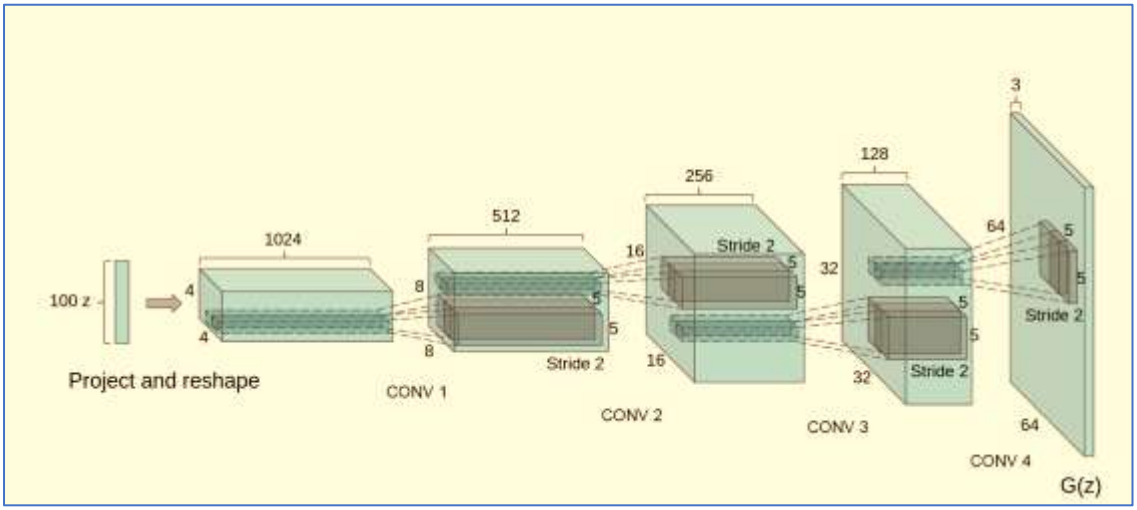
Deep Convolutional GAN with Keras

Deep Convolutional GAN (DCGAN) was proposed by a researcher from MIT and Facebook AI research. It is widely used in many convolution-based generation-based techniques. The focus of this paper was to make training GANs stable. Hence, they proposed some architectural changes in the computer vision problems. In this article, we will be using DCGAN on the fashion MNIST dataset to generate images related to clothes.

Need for DCGANs:

DCGANs are introduced to reduce the problem of mode collapse. Mode collapse occurs when the generator got biased towards a few outputs and can't able to produce outputs of every variation from the dataset. For example- take the case of mnist digits dataset (digits from 0 to 9) , we want the generator should generate all type of digits but sometimes our generator got biased towards two to three digits and produce them only. Because of that the discriminator also got optimized towards that particular digits only, and this state is known as mode collapse. But this problem can be overcome by using DCGANs.

Architecture:



The generator of the DCGAN architecture takes 100 uniform generated values using normal distribution as an input. First, it changes the dimension to 4x4x1024 and performed a fractionally stridden convolution 4 times with a stride of 1/2 (this means every time when applied, it doubles the image dimension while reducing the number of output channels). The generated output has dimensions of (64, 64, 3). There are some architectural changes proposed in the generator such as the removal of all fully connected layers, and the use of Batch Normalization which helps in stabilizing training. In this paper, the authors use ReLU activation function in all layers of the generator, except for the output layers. We will be implementing generator with similar guidelines but not completely the same architecture.

The role of the discriminator here is to determine that the image comes from either a real dataset or a generator. The discriminator can be simply designed similar to a convolution neural network that performs an image classification task. However, the authors of this paper suggested some changes in the discriminator architecture. Instead of fully connected layers, they used only strided-convolutions with LeakyReLU as an activation function, the input of the generator is a single image from the dataset or generated image and the output is a score that determines whether the image is real or generated**.**

Implementation:

In this section we will be discussing the implementation of DCGAN in Keras, since our dataset in the Fashion MNIST dataset, this dataset contains images of size (28, 28) of 1 color channel instead of (64, 64) of 3 color channels. So, we need to make some changes in the architecture, we will be discussing these changes as we go along.

In the first step, we need to import the necessary classes such as TensorFlow, Keras, matplotlib, etc. We will be using TensorFlow version 2. This version of **TensorFlow** provides inbuilt support for the Keras library as its default High-level API.

```
# code % matplotlib inline
import tensorflow as tf
from tensorflow import keras
import numpy as np
import matplotlib.pyplot as plt
from tqdm import tqdm
from IPython import display

# Check tensorflow version
print('Tensorflow version:', tf.__version__)
```

Now we load the fashion-MNIST dataset, the good thing is that the dataset can be imported from tf.keras.datasets API. So, we don't need to load datasets manually by copying files. This dataset contains 60k

training images and 10k test images for each dimension (28, 28, 1). Since the value of each pixel is in the range (0, 255), we divide these values by 255 to normalize it.

```
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.fashion_mnist.load_data()
x_train = x_train.astype(np.float32) / 255.0
x_test = x_test.astype(np.float32) / 255.0
x_train.shape, x_test.shape
```

```
((60000, 28, 28), (10000, 28, 28))
```

Now in the next step, we will be visualizing some of the images from the Fashion-MNIST dataset, we use matplotlib library for that.

```
# We plot first 25 images of training dataset
plt.figure(figsize =(10, 10))
for i in range(25):
plt.subplot(5, 5, i + 1)
plt.xticks([])
plt.yticks([])
plt.grid(False)
plt.imshow(x_train[i], cmap = plt.cm.binary)
plt.show()
```

Original Fashion MNIST images

Now, we define training parameters such as batch size and divide the dataset into batches, and fill those batches by randomly sampling the training data.

```
batch_size = 32
# replacing the selected elements with new elements.
def create_batch(x_train):
# Correct indentation here
dataset = tf.data.Dataset.from_tensor_slices(x_train).shuffle(1000)

# Combines consecutive elements of this dataset into batches
dataset = dataset.batch(batch_size, drop_remainder=True).prefetch(1)

# Creates a Dataset that prefetches elements from this dataset
return dataset
```

Now, we define the generator architecture, this generator architecture takes a vector of size 100 and first reshape that into (7, 7, 128) vector and then, it applies transpose convolution on that reshaped image in combination with batch normalization. The output of this generator is a trained image of dimension (28, 28, 1).

```
# code
num_features = 100

generator = keras.models.Sequential([
keras.layers.Dense(7 * 7 * 128, input_shape =[num_features]),
keras.layers.Reshape([7, 7, 128]),
keras.layers.BatchNormalization(),
keras.layers.Conv2DTranspose(
64, (5, 5), (2, 2), padding =&amp;quot;same&amp;quot;, activation =&amp;quot;selu&amp;quot;),
keras.layers.BatchNormalization(),
keras.layers.Conv2DTranspose(
1, (5, 5), (2, 2), padding =&amp;quot;same&amp;quot;, activation =&amp;quot;tanh&amp;quot;),
])
generator.summary()
```

```
Model:                                                          "sequential"
_________________________________________________________________
Layer  (type)                          Output  Shape                  Param  #
```

```
=================================================================
dense (Dense)                (None, 6272)              633472
_________________________________________________________________
reshape (Reshape)            (None, 7, 7, 128)         0
_________________________________________________________________
batch_normalization (BatchNo (None, 7, 7, 128)         512
_________________________________________________________________
conv2d_transpose (Conv2DTran (None, 14, 14, 64)        204864
_________________________________________________________________
batch_normalization_1 (Batch (None, 14, 14, 64)        256
_________________________________________________________________
conv2d_transpose_1 (Conv2DTr (None, 28, 28, 1)         1601
=================================================================
Total params: 840,705
Trainable params: 840,321
Non-trainable params: 384
_________________________________________________________________
```

Now, we define discriminator architecture, the discriminator takes an image of size 28*28 with 1 color channel and outputs a scalar value representing an image from either dataset or generated image.

```
discriminator = keras.models.Sequential([
keras.layers.Conv2D(64, (5, 5), (2, 2), padding =&quot;same&quot;, input_shape =[28, 28, 1]),
keras.layers.LeakyReLU(0.2),
keras.layers.Dropout(0.3),
keras.layers.Conv2D(128, (5, 5), (2, 2), padding =&quot;same&quot;),
keras.layers.LeakyReLU(0.2),
keras.layers.Dropout(0.3),
keras.layers.Flatten(),
keras.layers.Dense(1, activation ='sigmoid')
])
discriminator.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 14, 14, 64)        1664
_________________________________________________________________
leaky_re_lu (LeakyReLU)      (None, 14, 14, 64)        0
_________________________________________________________________
dropout (Dropout)            (None, 14, 14, 64)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 7, 7, 128)         204928
_________________________________________________________________
leaky_re_lu_1 (LeakyReLU)    (None, 7, 7, 128)         0
_________________________________________________________________
dropout_1 (Dropout)          (None, 7, 7, 128)         0
_________________________________________________________________
flatten (Flatten)            (None, 6272)              0
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 6273
=================================================================
Total params: 212,865
Trainable params: 212,865
Non-trainable params: 0
_________________________________________________________________
```

Now we need to compile our DCGAN model (combination of generator and discriminator), we will first compile the discriminator and set its training to False, because we first want to train the generator.

```
# compile discriminator using binary cross entropy loss and adam optimizer
discriminator.compile(loss =&amp;quot;binary_crossentropy&amp;quot;, optimizer =&amp;quot;adam&amp;quot;)
# make  discriminator no-trainable as of  now
discriminator.trainable = False
# Combine both generator and discriminator
gan = keras.models.Sequential([generator, discriminator])
# compile generator using binary cross entropy loss and adam optimizer

gan.compile(loss =&amp;quot;binary_crossentropy&amp;quot;, optimizer =&amp;quot;adam&amp;quot;)
```

Now, we define the training procedure for this GAN model, we will be using tqdm package which we have imported earlier., this package helps in visualizing training.

```
seed = tf.random.normal(shape =[batch_size, 100])

def train_dcgan(gan, dataset, batch_size, num_features, epochs = 5):
generator, discriminator = gan.layers
for epoch in tqdm(range(epochs)):
print()
print(&amp;quot;Epoch {}/{}&amp;quot;.format(epoch + 1, epochs))

for X_batch in dataset:
# create a random noise of sizebatch_size * 100
# to passit into the generator
noise = tf.random.normal(shape =[batch_size, num_features])
generated_images = generator(noise)

# take batch of generated image and real image and
#  use them to train  the discriminator
X_fake_and_real = tf.concat([generated_images, X_batch], axis = 0)
y1 = tf.constant([[0.]] * batch_size + [[1.]] * batch_size)
discriminator.trainable = True
discriminator.train_on_batch(X_fake_and_real, y1)

# Here we will be training our GAN model, in this step
#  we pass noise that uses generatortogenerate the image
#  and pass it with labels as [1] So, it can fool the discriminator
noise = tf.random.normal(shape =[batch_size, num_features])
y2 = tf.constant([[1.]] * batch_size)
discriminator.trainable = False
gan.train_on_batch(noise, y2)

# generate images for the GIF as we go
generate_and_save_images(generator, epoch + 1, seed)

generate_and_save_images(generator, epochs, seed)
```

Now we define a function that generates and save images from generator (during training). We will use these generated images to plot the GIF later.

```
def generate_and_save_images(model, epoch, test_input):
# Indent this line properly to be part of the function
predictions = model(test_input, training=False)

fig = plt.figure(figsize=(10, 10))
```

```
for i in range(25):
plt.subplot(5, 5, i + 1)
plt.imshow(predictions[i, :, :, 0] * 127.5 + 127.5, cmap='binary')
plt.axis('off')

# Save the generated images as a PNG file
plt.savefig('image_epoch_{:04d}.png'.format(epoch))
```

Now, we need to train the model but before that, we also need to create batches of training data and add a dimension that represents number of color maps.

```
# reshape to add a color map
x_train_dcgan = x_train.reshape(-1, 28, 28, 1) * 2. - 1.
# create batches
dataset = create_batch(x_train_dcgan)
# callthe training function with 10 epochs and record time %% time
train_dcgan(gan, dataset, batch_size, num_features, epochs = 10)
```

```
0%|                                                  |    0/10    [00:00<?,     ?it/s]
Epoch                                                                        1/10

 10%|?                                    |    1/10    [01:04<09:39,    64.37s/it]
Epoch                                                                        2/10

 20%|??                                   |    2/10    [02:10<08:39,    64.99s/it]
Epoch                                                                        3/10

 30%|???                                  |    3/10    [03:14<07:33,    64.74s/it]
Epoch                                                                        4/10

 40%|????                                 |    4/10    [04:19<06:27,    64.62s/it]
Epoch                                                                        5/10

 50%|?????                                |    5/10    [05:23<05:22,    64.58s/it]
Epoch                                                                        6/10

 60%|??????                               |    6/10    [06:27<04:17,    64.47s/it]
Epoch                                                                        7/10

 70%|???????                              |    7/10    [07:32<03:13,    64.55s/it]
Epoch                                                                        8/10

 80%|????????                             |    8/10    [08:37<02:08,    64.48s/it]
Epoch                                                                        9/10

 90%|?????????                            |    9/10    [09:41<01:04,    64.54s/it]
Epoch                                                                       10/10

100%|??????????|                          10/10          [10:46<00:00,    64.61s/it]
CPU    times:    user    7min    4s,    sys:    33.3    s,    total:    7min    37s
Wall time: 10min 46s
```

Now we will define a function that takes the saved images and convert them into GIF. We use this function from here

```
import imageio
import glob
```

```
anim_file = 'dcgan_results.gif'

with imageio.get_writer(anim_file, mode ='I') as writer:
filenames = glob.glob('image*.png')
filenames = sorted(filenames)
last = -1
for i, filename in enumerate(filenames):
frame = 2*(i)
if round(frame) &gt; round(last):
last = frame
else:
continue
image = imageio.imread(filename)
writer.append_data(image)
image = imageio.imread(filename)
writer.append_data(image)
display.Image(filename = anim_file)
```

Conditional Generative Adversarial Network

Architecture and Working of CGANs

Conditional GANs extend the basic GAN framework by conditioning both the generator and discriminator on additional information. This conditioning helps to direct the generation process helps in making it more controlled and focused.

1. Generator in CGANs: The generator creates synthetic data such as images, text or videos. It takes two inputs:

- **Random Noise (z)**: A vector of random values that adds diversity to generated outputs.
- **Conditioning Information (y)**: Extra data like labels or context that guides what the generator produces for example a class label such as "cat" or "dog".

The generator combines the noise and the conditioning information to produce realistic data that matches the given condition. For example if the condition y is "cat" the generator will create an image of a cat.

2. Discriminator in CGANs: The discriminator is a binary classifier that decides whether input data is real or fake. It also receives two inputs:

- **Real Data (x)**: Actual samples from the dataset.
- **Conditioning Information (y)**: The same condition given to the generator.

Using both the real/fake data and the condition, the discriminator learns to judge if the data is genuine and if it matches the condition. For example if the input is an image labeled "cat" the discriminator verifies whether it truly looks like a real cat.

3. Interaction Between Generator and Discriminator: The generator and discriminator train together through adversarial training:

- The generator tries to create fake data based on noise (z) and condition (y) that can fool the discriminator.
- The discriminator attempts to correctly classify real vs. fake data considering the condition (y).

The goal of the adversarial process is:

- **Generator**: Produce data that the discriminator believes is real.
- **Discriminator**: Accurately distinguish between real and fake data.

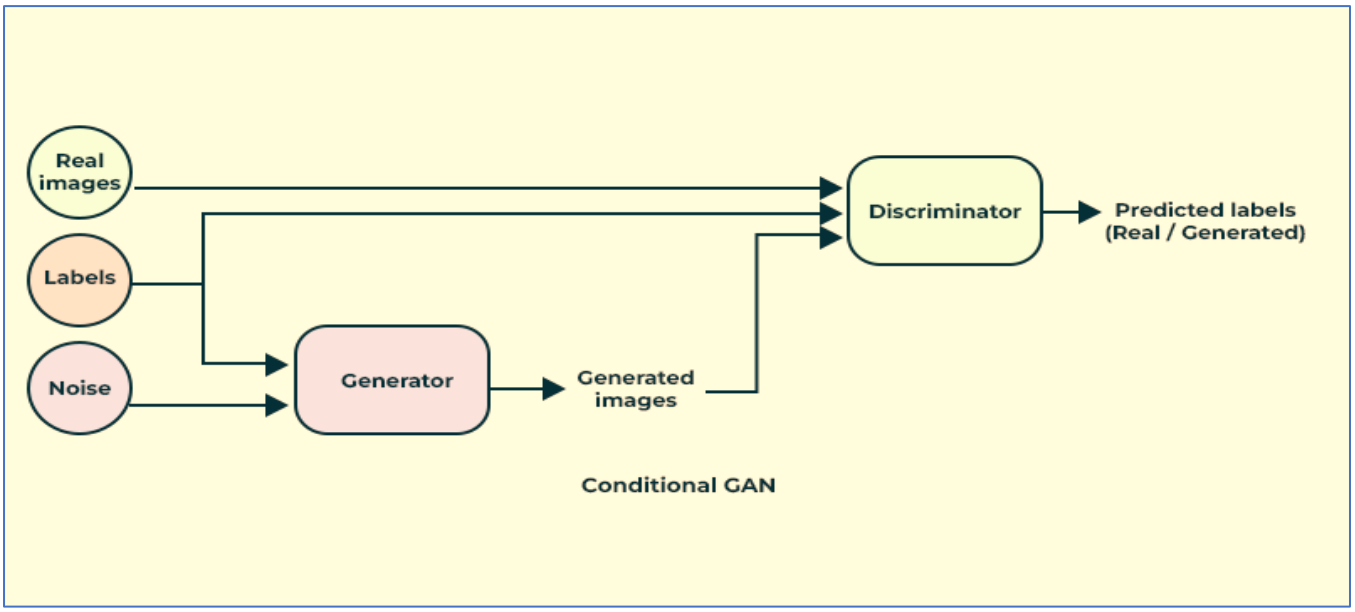
4. Loss Function and Training: Training is guided by a loss function that balances the generator and discriminator:

$\min G \max DV(D,G) = E_{x \sim pdata(x)}[\log D(x|y)] + E_{z \sim pz(z)}[\log(1 - D(G(z|y)))]$

- The first term encourages the discriminator to classify real samples correctly.
- The second term pushes the generator to produce samples that the discriminator classifies as real.

Here E represents the expected value $pdata$ is the real data distribution and pz is the prior noise distribution.

As training progresses both the generator and discriminator improve. This adversarial process results in the generator producing more realistic data conditioned on the input information.



Conditional Generative Adversarial Network

Implementing CGAN on CiFAR-10

We will build and train a Conditional Generative Adversarial Network (CGAN) to generate class-specific images from the CIFAR-10 dataset. Below are the key steps involved:

Step 1: Importing Necessary Libraries

We will import TensorFlow, NumPy, Keras and Matplotlib libraries for building models, loading data and visualization.

```
import tensorflow as tf
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.datasets import cifar10
from keras.preprocessing import image
import keras.backend as K
import matplotlib.pyplot as plt
import numpy as np
import time
from tqdm import tqdm
```

Step 2: Loading Dataset and Declaring Variables

- Load the CIFAR-10 dataset using TensorFlow datasets or tf.data.Dataset.
- Define global variables such as number of epochs, batch size and image dimensions.

```
batch_size = 16
epoch_count = 50
noise_dim = 100
n_class = 10
```

```
tags = ['Airplane', 'Automobile', 'Bird', 'Cat', 'Deer', 'Dog', 'Frog', 'Horse',
'Ship', 'Truck']
img_size = 32

(X_train, y_train), (_, _) = cifar10.load_data()

X_train = (X_train - 127.5) / 127.5

dataset = tf.data.Dataset.from_tensor_slices((X_train, y_train))
dataset = dataset.shuffle(buffer_size=1000).batch(batch_size)
```

Step 3: Visualizing Sample Images

Now we will visualize the images from the dataset to understand class distributions and data shape.

```
plt.figure(figsize=(2,2))
idx = np.random.randint(0,len(X_train))
img = image.array_to_img(X_train[idx], scale=True)
plt.imshow(img)
plt.axis('off')
plt.title(tags[y_train[idx][0]])
plt.show()
```

Step 4: Defining Loss Functions and Optimizers

In the next step we need to define the Loss function and optimizer for the discriminator and generator networks in a Conditional Generative Adversarial Network(CGANS).

- Use **Binary Cross-Entropy** Loss for both generator and discriminator.
- Define discriminator loss as sum of real and fake losses.
- The binary entropy calculates two losses: **real_loss**: Loss when the discriminator tries to classify real data as real and **fake_loss** : Loss when the discriminator tries to classify fake data as fake
- **d_optimizer** and **g_optimizer** are used to update the trainable parameters of the discriminator and generator during training.
- Use **Adam optimizer** for both networks.

```
bce_loss = tf.keras.losses.BinaryCrossentropy()


def discriminator_loss(real, fake):
real_loss = bce_loss(tf.ones_like(real), real)
fake_loss = bce_loss(tf.zeros_like(fake), fake)
total_loss = real_loss + fake_loss
return total_loss

def generator_loss(preds):
return bce_loss(tf.ones_like(preds), preds)

d_optimizer=Adam(learning_rate=0.0002, beta_1 = 0.5)
g_optimizer=Adam(learning_rate=0.0002, beta_1 = 0.5)
```

Step 5: Building the Generator Model

- Input: noise vector (latent space) and label.

- Convert label to a vector using an embedding layer (size 50).
- Process noise through dense layers with LeakyReLU activation.
- Reshape and concatenate label embedding with noise features.
- Use Conv2DTranspose layers to up-sample into 32×32×3 images.
- Output layer uses tanh activation to scale pixels between -1 and 1.

```
def build_generator():


in_label = tf.keras.layers.Input(shape=(1,))
li = tf.keras.layers.Embedding(n_class, 50)(in_label)

n_nodes = 8 * 8
li = tf.keras.layers.Dense(n_nodes)(li)
li = tf.keras.layers.Reshape((8, 8, 1))(li)
in_lat = tf.keras.layers.Input(shape=(noise_dim,))

n_nodes = 128 * 8 * 8
gen = tf.keras.layers.Dense(n_nodes)(in_lat)
gen = tf.keras.layers.LeakyReLU(alpha=0.2)(gen)
gen = tf.keras.layers.Reshape((8, 8, 128))(gen)
merge = tf.keras.layers.Concatenate()([gen, li])

gen = tf.keras.layers.Conv2DTranspose(
128, (4, 4), strides=(2, 2), padding='same')(merge)
gen = tf.keras.layers.LeakyReLU(alpha=0.2)(gen)

gen = tf.keras.layers.Conv2DTranspose(
128, (4, 4), strides=(2, 2), padding='same')(gen)
gen = tf.keras.layers.LeakyReLU(alpha=0.2)(gen)

out_layer = tf.keras.layers.Conv2D(
3, (8, 8), activation='tanh', padding='same')(gen)

model = Model([in_lat, in_label], out_layer)
return model


g_model = build_generator()
g_model.summary()
```

Output:

Building the Generator Model

Step 6: Building the Discriminator Model

- Input: image and label.
- Embed label into a 50-dimensional vector.
- Reshape and concatenate label embedding with the input image.
- Apply two Conv2D layers with LeakyReLU activations to extract features.
- Flatten features, apply dropout to prevent overfitting.
- Final dense layer with sigmoid activation outputs probability of real or fake.

```
def build_discriminator():


in_label = tf.keras.layers.Input(shape=(1,))
```

```
li = tf.keras.layers.Embedding(n_class, 50)(in_label)

n_nodes = img_size * img_size
li = tf.keras.layers.Dense(n_nodes)(li)

li = tf.keras.layers.Reshape((img_size, img_size, 1))(li)



in_image = tf.keras.layers.Input(shape=(img_size, img_size, 3))

merge = tf.keras.layers.Concatenate()([in_image, li])

fe = tf.keras.layers.Conv2D(128, (3,3), strides=(2,2), padding='same')(merge)
fe = tf.keras.layers.LeakyReLU(alpha=0.2)(fe)

fe = tf.keras.layers.Conv2D(128, (3,3), strides=(2,2), padding='same')(fe)
fe = tf.keras.layers.LeakyReLU(alpha=0.2)(fe)

fe = tf.keras.layers.Flatten()(fe)

fe = tf.keras.layers.Dropout(0.4)(fe)

out_layer = tf.keras.layers.Dense(1, activation='sigmoid')(fe)

model = Model([in_image, in_label], out_layer)

return model


d_model = build_discriminator()
d_model.summary()
```

Step 7: Creating Training Step Function

- Use TensorFlow's Gradient Tape to calculate and apply gradients for both networks.
- Alternate training discriminator on real and fake data.
- Train generator to fool discriminator.
- Use **@tf.function** for efficient graph execution.

```
@tf.function
def train_step(dataset):

real_images, real_labels = dataset

random_latent_vectors = tf.random.normal(shape=(batch_size, noise_dim))
generated_images = g_model([random_latent_vectors, real_labels])

with tf.GradientTape() as tape:
pred_fake = d_model([generated_images, real_labels])
pred_real = d_model([real_images, real_labels])

d_loss = discriminator_loss(pred_real, pred_fake)

grads = tape.gradient(d_loss, d_model.trainable_variables)
```

```
d_optimizer.apply_gradients(zip(grads, d_model.trainable_variables))


random_latent_vectors = tf.random.normal(shape=(batch_size, noise_dim))


with tf.GradientTape() as tape:
fake_images = g_model([random_latent_vectors, real_labels])
predictions = d_model([fake_images, real_labels])
g_loss = generator_loss(predictions)

grads = tape.gradient(g_loss, g_model.trainable_variables)
g_optimizer.apply_gradients(zip(grads, g_model.trainable_variables))

return d_loss, g_loss
```

Step 8: Visualizing Generated Images

- After each epoch we will generate images conditioned on different labels.
- Display or save generated images to monitor training progress.

```
def show_samples(num_samples, n_class, g_model):
fig, axes = plt.subplots(10,num_samples, figsize=(10,20))
fig.tight_layout()
fig.subplots_adjust(wspace=None, hspace=0.2)

for l in np.arange(10):
random_noise = tf.random.normal(shape=(num_samples, noise_dim))
label = tf.ones(num_samples)*l
gen_imgs = g_model.predict([random_noise, label])
for j in range(gen_imgs.shape[0]):
img = image.array_to_img(gen_imgs[j], scale=True)
axes[l,j].imshow(img)
axes[l,j].yaxis.set_ticks([])
axes[l,j].xaxis.set_ticks([])

if j ==0:
axes[l,j].set_ylabel(tags[l])
plt.show()
```

Step 9: Train the Model

- At the final step we will start training the model for specified epochs.
- Print losses regularly to monitor performance.
- Longer training typically results in higher quality images.

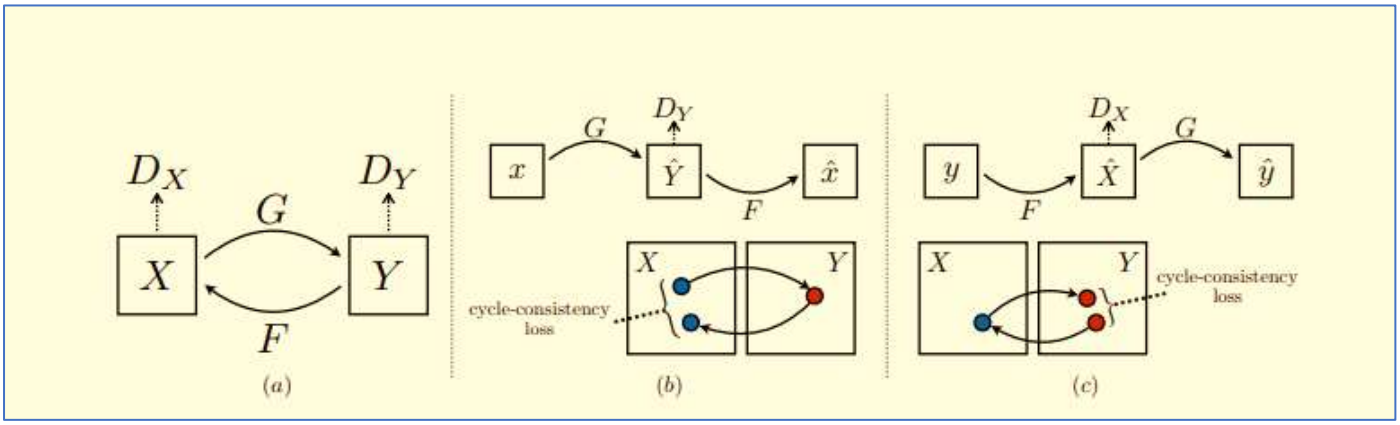
```
def train(dataset, epochs=epoch_count):

for epoch in range(epochs):
print('Epoch: ', epochs)
d_loss_list = []
g_loss_list = []
q_loss_list = []
start = time.time()

itern = 0
for image_batch in tqdm(dataset):
d_loss, g_loss = train_step(image_batch)
```

```
d_loss_list.append(d_loss)
g_loss_list.append(g_loss)
itern=itern+1

show_samples(3, n_class, g_model)

print (f'Epoch: {epoch} -- Generator Loss: {np.mean(g_loss_list)}, Discriminator
Loss: {np.mean(d_loss_list)}\n')
print (f'Took {time.time()-start} seconds. \n\n')


train(dataset, epochs=epoch_count)
```

Cycle Generative Adversarial Network (CycleGAN)

Architecture of CycleGAN

1. Generators: Create new images in the target style.



CycleGAN has two generators G and F:

- G transforms images from domain X like photos to domain Y like artwork.
- F transforms images from domain Y back to domain X.

The generator mapping functions are as follows:

$G:X \rightarrow Y F:Y \rightarrow X \quad G:X \rightarrow Y F:Y \rightarrow X$

where $X X$ is the input image distribution and $Y Y$ is the desired output distribution such as Van Gogh styles.

2. Discriminators: Decide if images are real (from dataset) or fake (generated).

There are two discriminators $D_x D_x$ and $D_\gamma D_\gamma$.

- $D_x D_x$ distinguishes between real images from $X X$ and generated images from $F(y) F(y)$.
- $D_\gamma D_\gamma$ distinguishes between real images from $Y Y$ and generated images from $G(x) G(x)$.

To further regularize the mappings the CycleGAN uses two more loss function in addition to adversarial loss.

1. Forward Cycle Consistency Loss: Ensures that when we apply G and then F to an image we get back the original image

For example: $x \rightarrow G(x) \rightarrow F(G(x)) \approx x \; x \rightarrow G(x) \rightarrow F(G(x)) \approx x$

2. Backward Cycle Consistency Loss: Ensures that when we apply **F** and then **G** to an image we get back the original image.

For example: $x \rightarrow G G(x) \rightarrow F F(G(x)) \approx x x G G(x) F F(G(x)) \approx x$

Generator Architecture

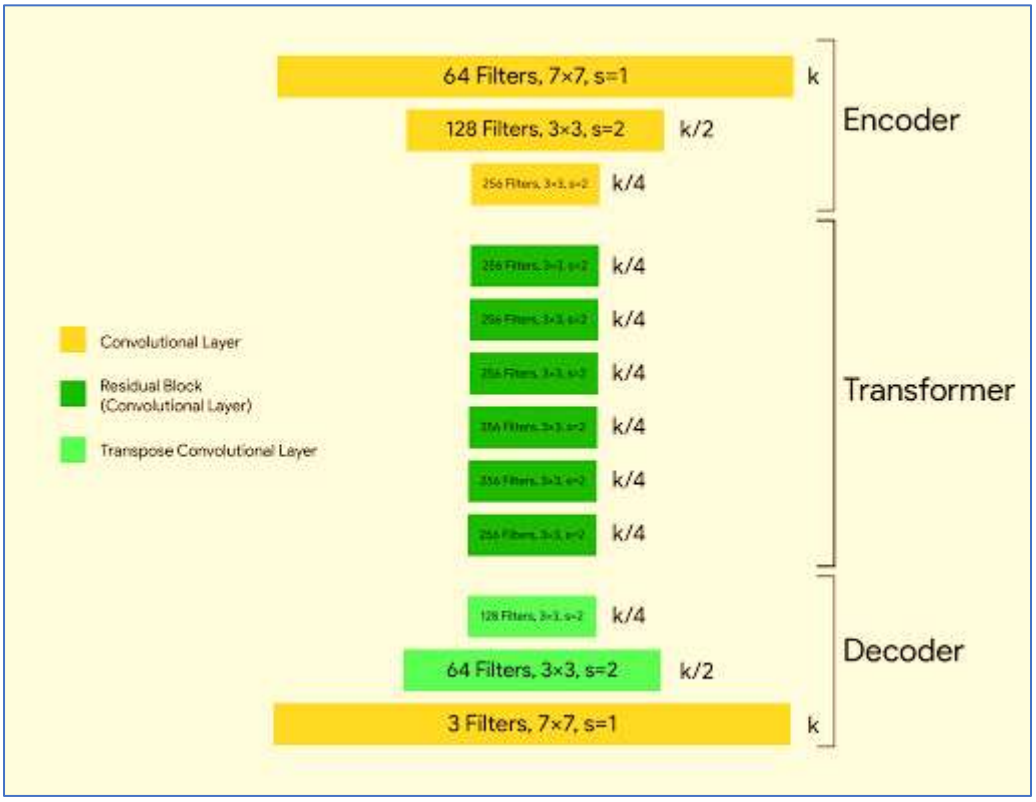
Each CycleGAN generator has three main sections:

1. **Encoder:** The input image is passed through three convolution layers which extract features and compress the image while increasing the number of channels. For example a 256×256×3 image is reduced to 64×64×256 after this step.
2. **Transformer:** The encoded image is processed through 6 or 9 residual blocks depending on the input size which helps retain important image details.
3. **Decoder:** The transformed image is up-sampled using two deconvolution layers and restoring it to its original size.

Generator Structure:

c7s1-64 → d128 → d256 → R256 (×6 or 9) → u128 → u64 → c7s1-3

- **c7s1-k**: 7×7 convolution layer with k filters.
- **dk**: 3×3 convolution with stride 2 (down-sampling).
- **Rk**: Residual block with two 3×3 convolutions.
- **uk**: Fractional-stride deconvolution (up-sampling).



Discriminator Architecture (PatchGAN)

In CycleGAN the discriminator uses a PatchGAN instead of a regular GAN discriminator.

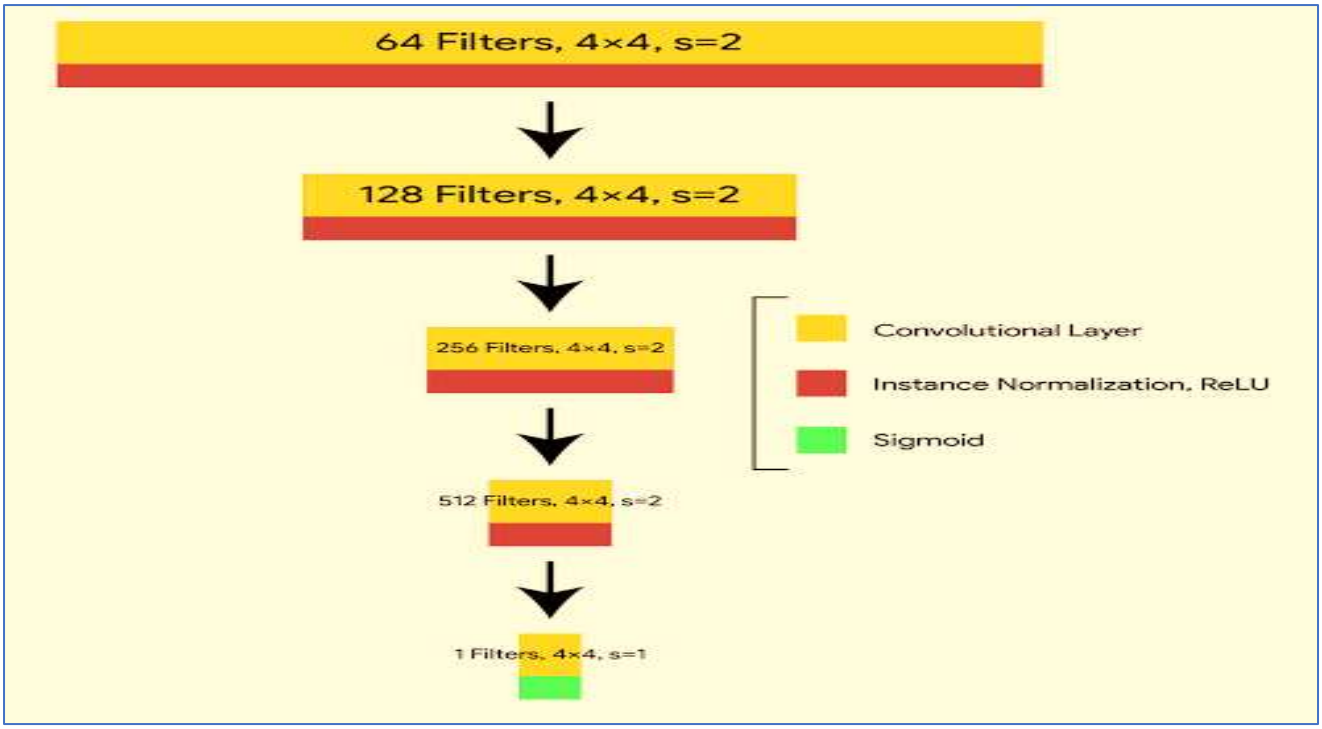
1. A regular GAN discriminator looks at the entire image (e.g 256×256 pixels) and outputs a single score that says whether the whole image is real or fake.
2. PatchGAN breaks the image into smaller patches (e.g 70×70 patches). It outputs a grid (like 70×70 values) where each value judges if the corresponding patch is real or fake.

This lets PatchGAN focus on local details such as textures and small patterns rather than the whole image at once it helps in improving the quality of generated images.

Discriminator Structure:

C64 → C128 → C256 → C512 → Final Convolution

- **Ck**: 4×4 convolution with k filters, InstanceNorm and LeakyReLU except the first layer.
- The final layer produces a 1×1 output and marking real vs. fake patches.



Cost Function in CycleGAN

CycleGAN uses a cost function or loss function to help the training process. The cost function is made up of several parts:

- **Adversarial Loss:** We apply adversarial loss to both our mappings of generators and discriminators. This adversary loss is written as :

$$\text{Lossadvers}(G,Dy,X,Y)=1m\sum(1-\text{Dy}(\text{G}(x)))2 \quad Lossadvers(G,Dy,X,Y)=m1\sum(1-Dy(G(x)))2$$

$$\text{Lossadvers}(F,Dx,Y,X)=1m\sum(1-\text{Dx}(\text{F}(y)))2 \quad Lossadvers(F,Dx,Y,X)=m1\sum(1-Dx(F(y)))2$$

- **Cycle Consistency Loss**: Given a random set of images adversarial network can map the set of input image to random permutation of images in the output domain which may induce the output distribution similar to target distribution. Thus adversarial mapping cannot guarantee the input xi to yi . For this to happen we proposed that process should be cycle-consistent. This loss function used in Cycle GAN to measure the error rate of inverse mapping G(x) -> F(G(x)). The behavior induced by this loss function cause closely matching the real input (x) and F(G(x))

$$\text{Losscyc}(G,F,X,Y)=1m[(\text{F}(\text{G}(xi))-xi)+(\text{G}(\text{F}(yi))-yi)] \quad Losscyc(G,F,X,Y)=m1[(F(G(xi))-xi)+(G(F(yi))-yi)]$$

The Cost function we used is the sum of adversarial loss and cyclic consistent loss:

$$L(G,F,Dx,Dy)=\text{Ladvers}(G,Dy,X,Y)+\text{Ladvers}(F,Dx,Y,X)+\lambda\text{Lcycl}(G,F,X,Y) \quad L(G,F,Dx,Dy)=Ladvers(G,Dy,X,Y)+Ladvers(F,Dx,Y,X)+\lambda Lcycl(G,F,X,Y)$$

and our aim is :

$$\text{argminG,FmaxDx,DyL}(G,F,Dx,Dy) \quad argG,FminDx,DymaxL(G,F,Dx,Dy)$$

Applications of CycleGAN in Image Translation

1. Collection Style Transfer: CycleGAN can learn to mimic the style of entire collections of artworks like Van Gogh, Monet or Cezanne rather than just transferring the style of a single image. Therefore it can generate different styles such as : Van Gogh, Cezanne, Monet and Ukiyo-e. This capability makes CycleGAN particularly useful for generating diverse artwork.

Style Transfer ResultsComparison of different Style Transfer Results

2. Object Transformation: CycleGAN can transform objects between different classes, such as turning zebras into horses, apples into oranges or vice versa. This is especially useful for creative industries and content generation.

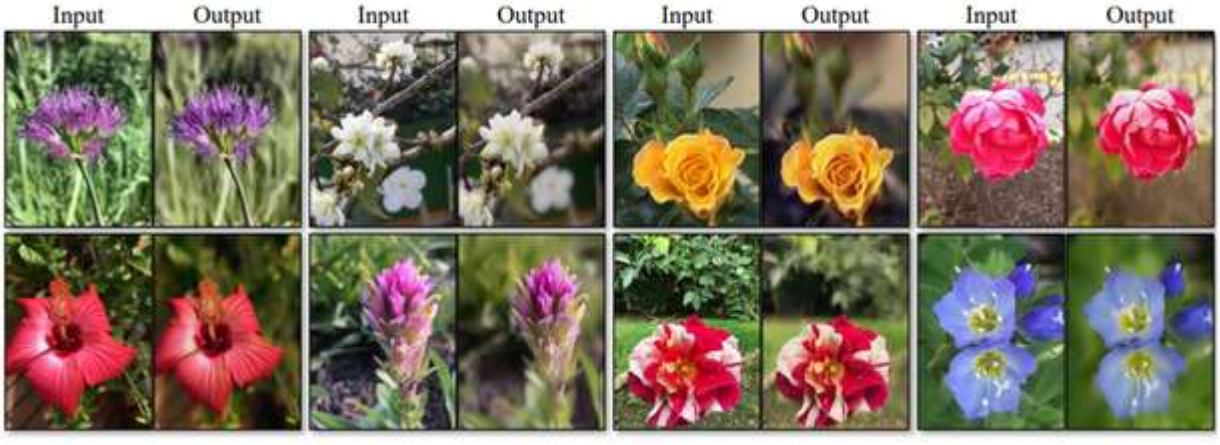
- **Apple <---> Oranges:**

3. Seasonal Transfer: CycleGAN can be used for seasonal image transformation, such as converting winter photos to summer scenes and vice versa. For instance, it was trained on photos of Yosemite in both winter and summer to enable this transformation.

4. Photo Generation from Paintings: CycleGAN can transform a painting into a photo and vice versa. This is useful for artistic applications where you want to blend the look of photos with artistic styles. This loss can be defined as :

$\text{Lidentity}(G,F)=\text{Ey p(y)}[\|\text{G(y)}-\text{y}\|1]+\text{Ex p(x)}[\|\text{F(x)}-\text{x}\|1]$ $Lidentity(G,F)=\text{E}y\ p(y)[\|G(y)-y\|1]+\text{E}x\ p(x)[\|F(x)-x\|1]$

5. Photo Enhancement: CycleGAN can enhance photos taken with smartphone cameras which typically have a deeper depth of field to look like those taken with DSLR cameras which have a shallower depth of field. This application is valuable for image quality improvement.



Evaluating CycleGAN's Performance

- **AMT Perceptual Studies**: It involve real people reviewing generated images to see if they look real. This is like a voting system where participants on Amazon Mechanical Turk compare AI-created images with actual ones.
- **FCN Scores**: It help to measure accuracy especially in datasets like Cityscapes. These scores check how well the AI understands objects in images by evaluating pixel accuracy and IoU (Intersection over Union) which measures how well the shapes of objects match real.

4. Deep Reinforcement Learning (DRL)

Deep Reinforcement Learning combines the representation learning power of deep learning with the decision-making ability of reinforcement learning. It helps agents to learn optimal behaviors in complex environments through trial and error using high-dimensional sensory inputs.

- Deep Reinforcement Learning
- Reinforcement Learning
- Markov Decision Processes

A Beginner's Guide to Deep Reinforcement Learning

Deep Reinforcement Learning

Deep Reinforcement Learning (DRL) is a revolutionary Artificial Intelligence methodology that combines reinforcement learning and deep neural networks. By iteratively interacting with an environment and making choices that maximise cumulative rewards, it enables agents to learn sophisticated strategies. Agents are able to directly learn rules from sensory inputs thanks to DRL, which makes use of deep learning's ability to extract complex features from unstructured data. DRL relies heavily on Q-learning, policy gradient methods, and actor-critic systems. The notions of value networks, policy networks, and exploration-exploitation trade-offs are crucial. The uses for DRL are numerous and include robotics, gaming, banking, and healthcare. Its development from Atari games to real-world difficulties emphasises how versatile and potent it is. Sample effectiveness, exploratory tactics, and safety considerations are difficulties. The collaboration aims to drive DRL responsibly, promising an inventive future that will change how decisions are made and problems are solved.

Core Components of Deep Reinforcement Learning

Deep Reinforcement Learning (DRL) building blocks include all the aspects that power learning and empower agents to make wise judgements in their surroundings. Effective learning frameworks are produced by the cooperative interactions of these elements. The following are the essential elements:

- **Agent**: The decision-maker or learner who engages with the environment. The agent acts in accordance with its policy and gains experience over time to improve its ability to make decisions.
- **Environment**: The system outside of the agent that it communicates with. Based on the actions the agent does, it gives the agent feedback in the form of incentives or punishments.
- **State**: A depiction of the current circumstance or environmental state at a certain moment. The agent chooses its activities and makes decisions based on the state.

- **Action**: A choice the agent makes that causes a change in the state of the system. The policy of the agent guides the selection of actions.Reward: A scalar feedback signal from the environment that shows whether an agent's behaviour in a specific state is desirable. The agent is guided by rewards to learn positive behaviour.\
- **Policy**: A plan that directs the agent's decision-making by mapping states to actions. Finding an ideal policy that maximises cumulative rewards is the objective.
- **Value Function**: This function calculates the anticipated cumulative reward an agent can obtain from a specific state while adhering to a specific policy. It is beneficial in assessing and contrasting states and policies.
- **Model**: A depiction of the dynamics of the environment that enables the agent to simulate potential results of actions and states. Models are useful for planning and forecasting.
- **Exploration-Exploitation Strategy**: A method of making decisions that strikes a balance between exploring new actions to learn more and exploiting well-known acts to reap immediate benefits (exploitation).
- **Learning Algorithm:** The process by which the agent modifies its value function or policy in response to experiences gained from interacting with the environment. Learning in DRL is fueled by a variety of algorithms, including Q-learning, policy gradient, and actor-critic.
- **Deep Neural Networks:** DRL can handle high-dimensional state and action spaces by acting as function approximators in deep neural networks. They pick up intricate input-to-output mappings.
- **Experience Replay:** A method that randomly selects from stored prior experiences (state, action, reward, and next state) during training. As a result, learning stability is improved and the association between subsequent events is decreased.

These core components collectively form the foundation of **Deep Reinforcement Learning**, empowering agents to learn strategies, make intelligent decisions, and adapt to dynamic environments.

How Deep Reinforcement Learning works?

In Deep Reinforcement Learning (DRL), an agent interacts with an environment to learn how to make optimal decisions. Steps:

1. Initialization: Construct an agent and set up the issue.
2. Interaction: The agent interacts with its surroundings through acting, which results in states and rewards.
3. Learning: The agent keeps track of its experiences and updates its method for making decisions.
4. Policy Update: Based on data, algorithms modify the agent's approach.
5. Exploration-Exploitation: The agent strikes a balance between using well-known actions and trying out new ones.
6. Reward Maximization: The agent learns to select activities that will yield the greatest possible total rewards.
7. Convergence: The agent's policy becomes better and stays the same over time.
8. Extrapolation: Skilled agents can use what they've learned in fresh circumstances.
9. Evaluation: Unknown surroundings are used to assess the agent's performance.
10. Use of the trained agent in practical situations.

Solving the CartPole Problem using Deep Q-Network (DQN)

Step 1: Import Required Libraries

```
# Import Required Libraries
import numpy as np
import tensorflow as tf
import gym
```

Step 2: Define the DQN Model

```
# Define the DQN Model
class DQN(tf.keras.Model):
def __init__(self, num_actions):
super(DQN, self).__init__()
self.dense1 = tf.keras.layers.Dense(24, activation='relu')
self.dense2 = tf.keras.layers.Dense(24, activation='relu')
self.output_layer = tf.keras.layers.Dense(
num_actions, activation='linear')

def call(self, inputs):
x = self.dense1(inputs)
x = self.dense2(x)
return self.output_layer(x)

# CartPole has 2 possible actions: push left or push right
num_actions = 2
dqn_agent = DQN(num_actions)
```

Step 3: Define the DQN Algorithm Parameters

```
# Define the DQN Algorithm Parameters
learning_rate = 0.001
discount_factor = 0.99
# Initial exploration probability
exploration_prob = 1.0
# Decay rate of exploration probability
exploration_decay = 0.995
# Minimum exploration probability
min_exploration_prob = 0.1
```

Step 4: Initialize the CartPole Environment

```
# Initialize the CartPole Environment
env = gym.make('CartPole-v1')
```

Step 5: Define the Loss Function and Optimizer

```
# Define the Loss Function and Optimizer
loss_fn = tf.keras.losses.MeanSquaredError()
optimizer = tf.keras.optimizers.Adam(learning_rate=learning_rate)
```

Step 6: Training the DQN

```
# Training the DQN
num_episodes = 1000
max_steps_per_episode = 500

for episode in range(num_episodes):
state = env.reset()
episode_reward = 0

for step in range(max_steps_per_episode):
# Choose action using epsilon-greedy policy
if np.random.rand() < exploration_prob:
action = env.action_space.sample()  # Explore randomly
else:
```

```
action = np.argmax(dqn_agent(state[np.newaxis, :]))

next_state, reward, done, _ = env.step(action)

# Update the Q-values using Bellman equation
with tf.GradientTape() as tape:
current_q_values = dqn_agent(state[np.newaxis, :])
next_q_values = dqn_agent(next_state[np.newaxis, :])
max_next_q = tf.reduce_max(next_q_values, axis=-1)
target_q_values = current_q_values.numpy()
target_q_values[0, action] = reward + discount_factor * max_next_q * (1 - done)
loss = loss_fn(current_q_values, target_q_values)

gradients = tape.gradient(loss, dqn_agent.trainable_variables)
optimizer.apply_gradients(zip(gradients, dqn_agent.trainable_variables))

state = next_state
episode_reward += reward

if done:
break

# Decay exploration probability
exploration_prob = max(min_exploration_prob, exploration_prob *
exploration_decay)
if (episode + 1)%100==0:
print(f"Episode {episode + 1}: Reward = {episode_reward}")
```

Output:

```
Episode 100: Reward = 20.0
Episode 200: Reward = 36.0
Episode 300: Reward = 12.0
Episode 400: Reward = 18.0
Episode 500: Reward = 65.0
Episode 600: Reward = 172.0
Episode 700: Reward = 52.0
Episode 800: Reward = 15.0
Episode 900: Reward = 146.0
Episode 1000: Reward = 181.0
```

Step 7: Evaluating the Trained DQN

```
# Evaluating the Trained DQN
num_eval_episodes = 10
eval_rewards = []

for _ in range(num_eval_episodes):
state = env.reset()
eval_reward = 0

for _ in range(max_steps_per_episode):
action = np.argmax(dqn_agent(state[np.newaxis, :]))
next_state, reward, done, _ = env.step(action)
eval_reward += reward
state = next_state

if done:
```

```
break

eval_rewards.append(eval_reward)

average_eval_reward = np.mean(eval_rewards)
print(f"Average Evaluation Reward: {average_eval_reward}")
```

Output:

```
Average Evaluation Reward: 180.1
```

Reinforcement Learning

Reinforcement Learning (RL) is a branch of machine learning that focuses on how agents can learn to make decisions through trial and error to maximize cumulative rewards. RL allows machines to learn by interacting with an environment and receiving feedback based on their actions. This feedback comes in the form of **rewards or penalties**.

Reinforcement Learning revolves around the idea that an agent (the learner or decision-maker) interacts with an environment to achieve a goal. The agent performs actions and receives feedback to optimize its decision-making over time.

- **Agent**: The decision-maker that performs actions.
- **Environment**: The world or system in which the agent operates.
- **State**: The situation or condition the agent is currently in.
- **Action**: The possible moves or decisions the agent can make.
- **Reward**: The feedback or result from the environment based on the agent's action.

How Reinforcement Learning Works?

The RL process involves an agent performing actions in an environment, receiving rewards or penalties based on those actions, and adjusting its behavior accordingly. This loop helps the agent improve its decision-making over time to maximize the **cumulative reward**.

Here's a breakdown of RL components:

- **Policy**: A strategy that the agent uses to determine the next action based on the current state.
- **Reward Function**: A function that provides feedback on the actions taken, guiding the agent towards its goal.
- **Value Function**: Estimates the future cumulative rewards the agent will receive from a given state.
- **Model of the Environment**: A representation of the environment that predicts future states and rewards, aiding in planning.

Reinforcement Learning Example: Navigating a Maze

Imagine a robot navigating a maze to reach a diamond while avoiding fire hazards. The goal is to find the optimal path with the least number of hazards while maximizing the reward:

- Each time the robot moves correctly, it receives a reward.
- If the robot takes the wrong path, it loses points.

The robot learns by exploring different paths in the maze. By trying various moves, it evaluates the rewards and penalties for each path. Over time, the robot determines the best route by selecting the actions that lead to the highest cumulative reward.

The robot's learning process can be summarized as follows:

1. **Exploration**: The robot starts by exploring all possible paths in the maze, taking different actions at each step (e.g., move left, right, up, or down).
2. **Feedback**: After each move, the robot receives feedback from the environment:
    - A positive reward for moving closer to the diamond.
    - A penalty for moving into a fire hazard.
3. **Adjusting Behavior**: Based on this feedback, the robot adjusts its behavior to maximize the cumulative reward, favoring paths that avoid hazards and bring it closer to the diamond.
4. **Optimal Path**: Eventually, the robot discovers the optimal path with the least number of hazards and the highest reward by selecting the right actions based on past experiences.

Types of Reinforcements in RL

1. Positive Reinforcement

Positive Reinforcement is defined as when an event, occurs due to a particular behavior, increases the strength and the frequency of the behavior. In other words, it has a positive effect on behavior.

- **Advantages**: Maximizes performance, helps sustain change over time.
- **Disadvantages**: Overuse can lead to excess states that may reduce effectiveness.

2. Negative Reinforcement

Negative Reinforcement is defined as strengthening of behavior because a negative condition is stopped or avoided.

- **Advantages**: Increases behavior frequency, ensures a minimum performance standard.
- **Disadvantages**: It may only encourage just enough action to avoid penalties.

CartPole in OpenAI Gym

One of the classic RL problems is the **CartPole environment** in **OpenAI Gym**, where the goal is to balance a pole on a cart. The agent can either push the cart left or right to prevent the pole from falling over.

- **State space**: Describes the four key variables (position, velocity, angle, angular velocity) of the cart-pole system.
- **Action space**: Discrete actions—either move the cart left or right.
- **Reward**: The agent earns 1 point for each step the pole remains balanced.

```
import gym
import numpy as np
import warnings

# Suppress specific deprecation warnings
warnings.filterwarnings("ignore", category=DeprecationWarning)

# Load the environment with render mode specified
env = gym.make('CartPole-v1', render_mode="human")

# Initialize the environment to get the initial state
state = env.reset()

# Print the state space and action space
print("State space:", env.observation_space)
print("Action space:", env.action_space)

# Run a few steps in the environment with random actions
for _ in range(10):
```

```
env.render()  # Render the environment for visualization
action = env.action_space.sample()  # Take a random action

# Take a step in the environment
step_result = env.step(action)

# Check the number of values returned and unpack accordingly
if len(step_result) == 4:
next_state, reward, done, info = step_result
terminated = False
else:
next_state, reward, done, truncated, info = step_result
terminated = done or truncated

print(f"Action: {action}, Reward: {reward}, Next State: {next_state}, Done: {done}, Info: {info}")

if terminated:
state = env.reset()  # Reset the environment if the episode is finished

env.close()  # Close the environment when done
```

Output:

```
State space: Box([-4.8000002e+00 -3.4028235e+38 -4.1887903e-01 -3.4028235e+38], [4.8000002e+00 3.4028235e+38 4.1887
Action space: Discrete(2)
Action: 0, Reward: 1.0, Next State: [-0.02765167 -0.24203628  0.03266023  0.32844445], Done: False, Info: {}
Action: 1, Reward: 1.0, Next State: [-0.03249239 -0.04739415  0.03922912  0.04623735], Done: False, Info: {}
Action: 0, Reward: 1.0, Next State: [-0.03344028 -0.24305603  0.04015387  0.35103476], Done: False, Info: {}
Action: 0, Reward: 1.0, Next State: [-0.0383014  -0.43872532  0.04717457  0.6561042 ], Done: False, Info: {}
Action: 0, Reward: 1.0, Next State: [-0.0470759  -0.6344712   0.06029665  0.9632605 ], Done: False, Info: {}
Action: 0, Reward: 1.0, Next State: [-0.05976533 -0.83034915  0.07956186  1.27426   ], Done: False, Info: {}
Action: 1, Reward: 1.0, Next State: [-0.07637231 -0.6363272   0.10504705  1.0075142 ], Done: False, Info: {}
Action: 1, Reward: 1.0, Next State: [-0.08909886 -0.44275263  0.12519734  0.74957997], Done: False, Info: {}
Action: 1, Reward: 1.0, Next State: [-0.09795391 -0.24955945  0.14018895  0.49877036], Done: False, Info: {}
Action: 1, Reward: 1.0, Next State: [-0.1029451  -0.05666344  0.15016435  0.25334558], Done: False, Info: {}
```

Application of Reinforcement Learning

1. **Robotics:** RL is used to automate tasks in structured environments such as manufacturing, where robots learn to optimize movements and improve efficiency.
2. **Game Playing:** Advanced RL algorithms have been used to develop strategies for complex games like chess, Go, and video games, outperforming human players in many instances.
3. **Industrial Control:** RL helps in real-time adjustments and optimization of industrial operations, such as refining processes in the oil and gas industry.
4. **Personalized Training Systems:** RL enables the customization of instructional content based on an individual's learning patterns, improving engagement and effectiveness.

Markov Decision Process

Markov Decision Process (MDP) is a way to describe how a decision-making agent like a robot or game character moves through different situations while trying to achieve a goal. MDPs rely on variables such as the environment, agent's actions and rewards to decide the system's next optimal action. It helps us answer questions like:

- What actions should the agent take?
- What happens after an action?

- Is the result good or bad?

In artificial intelligence Markov Decision Processes (MDPs) are used to model situations where decisions are made one after another and the results of actions are uncertain. They help in designing smart machines or agents that need to work in environments where each action might led to different outcomes.

Key Components of an MDP

An MDP has **five main parts**:

Components of Markov Decision Process

1. **States (S)**:A state is a situation or condition the agent can be in. For example, A position on a grid like being at cell (1,1).
2. **Actions (A)**: An action is something the agent can do. For example, Move UP, DOWN, LEFT or RIGHT. Each state can have one or more possible actions.

3. **Transition Model (T)**: The model tells us what happens when an action is taken in a state. It's like asking: "If I move RIGHT from here, where will I land?" Sometimes the outcome isn't always the same that's uncertainty. For example:

- 80% chance of moving in the intended direction
- 10% chance of slipping to the left
- 10% chance of slipping to the right

This randomness is called a **stochastic transition.**

4. **Reward (R)**: A reward is a number given to the agent after it takes an action. If the reward is positive, it means the result of the action was good. If the reward is negative it means the outcome was bad or there was a penalty help the agent learn what's good or bad. Examples:

- +1 for reaching the goal
- -1 for stepping into fire
- -0.1 for each step to encourage fewer moves

5. **Policy (π)**: A policy is the agent's plan. It tells the agent: "If you are in this state, take this action." The goal is to find the best policy that helps the agent earn the highest total reward over time.

Let's consider a 3x4 grid world. The agent starts at cell `(1,1)` and aims to reach the **Blue Diamond** at `(4,3)` while avoiding **Fire** at `(4,2)` and a **Wall** at `(2,2)`. At each state the agent can take one of the following actions: `UP, DOWN, LEFT or RIGHT`

Problem

1. Movement with Uncertainty (Transition Model)

The agent's moves are stochastic (uncertain):

- 80% chance of going in the intended direction.
- 10% chance of going left of the intended direction.
- 10% chance of going right of the intended direction.

2. Reward System

- +1 for reaching the goal.
- -1 for falling into fire.
- -0.04 for each regular move (to encourage shorter paths).
- 0 for hitting a wall (no movement or penalty).

3. Goal and Policy

- The agent's objective is to maximize total rewards.
- It must find an optimal policy: the best action to take in each state to reach the goal quickly while avoiding danger.

4. Path Example

- One possible optimal path is: `UP → UP → RIGHT → RIGHT → RIGHT`
- But because of randomness the agent must plan carefully to avoid accidentally slipping into fire.

Applications of Markov Decision Processes (MDPs)

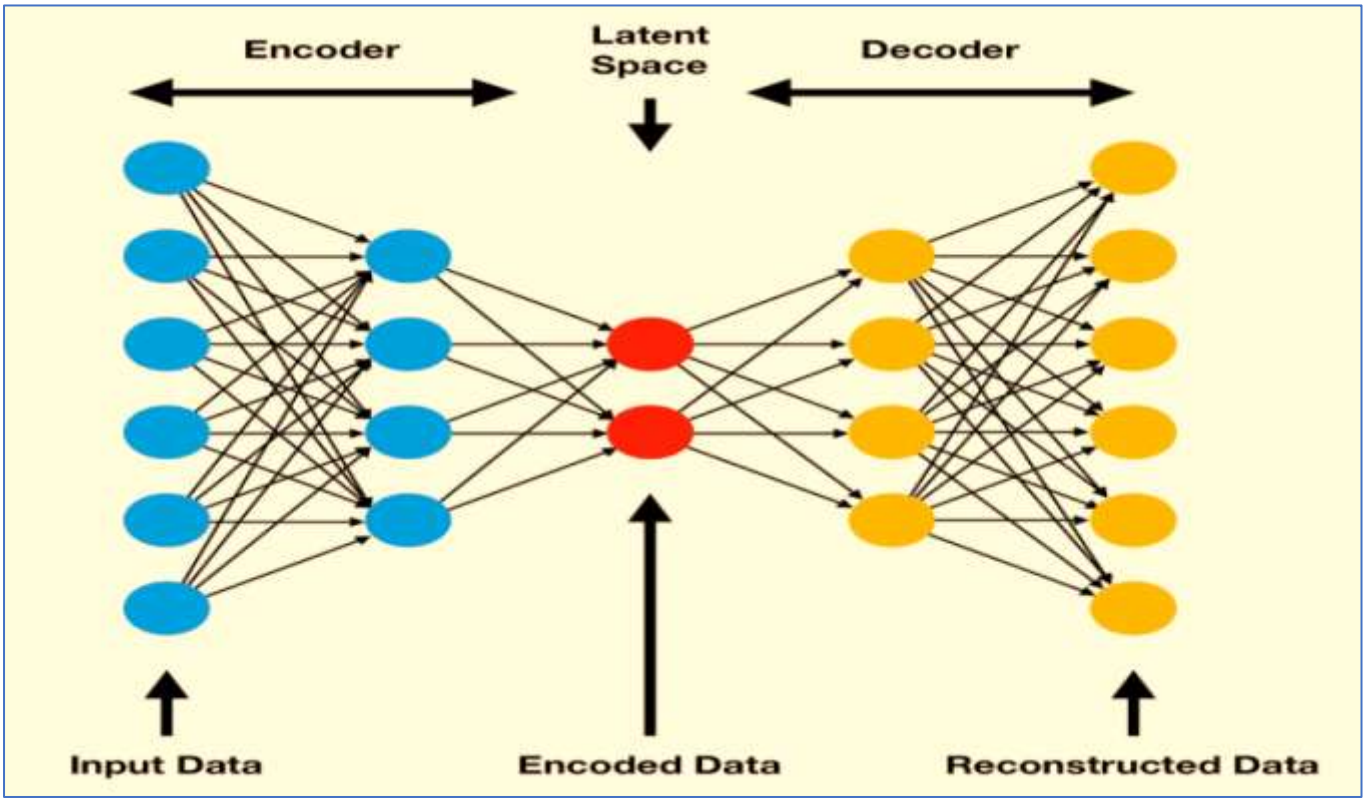
Markov Decision Processes are useful in many real-life situations where decisions must be made step-by-step under uncertainty. Here are some applications:

1. **Robots and Machines**: Robots use MDPs to decide how to move safely and efficiently in places like factories or warehouses and avoid obstacles.
2. **Game Strategy**: In board games or video games MDPs help characters to choose the best moves to win or complete tasks even when outcomes are not certain.
3. **Healthcare**: Doctors can use it to plan treatments for patients, choosing actions that improve health while considering uncertain effects.

4. **Traffic and Navigation**: Self-driving cars or delivery vehicles use it to find safe routes and avoid accidents on unpredictable roads.
5. **Inventory Management**: Stores and warehouses use MDPs to decide when to order more stock so they don't run out or keep too much even when demand changes.

Types of Autoencoders

Autoencoders are a type of neural network designed to learn efficient data representations. They work by compressing input data into a smaller, dense format called the latent space using an encoder and then reconstructing the original input from this compressed form using a decoder. This makes autoencoders useful for tasks such as dimensionality reduction, feature extraction and noise removal. In this article, we'll see various types of autoencoders and their core concepts.



Basic architecture of autoencoders

For more details refer to our article Autoencoders in Machine Learning.

Let's see common types of autoencoders which are designed with unique features to handle specific challenges and tasks in data representation and learning.

1. Vanilla Autoencoder

- Vanilla Autoencoder are the simplest form used for unsupervised learning tasks. They consist of two main parts an encoder that compresses the input data into a smaller, dense representation and a decoder that reconstructs the original input from this compressed form.
- Training minimizes reconstruction error which measures the difference between input and output. This optimization is done via backpropagation which helps in updating the network weights to improve reconstruction accuracy.
- They are foundational models helps in serving as building blocks for more complex variants.

Applications of Vanilla Autoencoders

Some key applications include:

1. **Data Compression:** They learn a compact version of the input data making storage and transmission more efficient.
2. **Feature Learning:** It extract important patterns from data which is useful in image processing, natural language processing and sensor analysis.
3. **Anomaly Detection:** If the reconstructed output is different from the original input, it can show an anomaly or outlier which makes autoencoders useful for fraud detection and system monitoring.

Now lets see the practical implementation.

Here we will be using Numpy, Matplotlib and Tensorflow libraries for its implementation and also we are using inbuilt dataset for this.

- **(x_train, _), (x_test, _) = fashion_mnist.load_data()**: Loads Fashion MNIST dataset into training and testing sets, ignoring labels.
- **encoded = tf.keras.layers.Dense(encoding_dim, activation='relu')(input_img)**: Encodes input into 32-dimensional vector with ReLU activation.
- **decoded = tf.keras.layers.Dense(784, activation='sigmoid')(encoded)**: Decodes the compressed vector back to 784 dimensions with sigmoid activation.
- **autoencoder = tf.keras.Model(input_img, decoded)**: Creates the autoencoder model connecting input to output.

```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras.datasets import fashion_mnist

(x_train, _), (x_test, _) = fashion_mnist.load_data()
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.

x_train_flat = x_train.reshape(len(x_train), 784)
x_test_flat = x_test.reshape(len(x_test), 784)

n = 10
encoding_dim = 32
input_img = tf.keras.Input(shape=(784,))
encoded = tf.keras.layers.Dense(encoding_dim, activation='relu')(input_img)
decoded = tf.keras.layers.Dense(784, activation='sigmoid')(encoded)

autoencoder = tf.keras.Model(input_img, decoded)
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')

autoencoder.fit(x_train_flat, x_train_flat,
epochs=50,
batch_size=256,
shuffle=True,
validation_data=(x_test_flat, x_test_flat))

decoded_imgs = autoencoder.predict(x_test_flat)

plt.figure(figsize=(20, 4))
for i in range(n):
ax = plt.subplot(2, n, i + 1)
plt.imshow(x_test_flat[i].reshape(28, 28), cmap='gray')
```

```
ax.axis('off')

ax = plt.subplot(2, n, i + 1 + n)
plt.imshow(decoded_imgs[i].reshape(28, 28), cmap='gray')
ax.axis('off')
plt.show()
```

2. Sparse Autoencoder

- Sparse Autoencoder add sparsity constraints that encourage only a small subset of neurons in the hidden layer to activate at once helps in creating a more efficient and focused representation.
- Unlike vanilla models, they include regularization methods like L1 penalty and dropout to enforce sparsity.
- KL Divergence is used to maintain the sparsity level by matching the latent distribution to a predefined sparse target.
- This selective activation helps in feature selection and learning meaningful patterns while ignoring irrelevant noise.

Applications of Sparse Autoencoders

1. **Feature Selection**: Highlights the most relevant features by encouraging sparse activation helps in improving interpretability.
2. **Dimensionality Reduction:** Creates efficient, low-dimensional representations by limiting active neurons.
3. **Noise Reduction**: Reduces irrelevant information and noise by activating only key neurons helps in improving model generalization.

Now lets see the practical implementation.

- **encoded = tf.keras.layers.Dense(encoding_dim, activation='relu', activity_regularizer=tf.keras.regularizers.l1(1e-5))(input_img):** Creates the encoded layer with ReLU activation and adds L1 regularization to encourage sparsity.

```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras.datasets import fashion_mnist

(x_train, _), (x_test, _) = fashion_mnist.load_data()
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.

x_train_flat = x_train.reshape(len(x_train), 784)
x_test_flat = x_test.reshape(len(x_test), 784)

n = 10
encoding_dim = 32
input_img = tf.keras.Input(shape=(784,))
encoded = tf.keras.layers.Dense(encoding_dim, activation='relu',
activity_regularizer=tf.keras.regularizers.l1(1e-5))(input_img)
decoded = tf.keras.layers.Dense(784, activation='sigmoid')(encoded)

sparse_autoencoder = tf.keras.Model(input_img, decoded)
sparse_autoencoder.compile(optimizer='adam', loss='binary_crossentropy')

sparse_autoencoder.fit(x_train_flat, x_train_flat,
```

```
epochs=50,
batch_size=256,
shuffle=True,
validation_data=(x_test_flat, x_test_flat))

decoded_imgs = sparse_autoencoder.predict(x_test_flat)

plt.figure(figsize=(20, 4))
for i in range(n):
ax = plt.subplot(2, n, i + 1)
plt.imshow(x_test_flat[i].reshape(28, 28), cmap='gray')
ax.axis('off')

ax = plt.subplot(2, n, i + 1 + n)
plt.imshow(decoded_imgs[i].reshape(28, 28), cmap='gray')
ax.axis('off')
plt.show()
```

Training

3. Denoising Autoencoder

- **Denoising Autoencoders** are designed to handle corrupted or noisy inputs by learning to reconstruct the clean, original data.
- Training involves feeding intentionally corrupted inputs and minimizing the reconstruction error against the clean version.
- This approach forces the model to capture robust features that are invariant to noise.

Applications of Denoising Autoencoders

1. **Image Denoising:** Removes noise from images to increase quality and improve downstream processing.
2. **Signal Cleaning**: Filters noise from audio and sensor signals helps in boosting detection accuracy.
3. **Data Preprocessing**: Cleans corrupted data before input to other models helps in increasing robustness and performance.

Now lets see the practical implementation.

```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras.datasets import fashion_mnist

(x_train, _), (x_test, _) = fashion_mnist.load_data()
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.

x_train_flat = x_train.reshape(len(x_train), 784)
x_test_flat = x_test.reshape(len(x_test), 784)

n = 10
encoding_dim = 32
input_img = tf.keras.Input(shape=(784,))
encoded = tf.keras.layers.Dense(encoding_dim, activation='relu')(input_img)
decoded = tf.keras.layers.Dense(784, activation='sigmoid')(encoded)
```

```
denoising_autoencoder = tf.keras.Model(input_img, decoded)
denoising_autoencoder.compile(optimizer='adam', loss='binary_crossentropy')

noise_factor = 0.5
x_train_noisy = x_train_flat + noise_factor * np.random.normal(loc=0.0, scale=1.0, size=x_train_flat.shape)
x_test_noisy = x_test_flat + noise_factor * np.random.normal(loc=0.0, scale=1.0, size=x_test_flat.shape)
x_train_noisy = np.clip(x_train_noisy, 0., 1.)
x_test_noisy = np.clip(x_test_noisy, 0., 1.)

denoising_autoencoder.fit(x_train_noisy, x_train_flat,
epochs=50,
batch_size=256,
shuffle=True,
validation_data=(x_test_noisy, x_test_flat))

decoded_imgs = denoising_autoencoder.predict(x_test_noisy)

plt.figure(figsize=(20, 6))
for i in range(n):
ax = plt.subplot(3, n, i + 1)
plt.imshow(x_test_flat[i].reshape(28, 28), cmap='gray')
ax.axis('off')

ax = plt.subplot(3, n, i + 1 + n)
plt.imshow(x_test_noisy[i].reshape(28, 28), cmap='gray')
ax.axis('off')

ax = plt.subplot(3, n, i + 1 + 2*n)
plt.imshow(decoded_imgs[i].reshape(28, 28), cmap='gray')
ax.axis('off')
plt.show()
```

4. Undercomplete Autoencoder

- **Undercomplete Autoencoders** intentionally restrict the size of the hidden layer to be smaller than the input layer.
- This bottleneck forces the model to compress the data helps in learning only the most significant features and discarding redundant information.
- The model is trained by minimizing the reconstruction error while ensuring the latent space remains compact.

Applications of Undercomplete Autoencoders

- **Anomaly Detection:** Detects unusual data points by capturing deviations in compressed features.
- **Feature Extraction:** Focuses on key data characteristics to improve classification and analysis.
- **Data Compression:** Encodes input data efficiently to save storage and speed up transmission.

Now lets see the practical implementation.

- **encoded = tf.keras.layers.Dense(encoding_dim, activation='relu',(input_img)**: Builds the encoder layer with ReLU activation.

```
import numpy as np
import matplotlib.pyplot as plt
```

```
import tensorflow as tf
from tensorflow.keras.datasets import fashion_mnist

(x_train, _), (x_test, _) = fashion_mnist.load_data()
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.

x_train_flat = x_train.reshape(len(x_train), 784)
x_test_flat = x_test.reshape(len(x_test), 784)

n = 10
encoding_dim = 16
input_img = tf.keras.Input(shape=(784,))
encoded = tf.keras.layers.Dense(encoding_dim, activation='relu')(input_img)
decoded = tf.keras.layers.Dense(784, activation='sigmoid')(encoded)

undercomplete_autoencoder = tf.keras.Model(input_img, decoded)
undercomplete_autoencoder.compile(optimizer='adam', loss='binary_crossentropy')

undercomplete_autoencoder.fit(x_train_flat, x_train_flat,
epochs=50,
batch_size=256,
shuffle=True,
validation_data=(x_test_flat, x_test_flat))

decoded_imgs = undercomplete_autoencoder.predict(x_test_flat)

plt.figure(figsize=(20, 4))
for i in range(n):
ax = plt.subplot(2, n, i + 1)
plt.imshow(x_test_flat[i].reshape(28, 28), cmap='gray')
ax.axis('off')

ax = plt.subplot(2, n, i + 1 + n)
plt.imshow(decoded_imgs[i].reshape(28, 28), cmap='gray')
ax.axis('off')
plt.show()
```

5. Contractive Autoencoder

- **Contractive Autoencoders** introduce an additional penalty during training to make the learned representations robust to small changes in input data.
- They minimize both reconstruction error and a regularization term that penalizes sensitivity to input perturbations.
- This results in stable, invariant features useful in noisy or fluctuating environments.

Applications of Contractive Autoencoders

1. **Stable Representation**: Learns features that remain consistent despite small input variations.
2. **Transfer Learning**: Provides robust feature vectors for tasks with limited labeled data.
3. **Data Augmentation**: Generates stable variants of input data to increase training diversity.

Now lets see the practical implementation.

```
import numpy as np
import matplotlib.pyplot as plt
```

```
import tensorflow as tf
from tensorflow.keras.datasets import fashion_mnist

(x_train, _), (x_test, _) = fashion_mnist.load_data()
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.

x_train_flat = x_train.reshape(len(x_train), 784)
x_test_flat = x_test.reshape(len(x_test), 784)

n = 10
encoding_dim = 32
input_img = tf.keras.Input(shape=(784,))
encoded = tf.keras.layers.Dense(encoding_dim, activation='relu')(input_img)
decoded = tf.keras.layers.Dense(784, activation='sigmoid')(encoded)

contractive_autoencoder = tf.keras.Model(input_img, decoded)

def contractive_loss(y_true, y_pred):
mse = tf.keras.losses.mean_squared_error(y_true, y_pred)
W = contractive_autoencoder.layers[1].kernel
dh = tf.gradients(contractive_autoencoder.layers[1].output, input_img)[0]
contractive = tf.reduce_sum(tf.square(W)) * tf.reduce_sum(tf.square(dh))
return mse + 1e-4 * contractive

contractive_autoencoder.compile(optimizer='adam', loss='binary_crossentropy')

contractive_autoencoder.fit(x_train_flat, x_train_flat,
epochs=50,
batch_size=256,
shuffle=True,
validation_data=(x_test_flat, x_test_flat))

decoded_imgs = contractive_autoencoder.predict(x_test_flat)

plt.figure(figsize=(20, 4))
for i in range(n):
ax = plt.subplot(2, n, i + 1)
plt.imshow(x_test_flat[i].reshape(28, 28), cmap='gray')
ax.axis('off')

ax = plt.subplot(2, n, i + 1 + n)
plt.imshow(decoded_imgs[i].reshape(28, 28), cmap='gray')
ax.axis('off')
```

6. Convolutional Autoencoder

- Convolutional Autoencoders use convolutional layers to effectively capture spatial and hierarchical features in high-dimensional data such as images.
- These models optimize reconstruction error using loss functions suited for images like mean squared error or binary cross-entropy.
- The architecture helps in handling structured inputs by preserving spatial relationships.

Applications of Convolutional Autoencoders

Convolutional autoencoders find applications in various domains where hierarchical features are important. Some applications include:

1. **Image Reconstruction**: Restores high-quality images from compressed latent codes.
2. **Image Denoising:** Removes noise while preserving spatial detail in images.
3. **Feature Extraction**: Captures hierarchical spatial features for tasks like classification and segmentation.

Now lets see the practical implementation.

```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras.datasets import fashion_mnist

(x_train, _), (x_test, _) = fashion_mnist.load_data()
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.

x_train_flat = x_train.reshape(len(x_train), 784)
x_test_flat = x_test.reshape(len(x_test), 784)

n = 10
input_img = tf.keras.Input(shape=(28, 28, 1))

x = tf.keras.layers.Conv2D(16, (3, 3), activation='relu', padding='same')(input_img)
x = tf.keras.layers.MaxPooling2D((2, 2), padding='same')(x)
x = tf.keras.layers.Conv2D(8, (3, 3), activation='relu', padding='same')(x)
encoded = tf.keras.layers.MaxPooling2D((2, 2), padding='same')(x)

x = tf.keras.layers.Conv2D(8, (3, 3), activation='relu', padding='same')(encoded)
x = tf.keras.layers.UpSampling2D((2, 2))(x)
x = tf.keras.layers.Conv2D(16, (3, 3), activation='relu', padding='same')(x)
x = tf.keras.layers.UpSampling2D((2, 2))(x)
decoded = tf.keras.layers.Conv2D(1, (3, 3), activation='sigmoid', padding='same')(x)

conv_autoencoder = tf.keras.Model(input_img, decoded)
conv_autoencoder.compile(optimizer='adam', loss='binary_crossentropy')

x_train_cnn = x_train.reshape(-1, 28, 28, 1)
x_test_cnn = x_test.reshape(-1, 28, 28, 1)

conv_autoencoder.fit(x_train_cnn, x_train_cnn,
epochs=50,
batch_size=256,
shuffle=True,
validation_data=(x_test_cnn, x_test_cnn))

decoded_imgs = conv_autoencoder.predict(x_test_cnn)

plt.figure(figsize=(20, 4))
for i in range(n):
ax = plt.subplot(2, n, i + 1)
plt.imshow(x_test_cnn[i].reshape(28, 28), cmap='gray')
ax.axis('off')
```

```
ax = plt.subplot(2, n, i + 1 + n)
plt.imshow(decoded_imgs[i].reshape(28, 28), cmap='gray')
ax.axis('off')
plt.show()
```

7. Variational Autoencoder

Variational Autoencoder (VAEs) extend traditional autoencoders by learning probabilistic latent distributions instead of fixed representations. Training optimizes the Evidence Lower Bound (ELBO) which balances:

1. **Reconstruction loss** to ensure accurate data reconstruction.
2. **KL Divergence** to regularize the latent space towards a standard Gaussian helps in preventing overfitting and smooth latent structure.

By balancing these two terms VAEs can generate meaningful outputs while keeping the latent space structured.

Applications of Variational Autoencoders (VAEs)

Here are some common applications:

1. **Image Generation**: Creates new realistic images by sampling from learned latent distributions.
2. **Anomaly Detection:** Identifies anomalies by measuring how well input data is reconstructed.
3. **Dimensionality Reduction**: Produces low-dimensional latent spaces useful for visualization and clustering.

Now lets see the practical implementation.

- **x_train = np.reshape(x_train, (len(x_train), 28, 28, 1))** : Reshapes training images to 28x28 with 1 channel for Conv2D input.
- **input_img = tf.keras.Input(shape=(28, 28, 1))** : Defines input layer for grayscale images with shape 28x28x1.
- **tf.keras.layers.MaxPooling2D((2, 2), padding='same')(x)** : Reduces spatial dimensions by half using max pooling with same padding.
- **decoded = tf.keras.layers.Conv2D(1, (3, 3), activation='sigmoid', padding='same')(x)** : Outputs reconstructed image with 1 channel and sigmoid activation for pixel values between 0 and 1.

```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras import backend as K
from tensorflow.keras.datasets import fashion_mnist

(x_train, _), (x_test, _) = fashion_mnist.load_data()
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
x_train_cnn = x_train.reshape(-1, 28, 28, 1)
x_test_cnn = x_test.reshape(-1, 28, 28, 1)

latent_dim = 2
n = 10

encoder_inputs = tf.keras.Input(shape=(28, 28, 1))
x = tf.keras.layers.Conv2D(32, 3, activation='relu', strides=2,
padding='same')(encoder_inputs)
```

```
x = tf.keras.layers.Conv2D(64, 3, activation='relu', strides=2, padding='same')(x)
x = tf.keras.layers.Flatten()(x)
x = tf.keras.layers.Dense(16, activation='relu')(x)
z_mean = tf.keras.layers.Dense(latent_dim)(x)
z_log_var = tf.keras.layers.Dense(latent_dim)(x)

def sampling(args):
z_mean, z_log_var = args
batch = tf.shape(z_mean)[0]
dim = tf.shape(z_mean)[1]
epsilon = tf.random.normal(shape=(batch, dim))
return z_mean + tf.exp(0.5 * z_log_var) * epsilon

z = tf.keras.layers.Lambda(sampling)([z_mean, z_log_var])

encoder = tf.keras.Model(encoder_inputs, [z_mean, z_log_var, z], name='encoder')

latent_inputs = tf.keras.Input(shape=(latent_dim,))
x = tf.keras.layers.Dense(7 * 7 * 64, activation='relu')(latent_inputs)
x = tf.keras.layers.Reshape((7, 7, 64))(x)
x = tf.keras.layers.Conv2DTranspose(64, 3, strides=2, padding='same', activation='relu')(x)
x = tf.keras.layers.Conv2DTranspose(32, 3, strides=2, padding='same', activation='relu')(x)
decoder_outputs = tf.keras.layers.Conv2DTranspose(1, 3, padding='same', activation='sigmoid')(x)

decoder = tf.keras.Model(latent_inputs, decoder_outputs, name='decoder')

outputs = decoder(z)

class VAELossLayer(tf.keras.layers.Layer):
def __init__(self, **kwargs):
super(VAELossLayer, self).__init__(**kwargs)

def call(self, inputs):
x, x_decoded, z_mean, z_log_var = inputs

reconstruction_loss = tf.keras.losses.binary_crossentropy(
K.flatten(x), K.flatten(x_decoded)
)
reconstruction_loss *= 28 * 28

kl_loss = 1 + z_log_var - K.square(z_mean) - K.exp(z_log_var)
kl_loss = K.sum(kl_loss, axis=-1)
kl_loss *= -0.5

total_loss = K.mean(reconstruction_loss + kl_loss)
self.add_loss(total_loss)
return x_decoded

outputs_with_loss = VAELossLayer()([encoder_inputs, outputs, z_mean, z_log_var])
```

```
vae = tf.keras.Model(encoder_inputs, outputs_with_loss, name='vae_with_loss')

vae.compile(optimizer='adam')

vae.fit(x_train_cnn,  epochs=50,  batch_size=256,  validation_data=(x_test_cnn,
None))

decoded_imgs = vae.predict(x_test_cnn)

plt.figure(figsize=(20, 4))
for i in range(n):
ax = plt.subplot(2, n, i + 1)
plt.imshow(x_test_cnn[i].reshape(28, 28), cmap='gray')
ax.axis('off')

ax = plt.subplot(2, n, i + 1 + n)
plt.imshow(decoded_imgs[i].reshape(28, 28), cmap='gray')
ax.axis('off')
plt.show()
```