AI Odyssey : Deep Dive into Transformers and its Application

IITK Consulting Group

CONTENTS

Introduction

Vision:
The vision of the project was to familiarise the mentees with the various tools of AI and understand the core principles and fundamentals of AI. It was achieved by first delving into the origins of important theories and mechanisms and then later mentees were asked to create their own implementation of various models.

Objectives:

1. Building familiarity with python and it's most basic libraries
2. Introducing the basics of Machine Learning and the mathematical logics behind it
3. Introducing various types of Models and Evaluation metrics.

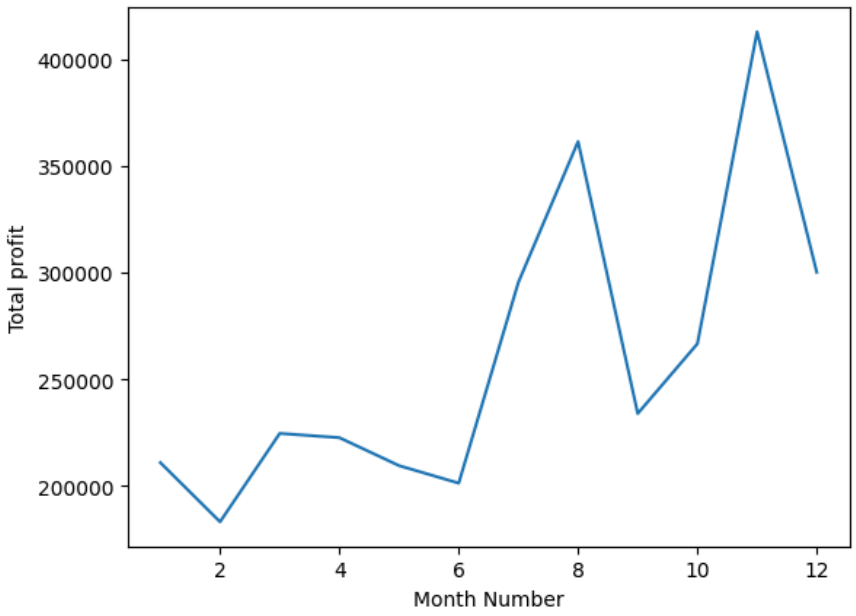
4. In-depth implementation of the models
5. Inculcating the habit of reading research papers and the ability to pick up important information from them in order to aid them in solving any other problems they may come across.
6. Learning the important mechanisms behind Computer Vision and Natural Language Processing

Basics of Python

Python plays a pivotal role in the fields of machine learning due to the simplicity of understanding and learning the language, combined with the extensive set of libraries it offers. Due to the extensivity, implementing AI models has become much more simpler in Python and hence knowledge of the language is absolutely necessary for comprehending how a computer can “learn” and recognize patterns.

Multiple resources were shared to cover the following libraries along with Assignment 0 which required implementation of various functions of these libraries. Furthermore, assignment 0 also introduced students to Google Colab and some basics of GitHub. Assignment 0 covered:

1) Basic data types of python like: Lists, Dictionaries and then how to manipulate them.
2) Defining functions and classes.
3) Numpy and some of its exercises as in adding and removing elements of Numpy Arrays or sorting them.
4) Pandas: Reading .csv or .txt files using Pandas and importing it to a dataframe/table, cleaning the datasets by removing or filling in the missing values with mean values.
5) Matplotlib: Visualising a few aspects of datasets into different type of plots for example line graphs and histograms.



*

Later on in the project, libraries like PyTorch and TensorFlow, which have specific implementations for machine learning and moreover, the ease of handling large datasets is very easy in python and hence makes it very important to have a working knowledge of python to implement models.

Machine Learning and Its Types

Machine Learning is a branch of Artificial Intelligence which focuses on the developments of algorithms and statistical models which allow computers to infer predictions and evolve behaviours from empirical data. What separates ML from regular coding is that ML algorithms are built in a way to learn patterns over time and get better over time.

Mentees were first introduced to the concept theoretically and then asked to implement it over time.
First supervised and unsupervised learning were covered

Supervised Learning

The model is trained on labelled datasets, i.e each training instance is paired with an output label. These types of models are widely used in the sectors of finance, marketing etc.
Students were tasked to import the NBA_logreg dataset, first clean and analyse it and then build the following models on it. First all the NaN values were dropped and then various features of the Dataframe like shape, columns, etc were assessed.

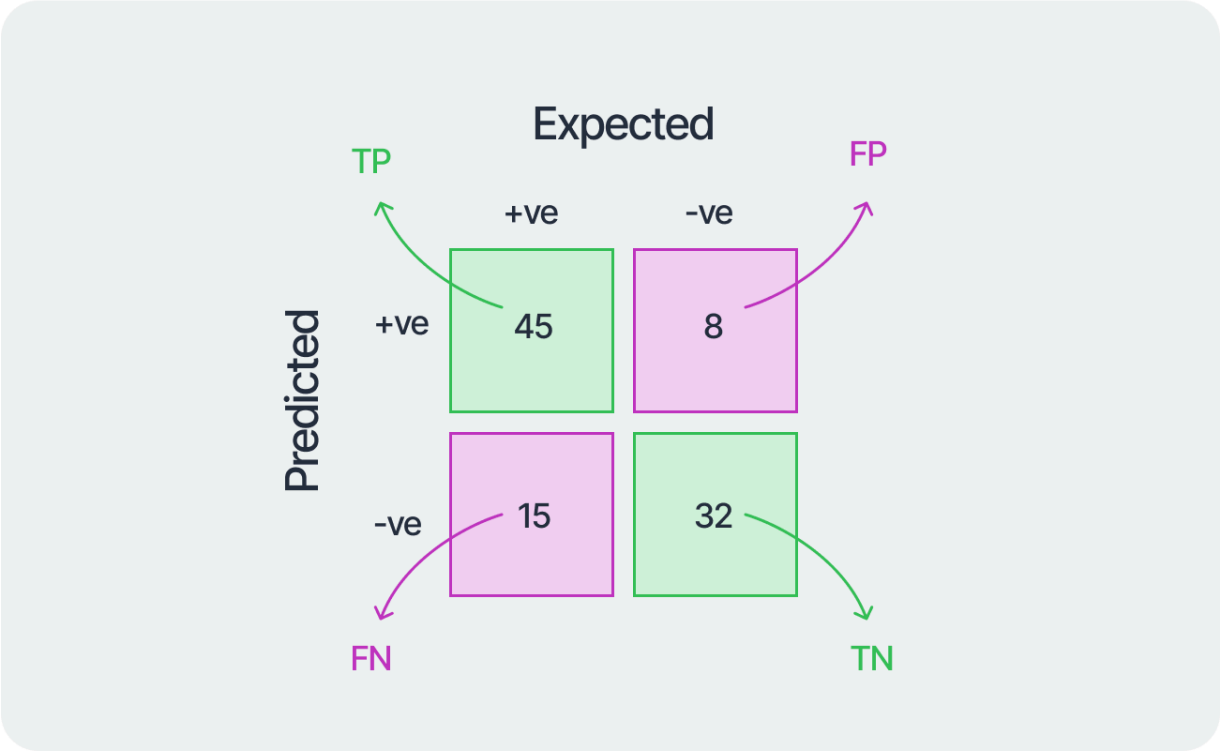
Next a Logistic Regression model was implemented with the help of the sklearn library. An example of the implementation: (Link to source code for all the following code)

```
from sklearn.linear_model import LogisticRegression
logreg = LogisticRegression()
logreg.fit(training_features, training_labels)

#training labels and features were defined in the code blocks before
this.
```

The model's metrics were evaluated by using Accuracy, F1 Score, Recall and Precision.
Accuracy = Total Number of Predictions / Number of Correct Predictions
F1 Score = TP / (TP + 0.5*(FP + FN)) = (2 * Precision * Recall) / (Precision + Recall)
Recall = TP / (TP + FN)
Precision = TP/ (TP + FP)

TP = True Positive
FP = False Positive
TN = True Negative
FN = False Negative



A confusion matrix from which all the above metrics can be derived.

The results of the above model were

Accuracy of the model is: 0.6691729323308271
F1 score of the model is: 0.7396449704142012
Precision of the model is: 0.7575757575757576
Recall of the model is: 0.7225433526011561

Similarly, a Support Vector Machine, a Multi-Class classification model and a KNN Model was implemented using the following code blocks.

The SVM and it's metrics:

```
from sklearn import svm
support_vm = svm.SVC(kernel='linear')
support_vm.fit(training_features, training_labels)

Accuracy of the model is: 0.6616541353383458
F1 score of the model is: 0.7383720930232557
Precision of the model is: 0.7696969696969697
Recall of the model is: 0.7094972067039106
```

For Multiclass classification the students were also tasked with understanding how to one hot encode the labels. The KNN model was implemented and it's metrics are as follows:

```
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier()
knn.fit(training_features, training_labels)

Accuracy of the model is: 0.972972972972973


The Confusion Matrix is:
[[19  0  0  0  0  0]
 [ 0 29  0  0  0  0]
 [ 0  0 22  0  0  1]
 [ 0  0  0 47  0  0]
 [ 0  2  0  0 30  0]
 [ 0  0  1  0  1 33]]
The classification report is:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        19
           1       0.94      1.00      0.97        29
           2       0.96      0.96      0.96        23
           3       1.00      1.00      1.00        47
           4       0.97      0.94      0.95        32
           5       0.97      0.94      0.96        35

    accuracy                           0.97       185
   macro avg       0.97      0.97      0.97       185
weighted avg       0.97      0.97      0.97       185
```

Similarly a Naive Bayes model was also implemented, after which we moved on to the implementation of Regression Models. For the regression models a csv dataset of IPL batters was used and the number of 4's was chosen as the target feature.
A simple Linear Regression model from sklearn.linear_model was implemented and in addition they were also taught about the Train-Test split and its importance.

```
from sklearn.linear_model import LinearRegression
linreg = LinearRegression()
linreg.fit(training_features, training_labels)

Mean Squared Error: 14.708082023642163
Root Mean Squared Error: 3.8351117354833564
Mean Absolute Error: 2.7164808918177603
```

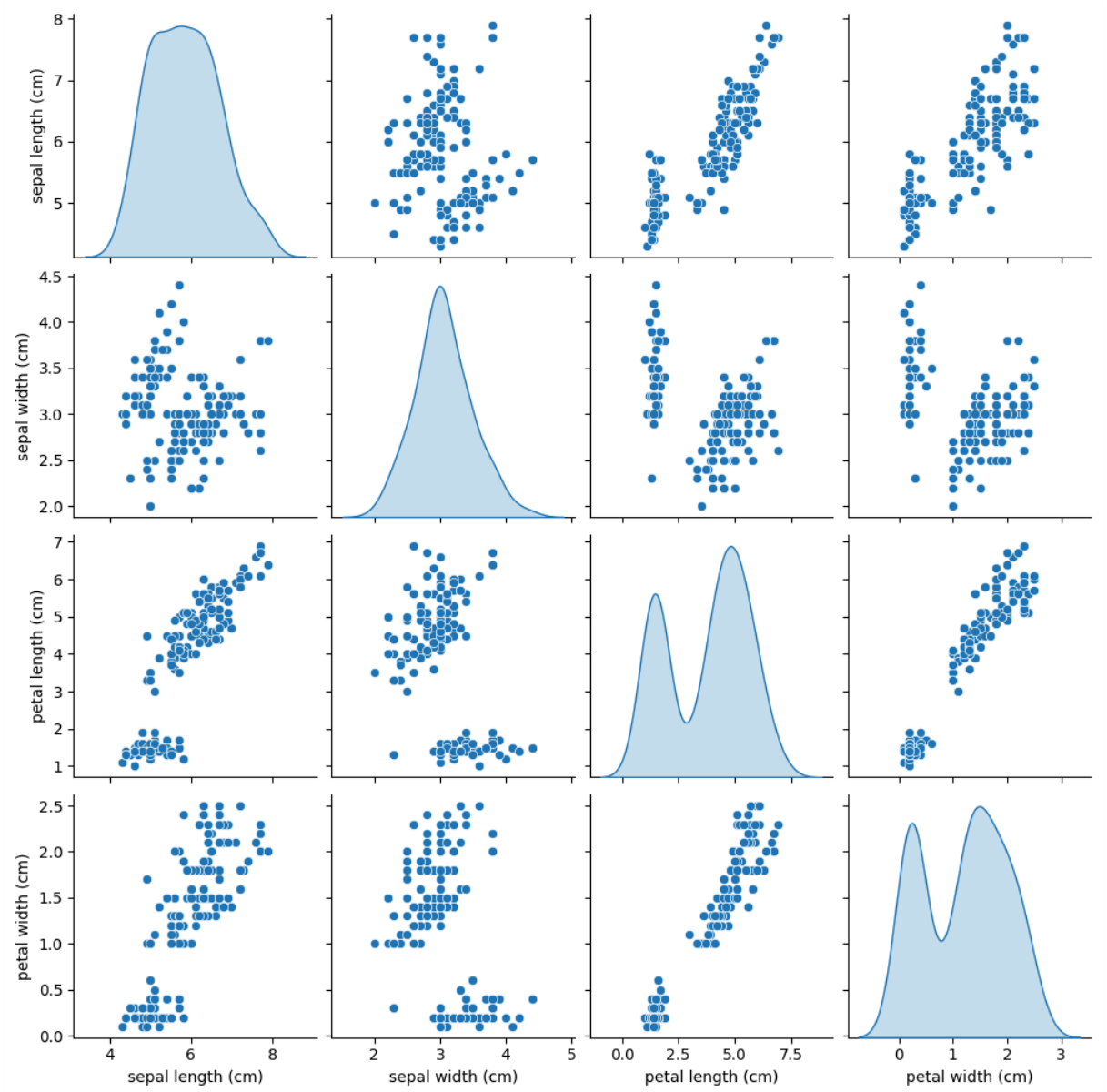
With this students were also pushed to learn what Mean Square Error, Root Mean Square error and Mean Absolute Error's are.

Unsupervised Learning

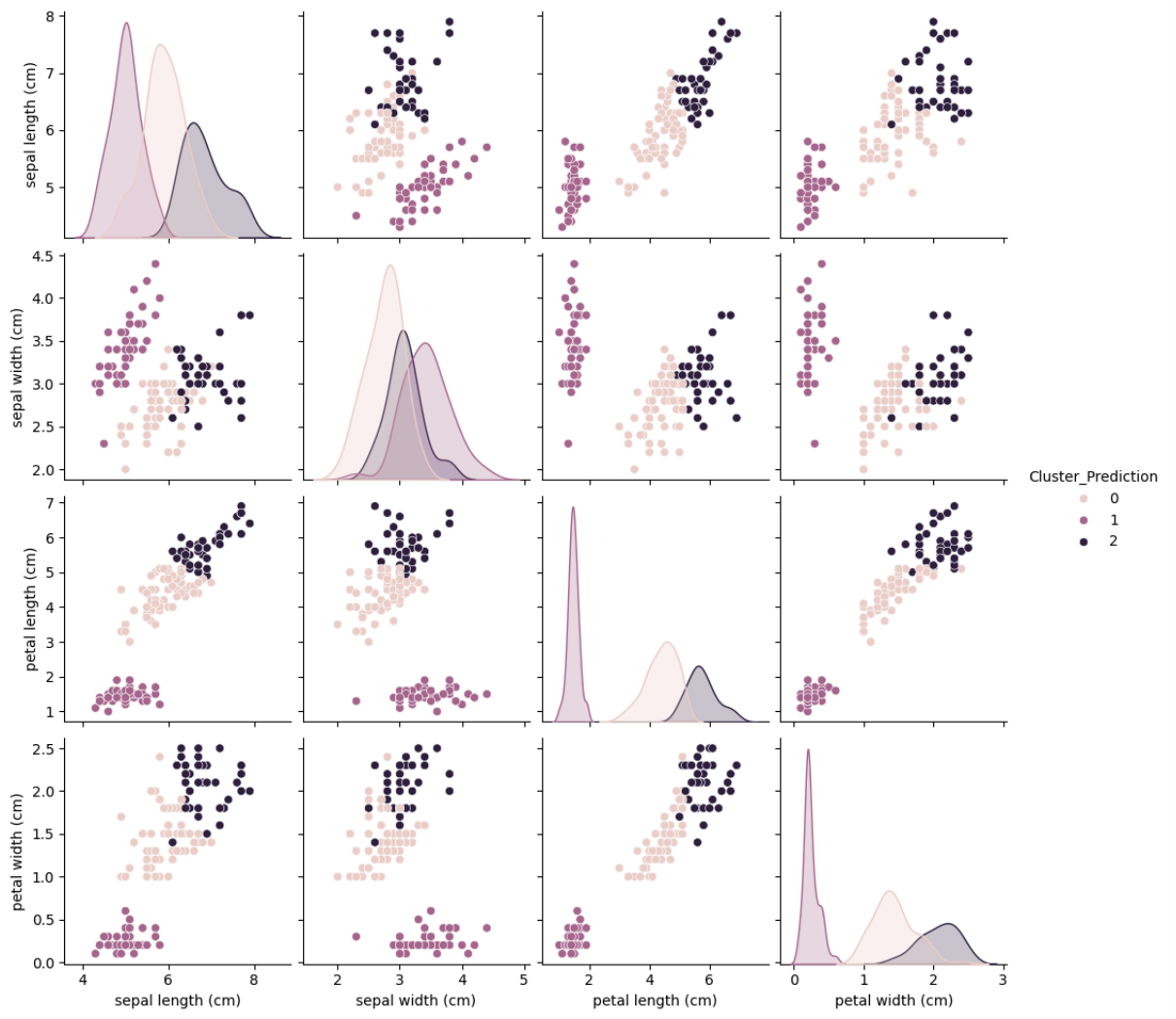
Unsupervised Learning is a type of machine learning where the model is trained on unlabeled data. The goal is to identify patterns, structures, and relationships within the data without any prior knowledge of the output labels.
The unsupervised learning model used the iris dataset to implement the model with the help of KMeans function from sklearn.cluster

```
kmeans = KMeans(n_clusters=3, random_state=42)
kmeans.fit(df)
```

Here are the datasets visualised before the model was applied:



And here is the image of the same dataset visualisations with the models clustering applied:



Neural Networks

Neural Networks are one of the most popular computational models designed to mimic the complex functioning of the human brain. They consist of interconnected nodes/neurons which process and learn from the data provided. They enable machines to complete tasks like decision making or pattern recognition. The entire topic ranging from activation functions, forward propagation, backward propagation, loss functions, the mathematics and theory behind it was covered through classes, a programming assignment and a theoretical assignment.

The Working

Understanding how a neural network learns is nothing but mathematics.

Forward Propagations:

1) *Input Layer:* Each feature throughout the network is represented by a node. The input layer has as many nodes as the number of features based on which we wish to make our decision. Hence this is how the model receives its data.

2) *Weights and Connections:* Any two neurons between two layers are connected and they have a weight associated with them indicating how strongly they are related or if they aren't.

3) *Hidden Layers:* The hidden layers process the inputs they get by multiplying the inputs with the associated weights and then adding them up. Then they are passed through an activation function (like RELU or softmax) to introduce non-linearity to the system.

4) *Output Layer:* The final output/conclusion is reached by passing the input through multiple input layers and producing a final output

Backward Propagation: This mechanism is an extremely efficient algorithm used to train feed-forward Neural Networks. It is an iterative algorithm which looks to minimise the loss function.

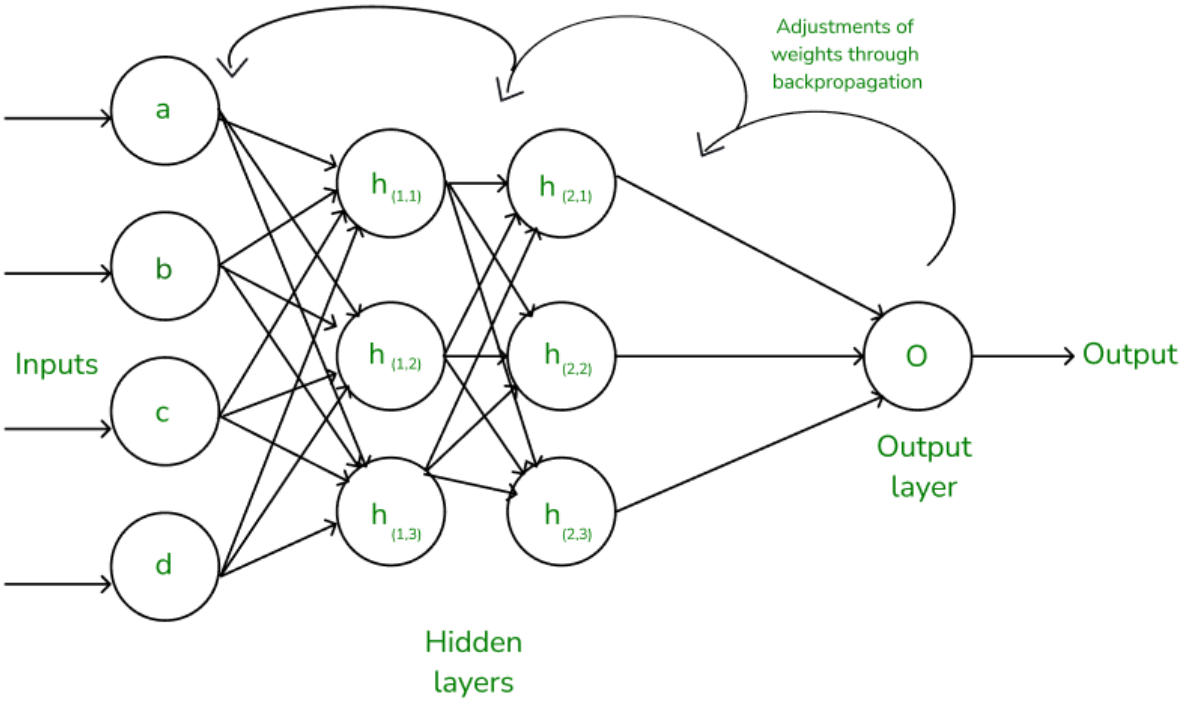
1) *Loss Calculation:* The network's output is evaluated using a loss function to measure the extent of how well the model is performing, i.e the lower the loss the better the performance. An example of the loss function for regression models is the Mean Squared Error:

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

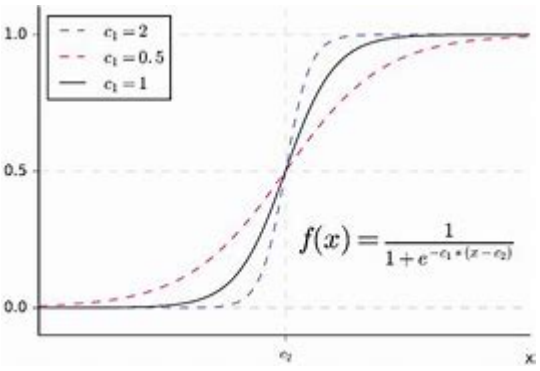
   Where
   N = number of instances

Y_i = Predicted Value
y_i(dash) = Actual Value

2) *Gradient Descent:* It is an optimization algorithm that is essentially used to minimise the loss function. To actually reduce the inaccuracy, the weights are changed on the basis of the derivative of the loss with respect to that particular weight. Accordingly the weights are adjusted by repeating this process, backward across the network.
3) *Training:* Training is done with different batches of the dataset to help ensure that it doesn't get overfitted



Activation Functions: It's purpose is to introduce non linearity to the model by functions like Rectified Linear Unit or Sigmoid. The decision to "fire" a neuron is based on the weighted input.
The sigmoid function:



The assignment covered a feed forward neural network which required the mentees to implement the forward propagation, backward propagation and activation functions by themselves and then implement the model on the MNIST dataset to recognize handwritten digits
This was the majority of code that was left for the mentees to complete themselves in order for them to get an idea of how to implement functions themselves (Link to source code)

```
class ANN:
    def __init__(self, input_size, output_size, learning_rate,
num_layers, num_of_nodes_layers):
        self.input_size = input_size
        self.output_size = output_size
        self.learning_rate = learning_rate
        self.num_layers = num_layers
        self.num_of_nodes_layers = num_of_nodes_layers
        self.weights_biases = {}
        self.activations = {}

    def initial_params(self):
        np.random.seed(20)
        self.weights_biases['W1'] =
np.random.rand(self.num_of_nodes_layers, self.input_size) - 0.5
        self.weights_biases['b1'] =
np.random.rand(self.num_of_nodes_layers, 1) - 0.5

        for i in range(2, self.num_layers + 1):
            self.weights_biases[f'W{i}'] =
np.random.rand(self.num_of_nodes_layers, self.num_of_nodes_layers) -
0.5
            self.weights_biases[f'b{i}'] =
np.random.rand(self.num_of_nodes_layers, 1) - 0.5

        self.weights_biases[f'W{self.num_layers + 1}'] =
np.random.rand(self.output_size, self.num_of_nodes_layers) - 0.5
        self.weights_biases[f'b{self.num_layers + 1}'] =
np.random.rand(self.output_size, 1) - 0.5

    def RELU(self, Z):
        return np.maximum(0, Z)

    def softmax(self, Z):
        Z_shift = Z - np.max(Z, axis=0, keepdims=True)
        exp_values = np.exp(Z_shift)
        exp_values_sum = np.sum(exp_values, axis=0, keepdims=True)
        return exp_values / exp_values_sum

    def forward_propagation(self, X):
        network = {}
        A = X
```

```
        for i in range(1, self.num_layers + 2):
            Z = np.dot(self.weights_biases[f'W{i}'], A) +
self.weights_biases[f'b{i}']
            if i == self.num_layers + 1:
A = self.softmax(Z)
            else:
                A = self.RELU(Z)

            network[f'Z{i}'] = Z
            network[f'A{i}'] = A

        return A, network

    def one_hot_encode(self, y):
        one_hot = np.zeros((y.max() + 1, y.size))
        one_hot[y, np.arange(y.size)] = 1
        return one_hot

    def total_loss(self, y_pred, Y):
        epsilon = 1e-8
        m = Y.shape[1]
        return -np.sum(Y * np.log(y_pred + epsilon)) / m

    def backward_prop(self, y_pred, Y, network):
        gradients = {}
        m = Y.shape[1]
        A_prev = X

        for i in reversed(range(1, self.num_layers + 2)):
          if i == self.num_layers + 1:
              dZ = network[f'A{i}'] - Y
          else:
              dA = np.dot(self.weights_biases[f'W{i+1}'].T, dZ)
              dZ = dA * (network[f'Z{i}'] > 0)

        dW = np.dot(dZ, network[f'A{i-1}'].T) / m if i > 1 else
np.dot(dZ, X.T) / m
        db = np.sum(dZ, axis=1, keepdims=True) / m
        gradients[f'dW{i}'] = dW
        gradients[f'db{i}'] = db

        return gradients

    def update_params(self, gradients):
        for i in range(1, self.num_layers + 2):
            self.weights_biases[f"W{i}"] -= self.learning_rate *
gradients[f"dW{i}"]
            self.weights_biases[f"b{i}"] -= self.learning_rate *
gradients[f"db{i}"]

    def train(self, X, y, num_iterations):
        self.initial_params()
Y = self.one_hot_encode(y)
```

```
        for i in range(num_iterations):
            y_pred, network = self.forward_propagation(X)
            loss = self.total_loss(y_pred, Y)
            if np.isnan(loss):
                print(f"NaN loss detected at iteration {i}")
                break
            gradients = self.backward_prop(y_pred, Y, network)
            self.update_params(gradients)
            if i % 100 == 0:
                print(f"Iteration {i}: Loss = {loss}")

    def predict(self, X):
        A, _ = self.forward_propagation(X)
        predictions = np.argmax(A, axis=0)
        return predictions
```

```
model = ANN(input_size=784, output_size=10, learning_rate=0.2,
num_layers=3, num_of_nodes_layers=64)
model.train(x_train, y_train, num_iterations=4000)
```

Moreover in the theoretical assignment mentees were encouraged to understand the concepts and mathematics of concepts like bagging, boosting, Decision Trees, KNNs and the factors that can influence them.

Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a specialised type of neural network designed to process and analyse grid-like data structures, most commonly used for image and video recognition tasks. CNNs are particularly effective at capturing spatial hierarchies and patterns in data through the use of convolutional layers.

Key Components:

- **Filters (Kernels)**: Small, learnable matrices that slide over the input data to detect specific features such as edges, textures, and patterns.

- **Convolution Operation**: The filter performs an element-wise multiplication with the input matrix and sums the results, creating a feature map (activation map).

  $(I*K)(i,j)=\sum_m\sum_n I(i+m,j+n)\cdot K(m,n)$

  I is the input matrix.
  K is the kernel (filter) matrix.
  (i,j) is the position in the output matrix.
  m and n are the kernel dimensions.

- **Stride**: The number of pixels by which the filter moves across the input matrix. A larger stride reduces the spatial dimensions of the output.

- **Padding**: Adding extra pixels around the input matrix to control the spatial size of the output. Common types are 'valid' (no padding) and 'same' (padding to keep output size equal to input size).

- **Pooling Layers:** Reduces the dimensions of the feature maps which reduces computational load and prevents the model from overfitting.

  Max Pooling: Max Pooling(x,y) = $\max_{i,j}(I(x+i,y+j))$

  Average Pooling: Average Pooling(x,y)= $(1 / n) \sum_{i,j} I(x+i,y+j)$

- **Fully Connected Layers:** The output from the convolutional layers, of all the feature maps are passed onto a typical ANN architecture which then produces the final output

  $y=\sigma(W.X+b)$
  σ = Activation Function

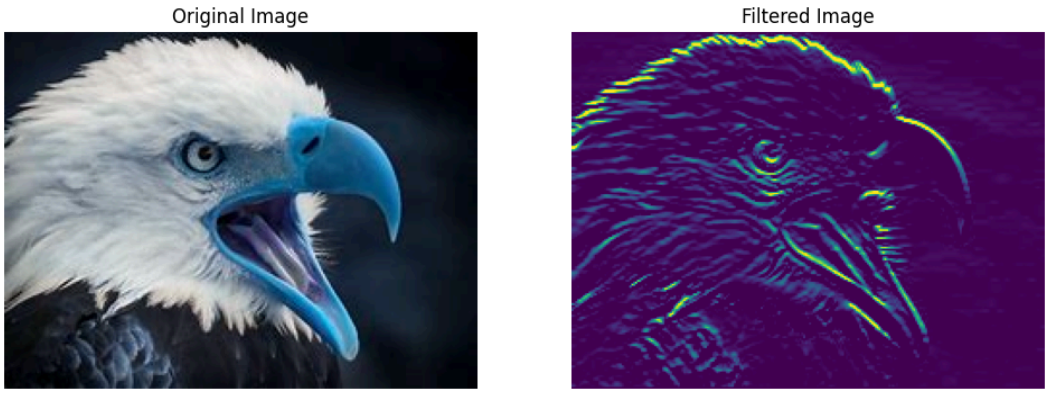
W_x = Weight Matrix
b = Bias Vector
X = Input Vector

- **Flattening:** The output 2D layers are converted to 1D output vectors

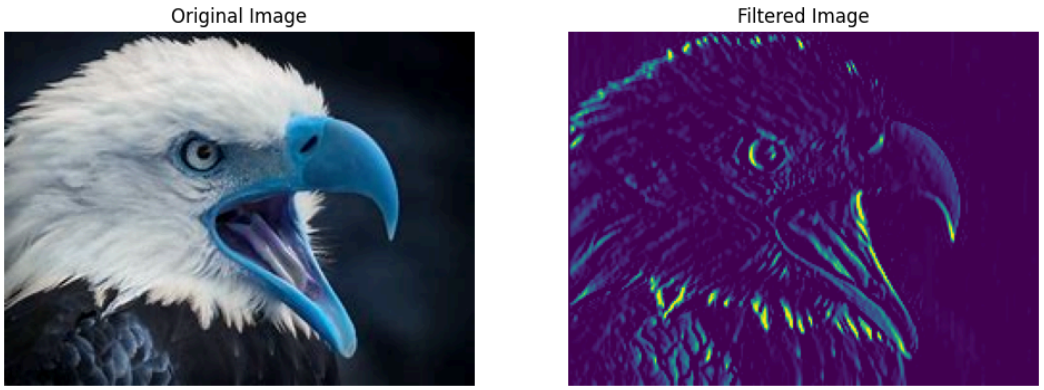
It is extremely useful in image classification, segmentation, object detection, facial recognition.

The assignment covered the implementation of a basic CNN Architecture to accomplish the same task as the ANN, i.e handwritten digit recognition using the MNIST database. Before we accomplished this, mentees were also asked to apply various image filters by themself so that they can understand the significance of all the filters. For example:

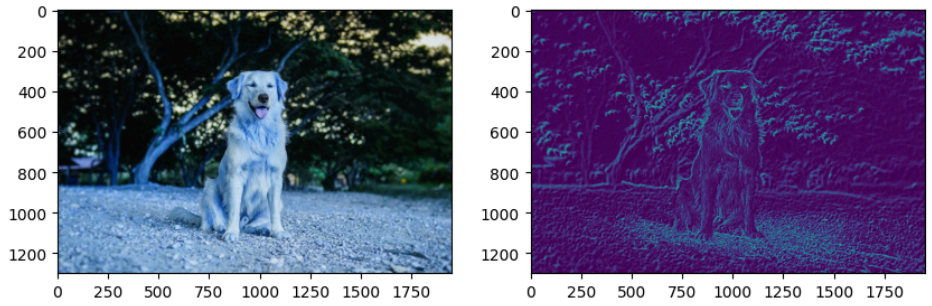
For Horizontal Edge Detection (Prewitt Horizontal):



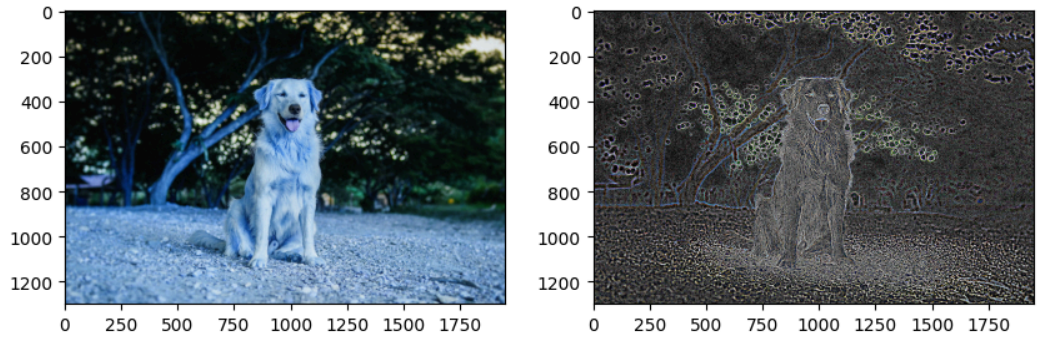
For vertical edge detection (Prewitt Vertical):



Edge Detected image using sobel filter:



Using a Laplacian Filter:



Now onto the actual model architecture and it's evaluation:

```
import tensorflow as tf
from tensorflow.keras import layers,models
from tensorflow import keras

convolutional_neural_network = models.Sequential([
    layers.Conv2D(filters=32, kernel_size=(3, 3), activation='relu',
input_shape=(28,28,1)),
    layers.MaxPooling2D((2, 2)),

    layers.Conv2D(filters=64, kernel_size=(3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),

    layers.Flatten(),
    layers.Dense(64, activation='relu'),
    layers.Dense(10, activation='softmax')
])
convolutional_neural_network.compile(optimizer='adam',
loss='sparse_categorical_crossentropy', metrics=['accuracy'])
convolutional_neural_network.fit(X_train, y_train, epochs=10)

And the final epoch had the following metrics:
Epoch 10/10
1500/1500 [==============================] - 41s 27ms/step - loss: 0.0092 -
accuracy: 0.9971
```

As we can see the final accuracy of the model is extremely high and the loss function is very low as well which just shows the mentees that CNNs are much more efficient at image classification than ANNs.

The classification report for the above model is as follows:

```
Classification Report:
              precision    recall  f1-score   support

           0       0.99      0.99      0.99       980
           1       1.00      0.99      1.00      1135
           2       0.99      0.99      0.99      1032
           3       0.99      1.00      0.99      1010
           4       1.00      0.98      0.99       982
           5       0.97      0.99      0.98       892
           6       0.99      0.98      0.99       958
           7       0.99      0.99      0.99      1028
           8       0.98      0.99      0.99       974
           9       0.99      0.99      0.99      1009

    accuracy                           0.99     10000
   macro avg       0.99      0.99      0.99     10000
weighted avg       0.99      0.99      0.99     10000

Confusion Matrix:
[[ 975    0    1    0    0    0    1    1    2    0]
 [   0 1127    2    2    0    0    2    1    1    0]
 [   0    0 1023    2    1    0    0    5    1    0]
 [   0    0    0 1005    0    3    0    1    1    0]
 [   0    0    1    0  964    0    4    0    4    9]
 [   0    0    0    6    0  884    1    0    1    0]
 [   4    1    0    0    1   12  938    0    2    0]
 [   0    1    5    0    0    0    0 1018    2    2]
 [   1    0    2    2    0    0    0    1  966    2]
 [   0    0    1    1    1    8    0    1    3  994]]
```

Following this they were asked to visualise what the images look like after the filter in each layer is applied and what the output of said layer.

RNN(Recurrent Neural networks)

Recurrent Neural Networks (RNNs) are a class of neural networks designed to recognize patterns in sequences of data. Unlike traditional feedforward neural networks, RNNs have connections that form directed cycles. This cyclical structure allows RNNs to maintain a form of memory by using previous outputs as part of the input for subsequent steps. This characteristic makes RNNs particularly suited for tasks where the order of inputs matters, such as language modeling or speech recognition.

- **How RNNs Work?**

At a high level, RNNs process sequences of data by passing the hidden state from one time step to the next. For each element in the sequence, the RNN computes a new hidden state based on the current input and the previous hidden state. Mathematically, this can be represented as: ht=f(Whht−1+Wxxt+b)h_t = f(W_h h_{t-1} + W_x x_t + b)ht=f(Whht−1+Wxxt+b) where hth_tht is the hidden state at time ttt, xtx_txt is the input at time ttt, WhW_hWh and WxW_xWx are weight matrices, bbb is a bias term, and fff is an activation function.

- **Key components**

1. Input Layer

- Purpose: The input layer receives the sequence data and feeds it into the network. For a sequence, each element of the data is processed one at a time.
- Details: The input at each time step ttt is represented as xtx_txt. This input is often a vector or tensor, depending on the complexity of the data.

2. Recurrent Layer

- Purpose: The core of the RNN where the sequence data is processed through recurrent connections.
- Details: At each time step, the RNN updates its hidden state based on the current input and the previous hidden state. The hidden state hth_tht at time ttt is updated using the formula: ht=f(Whht−1+Wxxt+b)h_t = f(W_h h_{t-1} + W_x x_t + b)ht=f(Whht−1+Wxxt+b) Here, WhW_hWh is the weight matrix for the hidden state, WxW_xWx is the weight matrix for the input, and bbb is a bias term. The activation function fff (often tanh\tanhtanh or ReLU) determines the output of the hidden layer.

3. Hidden State

- Purpose: Represents the memory of the network, carrying information from previous time steps.
- Details: The hidden state hth_tht is updated at each time step and used to influence future predictions. It captures the context from past inputs, which is crucial for tasks involving sequences.

4. Output Layer

- Purpose: Produces the final prediction or output based on the hidden state.
- Details: The output at time ttt, denoted as yty_tyt, is computed using: yt=Wyht+byy_t = W_y h_t + b_yyt=Wyht+by where WyW_yWy is the weight matrix for the output, and byb_yby is the output bias term. The output layer might include an activation function depending on the task (e.g., softmax for classification).

5. Activation Function

- Purpose: Introduces non-linearity into the network, allowing it to learn more complex patterns.
- Details: Common activation functions in RNNs include:
    - tanh\tanhtanh: Maps the input to a range between -1 and 1, which helps in normalizing the hidden state.
    - ReLU: Provides a piecewise linear function, which is useful for mitigating vanishing gradient issues.

6. Weight Matrices

- Purpose: Transform the input and hidden states to generate the output.
- Details: There are typically three weight matrices in an RNN:
    - WxW_xWx: Weight matrix for the input.
    - WhW_hWh: Weight matrix for the hidden state.
    - WyW_yWy: Weight matrix for the output.

7. Bias Terms

- Purpose: Adjust the output of neurons to fit the data better.
- Details: Biases bbb, byb_yby, and bhb_hbh (if applicable) are added to the weighted inputs and hidden states to allow for better model fitting.

8. **Backpropagation Through Time (BPTT)**

- Purpose: Training algorithm for RNNs, extending the backpropagation algorithm to handle sequences.
- Details: BPTT involves unfolding the RNN through time, computing gradients for each time step, and then updating the weights accordingly. It helps in adjusting the network parameters to minimize the loss function.

```
from tensorflow.keras.layers import Embedding, Conv1D, Bidirectional, LSTM, Dense, Input, Dropout
from tensorflow.keras.layers import SpatialDropout1D
from tensorflow.keras.callbacks import ModelCheckpoint
from tensorflow.keras.models import Model

sequence_input = Input(shape=(max_length,), dtype='int32')
embedding_sequences = embedding_layer(sequence_input)
x = SpatialDropout1D(0.2)(embedding_sequences)
x = Conv1D(128, 5, activation='relu')(x)
x = Bidirectional(LSTM(64, return_sequences=True))(x)
x = Bidirectional(LSTM(32))(x)
x = Dropout(0.5)(x)
x = Dense(64, activation='relu')(x)
x = Dropout(0.5)(x)
preds = Dense(1, activation='sigmoid')(x)
model = Model(sequence_input, preds)
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
checkpoint = ModelCheckpoint('model.h5', monitor='val_loss', save_best_only=True, verbose=1)
```

Here is the [link to assignment]

LSTM(Long-Short Term Memory Networks)

Long Short-Term Memory Networks (LSTMs) are a specialized type of RNN designed to address the limitations of standard RNNs. Introduced by Sepp Hochreiter and Jürgen Schmidhuber in 1997, LSTMs incorporate mechanisms to better capture long-term dependencies and mitigate the vanishing gradient problem.

1. Cell State (CtC_tCt)

- **Purpose:** The cell state acts as a memory that carries information across time steps.
- **Details:** Information flows along the cell state with minimal changes, allowing the network to retain long-term dependencies.

2. Hidden State (hth_tht)

- **Purpose:** The hidden state represents the output of the LSTM cell at each time step.
- **Details:** It is used for both current step processing and as part of the inputs for the next time step.

3. Forget Gate (ftf_tft)

- **Purpose:** Determines which information from the cell state should be discarded.
- **Details:** The forget gate outputs a number between 0 and 1 for each number in the cell state Ct−1C_{t-1}Ct−1, where 0 represents “completely forget” and 1 represents “completely keep”. It is computed as:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

- where σ\sigmaσ is the sigmoid function, WfW_fWf is the weight matrix, ht−1h_{t-1}ht−1 is the previous hidden state, xtx_txt is the current input, and bfb_fbf is the bias term.

4. Input Gate (iti_tit)

- **Purpose:** Decides which new information to add to the cell state.
- **Details:** The input gate controls how much of the new information C~t\tilde{C}_tC~t should be added to the cell state CtC_tCt. It is computed as:

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

Here, C~t\tilde{C}_tC~t is the candidate new cell state created by the current input xtx_txt and the previous hidden state ht−1h_{t-1}ht−1, and tanh\tanhtanh is the hyperbolic tangent function.

5. Cell State Update

- **Purpose:** Updates the cell state based on the forget and input gates.
- **Details:** The cell state CtC_tCt is updated using the outputs from the forget gate and the input gate:

$$C_t = f_t \cdot C_{t-1} + i_t \cdot \tilde{C}_t$$

6. Output Gate (oto_tot)

- **Purpose:** Determines the output of the LSTM cell.
- **Details:** The output gate decides which parts of the cell state should be output as the hidden state. It is computed as:

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$

The hidden state hth_tht is then updated based on the cell state CtC_tCt:

$$h_t = o_t \cdot \tanh(C_t)$$

Here is the link of code and the output of its working in one of our [assignments]

This was trained on a .txt file that contained a list of indian names. The assignment required the students. The implementation resulted in the production of names as follows:

```
please enter the first char of name:

m

please enter max length of the name required:

6
```

```
please enter how many names you require:

100

starting char 'm':

mangu

mango

manhu

masom

misar

mamin

moham

minal

muril

Maniu
```

The implementation of the dataset, model, and decoding of the generated output.

The Dataset implementation:

```
class NameDataset(Dataset):

    def __init__(self, filepath):

        with open(filepath, 'r') as file:

          self.names = file.read().splitlines()

        self.names = [self.clean_name(name) for name in self.names]

        self.chars = sorted(list(set(''.join(self.names))))

        self.char_to_index = {char: idx for idx, char in enumerate(self.chars)}

        self.index_to_char = {idx: char for idx, char in enumerate(self.chars)}

        self.vocab_size = len(self.chars)

    def clean_name(self, name):



        cleaned_name = ''.join(char for char in name if char not in string.punctuation and
not char.isdigit())

        return cleaned_name



    def __len__(self):
```

```
        return len(self.names)

    def __getitem__(self, idx):
        name = self.names[idx]
        name_indices = [self.char_to_index[char] for char in name]
        return torch.tensor(name_indices, dtype=torch.long)

    def one_hot_encode(self, index):
        one_hot = torch.zeros(self.vocab_size)
        one_hot[index] = 1
        return one_hot

    def decode(self, indices):
        return ''.join([self.index_to_char[idx] for idx in indices])

dataset = NameDataset('names (1).txt')
dataloader = DataLoader(dataset, batch_size=1, shuffle=True)
```

The model implementation:

```
class RNNModel(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(RNNModel, self).__init__()
        self.hidden_size = hidden_size
        self.rnn = nn.RNN(input_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x, hidden):
        out, hidden = self.rnn(x, hidden)
        out = self.fc(out[:, -1, :])
        return out, hidden

    def init_hidden(self):
        return torch.zeros(1, 1, self.hidden_size)
```

```
model = RNNModel(input_size=dataset.vocab_size, hidden_size=128,
output_size=dataset.vocab_size)
```

The task of generating the name

```
def generate_name(model, start_char, dataset, max_length=20):

    model.eval()

    if start_char not in dataset.char_to_index:

        raise ValueError(f"Start character '{start_char}' not found in dataset.")



    input_char = dataset.one_hot_encode(dataset.char_to_index[start_char]).view(1, 1, -1)

    hidden = model.init_hidden()

    name_generated = [start_char]



    for _ in range(max_length - 1):

        output, hidden = model(input_char, hidden)

        output_dist = output.data.view(-1).exp()

        top_i = torch.multinomial(output_dist, 1)[0]

        char = dataset.index_to_char[top_i.item()]



        if char == '\n':  # End of name

            break



        name_generated.append(char)

        input_char = dataset.one_hot_encode(top_i).view(1, 1, -1)



    return ''.join(name_generated)
```

The model was successful in generating Indian names when given the start character.

NLP(Natural language processing)

Natural Language Processing (NLP) is a subfield of artificial intelligence (AI) that focuses on the interaction between computers and humans through natural

language. The ultimate goal of NLP is to enable computers to understand, interpret, and generate human language in a way that is both meaningful and useful. NLP combines computational linguistics—rule-based modelling of human language—with statistical, machine learning, and deep learning models.

Key Components of NLP

1. **Tokenization**: Breaking down text into smaller units, such as words or phrases. For example, the sentence "NLP is fascinating" would be tokenized into ["NLP", "is", "fascinating"].
2. **Lemmatization and Stemming**: Reducing words to their base or root form. Lemmatization considers the context and converts words to their meaningful base form (e.g., "running" to "run"), whereas stemming cuts off prefixes or suffixes (e.g., "running" to "runn").
3. **Part-of-Speech (POS) Tagging**: Assigning parts of speech to each word in a sentence, such as nouns, verbs, adjectives, etc.
4. **Named Entity Recognition (NER)**: Identifying and classifying entities in text into predefined categories such as names of people, organisations, locations, dates, etc.
5. **Parsing**: Analysing the grammatical structure of a sentence. It includes syntactic parsing (sentence structure) and semantic parsing (meaning of the sentence).
6. **Sentiment Analysis**: Determining the sentiment or emotion expressed in a piece of text. This could be positive, negative, or neutral.
7. **Language Modelling**: Predicting the next word in a sentence. Language models are crucial for various NLP tasks like text generation and machine translation.
8. **Machine Translation**: Translating text or speech from one language to another.
9. **Text Classification**: Categorising text into predefined categories. Examples include spam detection in emails and topic classification in articles.
10. **Text Generation**: Creating new, meaningful text based on a given input. This includes tasks like writing essays, composing emails, and generating dialogue in chatbots

Attention Mechanisms and Transformers

Most of this section was based on the “Attention is all you need” paper which introduces the Transformer model, a novel architecture for sequence modelling and transduction tasks like machine translation. The key innovation is the use of self-attention mechanisms, which allow the model to weigh the importance of different words in a sentence relative to one another, without relying on recurrent or convolutional layers. The mentees were asked to watch various videos that explained the paper and encouraged to read the paper themselves and try to understand it to the best of their abilities.

Key Components of the Transformer architecture:

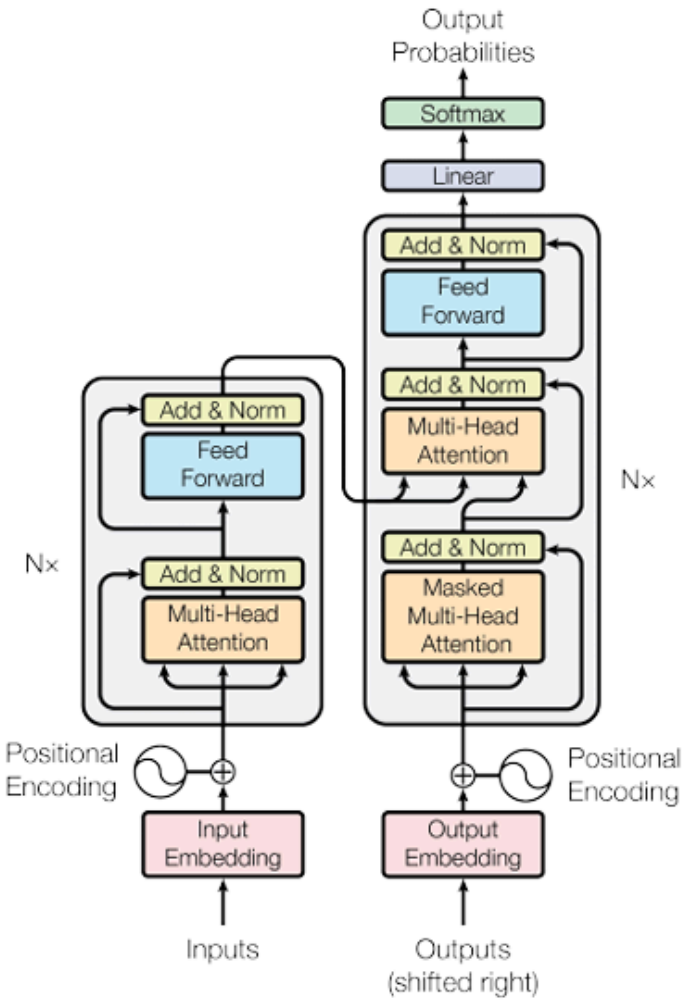


Figure 1: The Transformer - model architecture.

Self-Attention Mechanism:

- Allows each word to focus on different words in the sentence when creating its representation.
- Scaled Dot-Product Attention is the core mechanism:

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

- Where QQQ (queries), KKK (keys), and VVV (values) are derived from the input embeddings, and dkd_kdk is the dimension of the keys.

Multi-Head Attention:

- Enhances the model's ability to focus on different parts of the sentence from different perspectives:

$$\begin{aligned}\text{MultiHead}(Q, K, V) &= \text{Concat}(\text{head}_1, ..., \text{head}_\text{h})W^O \\ \text{where head}_\text{i} &= \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)\end{aligned}$$

- Each head_i is an independent self attention operation.

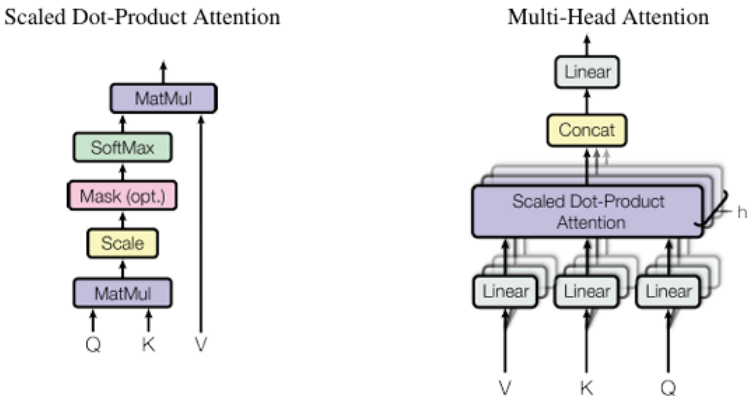


Figure 2: (left) Scaled Dot-Product Attention. (right) Multi-Head Attention consists of several attention layers running in parallel.

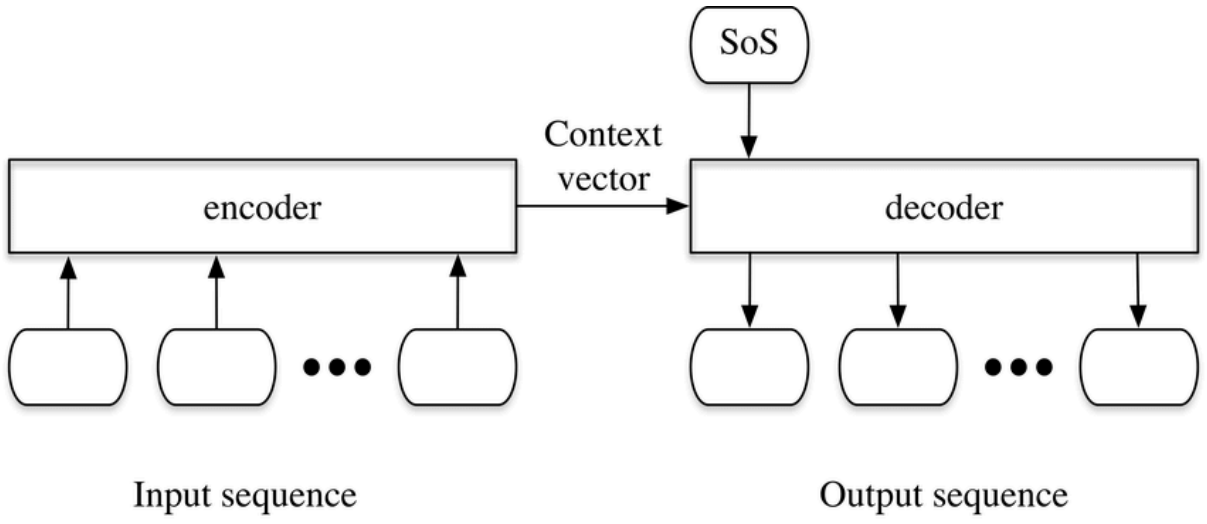
Positional Encoding:

- Adds information about the position of words in the sequence, since the model does not inherently capture order:

$$\begin{aligned}PE_{(pos,2i)} &= sin(pos/10000^{2i/d_\text{model}}) \\ PE_{(pos,2i+1)} &= cos(pos/10000^{2i/d_\text{model}})\end{aligned}$$

Encoder-Decoder Structure:

- The Transformer is composed of an encoder and a decoder, each consisting of a stack of layers.
- **Encoder**: Processes the input sequence and generates a set of encodings.
- **Decoder**: Uses the encoder's outputs to generate the output sequence, one word at a time.



All of this was rigorously covered in classes with special emphasis on trying to understand the paper of attention is all you need. Extreme emphasis and efforts were made by the mentors and mentees in attempting to understand this complex mechanism which makes it possible for Transformers, enabling the function of tools like ChatGPT.