WIDS PROJECT REPORT

Attention is all you Need:
Implementing a Generative AI Transformer

Created by Aditya Adhana , CSE'28

Week 1

In week 1, I was introduced to the topics such as **NumPy**, **pandas** and **matplotlib**. These libraries allow you to work with vectors and use many of their properties such as vector addition and subtraction to make the processing much easier.

```
[ ]  # < START >
     # Initialize an array ZERO_ARR of dimensions (4, 5, 2) whose every element is 0
     ZERO_ARR = np.empty((4,5,2))
     ZERO_ARR.fill(0)
     # < END >

     print(ZERO_ARR)

     # < START >
     # Initialize an array ONE_ARR of dimensions (4, 5, 2) whose every element is 1
     ONE_ARR = np.empty((4,5,2))
     ONE_ARR.fill(1)
     # < END >

     print(ONE_ARR)
```

Further I learnt how using vectorisation methods can greatly reduce the processing time compared to for-loops

```
import time

arr_nonvectorized = np.random.rand(1000, 1000)
arr_vectorized = np.array(arr_nonvectorized) # making a deep copy of the array

start_nv = time.time()

# Non-vectorized approach
# <START>



# <END>

end_nv = time.time()
print("Time taken in non-vectorized approach:", 1000*(end_nv-start_nv), "ms")

# uncomment and execute the below line to convince yourself that both approaches are doing the same thing
# print(arr_nonvectorized)

start_v = time.time()

# Vectorized approach
# <START>

# <END>

end_v = time.time()
print("Time taken in vectorized approach:", 1000*(end_v-start_v), "ms")

# uncomment and execute the below line to convince yourself that both approaches are doing the same thing
# print(arr_vectorized)
```

Further I learnt how to use matrices in python using NumpPy and slice them, generate matrices with random or preset values and use reshape and transpose functions.

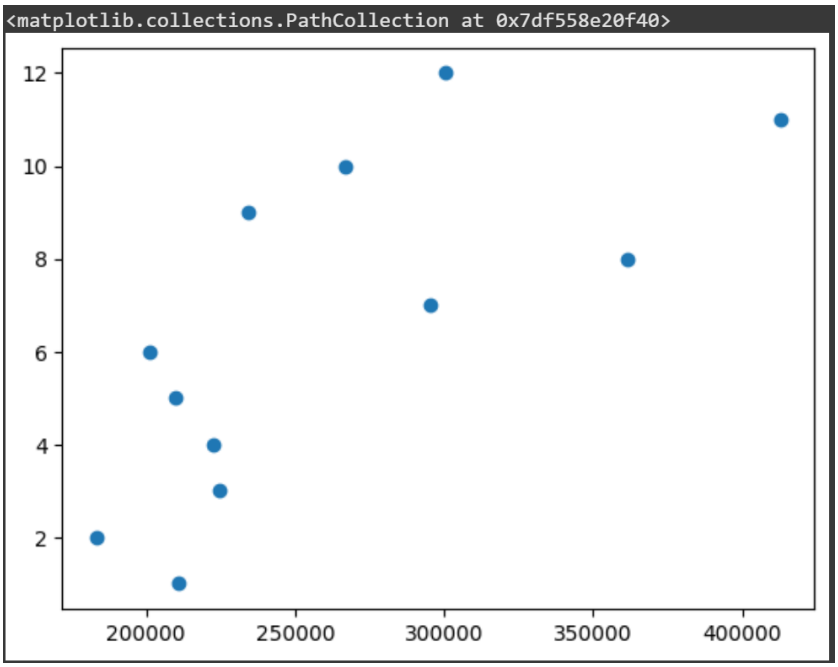
```
y = np.array([[1, 2, 3],
              [4, 5, 6]])

# < START >
# Create a new array y_transpose that is the transpose of matrix y

y_transpose = y.T

# < END >

print(y_transpose)

# < START >
# Create a new array y_flat that contains the same elements as y but has been flattened to a column array

y_flat = y.reshape((6,1))

# < END >

print(y_flat)
```

Next, I learnt Pandas for processing and presenting data

```
[ ] array = np.random.rand(100,5)

    # store the array in a csv file named numpy.csv use only numpy for this.

    np.savetxt('numpy.csv', array, delimiter=',')

    # store the array in a csv file named pandas.csv. This csv must also have a header. Column i must have heading 'column i'.

    df = pd.DataFrame(array, columns=[f'column {i+1}' for i in range(array.shape[1])])

    df.to_csv('pandas.csv', index=False)
```

```
    # load the data stored in pandas.csv

    # convert the pandas dataframe to numpy array

    # add 1 to all the elements of the numpy array

    # store the array to pandas_1.csv. This csv must also have a header. Column i must have heading 'column i'.

    import pandas as pd

    df = pd.read_csv('pandas.csv')

    array = df.to_numpy()

    for i in range(array.shape[0]):
      for j in range(array.shape[1]):
        array[i][j] += 1

    df = pd.DataFrame(array, columns=[f'column {i+1}' for i in range(array.shape[1])])

    df.to_csv('pandas_1.csv', index=False)
```

I learnt Matplotlib for processing graphical data, which is a pre requisite for plotting data. I also learnt graphic techniques for efficient management and interpretation of the data and results.



I also learnt the multivariant gradient descent method which utilises the property of approximations and computer to reach to an accurate answer

In this part of the assignment you have to implement multivariate gradient descent to find the minimas (local and global) of the given function: Note : you can find different minimas by changing your initialisation.

$f(x) = x^4 + x^2y^2 - y^2 + y^4 + 6$

```
[ ] import numpy as np
```

```
[ ] def call_func(func):
        def wrapper():
            func()
        return wrapper
```

```
[ ] def derivative(f, positions):
        positions = np.array(positions)
        print(positions)
        return np.array([4*positions[0]**3 + 2*positions[0]*positions[1]**2, 4*positions[1]**3 -2*positions[1] + 2*positions[1]*positions[0]**2])
```

```
[ ] def multivariable_gradient_descent(func, positions, alpha = 0.1):
        epsilon = 1e-5
        weights = np.ones(len(positions))
        count = 0
        while abs(sum(weights)) > epsilon:
            count += 1
            weights = derivative(func, positions)
            positions -= weights/max(100, abs(sum(weights)))
        return positions, count
```

Multivariate gradient descent is an extension of the basic gradient descent algorithm used for optimization in problems with multiple variables (or features). In simple linear regression, for instance, we typically deal with one variable. However, in multivariate gradient descent, we deal with several variables (features or predictors), and the goal is to minimize a cost function that depends on all of them. The core idea is to adjust the parameters (weights) iteratively by computing the gradient of the cost function with respect to each parameter. The gradients for each parameter tell us how the cost function changes as we change the parameter, and the algorithm adjusts the parameters in the opposite direction of the gradient to reduce the cost.

The method was later utilised for finding roots of the polynomial in a particular interval

```
[ ]  positions = [1,2]
     func = 5
     multivariable_gradient_descent(func, positions)
```

```
[1.74124204e-05 7.07106781e-01]
[1.72382962e-05 7.07106781e-01]
[1.70659133e-05 7.07106781e-01]
[1.68952541e-05 7.07106781e-01]
[1.67263016e-05 7.07106781e-01]
[1.65590386e-05 7.07106781e-01]
[1.63934482e-05 7.07106781e-01]
[1.62295137e-05 7.07106781e-01]
[1.60672186e-05 7.07106781e-01]
[1.59065464e-05 7.07106781e-01]
[1.57474809e-05 7.07106781e-01]
[1.55900061e-05 7.07106781e-01]
[1.54341061e-05 7.07106781e-01]
[1.52797650e-05 7.07106781e-01]
[1.51269673e-05 7.07106781e-01]
[1.49756977e-05 7.07106781e-01]
[1.48259407e-05 7.07106781e-01]
[1.46776813e-05 7.07106781e-01]
[1.45309045e-05 7.07106781e-01]
[1.43855954e-05 7.07106781e-01]
[1.42417395e-05 7.07106781e-01]
[1.40993221e-05 7.07106781e-01]
[1.39583289e-05 7.07106781e-01]
[1.38187456e-05 7.07106781e-01]
[1.36805581e-05 7.07106781e-01]
[1.35437525e-05 7.07106781e-01]
[1.34083150e-05 7.07106781e-01]
[1.32742319e-05 7.07106781e-01]
[1.31414895e-05 7.07106781e-01]
[1.30100746e-05 7.07106781e-01]
[1.28799739e-05 7.07106781e-01]
[1.27511742e-05 7.07106781e-01]
[1.26236624e-05 7.07106781e-01]
```

Week 2

In week 2, I finally applied all the gained knowledge about **NumPy** for creating logic gates AND, OR, NOR, NOT and using their combinations to create other logic gates such XOR.

XOR GATE

```
import numpy as np

def func4(output):
    if output==0:
        return 1
    else:
        return 0

def xor_gate(inp_1, inp_2):
    x1 = (-1)**(inp_1 + 1)
    x2 = (-1)**(inp_2 + 1)

    w1 = 0.5
    w2 = 0.5

    inp = np.array([[x1, x2]])
    w = np.array([[w1], [w2]])

    final = np.dot(inp, w)

    output = final.sum()

    return func4(output)

xor_gate(1,0)
```

Further, these concepts were used to create a full adder. A full adder is a digital circuit that performs addition of three input bits: two significant bits and a carry-in bit. It produces a sum bit and a carry-out bit as outputs. The sum represents the least significant bit of the total, while the carry-out indicates any overflow beyond the current bit. Full adders are used in binary addition.

FULL ADDER

```
def adder_sum(inp_1, inp_2 , cin):
  return xor_gate(xor_gate(inp_1, inp_2), cin)

def adder_cout(inp_1, inp_2 , cin):
  return or_gate(or_gate(and_gate(inp_1, inp_2), and_gate(inp_1, cin)), and_gate(inp_2, cin))

print(adder_sum(1,1,1))
print(adder_cout(1,1,1))
```

```
1
1
```

Finally, I created a ripple adder using all the gates and full adder

A ripple adder is a type of binary adder where multiple full adders are connected in series to add multi-bit binary numbers. In this arrangement, the carry-out from one adder is passed as the carry-in to the next adder in the chain. The term "ripple" refers to the delay in the carry signal as it propagates through each adder, which can result in slower performance for larger bit-widths. Ripple adders are simple and easy to design but are less efficient in terms of speed compared to more advanced adders like carry-lookahead adders.

RIPPLE ADDER

```
[ ]  def ripple_adder(inp_1 , inp_2):
       sum = 0
       cout = 0
       i =0
       while(inp_1 >0 or inp_2 >0):
         sum += adder_sum(inp_1%10, inp_2%10, cout)*(10**i)
         cout = adder_cout(inp_1%10, inp_2%10, cout)
         i+=1
         inp_1 = inp_1//10
         inp_2 = inp_2//10
       sum+= cout*(10**i)
       return sum

     ripple_adder(1111,1000)
```

```
10111
```

Week 3

In week 3, I created a simple layered network for identifying handwritten numbers, using MNIST. The pixels and coluorscales were scaled and subsequently fed to the network

```
import tensorflow as tf
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
import matplotlib.pyplot as plt
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.utils import to_categorical
```

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

# Preprocess the data
x_train = x_train / 255.0
x_test = x_test / 255.0
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
```

Then, I created custom layers to train the dataset using Soft Max and Relu function. The custom layers gave better accuracy than the default ones.

```
class CustomDenseReluLayer(tf.keras.layers.Layer):
    def __init__(self, units):
        super(CustomDenseReluLayer, self).__init__()
        self.units = units

    def build(self, input_shape):
        self.w = self.add_weight(shape=(input_shape[-1], self.units),initializer='random_normal',trainable=True)
        self.b = self.add_weight(shape=(self.units,),initializer='zeros',trainable=True)

    def call(self, inputs):
        z = tf.matmul(inputs, self.w) + self.b
        z = tf.nn.relu(z)
        return z

class CustomDenseSoftmaxLayer(tf.keras.layers.Layer):
    def __init__(self, units):
        super(CustomDenseSoftmaxLayer, self).__init__()
        self.units = units

    def build(self, input_shape):
        self.w = self.add_weight(shape=(input_shape[-1], self.units),initializer='random_normal',trainable=True)
        self.b = self.add_weight(shape=(self.units,),initializer='zeros',trainable=True)

    def call(self, inputs):
        z = tf.matmul(inputs, self.w) + self.b
        z = tf.nn.softmax(z)
        return z

class CustomFlattenLayer(tf.keras.layers.Layer):
    def call(self, inputs):
        return tf.reshape(inputs, (tf.shape(inputs)[0], -1))
```

The Boston Housing case

I implemented the learnings of week 3 to create a model for Boston Housing data

I first created a linear regression model, using a uniform scaling for all factors which did not give a good accuracy.

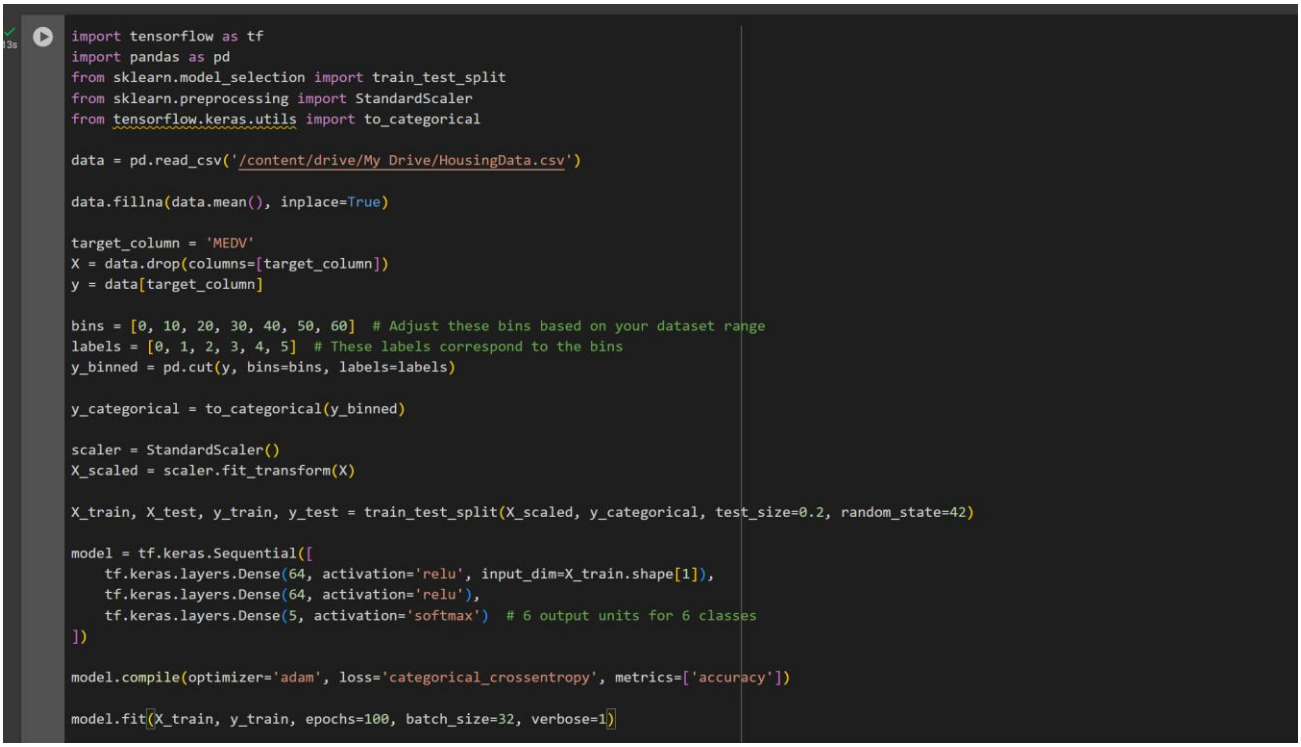
```python
import tensorflow as tf
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

data = pd.read_csv('/content/drive/My Drive/HousingData.csv')
data.fillna(data.mean(), inplace=True)

target_column = 'MEDV'
X = data.drop(columns=[target_column])
y = data[target_column]

scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

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

model = tf.keras.Sequential([
    tf.keras.layers.Dense(1, input_dim=X_train.shape[1], activation=None)
])

model.compile(optimizer='adam', loss='mean_squared_error')

model.fit(X_train, y_train, epochs=100, batch_size=32, verbose=1)

test_loss = model.evaluate(X_test, y_test)
print(f'Test Loss (MSE): {test_loss}')

y_pred = model.predict(X_test)

print("Predictions:", y_pred[:5])
print("Actual values:", y_test[:5].values)
```

To improve the accuracy, I studied the data and added custom scaling factors for each data column. This gave a much accuracy for the same model

```python
# Custom scaling factors
scaling_factors = {
    'CRIM': 100,
    'ZN': 100,
    'INDUS': 30,
    'CHAS': 2,
    'NOX': 1,
    'RM': 10,
    'AGE': 100,
    'DIS': 10,
    'RAD': 30,
    'TAX': 1000,
    'PTRATIO': 30,
    'B': 500,
    'LSTAT': 30
}
```

The Boston Housing case

Finally I created a feedforward Neural network for the same case using custom Softmax and Relu activation layers. The model implemented three layers – 1 flattened and 2 dense.



```
import tensorflow as tf
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from tensorflow.keras.utils import to_categorical

data = pd.read_csv('/content/drive/My Drive/HousingData.csv')

data.fillna(data.mean(), inplace=True)

target_column = 'MEDV'
X = data.drop(columns=[target_column])
y = data[target_column]

bins = [0, 10, 20, 30, 40, 50, 60]  # Adjust these bins based on your dataset range
labels = [0, 1, 2, 3, 4, 5]  # These labels correspond to the bins
y_binned = pd.cut(y, bins=bins, labels=labels)

y_categorical = to_categorical(y_binned)

scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

X_train, X_test, y_train, y_test = train_test_split(X_scaled, y_categorical, test_size=0.2, random_state=42)

model = tf.keras.Sequential([
    tf.keras.layers.Dense(64, activation='relu', input_dim=X_train.shape[1]),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(5, activation='softmax')  # 6 output units for 6 classes
])

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

model.fit(X_train, y_train, epochs=100, batch_size=32, verbose=1)
```

Week 4

In week 4, Finally I implemented all he knowledge of logics, layers and neural networks to create and the test the Generative AI transformer.

```
model.summary()
```

Model: "functional_8"

Layer (type)	Output Shape	Param #
input_layer_5 (InputLayer)	(None, 64)	0
token_and_position_embedding_1 (TokenAndPositionEmbedding)	(None, 64, 256)	33,024
transformer_block_4 (TransformerBlock)	(None, 64, 256)	1,184,512
transformer_block_5 (TransformerBlock)	(None, 64, 256)	1,184,512
transformer_block_6 (TransformerBlock)	(None, 64, 256)	1,184,512
dense_15 (Dense)	(None, 64, 65)	16,705

Total params: 3,603,265 (13.75 MB)
Trainable params: 3,603,265 (13.75 MB)
Non-trainable params: 0 (0.00 B)

I changed the parameters and found a good accuracy for the following parameters.

block_size = 64 # Reduced block size for faster training and to prevent overfitting

num_heads = 4 # Lower number of attention heads to reduce model complexity

num_transformer_blocks = 3 # Reduced layers to avoid overfitting

feed_forward_dim = 128 # Smaller feed-forward dimensions to reduce model complexity

dropout_rate = 0.3 # Increased dropout to prevent overfitting

batch_size = 16 # Smaller batch size for better generalization

SUMMARY

Throughout this project, I gained valuable insights into both computational techniques and digital logic design. One of the most significant aspects I learned was the power of **vectorization** in Python, particularly with the NumPy library. Vectorization allows for more efficient computations by eliminating the need for time-consuming for-loops, leading to faster and more scalable data processing. This understanding deepened my appreciation for how mathematical operations can be optimized and leveraged for better performance in real-world applications.

In addition to vectorization, I explored the manipulation of **matrices** using NumPy, which is foundational for tasks like linear algebra, machine learning, and data science. I now understand the utility of functions like reshaping, transposing, and slicing in matrix operations, which are essential tools for working with multidimensional data. These techniques play a key role in implementing algorithms like **multivariate gradient descent**.

Multivariate gradient descent, in particular, provided me with a deeper understanding of optimization in machine learning. The ability to iteratively adjust parameters by calculating gradients and minimizing a cost function is a fundamental concept for training machine learning models. I grasped how this approach is essential in problems with multiple features, and how it enables more accurate and efficient solutions by fine-tuning the model parameters.

The application of these computational principles to **digital logic design** further expanded my knowledge of how basic computational units, like logic gates and adders, are implemented. I developed an understanding of how fundamental logic gates like AND, OR, and XOR are the building blocks for more complex digital circuits. The **full adder** and **ripple adder** are key components in binary addition, and learning how to combine them showed me how digital systems process binary data, perform arithmetic operations, and handle carry-over values.

Overall, this project not only enhanced my technical skills in Python and digital logic design but also deepened my understanding of the theoretical concepts that drive many technologies, from machine learning algorithms to digital circuits. In generative AI, these models are trained on massive datasets to understand patterns and dependencies. Once trained, they can generate new data that is similar to the training data but novel in nature. For example, GPT (Generative Pre-trained Transformer) is a widely known Transformer-based model that generates coherent and contextually relevant text based on an input prompt. The ability to generate realistic outputs is powered by the model's deep layers and vast amount of parameters, making it versatile for a wide range of applications, from creative writing to problem-solving.

In the end, I would like to say that this was an excellent opportunity for me to learn the various perspectives of Logics as well as Numpy and other python libraries like Pandas. The concepts of approximations and logic gates as well as the ability to create adders was really insightful. Finally I created a generative AI transformer.

I would like to thank my mentors, Aditya Neeraje and Yash Sabale for the continuous guidance and aid in case of confusions.

THANK YOU