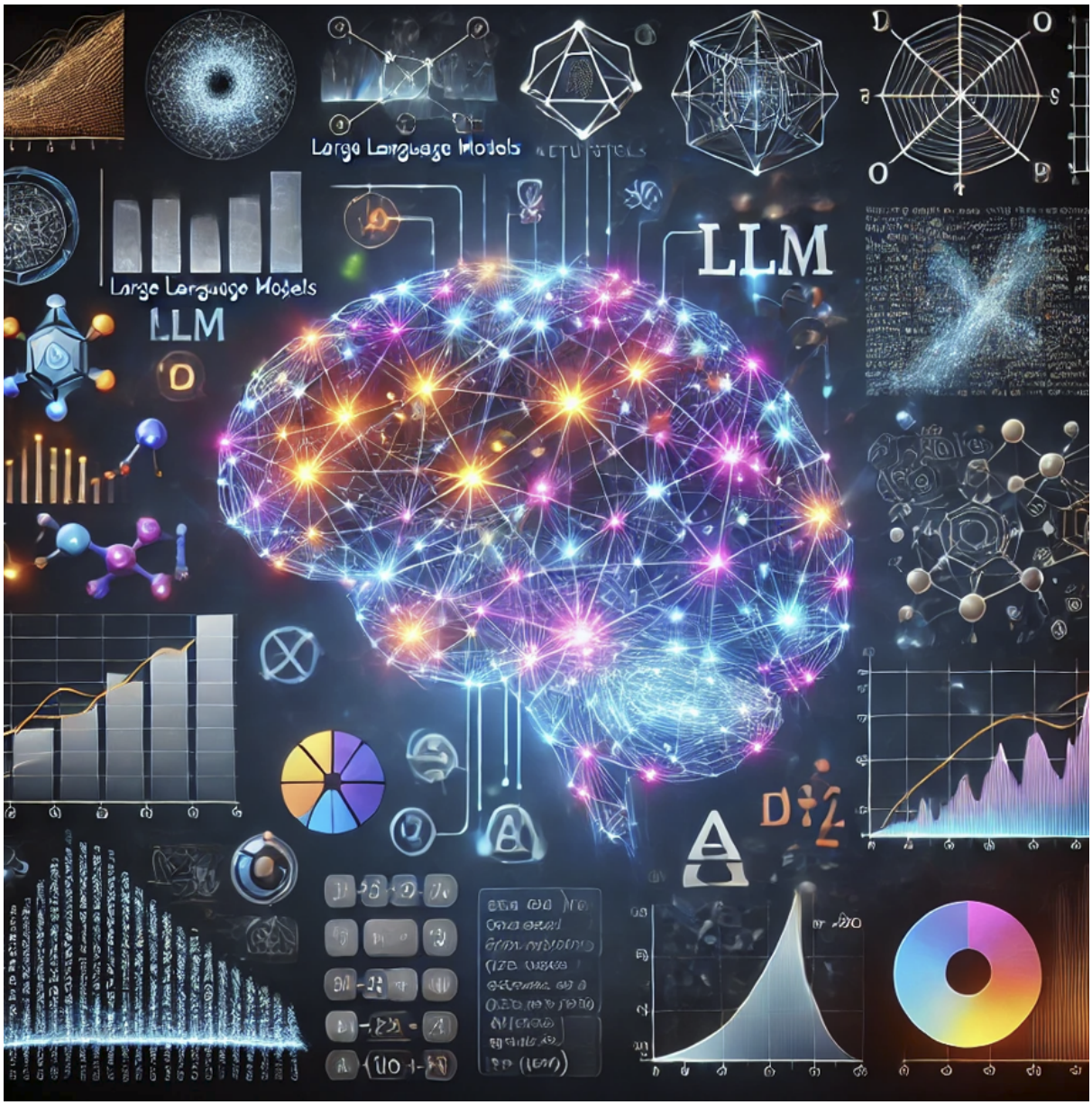
Generative Transformers

Shaashvat Sekhar - WIDS 2k24

14 Feb 2025



Contents

1 Introduction

The link to my entire project can be found on GitHub

`https://github.com/shaashvats/Generative-AI-Transformer/tree/main/`

a Transformers

Transformers have emerged as one of the most versatile and powerful architectures in modern machine learning, driving advances in natural language processing and generative AI.

Their ability to handle complex patterns and dependencies in data has made them foundational for large language models (LLMs) and other generative AI tools, which are rapidly shaping the future of technology and communication.

b Acknowledgments

My motivation to do this project was to learn the underlying maths of a transformer, and to learn how to implement a basic generative transformer.

I thank my mentors Aditya Neeraje, and Yash Sabale for giving me this opportunity, guiding me through this project, and providing useful resources.

I thank the Analytics Club IIT Bombay, and WiDS 4.0 for connecting me with my mentors.

Shaashvat Sekhar

2 Week One

In the first part of the project I familiarised myself with some python libraries and tools, and implemented gradient descent

a Familiarization, and Insights

- Python is a relatively easy language to write code in because it is a high-level language with english-like syntax and has lots of useful libraries for ML
- NumPy has a feature called broadcasting which enables simultaneous element-wise operation on arrays of different shapes using vectorization. It enables fast matrix operations, making it useful for ML
- Jupyter Notebooks are useful for creating projects because they have blocks which can be used to write and execute code systematically one step-at-a-time. They are also compatible with with LaTeX for equations, MatPlot Lib for visualising and all other python libraries

b Gradient Descent

- Gradient Descent is a method to find a local minima of a continuous function. It is based on the principles of multivariable calculus, and concepts of partial derivatives and gradient.
- The greatest descent of a function at a point x is in the direction of: $-\nabla f$
- By iteration of:

$$x_{i+1} = x - \alpha * \nabla f(x_i)$$

  Where α is called the "learning rate" and is a small positive constant, we can find a local minima of a functions (with some limitations)
- α Can be dynamic to do gradient descent in less iterations (epochs). The "adam" optimizer uses this idea.
- I used gradient descent to find a local minima of

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

- This function has two local minimas, and gradient descent based on the initial position, ends up at (0, -7.07) or (0, 0.707)
- This demonstrates that gradient descent can be used to find a local minima but not necesserily a global minima
- (code, outputs, visualisation: on next page)

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

✓ 0.0s

Figure 1: Gradient Descent Algo

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

[9] ✓ 0.0s

```
(array([9.94526076e-06, 7.07106781e-01]), 1071)
```

```python
positions = np.array([-2,-2], dtype='float64')
func = 5
multivariable_gradient_descent(func, positions)
```

[10] ✓ 0.0s

```
(array([-9.97043752e-06, -7.07106781e-01]), 1073)
```

Figure 2: Gradient Descent Output

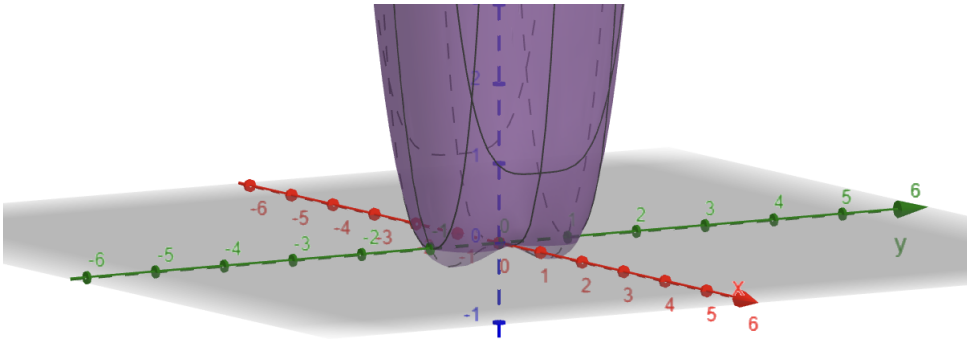


Figure 3: Function Visualisation

3 Week Two

In week two, I learned theory behind feedforward neural networks and used it to implement simple neural networks carrying out Boolean operations of two inputs

a Feedforward Neural Network

- A feedforward neural network has multiple layers of "neurons" $A^{(i)}$
- Each layer of neurons is created by a linear combination of the previous layer (inputs for layer-1) plus addition of bias, and finally a passthrough a non-linear activation function σ. This process is called **Forward Propogation.**

$$z^{(1)} = W^{(1)}x + b^{(1)}$$

$$a^{(1)} = \sigma(z^{(1)}) = \sigma(W^{(1)}x + b^{(1)})$$

$$z^{(2)} = W^{(2)}a^{(1)} + b^{(2)}$$

$$\hat{y} = \sigma_{\text{out}}(z^{(2)})$$

- Each neuron acts like a logistic regression. Having multiple neurons allows the model to "learn" different patterns. The "knowledge" is stored in the coefficients and biases which can create interesting effects because of the non-linear activation
- The generalised expression:

$$a^{(l)} = \sigma(W^{(l)}a^{(l-1)} + b^{(l)}), \quad \text{for } l = 2, 3, \ldots, L$$

$$a^{(0)} = x \quad \text{(input layer)}, \quad a^{(L)} = \hat{y} \quad \text{(output)}$$

- Loss is calculated through Binary Cross Entropy, and the sum of the loss across several test cases is used to determine how coefficients and biases are to be updated to improve the models functioning

$$\mathcal{L} = -\sum_{i=1}^{N} y_i \log(\hat{y}_i)$$

- We differenciate the loss with respect to the coefficients of different layers and update the coefficients based on the gradient. .We do not find the "exact" gradient with respect to all coefficients at once, rather, we go layer by layer and iterate each layer independantly. This process is called **Backward Propogation.** and it is used to make the model "learn"

$$\frac{\partial \mathcal{L}}{\partial W^{(l)}} = \frac{\partial \mathcal{L}}{\partial a^{(l)}} \cdot \frac{\partial a^{(l)}}{\partial z^{(l)}} \cdot \frac{\partial z^{(l)}}{\partial W^{(l)}}$$

$$W^{(l)} := W^{(l)} - \eta \frac{\partial \mathcal{L}}{\partial W^{(l)}} \quad , \quad b^{(l)} := b^{(l)} - \eta \frac{\partial \mathcal{L}}{\partial b^{(l)}}$$

- The inital coefficients and biases are set randomly in a process known as **Seeding**. They are set randomly instead of uniformly because of the way back propogation works, symmetric weights would make the model not learn diverse traits, rather, it would behave very primitively.

b Hyperparameter Tuning

- Hyperparameter tuning is done during the training of a machine learning model to find the best settings that improve how well the model learns before testing it on new data.
- It helps the model work better by avoiding mistakes like underfitting (too simple) or overfitting (too complex), and affecting how quickly the model learns
- What can be tuned:

  Learning Rate (η) determines how quickly the model changes coefficients and biases. A higher η can find a minima quicker, with a risk of overshooting.

  Number of Epochs determines how many iterations of the model are done. Too many can create overfitting, too few can create underfitting

  Activation function determines how loss is calculated and how the model behaves. If affects how the model learns, and the nature of outputs it can generate
- For the simple AND, XOR, Full-Adder Neural networks, hyperparameter tuning was not important because of how trivially the network was required to behave. However, for The Boston Housing Data, I did use it to improve model predictions

c Perceptron

- A perceptron is the simplest configuration of a feedforward neural net. It has no hidden layer. It has n inputs, one output, and only one activation.

$$z^{(1)} = W^{(1)}x + b^{(1)}$$

$$f(x) = h(z^{(1)})$$

h(u) is the Heaviside function where $h(u) = 1 \; \forall \; u \; \geq 1, h(u) = 0$ o.w.

- I made a slight change: I used the sigmoid activation instead of h(u). This allowed me to use binary cross entropy to calculate loss, and use the related formulas for backpropogation on g(x).

$$g(x) = \sigma(z^{(1)})$$

- I then rounded the outputs to get f(x), my output.

$$f(x) = \text{Round}(\sigma(z^{(1)}))$$

- The general idea of how a sigmoid activation works is that Z is a linear combination of inputs, and the sigmoid activation is a monotonically increasing function with a sharp rise from 0 to 1 around origin. This means that a sigmoid activation works like a classifier of a linear combination of inputs.
- For AND: This would work along the following lines:

  The output $y = a \wedge b$ is 1 only when a=1 and b=1. In all other cases, the output y = 0. We can see a clear linear demarcation of y on the basis of a+b where $y = 1 \iff a + b > 1$

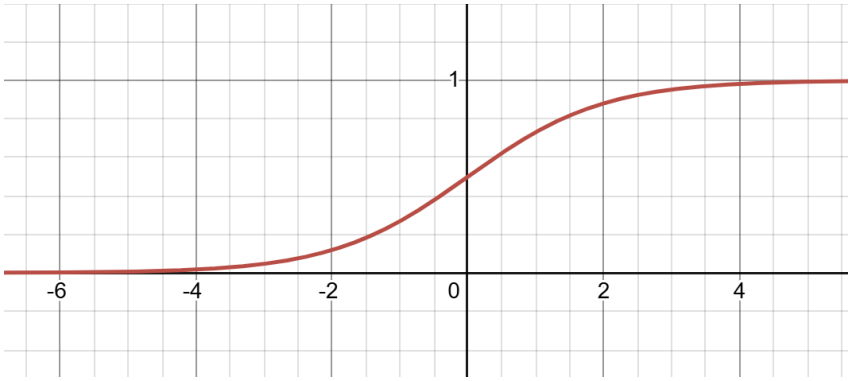


Figure 4: Sigmoid Function

We exploit this by using coefficients, bias: $z = \beta * (a + b - 1.5)$

Which ensures z would be positive only when both $a = 1$ and $b = 1$ and otherwise it would be negative.

A sufficiently large β would move z far enough from the origin that the output of the sigmoid activation would be very close to 0 when $z < 0$ and very close to 1 when $z > 0$. Thus a perceptron can very easily model an AND gate.

- For XOR: There is no linear combination of a,b that can demarcate (0,1) and (1,0) from (1,1) and (0,0). Hence a perceptron cannot model an XOR Gate

d Perceptron Implementation of AND

- I used sigmoid activation for backpropogation and then rounded to get results

  (see code in Week 2 of project)

- The final weights and bias came to be similar to what I had expected

  (see previous section)

- **XOR** The perceptron model couldn't model XOR gate as expected

```
Trained weights: [[7.41918007]
 [7.41918041]]
Trained bias: -11.301366033410076

Predictions:
[[0.]
 [0.]
 [0.]
 [1.]]
```

```
Trained weights: [[2.49330776e-16]
 [2.48742219e-16]]
Trained bias: -3.31850112596023e-16

Predictions:
[[0.]
 [0.]
 [0.]
 [0.]]
```

Figure 5: Successful AND (Left) - Unsuccessful XOR (right)

e Neural Net Implementation of XOR

- I used a neural net with one hidden layer. The hidden layer had two neurons.
- I rewrote the functions for forward propogation and backward propogation to account for the additional hidden layer item. The neural net was successfully.
- The intuition here is that (1,0) can be demarcated from the other 3 inputs and so can (0,1). One of the hidden neurons activates only for the prior and the other, for the latter. The output neuron then activates if atleast one of the hidden neurons does. The inference drawn here is that non convex behaviour can be modelled by many convex activations, something which one convex non-linear activation cannot do

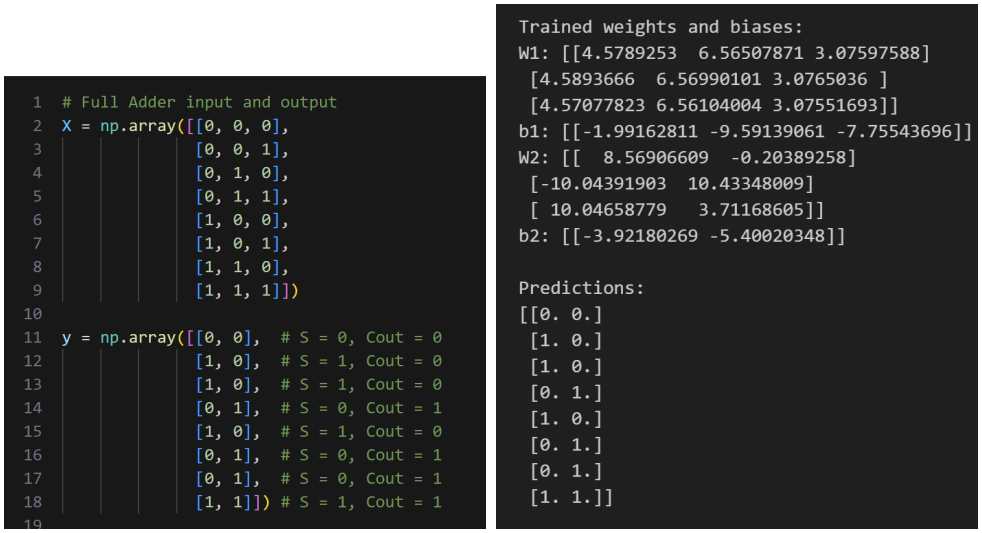
```
Trained weights and biases:
W1: [[4.57608671 6.51164718]
 [4.578482   6.52534109]]
b1: [[-7.00486476 -2.86347719]]
W2: [[-10.32773912]
 [  9.61540107]]
b2: [[-4.3991656]]

Predictions:
[[0.]
 [1.]
 [1.]
 [0.]]
```

Figure 6: Successful XOR using 2 Hidden Layer Neurons

f Neural Net Implementation of Full Adder

- In this case, two hidden neurons were insufficient but three were sufficient.
- Note: there are two outputs for a full adder, as opposed to one
- It seems obvious to me that the full adder requires more hidden neurons to "learn" more complex behaviour. However I couldn't predict for a more complicated circuit, or why its 3 and not 4 here.



```
# Full Adder input and output
X = np.array([[0, 0, 0],
              [0, 0, 1],
              [0, 1, 0],
              [0, 1, 1],
              [1, 0, 0],
              [1, 0, 1],
              [1, 1, 0],
              [1, 1, 1]])

y = np.array([[0, 0],  # S = 0, Cout = 0
              [1, 0],  # S = 1, Cout = 0
              [1, 0],  # S = 1, Cout = 0
              [0, 1],  # S = 0, Cout = 1
              [1, 0],  # S = 1, Cout = 0
              [0, 1],  # S = 0, Cout = 1
              [0, 1],  # S = 0, Cout = 1
              [1, 1]]) # S = 1, Cout = 1
```

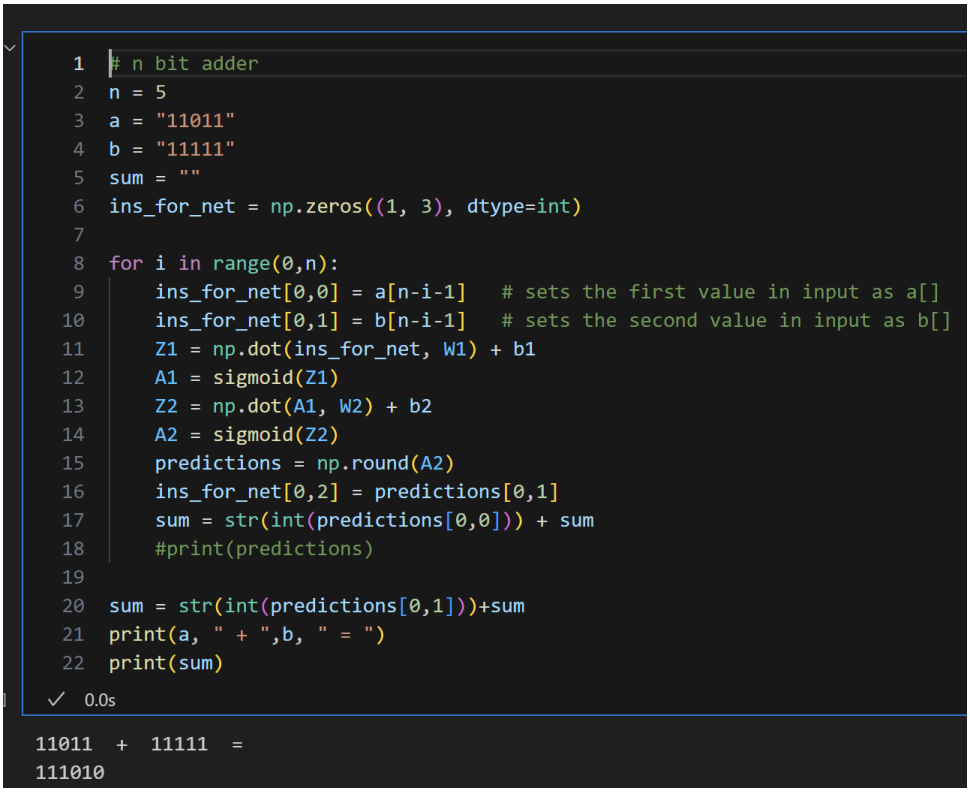
```
Trained weights and biases:
W1: [[4.5789253  6.56507871 3.07597588]
 [4.5893666  6.56990101 3.0765036 ]
 [4.57077823 6.56104004 3.07551693]]
b1: [[-1.99162811 -9.59139061 -7.75543696]]
W2: [[  8.56906609  -0.20389258]
 [-10.04391903  10.43348009]
 [ 10.04658779   3.71168605]]
b2: [[-3.92180269 -5.40020348]]

Predictions:
[[0. 0.]
 [1. 0.]
 [1. 0.]
 [0. 1.]
 [1. 0.]
 [0. 1.]
 [0. 1.]
 [1. 1.]]
```

Figure 7: Full Adder Training Data (left), Model (right)

g Ripple Carry Adder

- For the Ripple carry adder, I did multiple feedforwards using the Full Adder neural net coefficients I obtained. I cascaded the process to feed the computed carry-out of the n-th addition to the carry-in of the (n+1)-th



```
# n bit adder
n = 5
a = "11011"
b = "11111"
sum = ""
ins_for_net = np.zeros((1, 3), dtype=int)

for i in range(0,n):
    ins_for_net[0,0] = a[n-i-1]   # sets the first value in input as a[]
    ins_for_net[0,1] = b[n-i-1]   # sets the second value in input as b[]
    Z1 = np.dot(ins_for_net, W1) + b1
    A1 = sigmoid(Z1)
    Z2 = np.dot(A1, W2) + b2
    A2 = sigmoid(Z2)
    predictions = np.round(A2)
    ins_for_net[0,2] = predictions[0,1]
    sum = str(int(predictions[0,0])) + sum
    #print(predictions)

sum = str(int(predictions[0,1]))+sum
print(a, " + ",b, " = ")
print(sum)
```

✓ 0.0s

```
11011  +  11111  =
111010
```

Figure 8: Ripple Carry Adder

4 Week Three

I implemented a neural net to predict house prices in the well-known Boston Housing Dataset. I learnt to improve, and gauge model performance by tuning hyperparameters.

```
Number_of_train = 300
np.random.shuffle(data)
x_train = data[0:Number_of_train, :-1]
y_train = data[0:Number_of_train, -1]
x_test = data[Number_of_train:, :-1]
y_test = data[Number_of_train:, -1]
model = tf.keras.Sequential([
    tf.keras.layers.Dense(40, input_dim=13, activation='relu',
       use_bias=True),
    tf.keras.layers.Dense(40, activation='relu', use_bias=True),
    tf.keras.layers.Dense(1)
model.compile(optimizer='adam', loss='mean_squared_error')
model.fit(x_train, y_train, epochs=100, batch_size=300)
test_loss = model.evaluate(x_test, y_test)
y_pred = model.predict(x_test)
```

a Preprocessing

- I shuffled the data to ensure that the order in which data was collected did not affect which data was used to train the model and which would be used to test.
- I trained the model on 300 instances, and used the rest ≈ 200 to test.

b Model

- I changed different parameters till I could get a model that had really low loss, and had a similar loss on the train data as the test data. This was to neither overfit, nor underfit the model onto the data.
- This brought an interesting question to my mind: Given a training dataset, how well can we theoretically make predictions using a neural net on a test dataset?
- I used a high number of epochs for the model. This allowed the model to go very close to ideal behaviour. I lost count of the total number of epochs I used.
- A small batch size has a faster epoch time, but creates noisier updates. I tried using a combination of both small and large batch sizes. Given the size of the data, it didn't matter which batch size I used to save time, but an insight I drew into training larger models is that we can initially use small batch sizes to get close to a minima, and use a large batch sizes for fine-tuning the model as we get closer.
- I used relu as the activation function. I did this because the outputs were unbounded, and of an analog nature rather than a yes-no bounded nature. Sigmoid and tanh didn't work as they can't handle the range of values that a linear/polynomial function requires, that I tried to model prices as. Similarly, I used mean-square error to calculate loss.

- I used two hidden layers, each of 40 neurons. My intuition suggested that the model can learn traits in line with the minimum number of neurons it has in a layer. So when I created multiple layers I thought it would be best to have equal number of neurons in each layer. With fewer layers, the model had some instances with very large loss. This suggested that the model didn't have enough neurons to account for that pattern of behaviour. Increasing the number of neurons solved this issue.

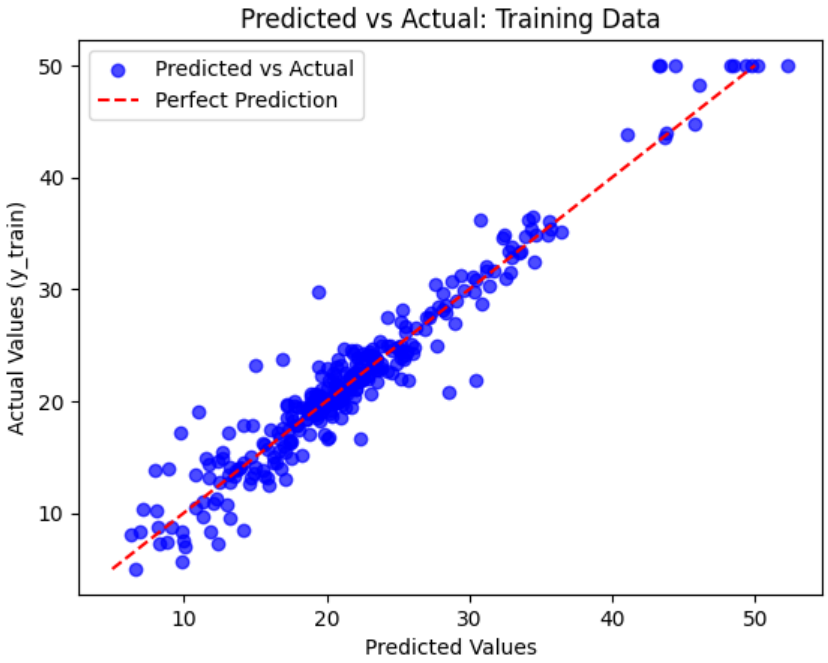


Figure 9: Model performance on training data

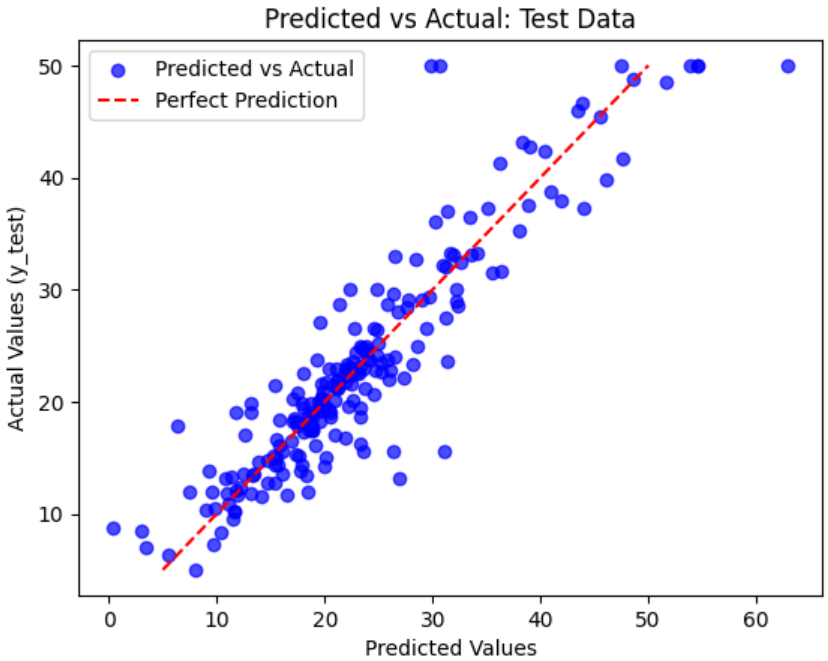


Figure 10: Model performance on test data

c Some Failures

- Tanh/Sigmoid in both layer activation couldn't model linear/polyonmial behaviour

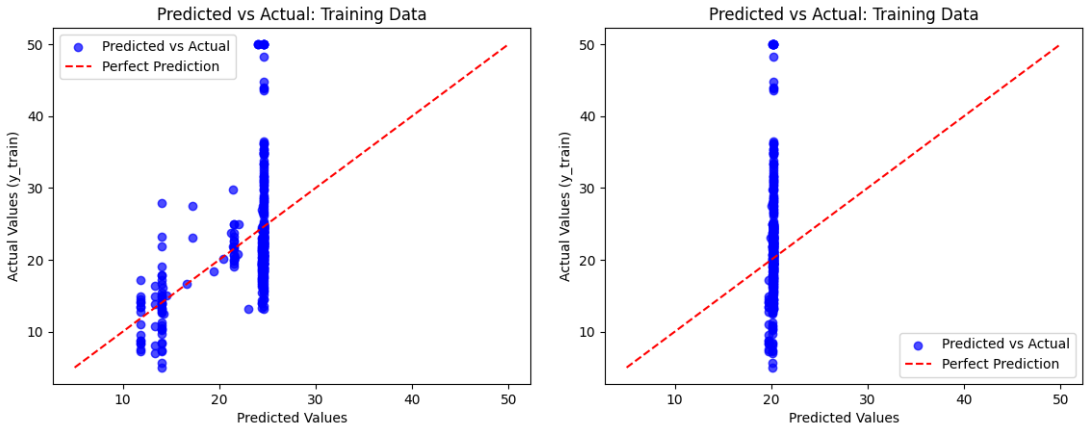


Figure 11: Tanh (left) - Sigmoid (right)

- Using a combination of tanh, relu couldn't model linear/polyonmial behaviour

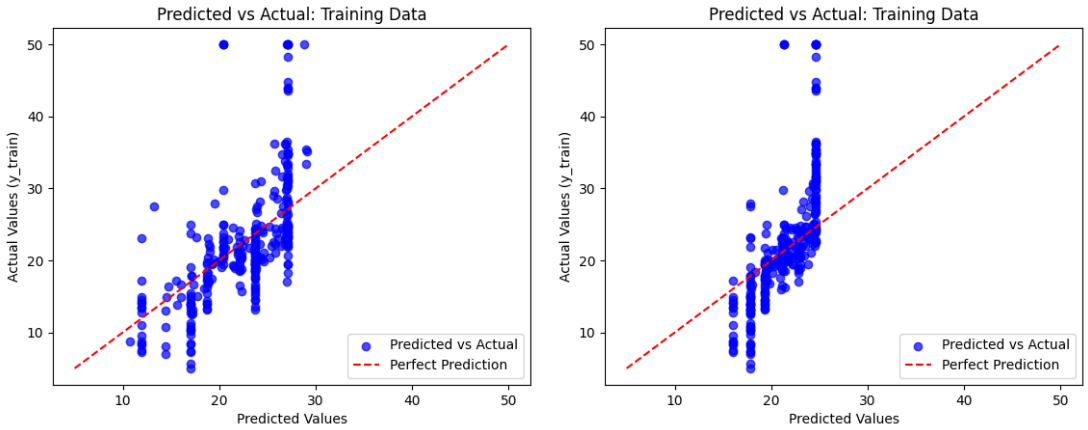


Figure 12: Tanh then Relu (left) - Relu then Tanh (right)

5 Week Four

In week four, I learnt the theory behind the architecture of transformers and implemented one to produce text like Shakespeare using generative character prediction.

a Transformer Theory

- **Tokenization** converts words to tokens, making a string a series of tokens
- **Token Embedding** assigns each token a vector in a high dimension space which begins to hold information of the meaning of the word
- **Positional Encoding** assings each position before the next word a vector in the same space as token embeddings
- **Attention** updates embeddings of latter embeddings based on their previous embeddings using Query, Key, and Value. Attention block helps the model learn context and how one word affects another after it.
- **Neural Net** predicts the next embedding based on the N embedding output of the attention block
- **Decoder**, finally, predicts the next word based on the embedding output of the neural net using softmax
- The oldest word is removed, the other words are shifted and the new word is added to the string. The new string is fed to the transformer and it agains repeats the process of predicting the next word.
- The transformer is trained similar to a neural network, using Backpropogation

b Character Tokenization

- In my first attempt at building a Large Language Model (LLM), I tokenized the characters not the words
- Given the sparse nature of training data, and very few epochs of training implemented, this resulted in some illegible text output (See next page)
- In some sense, it is a successful "stochastic parrot" because it is generating new tokens, however this is not what I had set out to achieve.
- Something I learnt from this endeavour is that LLMs take a lot of compute to train. This primitive one took ten hours for ten epochs.
- (Unofficial conclusion) I did not have the motivation to build the correct version of the Shakespeare parrot (where I tokenize words instead). This leaves me feeling a slight lack of closure, which I hope to channel some of into building more technical, detailed projects involving ai/ml/transformers/llms in the future. I hope that the efforts I have put into this project reflect a growing understanding of AI and my eagerness to learn.

```python
# Generate new text
generated_text = generate_text(model,
                               start_string="Why",
                               char_to_idx=char_to_idx,
                               idx_to_char=idx_to_char)
print(generated_text)
```

```
Whyos domanfinae, t l;
rothereansly: r aa,
r
t k
:
srnevey m,
si e
te
-'n on



,
d,
e
ow--paoIiN:
e
t.
igiso s
,

s
ted ! e
, llet.s
tusinothneethewaw,
lestotoforendgend; ao?uthau
...

ce
t kis m
den
```

Figure 13: Character tokenization

6 Conclusion

This is the first project in machine learning that I have completed. Over the course of the project, I have learned to work with TensorFlow, Keras, and NumPy, using them to implement various machine learning models. In weeks one and two, I implemented perceptrons and neural networks to model logical gates. In week three, I developed a neural network to predict house prices using the Boston housing dataset with TensorFlow. Finally, I implemented a transformer model for character generation, training it on Shakespeare's texts.

By building these models, I have gained a solid foundation in machine learning and artificial intelligence. Familiarizing myself with the transformer architecture has helped me understand its applications in various generative AI systems. This project has sparked my interest in exploring the theoretical foundations of artificial intelligence and in creating projects that leverage it to generate value for society.

7 Bibliography, References

- `https://github.com/AdityaNeeraje/Generative-AI-Transformer` by Aditya Neeraje and Yash Sabale
- Stanford CS229: Machine Learning, by Andrew Ng
- Deep learning Video Series, by 3b1b on YouTube
- ChatGPT and Claude.ai: for code and report-writing
- Boston Housing Data, by US Census Service
- Shakespeare's work, by Shakespeare