Generative Transformers

Shaashvat Sekhar - WIDS 2k24

14 Feb 2025

Contents

1 Introduction

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.

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 Yash Sabale, and Aditya Neeraje for giving me this opportunity.

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

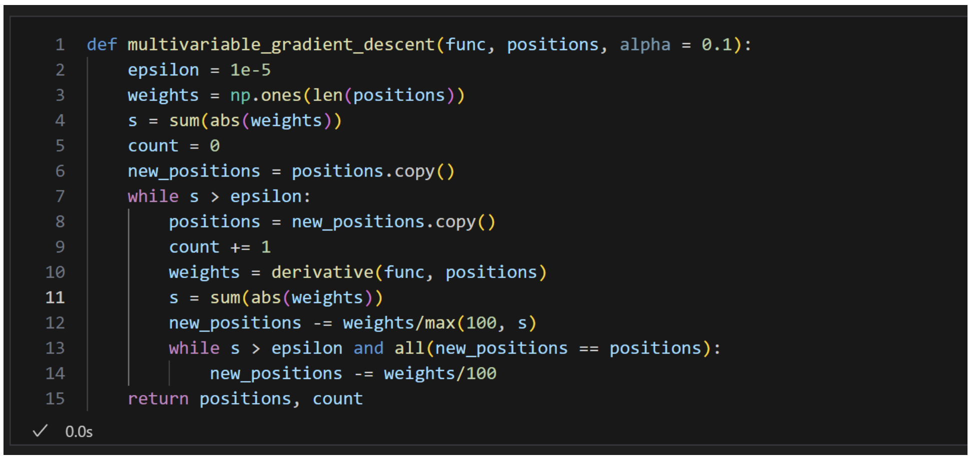
b Gradient Descent

- 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

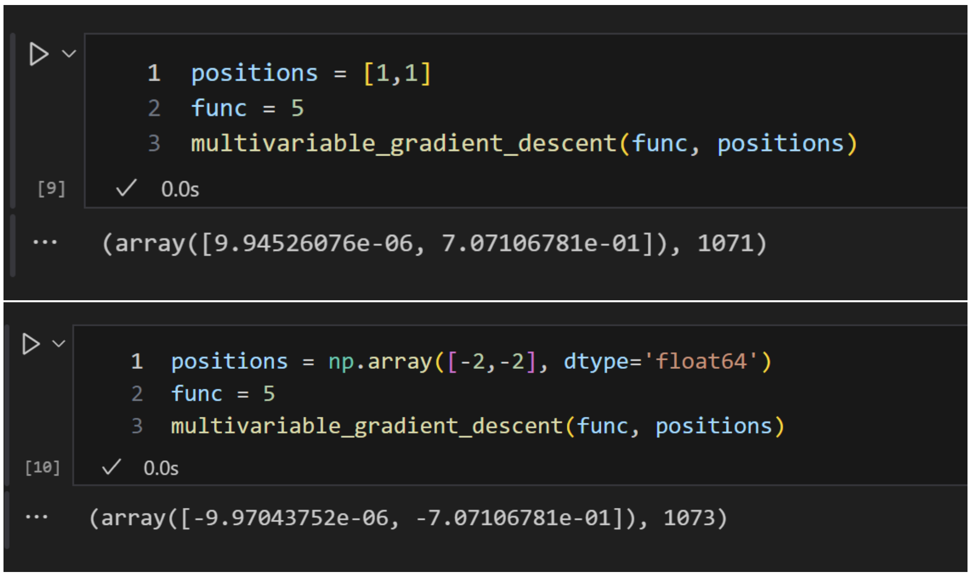
  $f(x) = x^4 + x^2y^2 - y^2 + y^4 + 6$
- This function has two local minimas, and gradient descent based on the initial position, ends up at (0, -7.07) or (0, 0.707)
- This demonstrates that gradient descent can be used to find a local minima but not necesserily a global minima
- (code, outputs, visualisation: 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

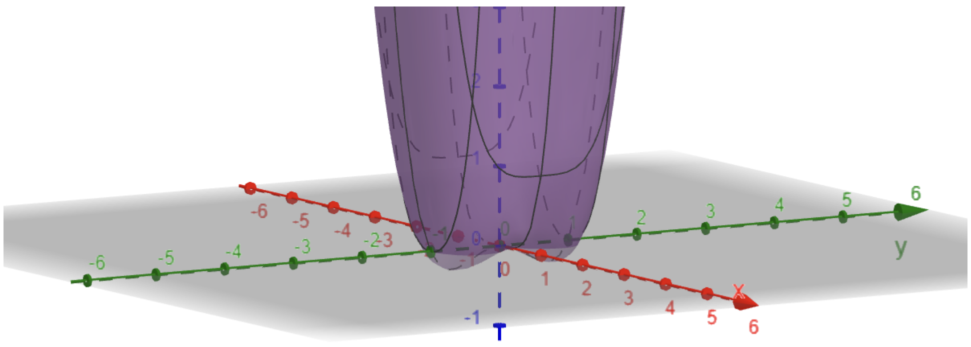


Figure 3: Function Visualisation

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

$$z^{(1)} = W^{(1)}x + b^{(1)}$$
$$a^{(1)} = \sigma(z^{(1)}) = \sigma(W^{(1)}x + b^{(1)})$$
$$z^{(2)} = W^{(2)}a^{(1)} + b^{(2)}$$
$$\hat{y} = \sigma_{\text{out}}(z^{(2)})$$

- 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:

$$a^{(l)} = \sigma(W^{(l)}a^{(l-1)} + b^{(l)}), \quad \text{for } l = 2, 3, \ldots, L$$
$$a^{(0)} = x \quad \text{(input layer)}, \quad a^{(L)} = \hat{y} \quad \text{(output)}$$

$$\mathcal{L} = \frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2$$

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

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

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

b Perceptrons

c Perceptron implementation of AND

-