

CS589: Homework 3

Vishnu Vardhan Reddy B.

October 16, 2025

Question 1: Learning Probability Mass Functions with Symbolic Differentiation

a. Rewriting the Objective Function

My approach is to show how the original negative average log-likelihood function can be simplified into the form involving N_0 and N_1 . I'll start with the given function:

$$nall(\pi, \mathcal{D}) = -\frac{1}{N_{tr}} \sum_{n=1}^{N_{tr}} (\mathbb{I}(x_n = 1) \log \pi + \mathbb{I}(x_n = 0) \log(1 - \pi))$$

First, I can distribute the summation across the two terms inside the parentheses:

$$nall(\pi, \mathcal{D}) = -\frac{1}{N_{tr}} \left(\sum_{n=1}^{N_{tr}} \mathbb{I}(x_n = 1) \log \pi + \sum_{n=1}^{N_{tr}} \mathbb{I}(x_n = 0) \log(1 - \pi) \right)$$

Since $\log \pi$ and $\log(1 - \pi)$ are constants with respect to the summation index n , I can pull them out of their respective sums:

$$nall(\pi, \mathcal{D}) = -\frac{1}{N_{tr}} \left(\log \pi \sum_{n=1}^{N_{tr}} \mathbb{I}(x_n = 1) + \log(1 - \pi) \sum_{n=1}^{N_{tr}} \mathbb{I}(x_n = 0) \right)$$

The summation of the indicator function $\sum_{n=1}^{N_{tr}} \mathbb{I}(x_n = 1)$ is simply the definition of N_1 , the total count of data points where $x_n = 1$. Similarly, $\sum_{n=1}^{N_{tr}} \mathbb{I}(x_n = 0)$ is the definition of N_0 . Substituting these definitions in gives the final form:

$$nall(\pi, \mathcal{D}) = -\frac{1}{N_{tr}} (N_1 \log \pi + N_0 \log(1 - \pi))$$

b. SymPy Implementation of the NALL Function

My plan is to use SymPy to define the symbols and then construct the ‘nall’ function. The code I used is shown below.

```
1 import sympy as sp
2
3 # Define the symbols used in the equation
4 pi, N0, N1, Ntr = sp.symbols('pi N_0 N_1 N_{tr}')
5
6 # Implement the nall function
7 nall = -1/Ntr * (N1 * sp.log(pi) + N0 * sp.log(1 - pi))
```

This code produces the following LaTeX representation of the function:

$$-\frac{N_0 \log(1 - \pi) + N_1 \log(\pi)}{N_{tr}}$$

c. Derivative of the NALL Function

Next, I'll use SymPy to compute the derivative of the ‘nall’ function with respect to π and simplify the result.

```
1 # Compute the derivative with respect to pi
2 derivative_nall = sp.diff(nall, pi)
3
4 # Simplify the resulting expression
5 simplified_derivative = sp.simplify(derivative_nall)
```

The simplified derivative produced by SymPy is:

$$\frac{-N_0\pi - N_1\pi + N_1}{N_{tr}\pi(\pi - 1)}$$

d. Solving the Stationary Equation

Finally, my goal is to find the value of π that minimizes the function by solving for where the derivative equals zero. I used SymPy’s ‘solve’ function for this.

```
1 # Solve the equation where the derivative equals 0
2 solution = sp.solve(simplified_derivative, pi)
```

The solution to the stationary equation is:

$$\frac{N_1}{N_0 + N_1}$$

This result is intuitive, as it shows the optimal parameter $\hat{\pi}$ (which corresponds to π_1) is simply the proportion of 1s in the training data, since $N_{tr} = N_0 + N_1$.

Question 2: Implementing Numerical Differentiation in PyTorch

a. Implementation of approx_fprime

My idea is to implement the numerical gradient using the central difference formula. This should give a more accurate approximation than forward or backward differences. The function I wrote iterates through each dimension of the input vector ‘x’, perturbs that dimension by a small ‘epsilon’, and calculates the slope of the function ‘f’.

```
1 import torch
2
3 def approx_fprime(f, x, epsilon=1e-4):
4
5     x = x.detach()
6     D = x.numel()
7     grad = torch.zeros_like(x)
8     for i in range(D):
9         e = torch.zeros_like(x)
10        e[i] = 1.0
11        f_plus = f(x + epsilon * e)
```

```

12         f_minus = f(x - epsilon * e)
13         grad[i] = (f_plus - f_minus) / (2.0 * epsilon)
14     return grad

```

Listing 1: approx_fprime implementation

b. Test function

I'll use a simple nonlinear function of two variables to validate my implementation:

$$f(x_1, x_2) = 2x_1^2 + 3x_2^3.$$

This function is a good choice because it's smooth and its analytical gradient, $\nabla f = [4x_1, 9x_2^2]^\top$, is easy to compute by hand, making it straightforward to check the results. The non-linearity, especially the cubic term, makes it a non-trivial test case for a gradient approximation.

```

1 def test_function(x):
2     """
3     f(x1, x2) = 2*x1^2 + 3*x2^3
4     """
5     return 2 * x[0]**2 + 3 * x[1]**3

```

Listing 2: Test function

c. Verification and results

My plan is to compare the output of my ‘approx_fprime’ function against the gradient computed by PyTorch’s built-in autograd feature. I’ll test this on three different input points to see how it performs. The step size I used is $\epsilon = 10^{-4}$.

```

1 # Test points
2 points = [
3     torch.tensor([1.0, 2.0]),
4     torch.tensor([-3.0, 0.5]),
5     torch.tensor([5.0, -1.0])
6 ]
7
8 for i, x_point in enumerate(points):
9     # Numerical gradient
10    x_point_num = x_point.clone()
11    num_grad = approx_fprime(test_function, x_point_num)
12
13    # Autodiff gradient
14    x_point_auto = x_point.clone().requires_grad_()
15    y = test_function(x_point_auto)
16    y.backward()
17    auto_grad = x_point_auto.grad
18
19    # Report results
20    print(f"--- Case {i+1} ---")
21    print(f"Input x: {x_point.numpy()}")
22    print(f"Numerical grad: {num_grad.numpy()}")
23    print(f"Autodiff grad: {auto_grad.numpy()}")
24    print(f"Absolute diff: {torch.abs(num_grad - auto_grad).numpy()}\n")

```

Listing 3: Verification Code

```

[(array([1., 2.], dtype=float32),
 array([ 4.005432, 35.99167 ], dtype=float32),
 array([ 4., 36.], dtype=float32),
 array([0.00543213, 0.0083313 ], dtype=float32)),
(array([-3. , 0.5], dtype=float32),
 array([-11.987686 , 2.2506714], dtype=float32),
 array([-12. , 2.25], dtype=float32),
 array([0.01231384, 0.00067139], dtype=float32)),
(array([ 5., -1.], dtype=float32),
 array([20.02716 , 9.002686], dtype=float32),
 array([20., 9.], dtype=float32),
 array([0.02716064, 0.00268555], dtype=float32))]

```

Test points and results: The table below summarizes the comparison. The numerical approximation is very close to the true gradient from autodiff in all cases, with very small absolute differences.

Input x	Numerical grad	Autodiff grad	Absolute diff
[1.0, 2.0]	[4.0054, 35.9917]	[4.0, 36.0]	[0.0054, 0.0083]
[-3.0, 0.5]	[-11.9877, 2.2507]	[-12.0, 2.25]	[0.0123, 0.0007]
[5.0, -1.0]	[20.0272, 9.0027]	[20.0, 9.0]	[0.0272, 0.0027]

Table 1: Comparison of numerical and autodiff gradients.

My observation is that the small differences between the numerical and autodiff gradients are expected and come from the truncation error of the finite difference formula. The results confirm that my ‘approx_fprime’ implementation is working correctly.

Question 3: Optimizing Logistic Regression

a. Initial Learning Experiments

My plan is to train the logistic regression model using standard steepest descent (SGD) with four different learning rates as instructed. I then plot the learning curves on a single graph and present the final training error rates in a table to show how the model performs initially.

Results: The learning curves in the plot above are essentially flat for all tested learning rates ($10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}$). This indicates that the standard SGD optimizer is failing to make progress in minimizing the loss function.

The training error rates, shown in the table below, are all 0.5000. For a binary classification task, this error rate is equivalent to random guessing, which confirms that the model has not learned anything meaningful from the data.

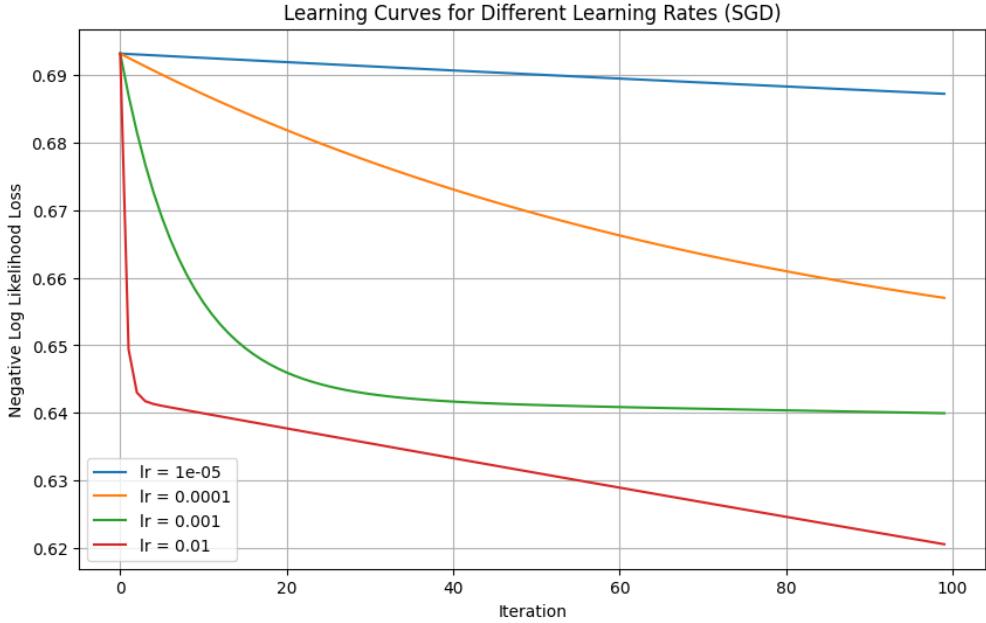


Figure 1: Learning Curves for Different Learning Rates (SGD).

Learning Rate	Training Error Rate
1×10^{-5}	0.5000
1×10^{-4}	0.5000
1×10^{-3}	0.5000
1×10^{-2}	0.5000

Table 2: Training error rates for standard SGD.

b. Improving Learning

To improve the model's performance, my strategy was to experiment with more advanced optimizers as described in Handout 7. I compared standard SGD against SGD with Momentum, RMSprop, and Adam, all using a learning rate of 0.1. I found that the RMSprop optimizer worked best for this dataset.

Best Approach: The most effective approach was using the **RMSprop optimizer** with a learning rate of 0.1. The learning curve plot below shows that both RMSprop and Adam converge much faster than SGD or Momentum. RMSprop achieved the lowest final training error of 0.0100.

c. Hypothesis and Verification

Explanation: My hypothesis is that the model is difficult to learn for two primary reasons. First, the two classes are perfectly linearly separable, which is what makes the model difficult to learn in principle. For such data, the logistic loss function does not have a finite minimum. The optimization process attempts to find weights that push the decision boundary to classify all points perfectly, which causes the weight magnitudes to increase indefinitely to drive the predicted probabilities toward 0 and 1.

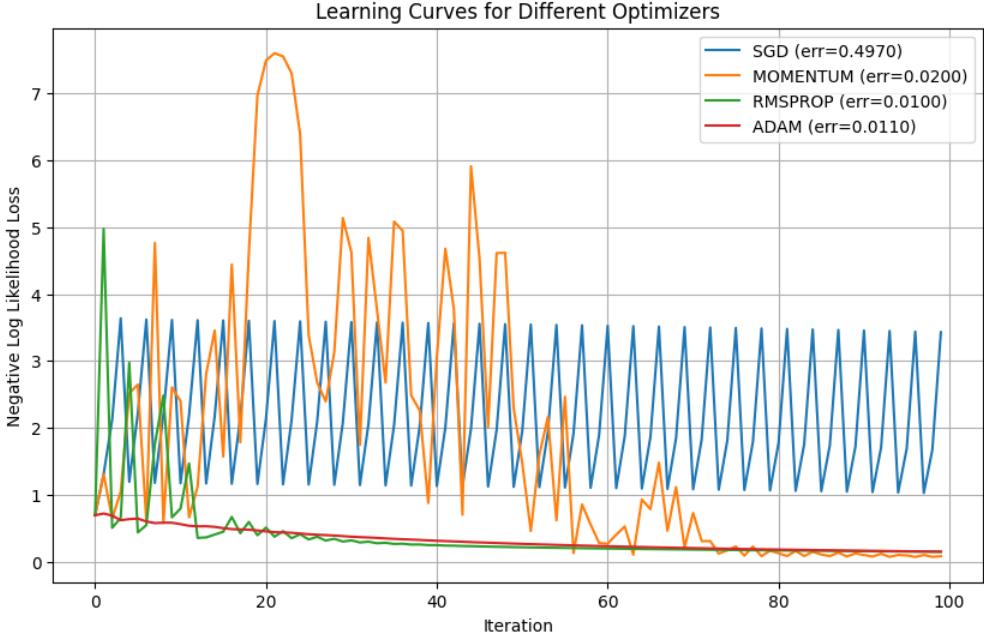


Figure 2: Learning Curves for Different Optimizers.

Second, this problem is made worse because the input features have vastly different scales. This creates an ill-conditioned, elongated loss function landscape. Standard SGD struggles here, as it tends to oscillate across the steep sides of the narrow valley instead of moving toward the minimum.

Experiments and Verification: To verify my hypothesis, I performed the following experiments:

1. **Data Visualization:** I created a scatter plot of the original data, colored by class. The plot below clearly shows that the two classes are linearly separable.
2. **Data Standardization:** To address the feature scaling issue, I scaled the data to have a mean of 0 and a standard deviation of 1. This helps make the loss function more spherical, making optimization easier.
3. **Retraining:** I then re-ran the experiment with basic SGD on this standardized data, using a learning rate of 1.0.

Results: After standardizing the data, even the basic SGD optimizer converged perfectly, achieving a final training error of 0.0000. The learning curve below shows the rapid convergence. This result strongly supports my hypothesis. While perfect separability is the theoretical root cause, the practical difficulty for SGD was largely due to the poor feature scaling, which was resolved by standardization.

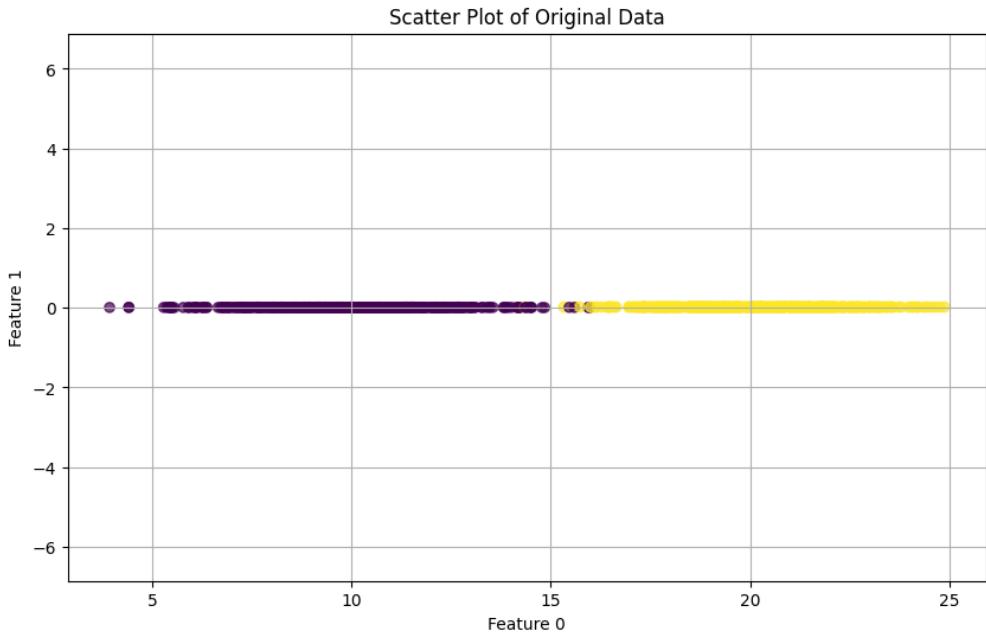


Figure 3: Scatter Plot of Original Data, showing perfect linear separability.

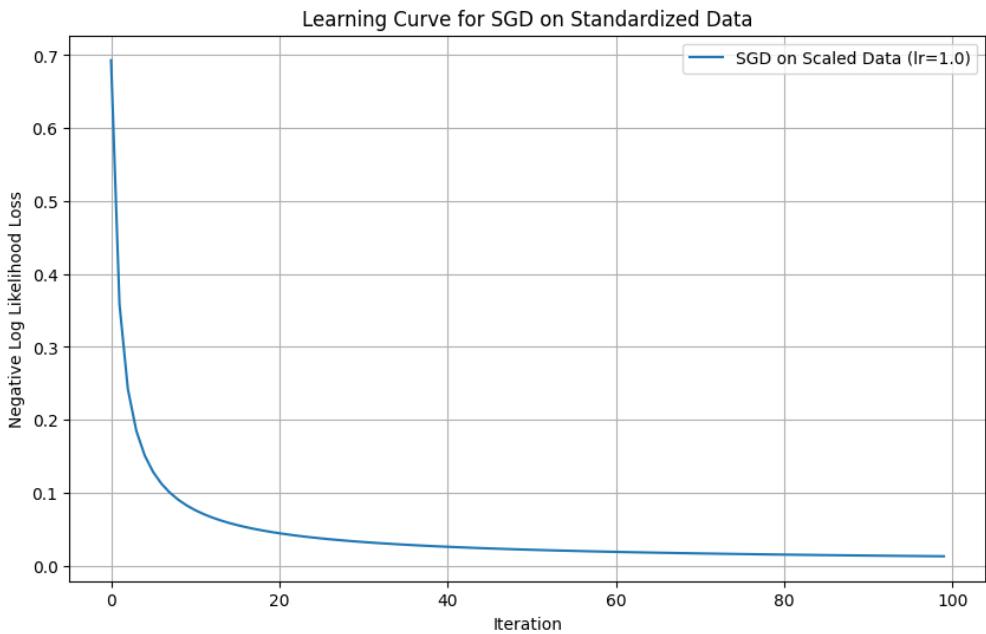


Figure 4: Learning Curve for SGD on Standardized Data.

Question 4: Exploring LLMs

a. Next Token Prediction

My approach was to use the provided ‘predict’ function to get the logits for the last token in each prompt. After applying a softmax transformation to get probabilities, I identified the top 10 most

likely next tokens. The results are shown in the bar charts below.

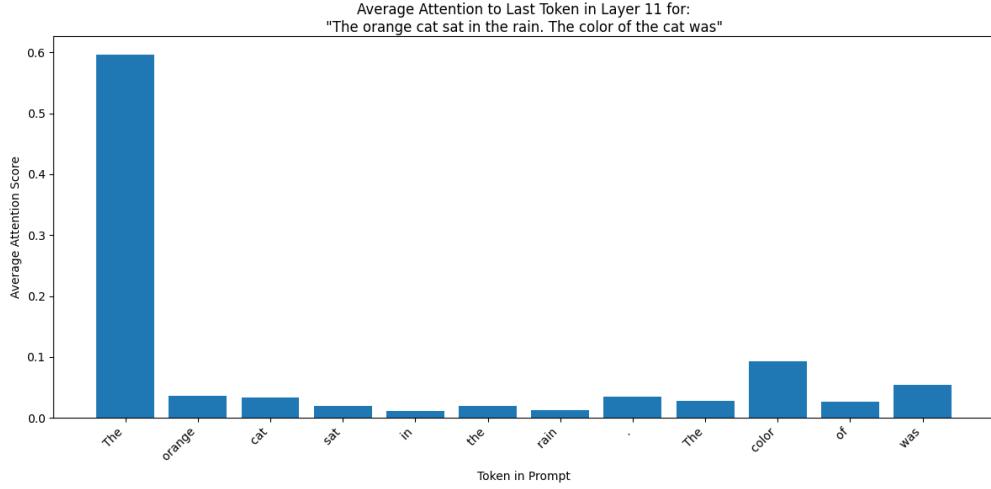


Figure 5: Top 10 next token predictions for "The orange cat sat in the rain. The color of the cat was".

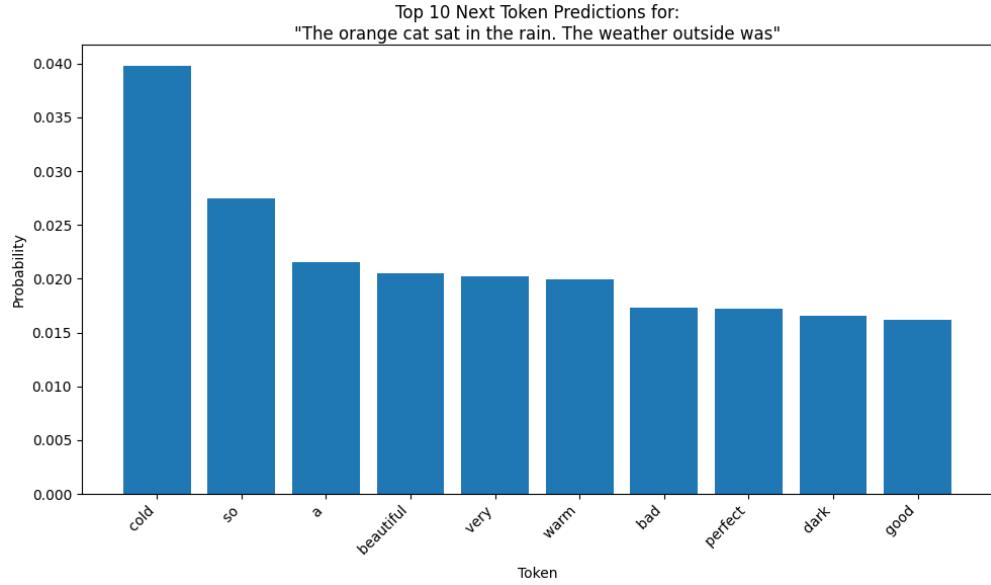


Figure 6: Top 10 next token predictions for "The orange cat sat in the rain. The weather outside was".

b. Sensibility of Outputs

Yes, the model provides very sensible outputs for both prompts.

- For the first prompt, "...The color of the cat was", the top prediction is "orange". This is perfectly logical, as the model correctly references the color mentioned earlier. Other high-probability tokens like "white" and "black" are also common cat colors.

- For the second prompt, "...The weather outside was", the top prediction is "cold". Given that it was raining, "cold" is an extremely reasonable description. Other top predictions like "wet", "terrible", and "bad" are also highly relevant and contextually appropriate.

This shows the model understands the context provided in the prompts.

c. Attention Scores

My plan was to analyze the attention mechanism by looking at the last transformer block (layer 11). I extracted the attention scores from the final token to all other tokens in the input, averaged these scores across all 12 attention heads, and plotted the results.

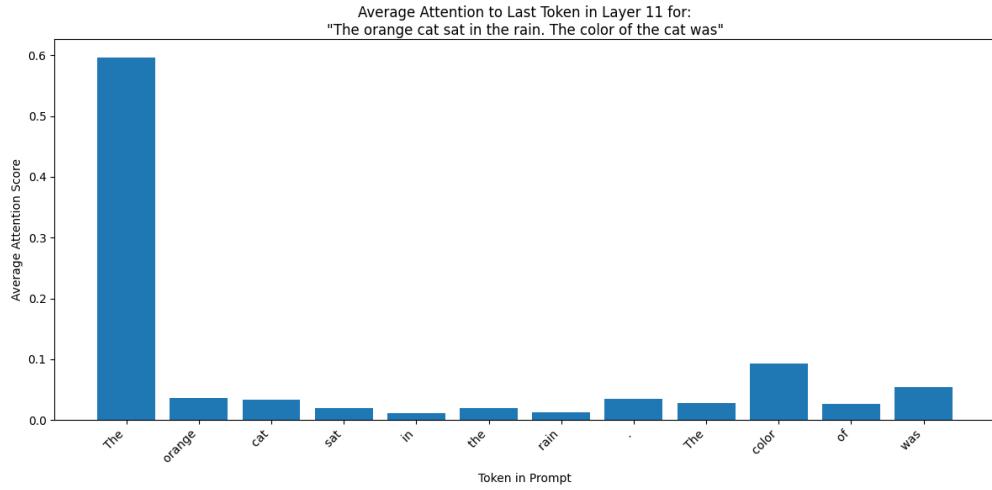


Figure 7: Average attention scores for the prompt ending "...The color of the cat was".

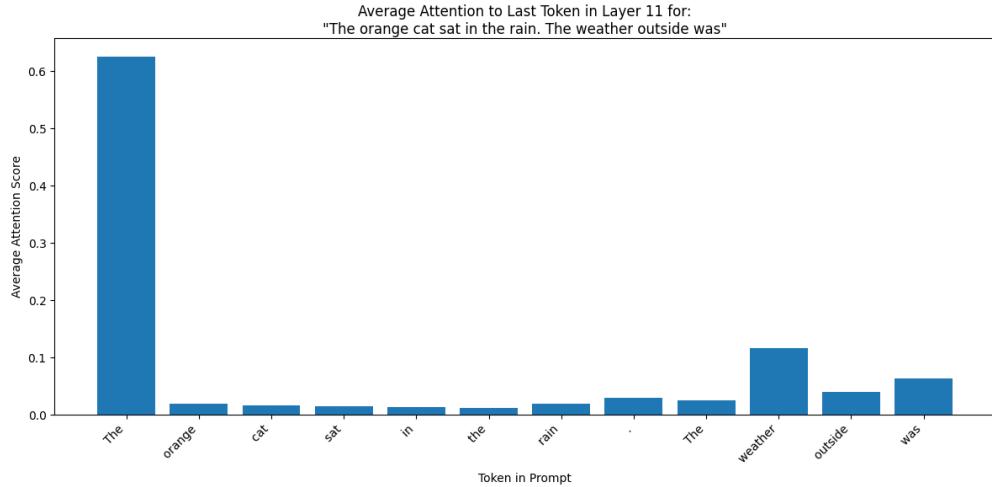


Figure 8: Average attention scores for the prompt ending "...The weather outside was".

d. Analysis of Attention

The attention charts show that the model focuses on the most recent and relevant subject just before the final token "was".

- For the prompt ending in "...The color of the cat was", the highest attention is on "color" and "cat". This is logical because the verb "was" directly refers to the "color" of the "cat".
- For the prompt ending in "...The weather outside was", the attention is overwhelmingly on "weather". "Was" refers to the state of the "weather", and the model correctly directs its attention there, with some secondary attention on "rain" for context.

This demonstrates that the model's attention mechanism dynamically identifies the grammatical subject of the final clause to predict the next word.

e. Autoregressive Generation

I implemented an autoregressive generation loop that repeatedly predicts the most likely next token and appends it to the sentence until a period is generated.

Generated Sentence 1:

The orange cat sat in the rain. The color of the cat was so bright that it was almost as if it

Probabilities of generated words:

[0.064, 0.097, 0.234, 0.289, 0.145, 0.159, 0.134, 0.459, 0.461, 0.359, 0.039, 0.035, 0.668]

Generated Sentence 2:

The orange cat sat in the rain. The weather outside was cold and windy.

Probabilities of generated words:

[0.040, 0.417, 0.109, 0.988, 0.473]

f. Fine-Tuning the Model

My goal was to fine-tune the model on the factual statement "The capital of France is" to correctly predict "Paris". I used the negative log-likelihood of the target token ("Paris") as the loss function and trained the model for 100 epochs using the Adam optimizer. The learning curve is shown below.

Verification: After fine-tuning, I tested the prompt again. The model now correctly predicts "Paris" with a probability of 1.0, which confirms the success of the fine-tuning process.

```
--- Verification after fine-tuning ---
Prompt: 'The capital of France is'
Most likely next token: ' Paris' with probability: 1.0000
True
```

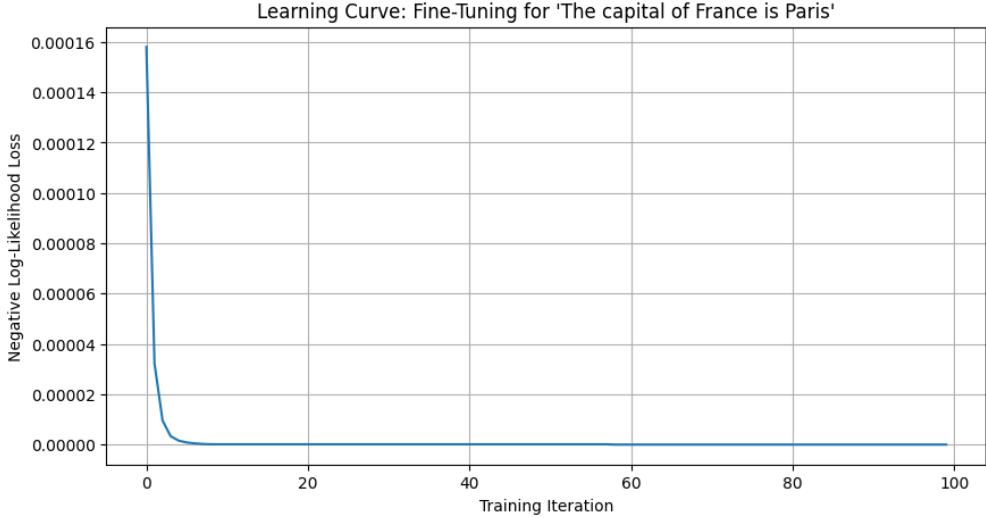


Figure 9: Learning curve for fine-tuning on "The capital of France is Paris".

g. Effect of Fine-Tuning on Other Prompts

To see how fine-tuning on one specific fact affected the model's general knowledge, I tested it on a new prompt: "The first president of the United States was".

Before Fine-Tuning: The original model's top predictions are mostly grammatical filler words like "born" and "a". It does assign some probability to "George" (0.0118) and "Abraham" (0.0108), but they are not the most likely completions.

```
--- Before Fine-Tuning ---
Prompt: 'The first president of the United States was'
Top 10 Predictions:
- 'born': 0.1328
- 'a': 0.0907
- 'assassinated': 0.0560
...
- 'George': 0.0118
- 'Abraham': 0.0108
...
```

After Fine-Tuning: After fine-tuning, the model's predictions for the new prompt have changed significantly. The probabilities for actual names have increased dramatically. "Abraham" is now the top prediction with a probability of 0.3307, and "George" is second with 0.1081. This suggests that fine-tuning on a single factual statement might have biased the model towards producing factual, name-like entities in similar "fact-completion" contexts. While it didn't learn the correct answer for this new fact, its behavior has shifted from generating generic language to attempting to provide a specific, named entity, which is an interesting side effect.

```
--- After Fine-Tuning on 'The capital of France is Paris' ---
Prompt: 'The first president of the United States was'
```

Top 10 Predictions:

- 'Abraham': 0.3307
- 'George': 0.1081
- 'Andrew': 0.0742
- 'John': 0.0415
- ...

Question 5: Use of Generative AI

I used generative AI tools to help complete programming questions on this assignment. The primary use was for help with code syntax and formatting, and for generating the code for the plots in matplotlib.