

Assignment 1

Student: INNOCENT KISOKA
Student's email: innocent.kisoka@usi.ch

Polynomial regression

1 Question 2:

I defined a function `plot_polynomial` that takes the polynomial coefficients, a range of values for z , and the desired plot color.

```
9 # Function to plot a polynomial
10 def plot_polynomial(coeffs, z_range, color='b'):
11     z_vals = np.linspace(z_range[0], z_range[1], 500) # Create 500 points in the given range
12     poly_vals = np.polyval(coeffs, z_vals) # Evaluate polynomial at these points
13     plt.plot(z_vals, poly_vals, color=color)
14     plt.xlabel('z')
15     plt.ylabel('p(z)')
16     plt.grid(True)
17     plt.savefig('my_plot1.png')
18     plt.show()
19
20 # Define coefficients and plot the polynomial
21 coeffs = np.array([1, -2, 3, -4, 1, 1/30])
22 plot_polynomial(coeffs, [-4, 4], color='b')
```

Figure 1: Polynomial Code

I plotted the polynomial as shown in Figure 13. The plot shows the polynomial $p(z)$ over the interval $[-4, 4]$. It captures the shape of the polynomial that can be explicitly seen on the graph provided here.

2 Question 3:

This function generates noisy polynomial data based on the provided coefficients. It also incorporates random noise to simulate real-world data with variability.

```
24 # Function to create a dataset with noise
25 def create_dataset(coeffs, z_range, sample_size, sigma, seed=42):
26     np.random.seed(seed)
27     z = np.random.uniform(z_range[0], z_range[1], sample_size) # Generate random z values
28     noise = np.random.normal(0, sigma, sample_size) # Generate Gaussian noise
29     y = sum(coeffs[i] * z**i for i in range(len(coeffs))) + noise # Create y values with noise
30     X = np.array([z**i for i in range(len(coeffs))]).T # Create feature matrix
31     return torch.tensor(X, dtype=torch.float32), torch.tensor(y, dtype=torch.float32)
```

Figure 2: create dataset code

3 Question 4:

- This function is parameterized with the same polynomial coefficients and range of z values but utilizes different random seeds to ensure the datasets are independent and not duplicates.

```

23 # Create training and validation datasets
24 X_train, y_train = create_dataset(coefficients, -1, 5, -0.1, 1/50, x_range=-2, 2, sample_size=500, sigma=4.5, seed=0)
25 X_val, y_val = create_dataset(coefficients, -1, 5, -0.1, 1/50, x_range=-2, 2, sample_size=50, sigma=4.5, seed=1)
26
27
28

```

Figure 3: Train and evaluation code

4 Question 5

Here I defined a function to visualize the polynomial and the dataset

```

10 # Function to visualize the data and true polynomial
11 def visualize_data(x, y, coeffs, x_range, title):
12     x = np.linspace(x_range[0], x_range[1], 100)
13     p_z = sum([coeffs[i] * x**i for i in range(len(coeffs))])
14     plt.plot(x, p_z, label='True Polynomial', color='r')
15     plt.scatter(x, y, label='Noisy Data')
16     plt.title(title)
17     plt.xlabel('x')
18     plt.ylabel('p(x)')
19     plt.legend()
20     plt.grid(True)
21     plt.savefig('my_plot2.png')
22     plt.show()
23
24 visualize_data(X_train, y_train, coeffs, x_range=-2, 2, title='Training Data Visualization')

```

Figure 4: Visualization code

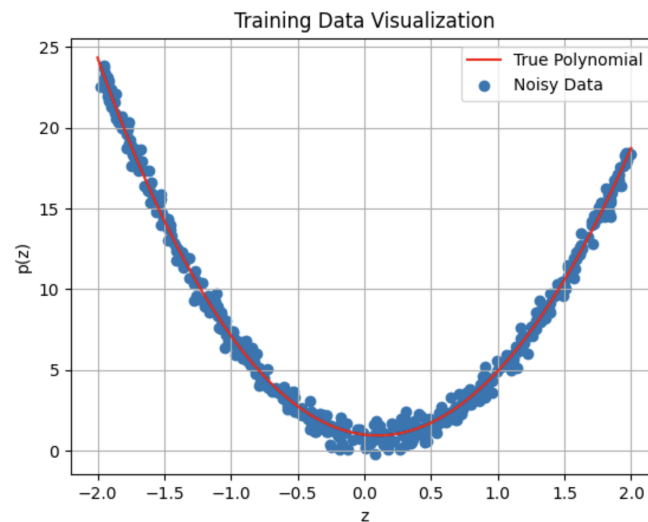


Figure 5: Training data visualization

5 Question 6

I implemented gradient descent using PyTorch's `torch.nn.Linear` to estimate the polynomial coefficients. The most important points are:

- **Learning rate:** I started with a very moderate value of 0.01. This is because if the learning rate is too small, the convergence will be slow and if it's too large, the optimization process will diverge.
- **Bias in `torch.nn.Linear`:** I set `bias=False` since the model already has an intercept term in the dataset (the 1 component in `x`). This constant

"1" is a part of the input, allowing the model to learn a constant shift during training.

- **Number of steps:** I choose 500 epochs which I think it's a reasonable number of steps, depending on trial and error. As I expect that the model would have learned enough and produce more refined results after this amount of iterations.

6 Question 7

`PolynomialRegressionModel` has a single linear layer with $(\text{degree} + 1)$ inputs to account for the polynomial terms (for degree 4, it will have 5 inputs). Here is the plot of training and validation loss as functions of iterations.

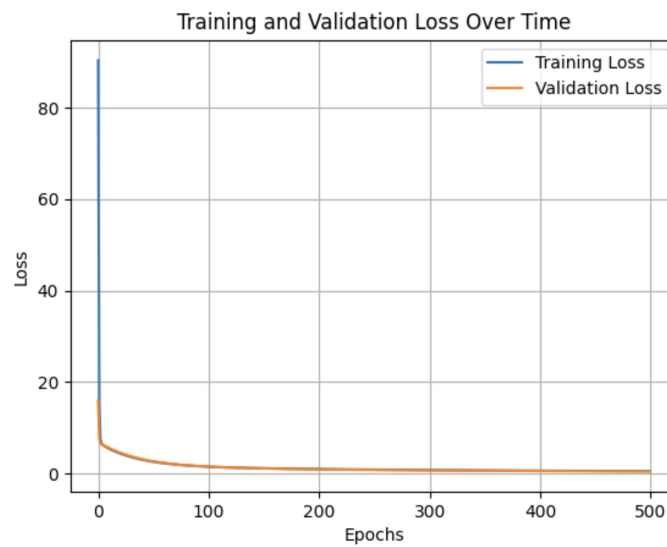


Figure 6: Training and Validation plot

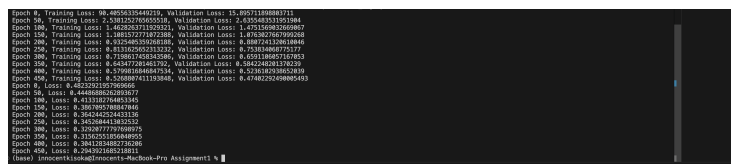


Figure 7: Training and validation results

7 Question 8

A Plot of the Estimated and True Polynomial

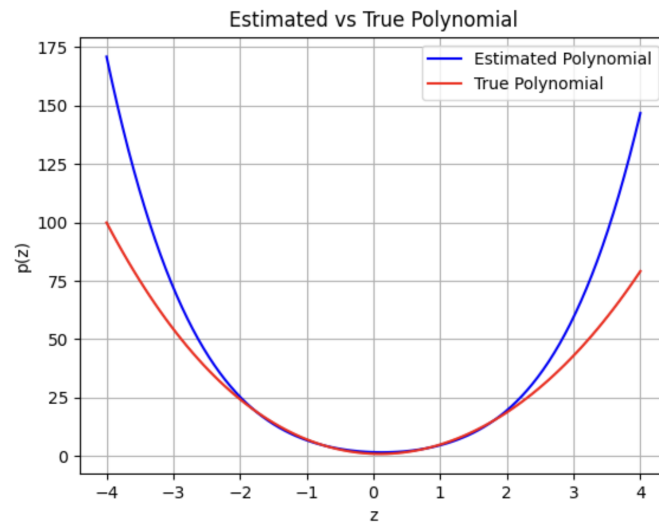


Figure 8: Comparison of estimated polynomial (blue) and true polynomial (red)

8 Question 9

The parameter values are tracked during training and compared to their true values as shown in Figure 9

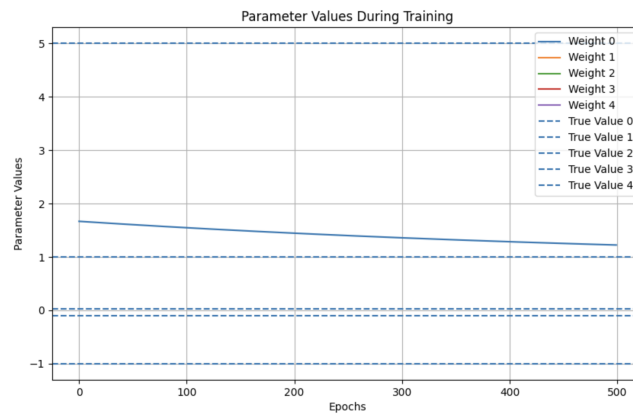


Figure 9: Parameter values during training with true values indicated by dashed lines

9 Question (5 pts)

The main difference between manual learning rate selection and adaptive methods like AdaGrad is in their approach to adjusting learning rates during training[1]:

Manual selection:

- Uses a global or layer-specific learning rate
- Requires expertise and tuning
- Static approach, not adaptive to data characteristics

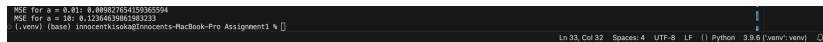
Adaptive methods (like AdaGrad):

- Automatically adjust learning rates for each parameter
- Use historical gradient information
- Adapt to feature frequency and gradient magnitude
- Offer advantages like automatic tuning, improved performance on sparse data, and less sensitivity to initial choices

AdaGrad can suffer from diminishing learning rates over time, it paved the way for more advanced adaptive methods like RMSprop and Adam.[2]

10 Bonus question* (5pts)

Results: After executing the code I obtained the following results :



```
MSE for a = 0.01: 0.00952938137851114
MSE for a = 10: 0.1272485031272189
(.venv) (base) innocents@innocents-MacBook-Pro Assignment1 %
```

Figure 10: MSE results for $a=0.01$ and $a=10$

Case 1 ($a = 0.01$):

- MSE = 0.00952938137851114
- Linear regression fits well
- Reason: In $[-0.05, 0.01]$, the logarithmic function is nearly linear
- Performance: Good

Case 2 ($a = 10$):

- MSE = 0.1272485031272189
- Linear regression fits poorly
- Reason: In $[-0.05, 10]$, the logarithmic function's curvature is pronounced
- Performance: Poor

Therefore, Linear regression performs well when the function is approximately linear in the given domain. Linear regression performs poorly over larger intervals where logarithmic functions deviate significantly from linearity.

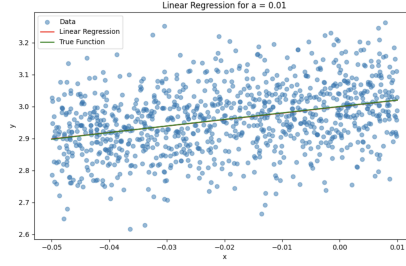


Figure 11: Linear Regression for a = 0.01

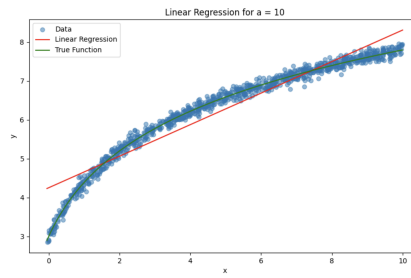


Figure 12: Linear Regression for a = 10

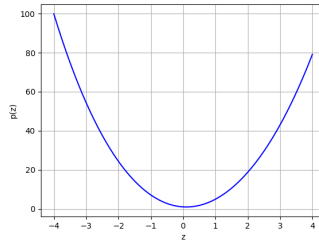


Figure 13: Plot of the polynomial $p(z)$ over the range $[-4, 4]$.

References

- [1] Jun Lu. Adasmoother: An adaptive learning rate method based on effective ratio. In *Sentiment Analysis and Deep Learning: Proceedings of ICSADL 2022*, pages 273–293. Springer, 2023.
- [2] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7), 2011.