Assignment 1

Student: Harkeerat Singh Sawhney
Student's email: sawhnh@usi.ch

Polynomial regression

Write answers to questions here. Please, keep the original enumeration.

1. In Question 1 it was asked to implement a function PlotPolynomial, which plots the function $p(z) = \sum_{k=1}^{4} z^k w_k$. The function took *coeffs* which is a `np.array` and contains the 5 weights w_0, w_1, w_2, w_3, w_4. In that $w_0 = 0$. The function also took a z-range which is the range for the z value. The function then plots the polynomial function $p(z)$ for the given z-range. This was done by the use of the python library `matplotlib.pyplot`. To test the function out the following variable were used.

```
coeffs = np.array([0, -10, 1, -1, 1 / 1000])
z_range = [-10, 10]
color = "b"
plot_polynomial(coeffs, z_range, color)
```

   The figure generate from the function can be seen in Figure 1.

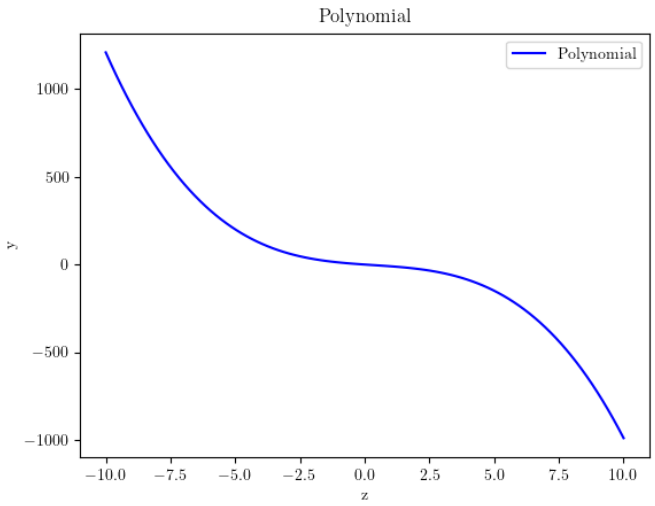


Figure 1: Polynomial function generated from the function `plot_polynomial`

2. In Question 2 we were asked to make a function `createDataset`, which is similar to the function `createDataset` from the previous assignment. The function take the following parameters, `coeffs` which is a `np.array` containing the weights w_0, w_1, w_2, w_3, w_4, `z_range` which is the range of the z values, `N` which is the number of samples to be generated, `sigma` which is the standard deviation of the noise and `seed` which is the seed for the random number generator.

   First a random state is initialized with the seed provided, then through zMin and zMax the set of sampleSize z values are generated. Lastly through the definition of the D‘ dataset the x values are computed for the power of 0 to 4. Hence at last x is a matrix of size $sampleSize \times 5$.

   After that the value of y had to be computed using the formula $y = xw + \epsilon$, where ϵ is the noise. This was done by first generating a noise vector of size $SampleSize \times 1$ and then adding it to the xw vector. The noise vector was generated using the `randn` function from the `numpy.random` library. The function at last returns the x and y values.

   The function starts from line 54.

3. In Question 3 we were given the values of the parameters for the function `createDataset` and were asked to generate the dataset. We were asked to generate 2 dataset, one for the training and one for the testing. The training dataset was generated using the following parameters.

```
1  coeffs = np.array([0, -10, 1, -1, 1 / 1000])
2  z_range = [-3, 3]
3  sigma = 0.5
4  sample_size_train = 500
5  sample_size_eval = 500
6  seed_train = 0
7  seed_eval = 1
8
9  # Creating the dataset
10 X_train, y_train = create_dataset(coeffs,
       z_range, sample_size_train, sigma, seed_train)
11 X_eval, y_eval = create_dataset(coeffs, z_range,
       sample_size_eval, sigma,seed_eval)
```

4. In Question 4 we were asked to implement the function `visualizeDataset`. The function takes the x and y value which the function `createDataset` returns. The function then plots the x and y values using the `matplotlib.pyplot` library. The function also plots the polynomial function $p(z)$ using the `plot_polynomial` function.

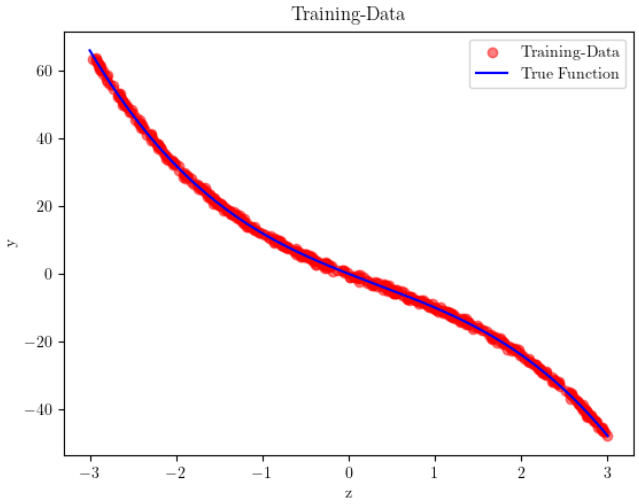


Figure 2: Training Data vs True Polynomial

Figure 2 shows the plot of the training data and the true polynomial function. As it can be seen from the plot that the data is not exactly on the True Polynomial function, because of the noise which is added to it.

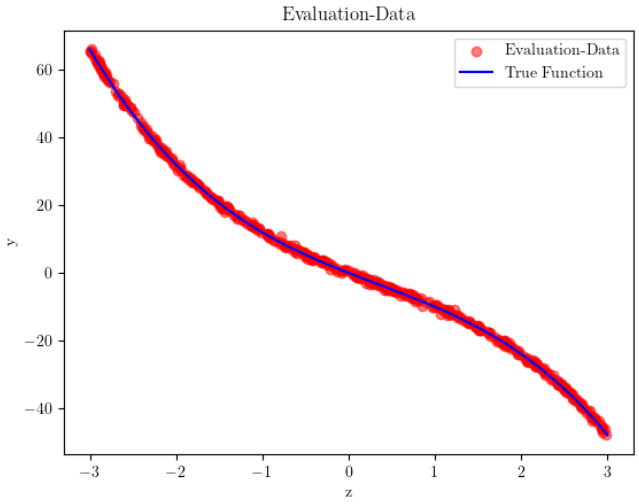


Figure 3: Evaluation Data vs True Polynomial

Figure 3 shows the plot of the evaluation data and the true polynomial function. Again as it can be seen from the plot that the data is not exactly on the True Polynomial function, because of the noise which is added to it.

5. In this Question we were asked to perform Polynomial Regression on the dataset D. To achieve this there were in total 3 steps. First we had to preprocess the data, then we had to train the model and lastly we had to evaluate the model.

   In order to preprocess the data we first need to reshape both training and evaluation data into the input and output dimensions. For the `X_train` and `X_eval` we need to reshape them into $sampleSize \times 5$ and for the `y_train` and `y_eval` we need to reshape them into $sampleSize \times 1$. The reason why for X we need to reshape it into $sampleSize \times 5$ is because we have 5 features (which are the weights) and for y we need to reshape it into $sampleSize \times 1$ because we have only 1 output. After that the data is converted into PyTorch tensors of type float32, and they were moved on the GPU for faster computation. With this data is ready to be trained on.

   The second step is to prepare the model itself, and for this we created a class `PolynomialRegression`. The class inherits from the `nn.Module` class, since we want to make a basic neural network module. The neural network is very simple as it has only one layer which takes 5 inputs and gives out 1 input. In the one fully connect layer it performs Linear Regression on the data.

   At last third step is to put train the model with the processed data. For this we first initialize the model and then we define the loss function and the optimizer. The loss function is the Mean Squared Error (MSE) and the optimizer is the Stochastic Gradient Descent (SGD). The model is then trained for 5000 epochs and the loss is printed after every 100 epochs. The learning rate which was used in this case is 0.001. In the for loop the model is trained on the training data, and then is evaluation on the evaluation data. The loss is calculated for both the training and the evaluation data.

   Some observations which I had noticed was with regarding the learning rate. If the learning rate is very high, then the loss would go to infinity and would not come down, but in the other hand if the learning rate is very low, it would never get towards the minimum loss. Hence the learning rate has to be chosen very carefully. The learning rate which I had chosen was 0.001, and it gave me the best results. Also another thing which I noticed was that initially in my neural network I had multiple layers and many neurons. This was mainly due to my misunderstanding of the question, but what I had observed was that the learning rate had to be much smaller due to the increase in the size of the neural network.

   Bias term in `torch.nn.linear` is the first weight, and since the first weight in our case is always 0 we do not need to add the bias term. Hence the bias term in `torch.nn.linear` can be false.

   The code for this question starts from line 98.

6. In this Question we were asked to plot the training and evaluation loss as functions of the epoch number. The plot can be seen in Figure 4.

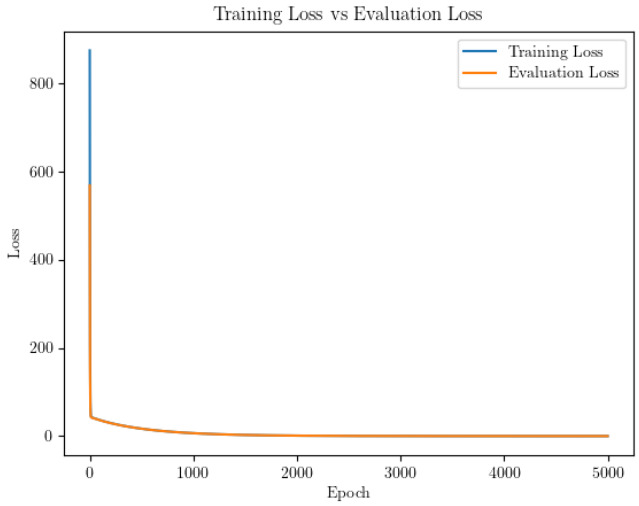


Figure 4: Training and Evaluation Loss vs Epochs

As it can be seen from the Figure 4 that the training loss and evaluation loss is decreasing at the same rate and they are very close to each other. This means that the model is not overfitting and is generalizing well. However in the beginning the Training Loss is very high as compare to the Evaluation Loss, but in couple of epochs it comes down and then both the losses are very close to each other.

7. In this Question we were asked to plot the polynomial from the true coefficients and the polynomial from the learned coefficients.

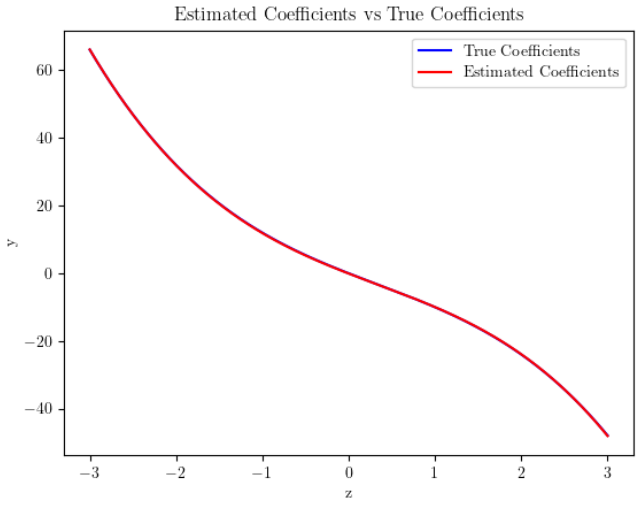
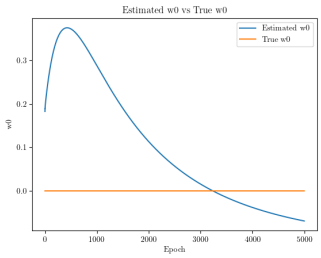


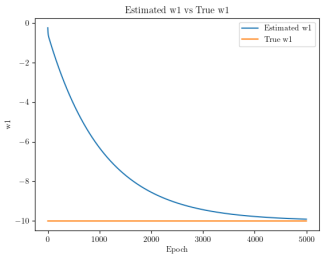
Figure 5: Estimated Coefficients vs True Coefficients

In the Figure 5 we can see the plot of the estimated coefficients vs the true coefficients. As it can be seen from the plot that the estimated coefficients are very close to the true coefficients. This means that the model has learned the coefficients very well.

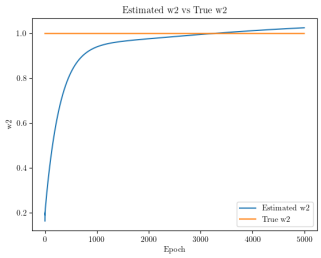
8. Bellow are the 5 Figures showing the true weights vs the estimated weights per epoch.



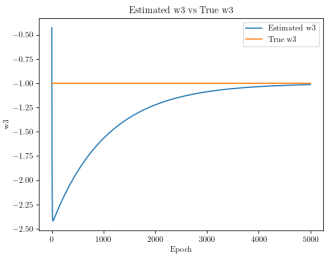
(a) Estimated w_0 vs True w_0



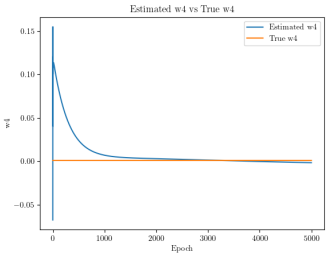
(b) Estimated w_1 vs True w_1



(c) Estimated w_2 vs True w_2



(d) Estimated w_3 vs True w_3



(e) Estimated w_4 vs True w_4

Figure 6: Comparison of Estimated and True w values

9. As it can be seen from the Figure 6 that the estimated weights are very close to the true weights. This means the model has been trained very well and has learned the weights very well.

```python
coeffs = np.array([0, -10, 1, -1, 1 / 1000])
z_range = [-3, 3]
sigma = 0.5
sample_size_train = 10
sample_size_eval = 500
seed_train = 0
seed_eval = 1
# Creating the dataset
X_train, y_train = create_dataset(
coeffs, z_range, sample_size_train, sigma,
    seed_train
)
X_eval, y_eval = create_dataset(coeffs, z_range,
    sample_size_eval, sigma, seed_eval)
```

As it can be seen the main difference is the sample size of the training data. In the previous case the sample size was 500, but in this case the sample size is 10.

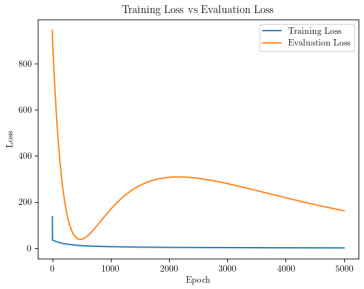


Figure 7: Training and Evaluation Loss vs Epochs

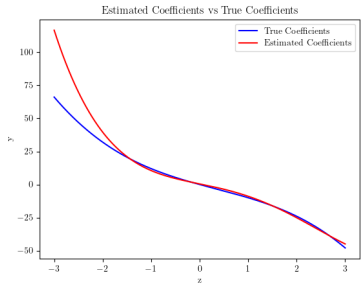


Figure 8: Estimated Coefficients vs True Coefficients

Figure 9: Comparison of Loss and Coefficients

From the Figure 9 we can see that the loss is very high and the estimated coefficients are very far from the true coefficients. This means that the model has not been trained well and has not learned the coefficients well. This is mainly due to the fact that the trained data is very less and because of the size difference it is not able to learn the coefficients well.

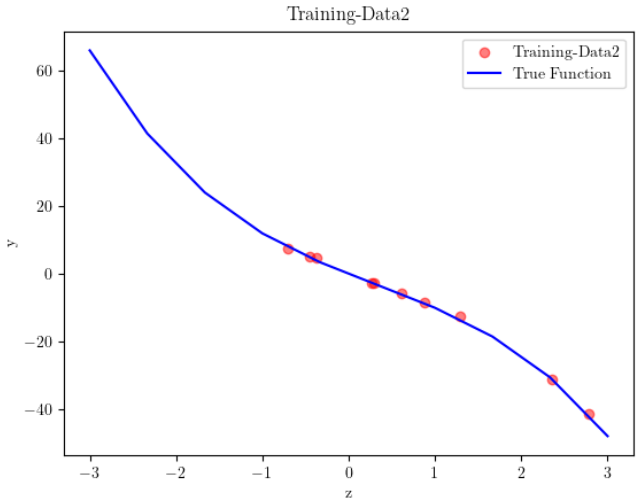
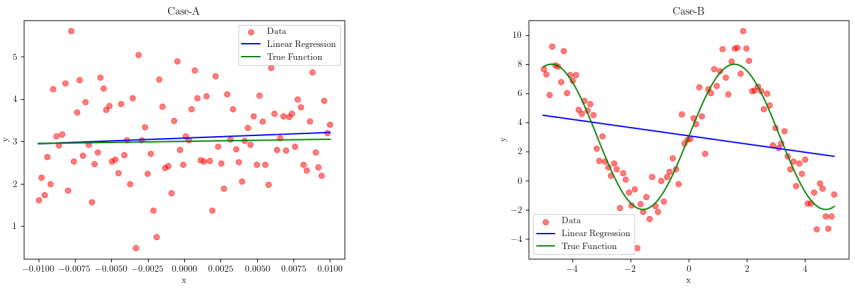


Figure 10: Training Data vs True Polynomial

As it can be seen from the Figure 10 that the due to small sample size of the training data, it does not cover the whole polynomial function. This is the reason why the model is not able to learn the coefficients well.

10. The value for a which should be selected is option A being the value 0.01. This is because if we select the option B in which the value of a is 5, then the function draws a polynomial line which is very tough to be learned with linear regression. In the other hand with option A since the range is very small, the polynomial line becomes linear and hence it is very easy to be learned with linear regression.



(a) Case A (b) Case B

Figure 11: Comparison of Case A and Case B

As it can be seen from the Figure 11 that the Case A is very easy to be learned with linear regression, but in the other hand Case B is very hard to be learned with linear regression. This is because in Case A the polynomial line is linear, but in Case B the polynomial line is not linear and hence it is very hard to be learned with linear regression.

Questions

(a) No we do not need to chose different values of alpha. In Linear Regression it is common to use the same learning rate for the parameters. Usually you would want to use different learning rates if if you want to find a complex Global Minimum, but in this case of our Linear Regression we have only 2 parameters which does not require us to use different learning rates.

(b) The main principle behind choosing two different learning rates is to find a complex Global Minimum. In such cases there tends to be many local minimums which with a single learning rate is very is to converge to. Hence in such cases we should use different learning rates to find the complex Global Minimum. AdaGrad is such algorithm which follows the same principle, in which it uses different learning rates for different parameters. The learning rate for each parameter is inversely proportional to the sum of the squares of the gradients. Hence if the gradient is very large then the learning rate will be very small and if the gradient is very small then the learning rate will be very large. This helps in finding the complex Global Minimum.