Assignment 1

Student: Stefano Quaggio
Student's email: quaggs@usi.ch

October 14, 2024

Polynomial regression

The goal of this assignment is to perform polynomial regression using gradient descent with Pytorch to estimate the w of the polynomial:

$$p(z) = \frac{z^4}{30} - \frac{z^3}{10} + 5z^2 - z + 1 = \sum_{k=0}^{4} z^k w_k$$

Python file notes: Extra functions were added for greater code compactness and readability, trying as much as possible to avoid duplicating code. Each function parameter has been typed.

Answers to the questions:

2. *plot_function* was created using 2 main functions: `linspace` and `poly1d` both from the Numpy library. Once the maximum and minimum values of the variable z are obtained, 50 evenly spaced samples, calculated over the interval, are extracted using `linspace`. To construct the polynomial I used the function `poly1d`, which allows, given the coefficients, to construct a 1-dimensional polynomial as in our case. The idea, later used in other functions, is to have the same mathematical logic to obtain the y-values, simply by applying $p(z)$ where p is the polynomial obtained from `poly1d` and z are all the values on the x-axis obtained from the *z_range*. The result is shown in Figure 1.

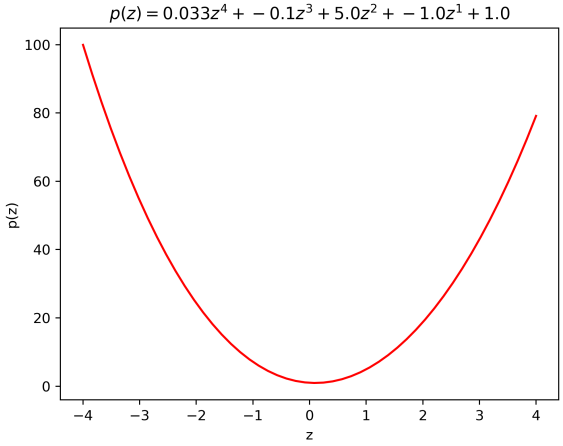
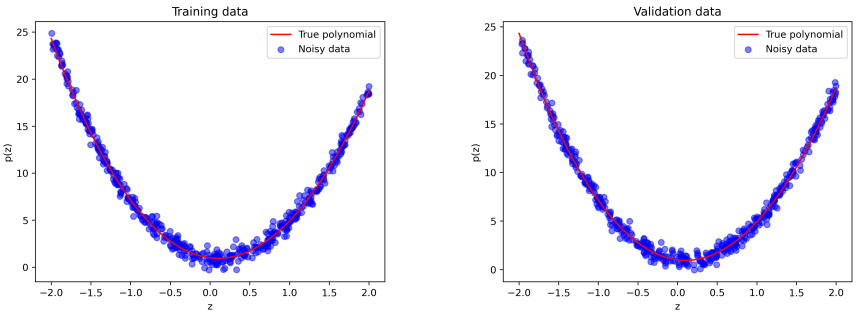


Figure 1: Polynomial function

3. *create_dataset* allows to generate the dataset D', thus returning a noisy dataset. First, I went to reuse the code seen during lectures to generate "sample sizes" of points from a uniform distribution, rescaling them between the minimum and maximum values of the range passed as a parameter. For the creation of *y_hat*, instead, I applyed to the Numpy `poly1d` function (seen in step 2) of my polynomial, my values of z, then creating a tensor. The result is a tensor of real values of my polynomial. From this, I added Gaussian noise with the sigma value as input to the function. At the end, for getting $x^i = \left[1, z_i, z_i^2, z_i^3, z_i^4\right]^T$, I used a Pytorch function called "stack". I basically stack the values obtained from a single z from time to time, applying the various elevations, resulting in a 2-dimensional, 500 row x 5 column tensor. Each row represents a 1-dimensional tensor of increasing powers of the value of z, while the columns the different powers from z^0 to z^4.

5. *visualize_data* function uses the same logic as the *plot_polynomial* function in Step 1, with only the addition of the scatter plot relative to the data generated by *create_dataset*. Since X is a 500 x 5 tensor, I extracted only the first column related to the power z^0, thus having the value of x independent of z. Figure 2 shows the plot of the training and validation data.



(a) Noisy training dataset (b) Noisy validation dataset

Figure 2: *visualize_data* function's plot

Up to this point, the vector of coefficients used has been initialized with values inverted from what the delivery indicates, thus of the type $w = [w_4, w_3, w_2, w_1, w_0]$, since the `poly1d` function uses the coefficient at the first position of the vector as the power of maximum degree, decreasing the degree at subsequent positions. By doing this, I avoided having to reorder the vector each time to use it in functions.

6. To perform polynomial regression on dataset D, I created a general function called *create_train_model*, which can also be used for the bonus question by adjusting the input size of the linear regression model and the boolean value *bonus_question*. The code is very similar to what was shown during the lectures, as we can use linear regression with an input size of 5 (features) and gradient descent to perform our task. Therefore, the model creation, optimizer, and evaluation are the same as what we used in class. The adjustment I made relates to the shape of the y tensors, as the *create_dataset* function returns a tensor with one less dimension than the X tensor, which caused issues during training. I'm also saving the estimated weights at each step so they can be plotted in subsequent questions. After multiple tests, I found that 0.027 could be the highest value for the learning rate, beyond which the model would no longer converge (i.e. it would stop learning) during gradient descent. If I set a lower value, the model might take too long to converge (in the worst case, it might even get stuck in a local minimum). Regarding the bias parameter for the neural network, I set it to False because the constant term of the function is already provided to the model (which represents the bias), so we are essentially telling the model to learn this without adding a redundant layer of bias. As for the number of steps, I performed several manual tests to tune this parameter, and I noticed that the ratio between steps and loss improvement tended to diminish after around 600 steps, so I decided to stop the training after 620 steps. The final evaluation loss value is around 0.26.

7. I created a specific function called *visualize_loss* where I plot the train and validation loss over time. It is clear from the Figure 3 that both loss decrease until they assest themself on around 0.26 after 620 steps.

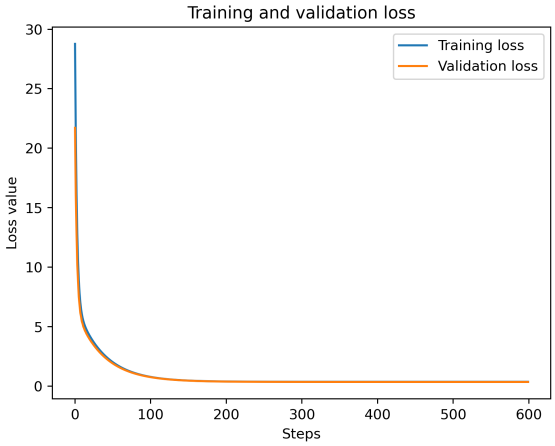


Figure 3: Loss values for train and validation

8. In Figure 4 you can see the true vs the estimated polynomial by the model previously trained. As you can see you are almost unable to view the true polynomial since the estimated one is very close to it.

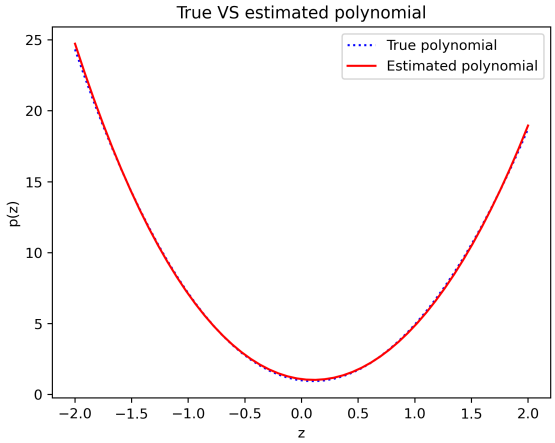


Figure 4: True and estimated polynomial

9. Here it's the first time I needed to reorder the coefficients, so that I iterate only once for the plot related to the estimated weights and to have the correct numbering. For line 226, a sequence of values from 0 to step - 1 is generated for the x-axis, while extracting by slicing the column related to the specific weight of its values over time. I used the axhline function to draw the reference line of the actual value of the weight. As seen in Figure 5, the weight w_4, corresponding to the highest degree term in the polynomial, undergoing small variations can have a large impact on the output, causing these fluctuations as the model tries to balance it.

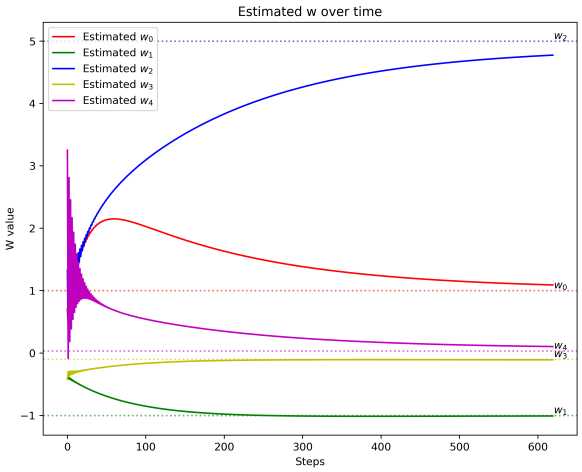


Figure 5: Estimated weights over 620 steps

Question

The learning rate is a hyperparameter used during the training of neural networks, which determines the step size that the optimization function has to perform. Its choice, therefore, affects its characteristics: if we set a high value, it allows us larger updates and potentially faster convergence, but risks causing oscillations or exceeding the optimal minimum. Conversely, a low value is potentially more stable, but risks taking longer to converge and getting trapped in local minima. Its choice, therefore, leads to the performance of many tests to find the optimal value. In contrast, adaptive methods, such as Adagrad, use techniques to adjust the learning rate during the training phase. It modifies the learning reate at each step for each parameter, based on the value of the past gradients. The advantages of theese are avoiding manual tuning of learning rates and it works very well with scattered data since it makes sure that parameters that receive few updates get higher learning rates, which speeds up convergence.

Bonus question

The goal is to approximate by linear regression the following function and evaluate the performance of the model in the intervals [-0.05, 0.01] and [-0.05, 10]:

$$f(x) = 2\log(x+1) + 3$$

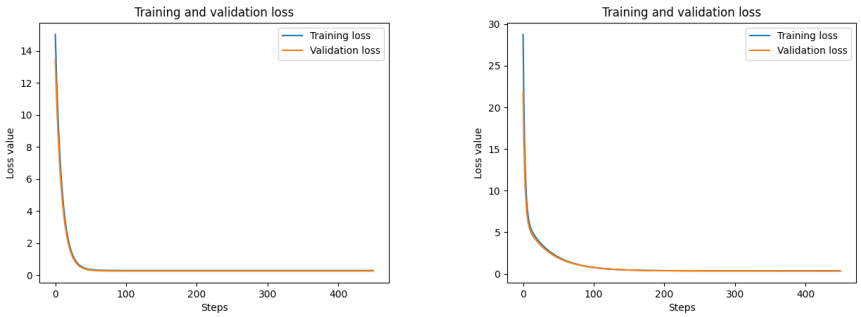
Basically what I did to answer the question is to reuse the same logic to generate the noisy dataset and the various visualization plots seen in the questions above. Small differences from the previous functions:

- *bonus_create_dataset*: allows generating noisy datasets for train and validation for the real function.

- *create_train_model*: input size for the nn.Linear of 1 (since we have only one feature to learn) and I set a boolean variable *bonus_question* to True, which allows to set the learning rate of the model to 0.03 (best value found after several tests) and to reshape the tensors of X (it was missing a dimension with respect to the tensors of y). In this case I set the bias to True since I wanted to estimate the value, which in the original function is +3. Finally I used 450 steps because they are a good trade off between efficiency and loss value obtained.

- *bonus_plot_function_compare*: inside I created the estimated function from the model, using the values of X previously created by the `linspace` function, multiply by the estimated coefficients and add the estimated bias.

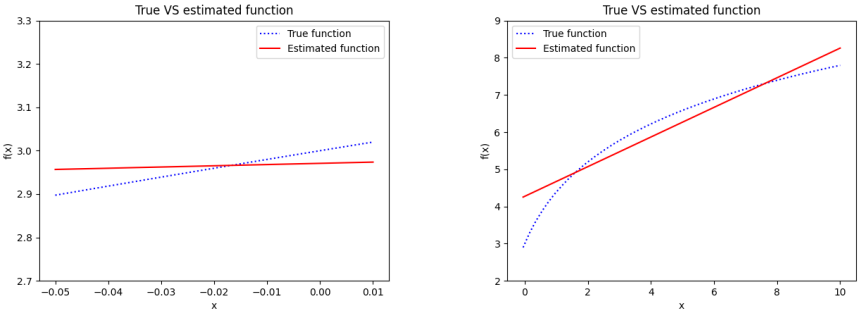
In Figure 6, the values of the loss function over time are displayed for training and validation. As could be guessed, the rate of improvement is higher in the first range since the function is very similar to a straight line. The final values are 0.25 and 0.34.



(a) Loss function range [-0.05, 0.01] (b) Loss function range [-0.05, 10]

Figure 6: Loss function for different value range

The function estimated by the model performs best in the first range of values, as can be seen in Figure 7



(a) Estimated function [-0.05, 0.01] (b) Estimated function [-0.05, 10]

Figure 7: Loss function for different value range