# Assignment 1

Student: Damiano Ficara
Student's email: damiano.ficara@usi.ch

## Polynomial regression

2. To realize the function `plot_polynomial`, that it's possible to find from line 15 to 29, this code creates a function that, given a set of coefficients of a polynomial and a range of z values, draws the graph of that polynomial in that range. The result is a plot shown in Figure 1

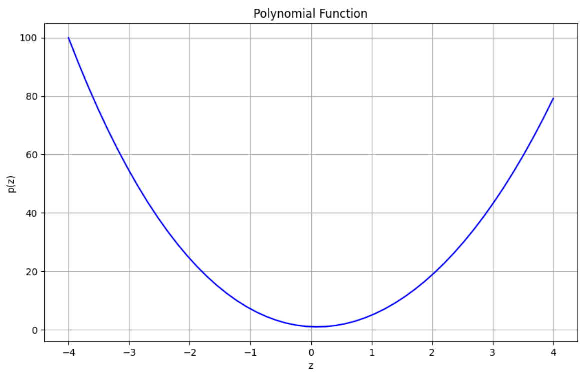

Figure 1: Polynomial function

3. To start from a reliable point, I used the code of `linear-regression.ipynb` saw during the lesson about linear regression. I adapted the original `create_dataset` function to fit the polynomial regression requirements. The implementation is available from line 33 to 48, uses evenly spaced z values instead of random ones, and constructs a design matrix for polynomial features. I adjusted the y_hat calculation to use matrix multiplication with the provided coefficients, enabling polynomial regression. The core logic of dataset generation, including adding Gaussian noise, remains similar to the base function. This approach allows for creating a dataset suitable for polynomial regression up to the 4th degree while maintaining the essential structure of the original function. In particular I used the function `torch.stack()`, that is a method joins a sequence of tensors along a new dimension

5. The function visualize_data plots both the *true polynomial* and *the generated data*. It takes in the dataset (X and y), polynomial coefficients

(coeffs), a range for the independent variable (z_range), and an optional title. The function generates the true polynomial using the coefficients, then creates two plots: one with the training data shown in Figure 2 and one with the validation data shown in figure 3, overlaying scatter plots of the data points alongside the true polynomial curve.

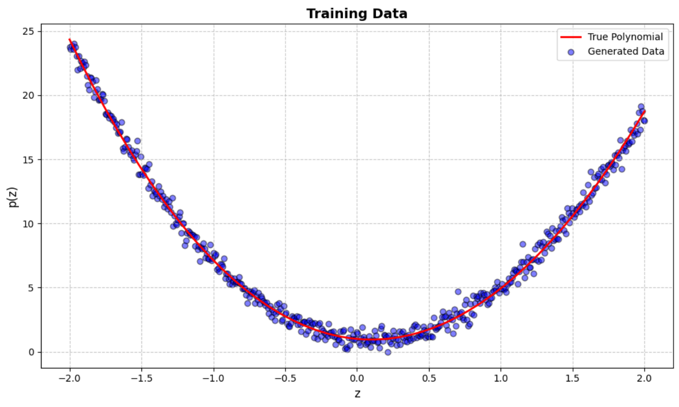

Figure 2: Training data

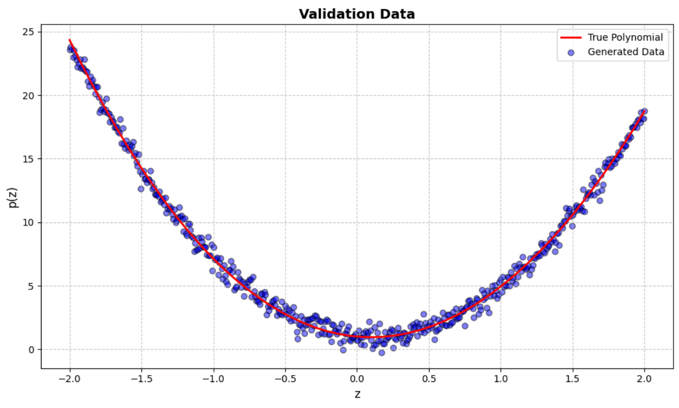

Figure 3: Validation data

6. For this task, I adapted the code from `linear-regression.ipynb`, specifically the section from line 130 to 173. I made the following key modifications and choices:

   - Increasing the number of steps from 10 to 600 to ensure sufficient training time for the polynomial regression model.
   - Setting the learning rate to 0.01, which I found to be effective through experimentation. A smaller learning rate would result in slower convergence, while a higher rate might cause instability or overshooting of the optimal parameters.

- Establishing the bias parameter in `torch.nn.Linear` to False because the polynomial features already include a constant term ($z^0$), making an additional bias term unnecessary.
- Adding a list `weights` to store the weights at each step, allowing for analysis of parameter evolution during training.
- Adjusting the model architecture to have 5 input features (corresponding to the polynomial degrees 0 to 4) and 1 output.
- Using the same loss function (MSE) and optimizer (SGD) as in the original notebook, as they are suitable for this regression task.

The 600 steps were chosen based on observing the convergence of the loss values. This number of steps appeared sufficient for the model to reach a stable state without overfitting.

7. Between line 179 to 186, to visualize the training process, I plotted the training and validation loss as functions of the iterations. The graph, shown in Figure 4 shows both loss curves decreasing over time, which is the expected behavior for a well-functioning model. The training loss (blue line) decreases more rapidly and reaches a lower value compared to the validation loss (orange line). This pattern is typical and indicates that the model is learning the training data well while still generalizing to unseen data. The convergence of both curves towards the end of training suggests that the chosen number of iterations was sufficient for the model to reach a stable state.

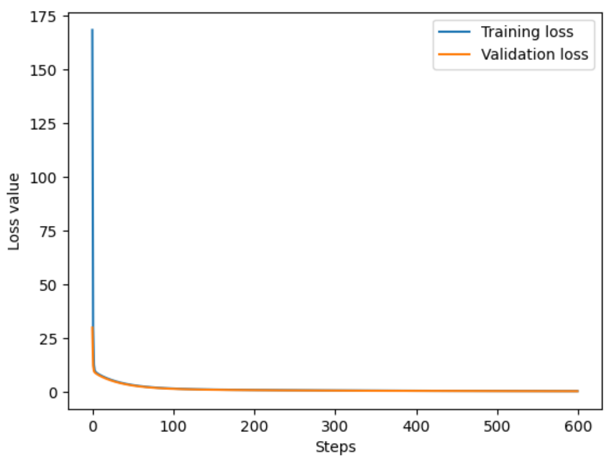

Figure 4: Training and Validation loss

8. Between line 188 to 204 I created a plot, shown in Figure 5 as comparing the true polynomial function with the estimated polynomial using the coefficients learned by the model. The close alignment between these two curves demonstrates that our model has successfully approximated the underlying polynomial function. There are minor discrepancies visible,

which is expected due to the added noise in our training data and the intrinsic difficulties associated with fitting higher-degree polynomials

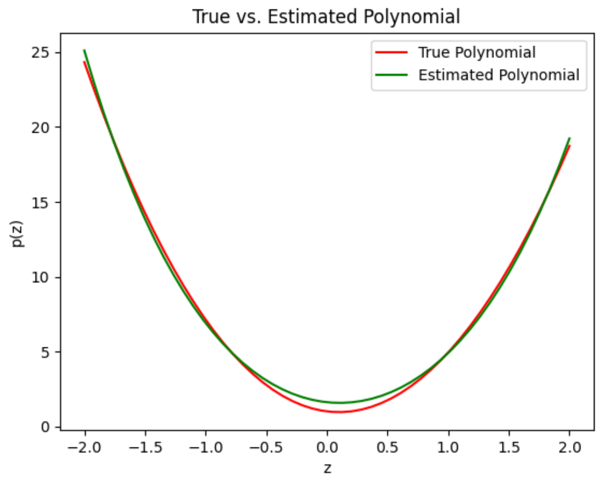

Figure 5: True polynomial function with the estimated polynomial

9. Between line 208 to 221 to gain insight into the learning process, I plotted, as possible to see in Figure 6, the evolution of each parameter (w0 to w4) over the course of training, along with their true values. Each parameter is represented by a different color, with the estimated values shown as solid lines and the true values as dashed horizontal lines of the same color. This visualization reveals how the parameters converge towards their true values over time. Some parameters converge more quickly than others, which is common in polynomial regression due to the different scales of the terms. The plot also shows some fluctuations in the parameter values during training, which is characteristic of the stochastic gradient descent optimization process used. Overall, the convergence of the estimated parameters to their true values confirms the effectiveness of our training procedure.

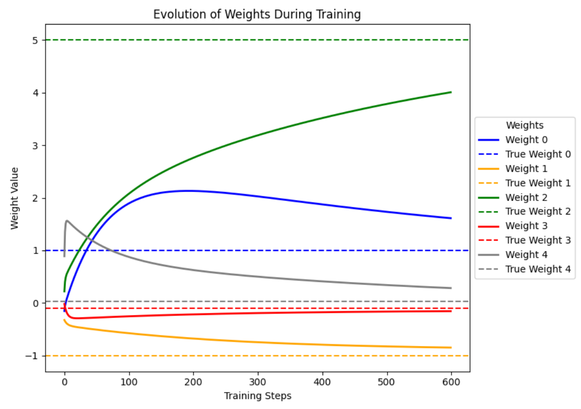

Figure 6: True polynomial function with the estimated polynomial

## Questions

The learning rate is a crucial hyperparameter that determines the magnitude of parameter updates during each iteration of optimization algorithms, such as stochastic gradient descent (SGD) and its variants. An excessively high learning rate may cause the optimization process to overshoot the optimal solution, resulting in oscillations or divergence. Conversely, an overly low learning rate can lead to slow convergence and potentially trap the model in local minima. Identifying this optimal value can be challenging, particularly when working with complex datasets or model architectures. To address this issue, adaptive methods have been developed to dynamically adjust the learning rate based on feedback received during training. By modifying the learning rate , adaptive scheduling techniques aim to achieve faster convergence and enhanced generalization performance. One such example is Adagrad,that adapts the learning rate for each parameter individually. Frequently updated parameters receive smaller learning rates, while infrequently updated ones get larger rates. This balances updates across all parameters, improving model convergence.

## Bonus question

This analysis evaluates linear regression's performance in approximating $f(x)$ over two intervals: $[-0.05, 0.01]$ (Case 1) and $[-0.05, 10]$ (Case 2). The implementation follows structures from Lecture 3.

Key implementation details:

- Bias in `nn.Linear(1, 1)` is set to `True` for the constant +3.
- Learning rate is 0.01, with 600 steps and sample size of 1000 for both training and validation.
- `create_dataset_log` generates noisy data based on $f(x)$.
- `train_and_evaluate` tracks loss during training.

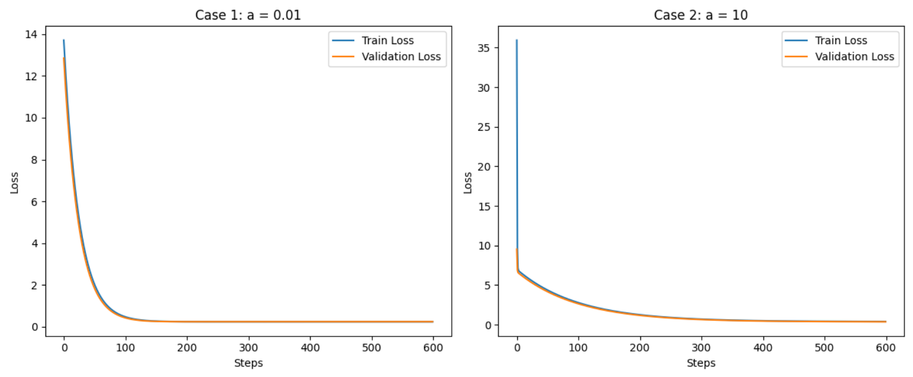

Figure 7: Loss curves for Case 1 and Case 2

Figure 7 shows that case 1 converges to a lower final loss ($\sim 0.25$) than case 2 ($\sim 0.36$), indicating better performance over smaller intervals.

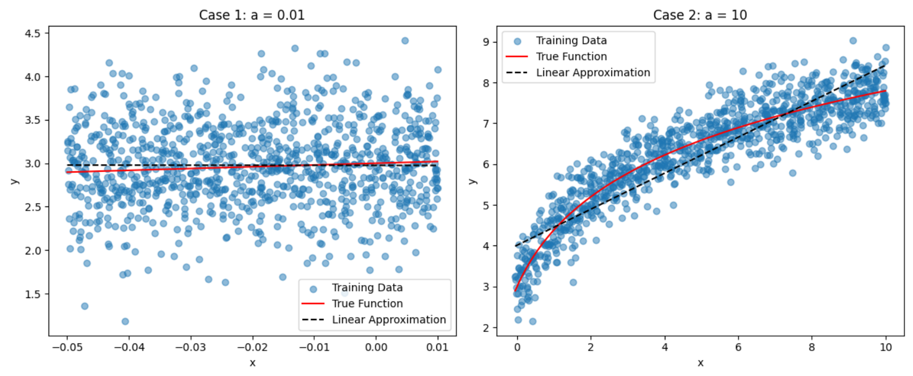

Figure 8: Function approximations for Case 1 and Case 2

Figure 8 shows that case 1 provides a better fit since $f(x)$ behaves more linearly over small intervals. case 2 struggles with non-linearity. An insightful analysis was conducted to explore the impact of the number of training steps on model performance. The final validation losses for various training steps are presented in the table 1

| **Steps** | **Case 1** ($a = 0.01$) | **Case 2** ($a = 10$) |
|---|---|---|
| 200 | 0.240824 | 1.175721 |
| 300 | 0.241830 | 0.640952 |
| 400 | 0.242431 | 0.448782 |
| 500 | 0.242518 | 0.380204 |
| 1000 | 0.242524 | 0.343842 |

Table 1: Final validation losses for different training steps.

Increasing training steps improves performance for Case 2, but the gains become marginal after 500 steps.