Assignment 1

Student: EMANUELE BELLINI
Student's email: belliem@usi.ch

Polynomial regression

Question 2

Define a function plot_polynomial(coeffs, z_range, color='b') :

```
def plot_polynomial(coeffs, z_range, color='b'):
    # Plot the polynomial defined by the coefficients
    z = np.linspace(z_range[0], z_range[1], 500)
    # coeffs[::-1] is used to have the coefficients
    in the correct order required by np.polyval
    p = np.polyval(coeffs[::-1], z)
    plt.figure(figsize=(12, 8))
    plt.plot(z, p, color=color, label='Polynomial')
    plt.xlabel('z')
    plt.ylabel('p(z)')
    plt.title('Polynomial Plot')
    plt.legend()
    plt.savefig('polynomial_plot.png', dpi=300, bbox_inches='tight')
    plt.close()
```

Since coeffs = np.array([1, -1, 5, -0.1, 1/30]) , and the np.polyval() function requires coefficients ordered from the highest degree term to the lowest, I reverse the order of coeffs elements. The corresponding plot is shown in Figure 1 .

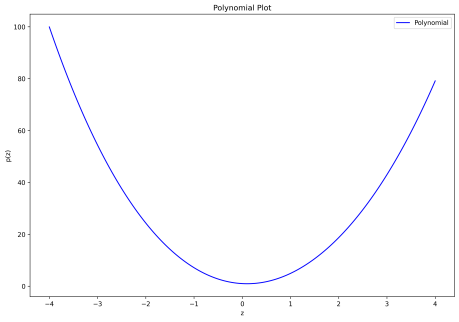

Figure 1: Polynomial plot

Question 3

```
def create_dataset(coeffs, z_range, sample_size, sigma, seed=42):
    np.random.seed(seed)
    # Generate random values of z
    z = np.random.uniform(z_range[0], z_range[1], sample_size)
    # Create X.T = [1, z, z^2, z^3, z^4]
    X = np.column_stack([z**i for i in range(5)])
    # Create y = p(z) + noise
    y = np.polyval(coeffs[::-1], z)
    + np.random.normal(0, sigma, sample_size)
    # Convert to torch tensors
    return torch.from_numpy(X).float(),
    torch.from_numpy(y).float().unsqueeze(1)
```

Doing that, I build the dataset $D' := \{(\mathbf{x}i, yi)\}_{i=1}^{N}$, by returning the torch tensors $\mathbf{X}$ and $\mathbf{y}$

Question 4

```
# Coefficients in ascending order of degree as requested
coeffs = np.array([1, -1, 5, -0.1, 1/30])
z_range = [-2, 2]
sigma = 0.5
sample_size_train = 500
sample_size_eval = 500

X_train, y_train = create_dataset(coeffs, z_range,
sample_size_train, sigma, seed=0)
X_eval, y_eval = create_dataset(coeffs, z_range, sample_size_eval,
sigma, seed=1)
```

1 Question 5

```
def visualize_data(X, y, coeffs, z_range, title=""):
plt.figure(figsize=(12, 8))
z = np.linspace(z_range[0], z_range[1], 500)
p = np.polyval(coeffs[::-1], z)
plt.plot(z, p, label='True Polynomial')
# Use the second column of X as the z values
plt.scatter(X[:, 1].numpy(), y.numpy(), alpha=0.6, label='Data')
plt.xlabel('z')
plt.ylabel('p(z)')
plt.title(title)
plt.legend()
# Use the title to save the plot with the correct name
plt.savefig(f'{title}.png', dpi=300, bbox_inches='tight')
plt.close()
```

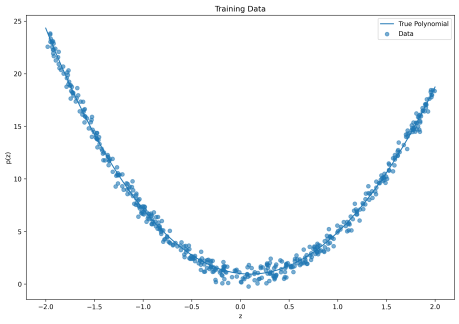

Figure 2: Training Data vs Polynomial plot

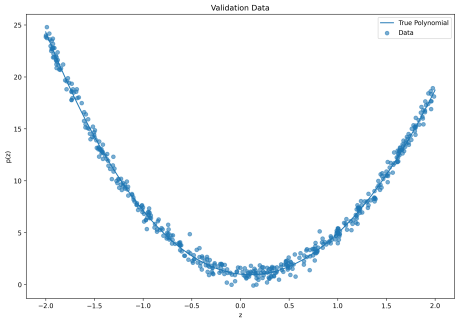

Figure 3: Validation Data vs Polynomial plot

Here I pass the title as parameter in order to save 2 different images: one for the training data (Figure 2) and one for the validation (Figure 3).

Question 6

```
model = nn.Linear(5, 1, bias = False)
criterion = nn.MSELoss()
learning_rate = 0.01
optimizer = optim.SGD(model.parameters(), lr=learning_rate)

n_epochs = 5
steps_per_epoch = 300
total_steps = n_epochs * steps_per_epoch
```

The learning rate of 0.01 assures fast convergence while maintaining stability.

If I set learning rate too low (e.g. 0.0001) then model update weights very slow and it significantly increased the time to converge. This is ineffective for training as I will need to take many small steps with my optimizer, on the way to minimum of loss function. In addition, the model might be stuck in local minima or saddle points since it does not have enough momentum to escape. In my case when I decrease the learning rate from 0.01 to 0.005, final loss value

goes up from 0.2515 to 0.3168 meaning that with smaller updates, the model was not able to find a better minimum within the allocated training steps and epochs.

However, when the learning rate is too high (e.g., 0.03), the model's weights change too drastically with each update. This can result in the model overshooting the optimal minimum such that instead of steadily diminishing, loss starts to fluctuate markedly up and down. As a result, the training process can become unstable and may never converge, leading to divergence. Setting the learning rate to 0.03 caused the model to diverge (with n_epochs = 5 and steps_per_epochs = 300), meaning that instead of minimizing, the loss increased over time or oscillated indefinitely.
More epochs will not enhance the model precision because the validation loss would increase. The same thing applies to steps: even if I set 3000 steps, the evaluation loss will only increase (slightly).

Output:

```
Epoch [1/5], Step [300/1500], Train Loss: 0.8089, Val Loss: 0.7454
Epoch [2/5], Step [600/1500], Train Loss: 0.4409, Val Loss: 0.3961
Epoch [3/5], Step [900/1500], Train Loss: 0.3138, Val Loss: 0.2900
Epoch [4/5], Step [1200/1500], Train Loss: 0.2675, Val Loss: 0.2589
Epoch [5/5], Step [1500/1500], Train Loss: 0.2504, Val Loss: 0.2519
True coefficients: [ 1.   -1.    5.   -0.1    0.03333333]
Estimated coefficients: [ 1.04  -1.00   4.82  -0.10   0.08]
Training done, with an evaluation loss of 0.2518561780452728
Final w: Parameter containing:
tensor([[ 1.0400, -1.0036,  4.8221, -0.0987,  0.0829]], requires_grad=True)
Final b:       None
```

The bias is set as False because I have included w_0 as coefficient ($1 = w_0$), so the model will find 5 weights and no bias. A different solution could be to set bias = True and to use 4 coefficients (w_1, w_2, w_3, w_4) and the bias $b = 1 = w_0$, but this solution led to less precise results.

Question 7

In Figure 4, you can see the update in the training and validation loss function during the training process (updates are performed each step). Both the training and validation losses converge rapidly in the early stages of training, indicating that the model learns quickly at first. Also, after the initial rapid drop, both losses continue to decrease smoothly, indicating steady learning throughout the training process. Finally, the similarity between both training and validation losses suggests the model's good generalization ability.

Question 8

As you can see in Figure 5, the true and the estimated polynomial are very similar.

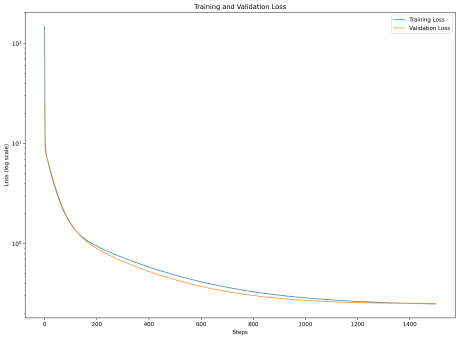

Figure 4: Training and Validation loss curves

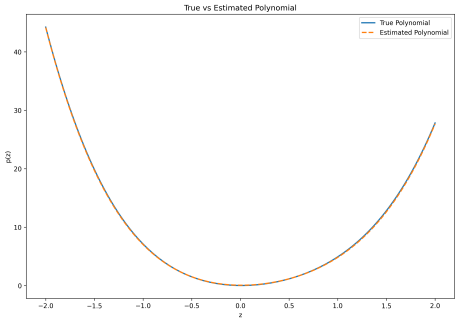

Figure 5: Comparison between the true and the estimated polynomial

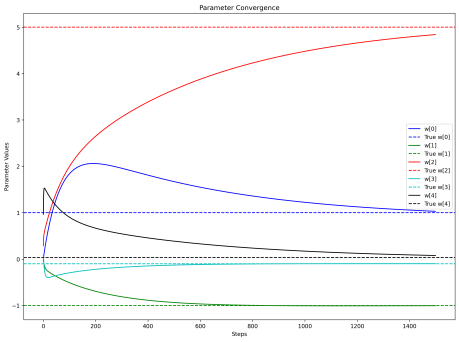

Figure 6: Convergence of parameters

Question 9

In Figure 6 I show how the parameters effectively converge to the true value during the training process. Notice that on the x axis we have the number of steps.

Question 10

The choice of static learning rate (either large or small) applies the same step size uniformly across all the parameters and across time. A too small learning rate makes the training stable but slows down the convergence, whereas a larger one speeds up the learning but at the risk of overshooting or instability. This is difficult to balance by hand because different parameters often require different step sizes, and the best rate may also change over time.
Adaptive methods like Adagrad tackle this issue by adjusting the learning rate for each parameter individually. Adagrad accumulates the squared gradients, reducing the effective learning rate for frequently updated parameters, while giving higher rates to those with smaller gradients. This ensures efficient convergence in flat regions and smaller steps in steep ones, making optimization more stable. However, Adagrad's learning rate often becomes very small over time, resulting in stagnation in training.