Assignment 1

Student: Fabian Gobet
Student's email: gobetf@usi.ch

19 October 2023

Polynomial regression

1. In order to plot the polynomial given in the exercise sheet, I defined two functions:

   - $p\_func(coeffs,\ Z)$
   - $plot\_polynomial(coeffs,\ z\_range,\ color='b',\ func = p\_func)$

   The $p\_func()$ function takes up two arguments: $coeffs$ and Z; and computes X, the Vandermonde matrix for the elements of the vector Z (the features matrix), and returns both X and the matrix product of X by $coeffs$. The matrix product effectively computes the polynomial of degree $len(coeffs) - 1$ at each point of X, admitting that the first element of $coeffs$ is the bias of the polynomial, and returns both X and the matrix product as np.float64.

```python
def p_func(coeffs,Z):
    X = np.vander(Z, N=len(coeffs), increasing=True).astype(np.float64)
    return X, (X@coeffs).astype(np.float64).reshape(-1,1)
```

   The $plot\_polynomial()$ function has a slightly modified signature from the original one, having an extra pre-defined parameter that assumes a function to which to do the plot. The reason for this is because I wanted to recycle some code so later I could plot the sine function.

   Here, I first determine the min and max range and create a numpy array with 1000 equidistant points from min to max. Then, i retrieve the features of $coeffs$ for the function in parameter, set conditions for color and label and plot the computed points along the generated 'linspace' points.

```python
def plot_polynomial(coeffs, z_range, color='b', func=p_func):
    z_min, z_max = z_range
    sample_size = 1000
    z = np.linspace(z_min, z_max, num=sample_size).astype(np.float64)
    _,y = func(coeffs,z)
    is_p = True if color=="b" else False
    lab = "f(x)" if is_p else "predicted␣f(x))"
    lin_width = 1 if is_p else 1.5
    plt.axhline(0, color='black',linewidth=1)
    plt.axvline(0, color='black',linewidth=1)
    plt.plot(z, y, color=color, label=lab, linewidth=lin_width)
    plt.xlabel('x')
    plt.ylabel('f(x)')
```

   As a dummy test i wrote the following code:

```python
z_range = [-3,3]
coeffs = np.array([0,-10,1,-1,1/100], dtype=np.float64).reshape(-1,1)
plot_polynomial(coeffs, z_range, color='b')
plt.legend()
plt.show()
```

Obtaining the following graph in Figure 1

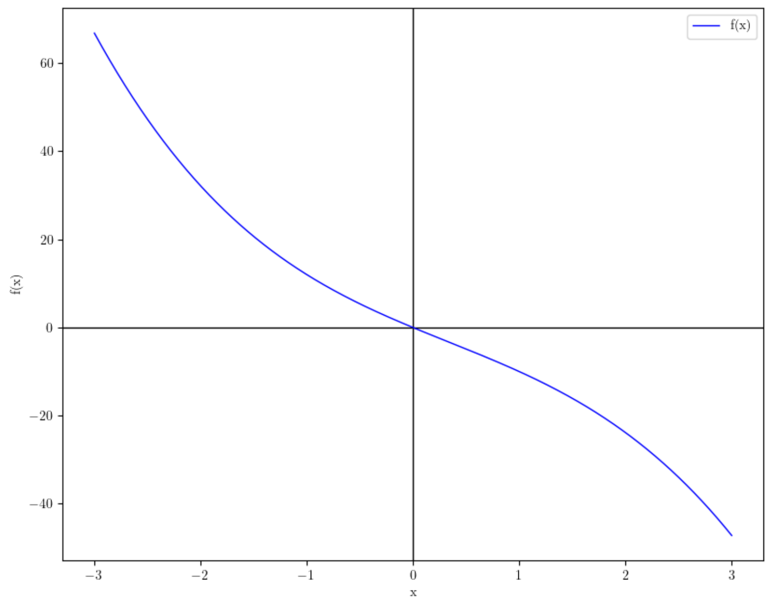


Figure 1: p(x) plot

2. For this exercise we were asked to implement a function *create_dataset*() that creates a data set based on interval, sample size and a sigma for the normal distribution with mean 0. Again, since I wanted to recycle this function to use in the sine exercise, I changed it's signature, having now an extra predefined parameter for the function to which the points are generated and then the noise is added.

   In this function I first create a numpy random state so the seed is used locally. I use that random state to sample '*sample_size*' points from the uniform distribution between the lower and upper bounds of z_range. Then, I compute the features and the value of the points through the given function, and add some noise using a normal distribution of mean 0.0, standard deviation *sigma* based on the random state previously created. Finally, I return the samples vector, the features matrix and the values vector of the function at each sample point.

```python
def create_dataset(coeffs, z_range, sample_size, sigma, seed=42,func=p_func
    ):
    random_state = np.random.RandomState(seed)
    z_min, z_max = z_range
    Z = random_state.uniform(z_min,z_max,(sample_size)).astype(np.float64)
    x,y = func(coeffs,Z)
    y += random_state.normal(0.0, sigma, y.shape).astype(np.float64)
    return Z,x,y
```

3. Using the previous code, I created two data sets with different seeds, 0 for test and 1 for evaluation, using the p_func function. In the polynomial exercises we are not interested in retrieving the vector Z from the *create_dataset* function because we can retrieve them later in the *visualize_data* function using the first column of the features matrix. Therefore, for now we just set the first retrieving parameter as _. Hence, we have the following code:

```
z_range = [-3,3]
coeffs = np.array([0,-10,1,-1,1/100], dtype=np.float64).reshape(-1,1)
_, x_train, y_train = create_dataset(coeffs,z_range,500,0.5,seed=0)
_, x_eval, y_eval = create_dataset(coeffs,z_range,500,0.5,seed=1)
```

4. In order to visualize the data, given an X matrix (which may be a 1D vector), the y values, a range and the coefficients (for the polynomial function), I defined the function *visualize_data* with an extra predefined parameter *func* to accommodate the utility of the function for the sine function.

   The *visualize_data* function first looks at X to see its dimension size. If X is the features matrix then I know we're trying to visualize the polynomial, thus the x values we're looking for will be in the second columns (index 1) of X (which will be the Vandermonde matrix). Otherwise, we'll use the same X.

   Then, I do a scatter plot for my processed X against the values of y and call the function plot_polynomial, given as parameter, to also plot the exact values of the function in the same graph. This function calls out for *plt.show*(), whereas plot_polynomial doesn't.

```
def visualize_data(X, y, coeffs, z_range, title="", func=p_func):
    x = X[:,1] if X.ndim > 1 else X
    plt.scatter(x,y,color='red', marker="x", label = title, s=8)
    plt.title("f(x)␣and␣"+title)
    plot_polynomial(coeffs, z_range, func, color='b')
    plt.legend()
    plt.show()
```

   After having implemented the function, I ran it both data sets created in exercise 3, generated Figure 2 and Figure 3

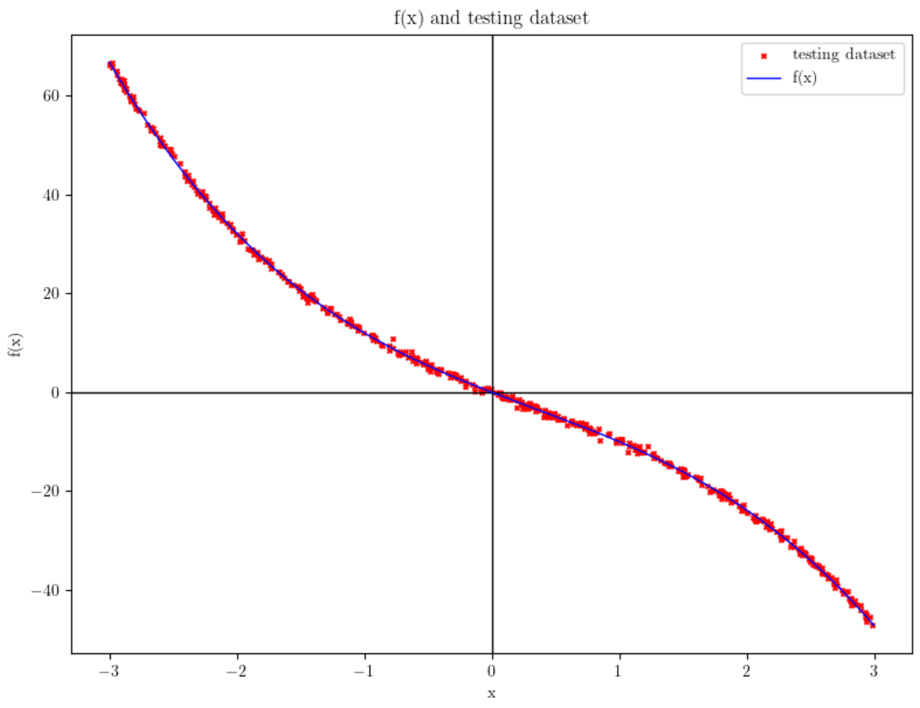


Figure 2: p(x) and training data

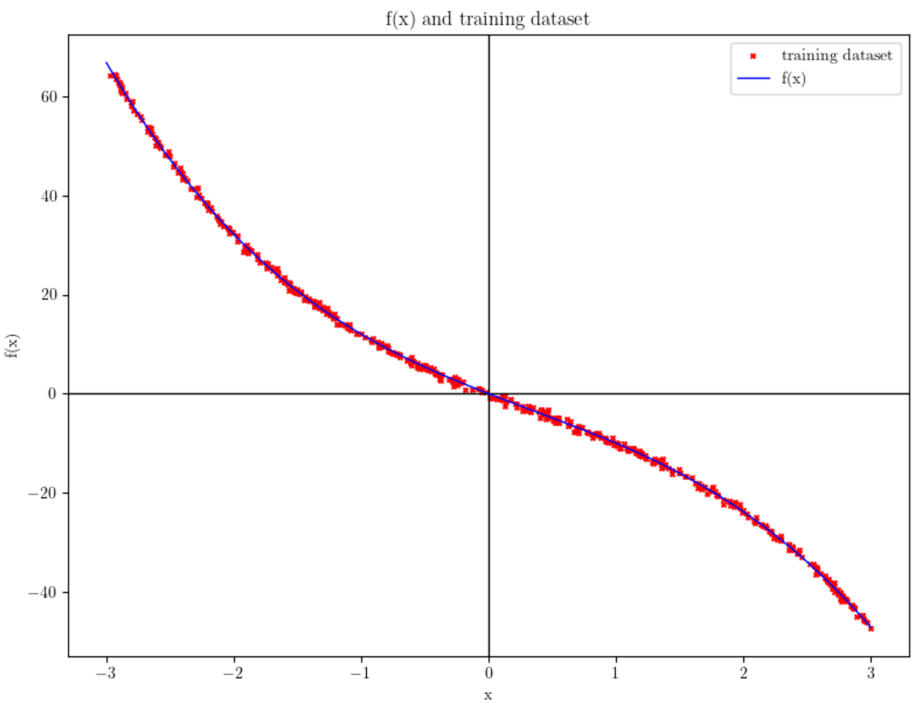


Figure 3: p(x) and testing data

5. Since the data was generated by me and according to the plots on Figure 2 and Figure 3, there shouldn't be any outliers to which I should attend. Furthermore, the added noise will make it so that the model can learn the underlying structure without over fitting, given the correct hyper-parameters, thus reflecting an accurate prediction for the evaluation data set.

   The task at hand is to perform linear regression on a polynomial of degree > 1, so as a data pre-processing step it is important to compute the features of each element x to be evaluated. This is achieved through the deployment of the *p_func* function, as explained Exercise 1.

   On this exercise I started out by defining a very simple *nn.Linear* model with *nn.MSELoss* and *nn.Adam* optimizer. I wasn't happy with the rate of convergence, even by tweaking the learning rate and adding weight decay to provide an L2 regularization effect. So I implemented a function to create mini-batches for my data sets, and then ran the model on it. Therefore, I will begin to explain the mini-batches code implementation and then I'll go through the rest of the code regarding the model and the results vs. hyper-parameters.

   The *create_batches*() function takes as arguments the sets X and Y, the training data and the expected result respectively, a batch size, a device and a seed. It then extracts the number n of training elements, creates a numpy random state based off of the seed and shuffles an array containing all indices from 0 to $n-1$. After these steps, it begins a for loop from 0 to $n-1$, with a step of *batch size*. In each of these steps, it grabs the portion of indices from the shuffled array, extracts the elements of X and Y corresponding to those indices and appends them to a respective list. If the argument '*device*' is not *None*, it also loads the batches to the device. At the end of the for loop results two lists for the X batches and the Y batches, properly congruent for the indices element wise.

```
def create_batches(X,Y,batch_size=1,device=None,seed=42):
    number_training = X.shape[0]
    shuffle_state = np.random.RandomState(seed)
    indices = np.arange(number_training)
    shuffle_state.shuffle(indices)

    X_batches = []
    Y_batches = []

    for start_idx in range(0,number_training,batch_size):
        end_idx = min(start_idx + batch_size, number_training)
        batch_indices = indices[start_idx:end_idx]
        X_batch = torch.tensor(X[batch_indices], dtype=torch.float64)
        Y_batch = torch.tensor(Y[batch_indices], dtype=torch.float64)
        if device is not None:
            X_batch = X_batch.to(device)
            Y_batch = Y_batch.to(device)
        X_batches.append(X_batch)
        Y_batches.append(Y_batch)

    return X_batches, Y_batches
```

Having implemented the prior function, I defined the following set of hyper-parameters and model for this exercise:

```
DEVICE = torch.device("cuda:0" if torch.backends.mps.is_available() else "
    cpu")
learning_rate = 0.5
batch_size = 250
num_epochs = 1000
iterations_limit = 1000

model = nn.Linear(coeffs_size,1,bias=False, dtype=torch.float64)
loss_fn = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

x_eval = torch.tensor(x_eval, dtype=torch.float64)
y_eval = torch.tensor(y_eval, dtype=torch.float64)

model = model.to(DEVICE)
x_eval = x_eval.to(DEVICE)
y_eval = y_eval.to(DEVICE)

batch_size = min(batch_size, x_train.shape[0])
X_batches, Y_batches = create_batches(x_train,y_train, batch_size=
    batch_size, device=DEVICE, seed=42)

train_loss = []
eval_loss = [loss_fn(model(x_eval),y_eval).item()]
parameters = [model.weight.data.detach().cpu().numpy().squeeze().copy()]
num_iterations = 0
```

The hyper-parameters chosen do not correspond to the initial ones. These had to be tweaked according to the stability and difference between training losses and evaluation losses, while also taking into account the rate of convergence. There is also a train loss and evaluation loss list that I used to keep track of the progression performance of the model. Furthermore, a parameters list was kept in order to account for exercise 8.

It is note worthy that the meaning of iteration was not well defined given that mini-batches were employed, therefore I accounted each mini-batch pass as an iteration, and kept both the limit of number of epochs and iterations to 1000.

Since i was using mini-batches, these provide a slight regularization effect and allow for bigger learning rates. If the learning rate is too small, it takes a lot of time for the model to converge to a solution. On the other hand, if the learning is too big, we may never step into a minima of the function we are trying to minimize. This latter case can be identified by the inconsistency of decay for the training loss values. The Adam optimizer works well here because it is an adaptive method for the learning rate, allowing some small degree of freedom regarding the magnitude of the initial learning rate.

We are including the bias in our weight vector, so we should define it as false in the definition of *nn.Linear*.

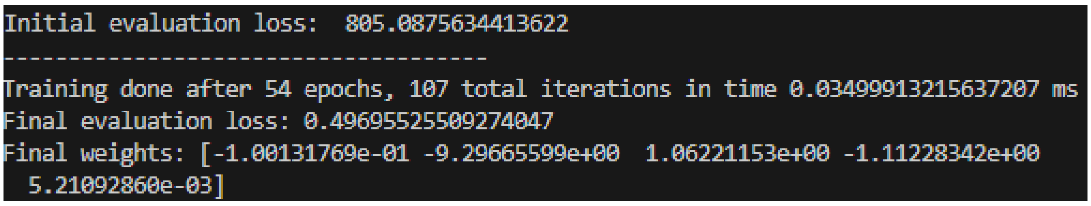
Given the prior considerations, the model code looks like the following:

```
start = time.time()
    for epoch in range(num_epochs):
        l = 1
        model.train()
        for x_batch, y_batch in zip(X_batches,Y_batches):
            num_iterations+=1
            optimizer.zero_grad()
            y_hat = model(x_batch)
            loss = loss_fn(y_hat,y_batch)
            train_loss.append(loss.item())
            loss.backward()
            optimizer.step()

            model.eval()
            parameters.append(model.weight.data.detach().cpu().numpy().
                squeeze().copy())
            with torch.no_grad():
                y_hat_eval = model(x_eval)
                loss_eval = loss_fn(y_hat_eval,y_eval)
                l = loss_eval.item()
                eval_loss.append(l)
                if (num_iterations) % 200 == 0:
                    print("Number␣of␣iterations:␣", num_iterations, "-␣Loss
                       ␣eval:", l)

                if l < 0.5 or num_iterations >= iterations_limit:
                    break
        if l < 0.5 or num_iterations >= iterations_limit:
          print("\n-------------------------------------")
          print("Training␣done␣after␣{}␣epochs,␣{}␣total␣iterations␣in␣time
              ␣{}␣ms".format(epoch+1,num_iterations, time.time() - start))
          print("Final␣evaluation␣loss:␣{}".format(l))
          break
```

After fine tuning the hyper-parameters and setting them initially as previously mentioned, I was able to converge to a solution in 54 epochs, 107 total iterations. More details about the convergence can be seen in the following image:



```
Initial evaluation loss:  805.0875634413622
-------------------------------------
Training done after 54 epochs, 107 total iterations in time 0.03499913215637207 ms
Final evaluation loss: 0.49695525509274047
Final weights: [-1.00131769e-01 -9.29665599e+00  1.06221153e+00 -1.11228342e+00
  5.21092860e-03]
```

Figure 4: Exercise 5 results

6. As a result of my model and the saved values for training and evaluation losses, we can see the following graph where the y-scale e set to a log scale so as to not see just a steep descent of the loss values.

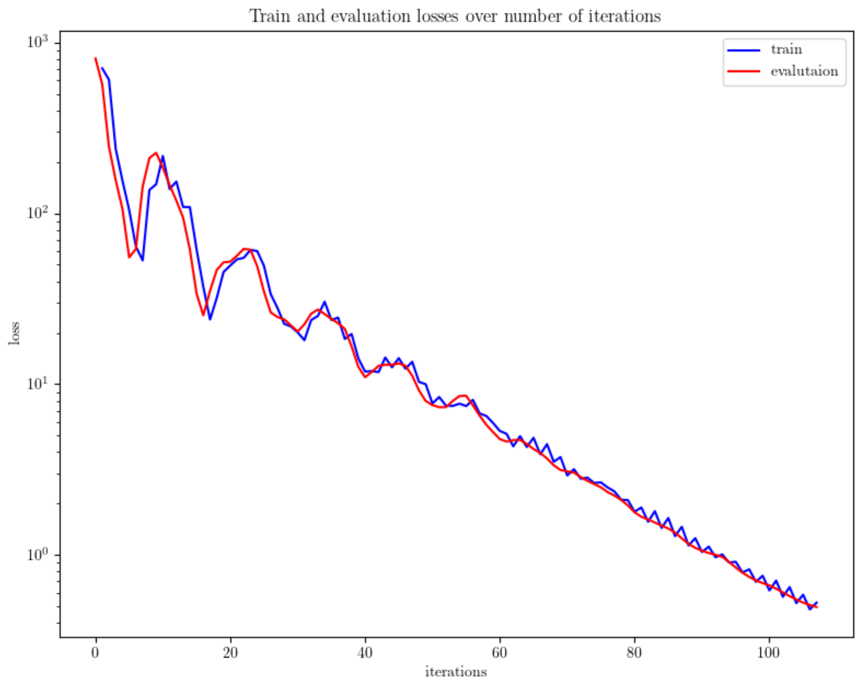


Figure 5: Train and evaluation losses (y in log scale)

It is worth noting that the unstable behaviour of the lines can be explained by the employment of mini-batches and the adjustment of the learning rate by the Adam algorithm.

Furthermore, if the y-scale were not logarithmic, we would see a steep descent in the first iterations and then a stabilization period towards the end.

The code employed to plot the losses was the following:

```
plt.plot(range(1,num_iterations+1),train_loss, color="blue", label="train")
plt.plot(range(0,num_iterations+1),eval_loss, color="red", label="
    evalutaion")
plt.title("Train␣and␣evaluation␣losses␣over␣number␣of␣iterations")
plt.xlabel('iterations')
plt.ylabel('loss')
plt.yscale("log")
plt.legend()
plt.show()
```

7. Because we saved all the computed parameters along the iterations of the model, the final result is now accessible as the last element of the list *parameters*. When plotting those against the original function in the interval defined in exercise 3 i got the following graph.

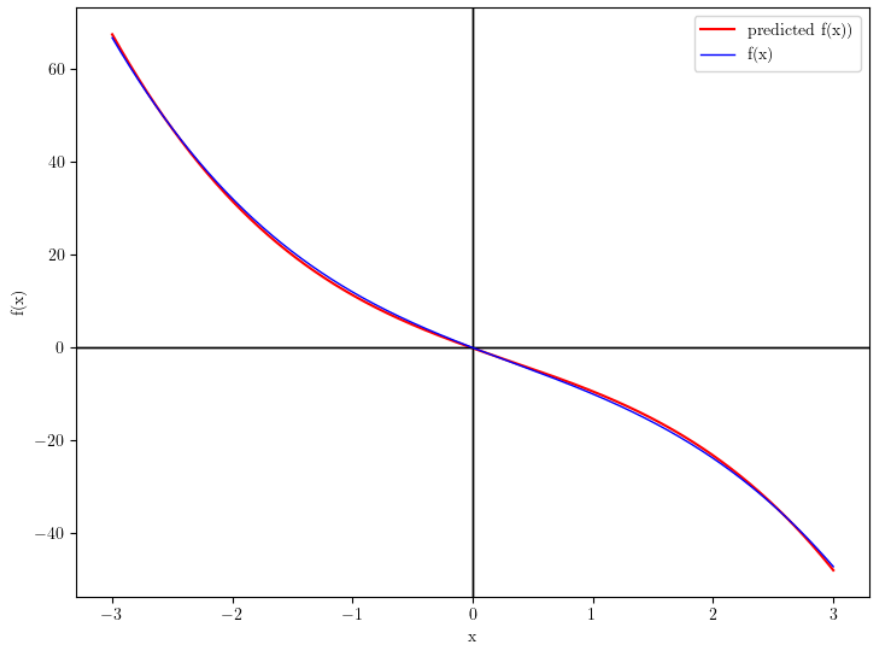


Figure 6: Original p(x) vs. model p(x)

The code employed to plot the losses was the following:

```
plot_polynomial(parameters[-1], z_range, color='r')
plot_polynomial(coeffs, z_range, color='b')
plt.legend()
plt.show()
```

8. With the aid of the *parameters* list I can effectively graph how each component changed throughout the iterations. In order to condense this information in a appealing five figured graph, I used *plt.subplots*() method, resulting in the following graph:

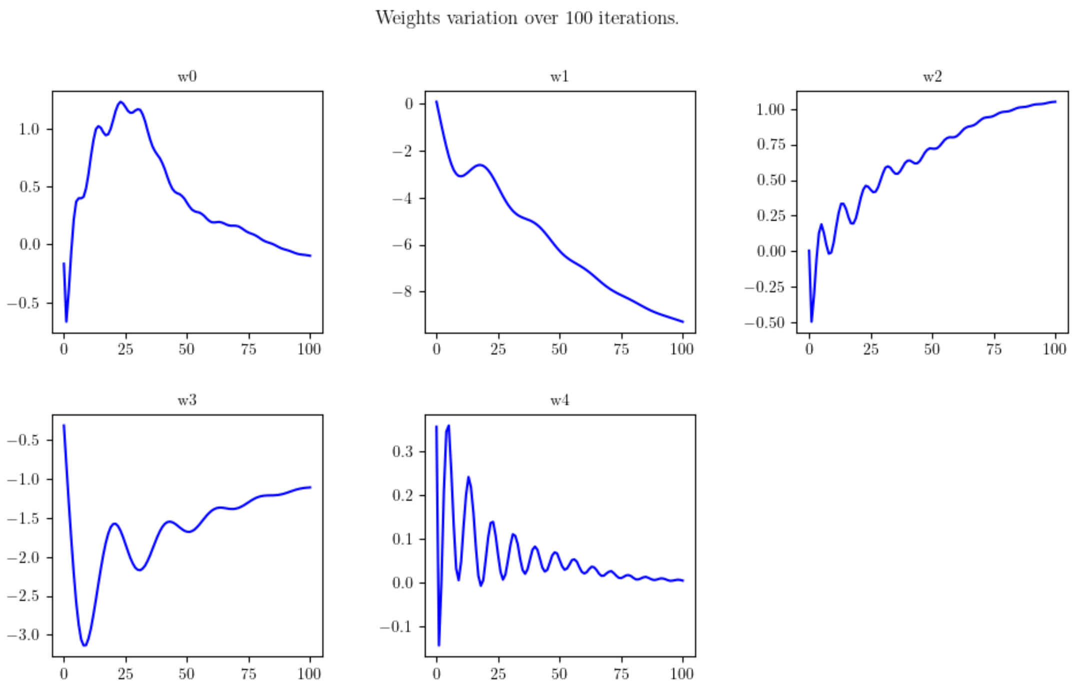


Figure 7: Model weights progression

Using the following code to obtain figure 7:

```
iterations_indices = np.arange(0,num_iterations+1)
num_cols = coeffs_size//2 + coeffs_size%2
fig, axs = plt.subplots(2, num_cols, squeeze=False)
fig.suptitle("Weights␣variation␣over␣"+str(num_iterations)+"␣iterations.")

for l in range(2):
    r = num_cols if (l+1)*num_cols <= coeffs_size else (coeffs_size%2)+1
    for c in range(num_cols):
        ax = axs[l,c]
        if(l*num_cols + c + 1 > coeffs_size):
            ax.set_visible(False)
        else:
            i = num_cols*l+c
            wi = np.array([vec[i] for vec in parameters], dtype=np.float64)
            ax = axs[l,c]
            ax.plot(iterations_indices, wi, color="blue")
            ax.set_title("w"+str(num_cols*l+c), fontsize=10)
fig.tight_layout()
plt.show()
```

9. For exercise 9 I copied the exact same code and hyper-parameters as stated in exercise 5, only changing the names of the variables and generating two data sets with the intended specifications for this exercise. In essence, the only difference relies on the following lines of code:

```
_, x_train_10, y_train_10 = create_dataset(coeffs,z_range,10,0.5,p_func,
    seed=0)
_, x_eval_10, y_eval_10 = create_dataset(coeffs,z_range,500,0.5,p_func,seed
    =1)
```

As a result, we obtained the following graphs for the training and evaluation losses, and the final model polynomial vs. the original one.

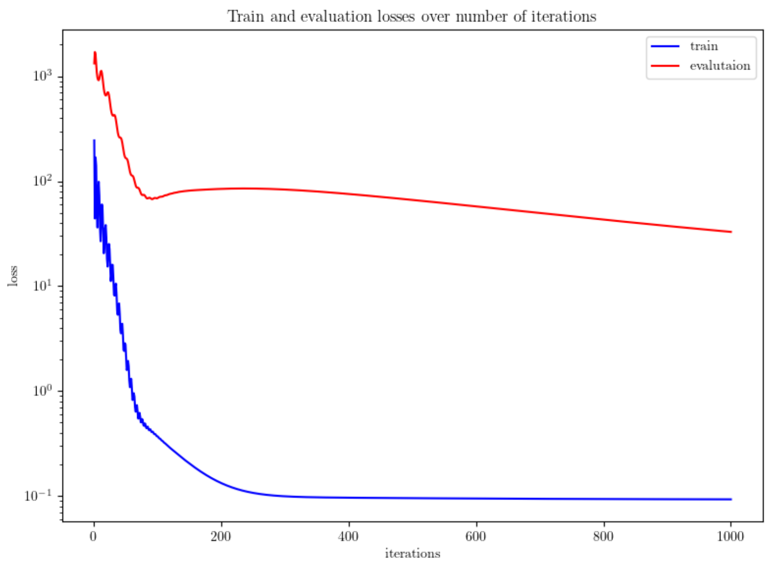


Figure 8: Train and evaluation losses of model_10 (y in log scale)

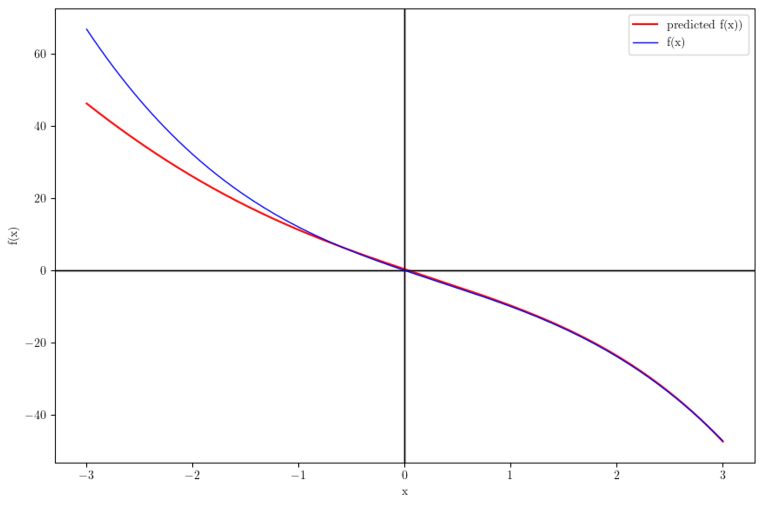


Figure 9: Original p(x) vs. model_10 p(x)

Because the data set used to train only has 10 training examples, the model is unable to generalize well for the intended function to learn. On the other hand, the evaluation data set has a substantial amount of examples which reflect the underlying structure of the intended graph.

Because of this, the model performs well on the training data set, having a considerate descending loss value throughout the iterations. But since these ten training examples don't represent well the intended function, the evaluation loss doesn't decrease as intended and even stagnates.

We can see that as a result of this the final function obtained through the model does not reflecting the original polynomial function.

10. In order to to use linear regression model to learn the sine function, some initial considerations must be taken. Namely, the way we will compute features that can represent the sine function. In order to achieve, taking into account that the sine function is analytical in $\mathbb{R}$, we can use the Taylor expansion. But then the questions arises: to what degree should we consider the polynomial that comes as a result of applying the Taylor expansion? In order to solve this, we can use Lagrange's Remainder Theorem and, given an error bound, effectively compute the k degree to which we can assure that the error is inferior to the error bound. Furthermore, for this exercise we're considering symmetric intervals given a positive integer a, which intuitively suggests that the Taylor expansion for the sine function should be done around $x = 0$, also known Mclaurin serie.

Using the Remainder theorem we know that

$$R_n(x) = \frac{f^{(k+1)}(c)}{(k+1)!}x^{k+1}, \text{ for a } c \text{ between } 0 \text{ and } x$$

.

If we want to bound the error, then we can simply take the absolute value of the remainder and find and appropriate majorant for the it. We know that $0 \leq |c| \leq a$ and $\left|f^{(k+1)}\right|(x) \leq 1$, hence we have:

$$|R_n(x)| = \left|\frac{f^{(k+1)}(c)}{(k+1)!}x^{k+1}\right| \leq \frac{5a^{k+1}}{(k+1)!}$$

The error bound represents a transcendental function of k so we can't solve it analytically for k given an maximum error allowed. But because the denominator has a factorial, it is easy to compute iteratively which k can satisfy the condition.

Given the prior facts, I implemented a function *get_min_k_degree*() that takes as arguments a and the maximum error allowed and returns the minimum k degree that satisfies the condition. It is important to note that this error bound was designed for the function $f(x) = 5\sin x + 3$

```
def get_min_k_degree(a,error_base_10=5e-1):
  error = np.log10(error_base_10)
  k = 1
  R = float(5*(a**(k+1))/(k+1))

  while(np.log10(R)>=error):
    k += 1
    R = (R*a)/(k+1)
  return k
```

I also implemented a function $sine\_func()$, similar to p_func, which given a dummy vector and the points in the x axis, computes the the features matrix to the degree of $2 \times len(dummy\_vector) - 1$ and the true value of $f(X)$, returning both these at the very end. It also worth taking note that because the expansion of the sin function iterates through odd powers and the bias doesn't appear in the Vandermonde matrix first column, some pre-processing of the matrix must be taken into account.

```
def sine_func(dummy_coeffs,Z):
  num_coffs = 2*len(dummy_sin_coeffs)-1
  vand_even_cols = np.vander(Z, N=num_coffs, increasing=True).astype(np.
      float64)[:,1::2]
  vand_even_cols = np.hstack((np.full((len(Z), 1), 3, dtype=np.float64),
      vand_even_cols))

  return vand_even_cols, (5*np.sin(Z)+3).astype(np.float64).reshape(-1,1)
```

We are now in conditions of proceeding to the implementation of the model. As in the prior exercises, we first define a set of conditions to which we want to run the model on, create the data sets and visualize them.

```
a = 0.01
a_z_range = [-a, a]
max_error = 5e-1
degree = get_min_k_degree(a,max_error)
degree = degree if degree%2 != 0 else degree+1
num_coeffs_sin = int((degree+1)/2 + 1)
dummy_sin_coeffs = np.empty((num_coeffs_sin,1))

Z_train_sin, x_train_sin, y_train_sin = create_dataset(dummy_sin_coeffs,
    a_z_range, 1000, 0.5, seed=0, func = sine_func)
Z_eval_sin, x_eval_sin, y_eval_sin = create_dataset(dummy_sin_coeffs,
    a_z_range, 500, 0.5, seed=1,  func = sine_func)

visualize_data(Z_train_sin,y_train_sin None,a_z_range,title="training␣
    dataset",func=sine_func)
visualize_data(Z_eval_sin,y_eval_sin,None,a_z_range,title="evaluation␣
    dataset",func=sine_func)
```

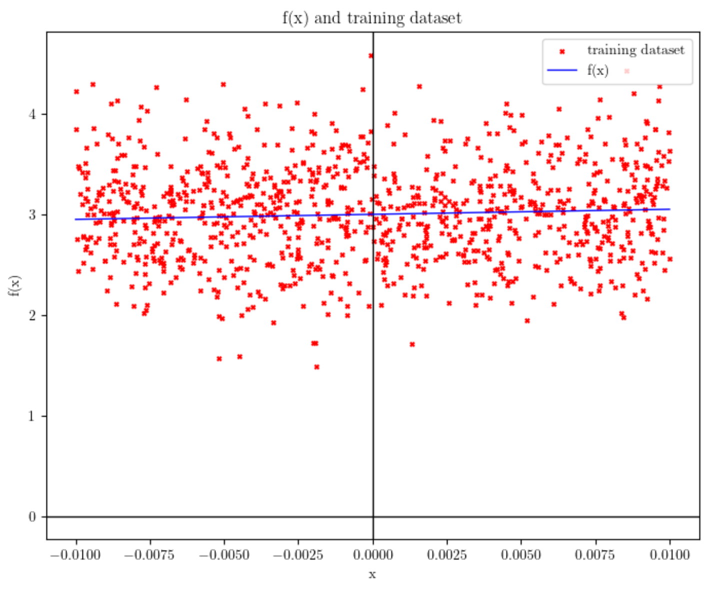


Figure 10: f(x) and training data (a=0.001)

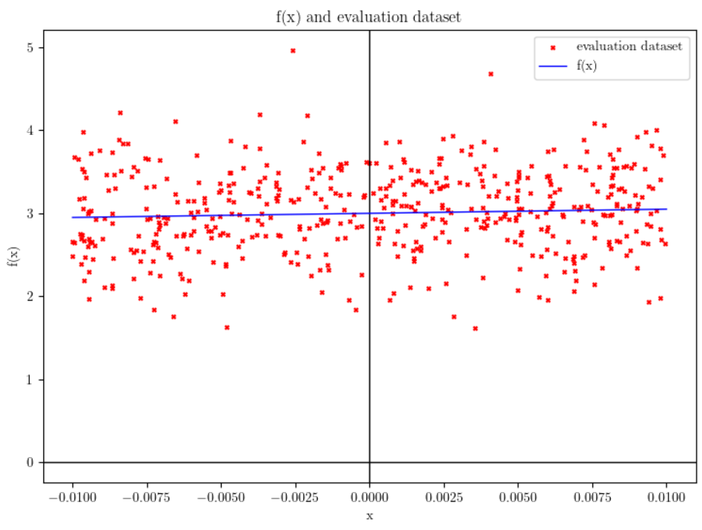


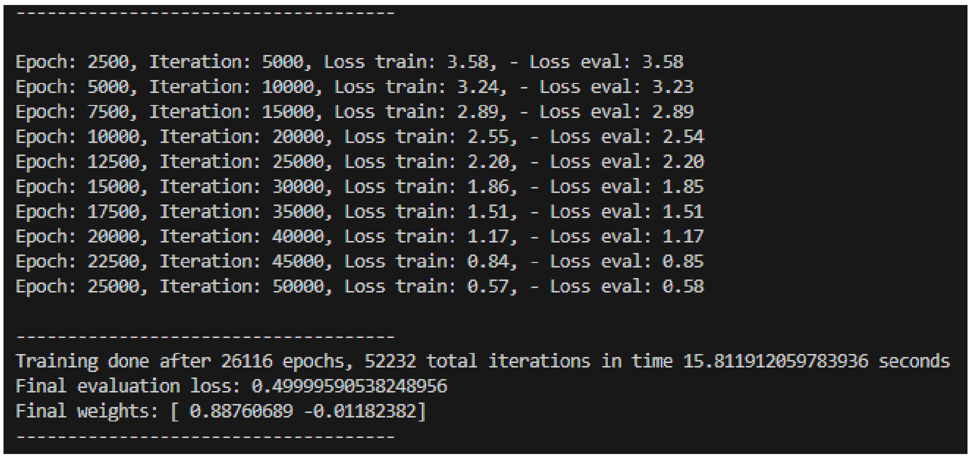
Figure 11: f(x) and evaluation data (a=0.001)

We now want to begin defining the hyper-parameters and the model itself. Unlike the prior exercises, here I had to employ the use of some weight decay, increase the number of maximum epochs and fine-tune the hyper-parameters to a greater degree. I also used the *nn.L1Loss* function because of the instability I was experiencing during gradient descent and how it reflected on the volatility of the training losses. The specifications that diverge from the prior exercises are the following (the rest follows the same procedure):

```
learning_rate_sin = 2.3e-5
w_decay = 4e-4
batch_size_sin = 500
num_epochs_sin = 32500

model_sin = nn.Linear(num_coeffs_sin,1,bias=False, dtype=torch.float64)
loss_fn_sin = nn.L1Loss(reduction='mean')
optimizer_sin = optim.Adam(model_sin.parameters(),lr=learning_rate_sin,
    weight_decay=w_decay)
```

After running the model on the training set I got the following stepping stones of information:



```
------------------------------------

Epoch: 2500, Iteration: 5000, Loss train: 3.58, - Loss eval: 3.58
Epoch: 5000, Iteration: 10000, Loss train: 3.24, - Loss eval: 3.23
Epoch: 7500, Iteration: 15000, Loss train: 2.89, - Loss eval: 2.89
Epoch: 10000, Iteration: 20000, Loss train: 2.55, - Loss eval: 2.54
Epoch: 12500, Iteration: 25000, Loss train: 2.20, - Loss eval: 2.20
Epoch: 15000, Iteration: 30000, Loss train: 1.86, - Loss eval: 1.85
Epoch: 17500, Iteration: 35000, Loss train: 1.51, - Loss eval: 1.51
Epoch: 20000, Iteration: 40000, Loss train: 1.17, - Loss eval: 1.17
Epoch: 22500, Iteration: 45000, Loss train: 0.84, - Loss eval: 0.85
Epoch: 25000, Iteration: 50000, Loss train: 0.57, - Loss eval: 0.58

------------------------------------
Training done after 26116 epochs, 52232 total iterations in time 15.811912059783936 seconds
Final evaluation loss: 0.49999590538248956
Final weights: [ 0.88760689 -0.01182382]
------------------------------------
```

Figure 12: Stepping stones information of training (a=0.001)

As we can see, compared to the polynomial function, convergence to a small error threshold is much harder. This convergence tends to be more difficult the bigger given interval [-a,a], which translates to a bigger kth degree.

The loss plots seem to descend quite evenly. At the very end we can notice a slight divergence from the training loss and evaluation loss plots, suggesting the some over fitting might have occurred.

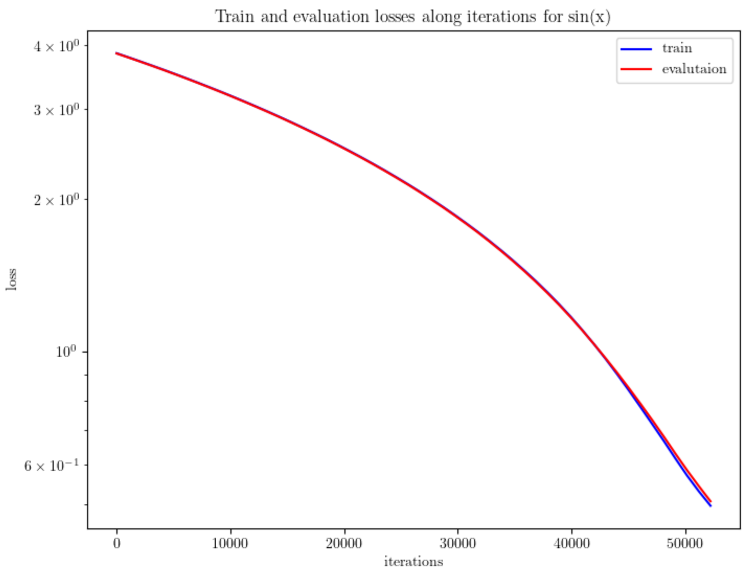


Figure 13: Train and evaluation losses of sin_model (y in log scale, a=0.001)

As a result of this model, we end with the following plots for $f(x)$ and the computed Taylor polynomial by the model.

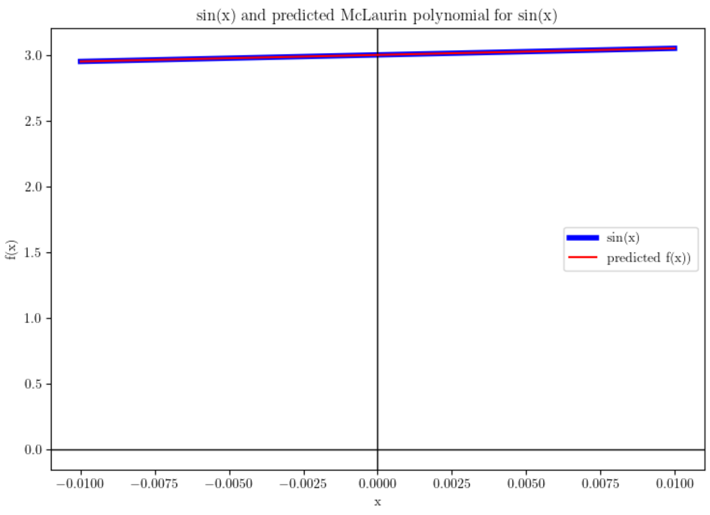


Figure 14: f(x) and sin_model function (a=0.001)

For the remainder of this exercise I will repeat the exact same procedure, only changing the value of a from 0.01 to 5.

Since the procedure is essentially the same, the only notes worth taking is that training was a lot harder compared to the scenario where a=0.001, and even though I couldn't meat the threshold for the evaluation loss value, the model still computed a good approximation to the function. The fact that the error for training and evaluation was didn't meet the threshold can be explained by looking at the points generated with noise. The model effectively learned the underlying structure of the $f(x)$ function. But because the generated points had so much noise when compared with $f(x)$, the threshold that error for the evaluation data did not account for this. Meeting the threshold could actually mean that the model was learning the noise instead of the underlying structure (overfitting). On the other hand, because we limited the polynomial degree and correctly chose a number of parameters to meet, it is difficult for the model to accommodate for the noise. So in reality, the only problem here was that the chosen train stopping error was unrealistic.

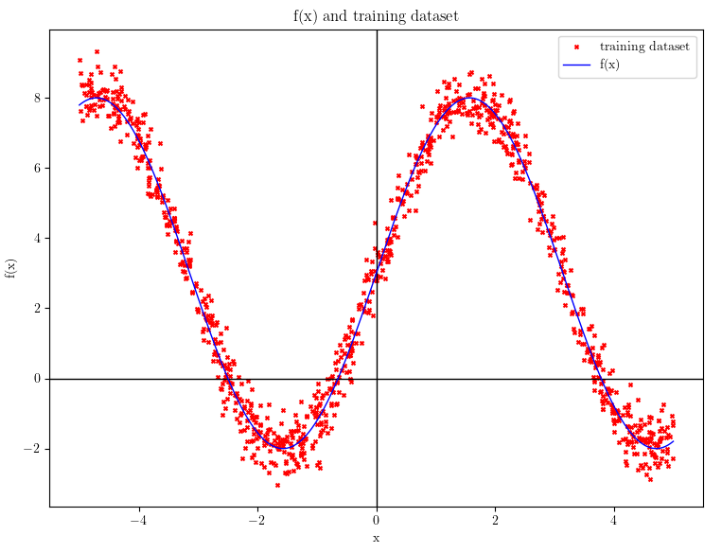


Figure 15: f(x) and training data (a=5)

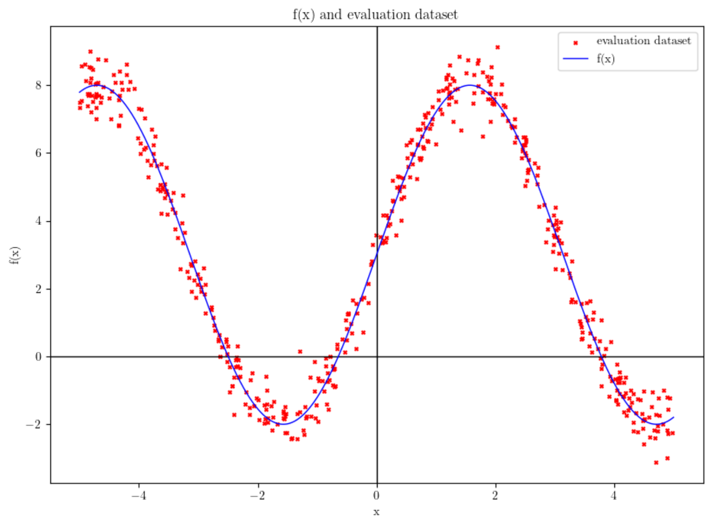
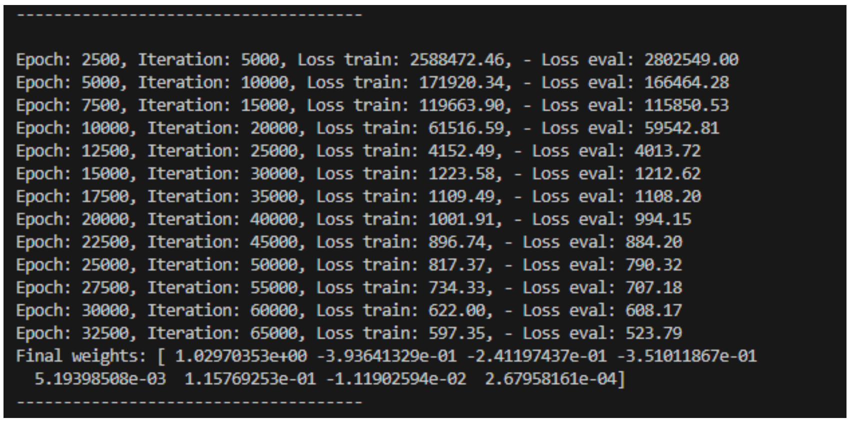


Figure 16: f(x) and training data (a=5)



```
------------------------------------

Epoch: 2500, Iteration: 5000, Loss train: 2588472.46, - Loss eval: 2802549.00
Epoch: 5000, Iteration: 10000, Loss train: 171920.34, - Loss eval: 166464.28
Epoch: 7500, Iteration: 15000, Loss train: 119663.90, - Loss eval: 115850.53
Epoch: 10000, Iteration: 20000, Loss train: 61516.59, - Loss eval: 59542.81
Epoch: 12500, Iteration: 25000, Loss train: 4152.49, - Loss eval: 4013.72
Epoch: 15000, Iteration: 30000, Loss train: 1223.58, - Loss eval: 1212.62
Epoch: 17500, Iteration: 35000, Loss train: 1109.49, - Loss eval: 1108.20
Epoch: 20000, Iteration: 40000, Loss train: 1001.91, - Loss eval: 994.15
Epoch: 22500, Iteration: 45000, Loss train: 896.74, - Loss eval: 884.20
Epoch: 25000, Iteration: 50000, Loss train: 817.37, - Loss eval: 790.32
Epoch: 27500, Iteration: 55000, Loss train: 734.33, - Loss eval: 707.18
Epoch: 30000, Iteration: 60000, Loss train: 622.00, - Loss eval: 608.17
Epoch: 32500, Iteration: 65000, Loss train: 597.35, - Loss eval: 523.79
Final weights: [ 1.02970353e+00 -3.93641329e-01 -2.41197437e-01 -3.51011867e-01
  5.19398508e-03  1.15769253e-01 -1.11902594e-02  2.67958161e-04]
------------------------------------
```

Figure 17: Stepping stones information of training (a=5)

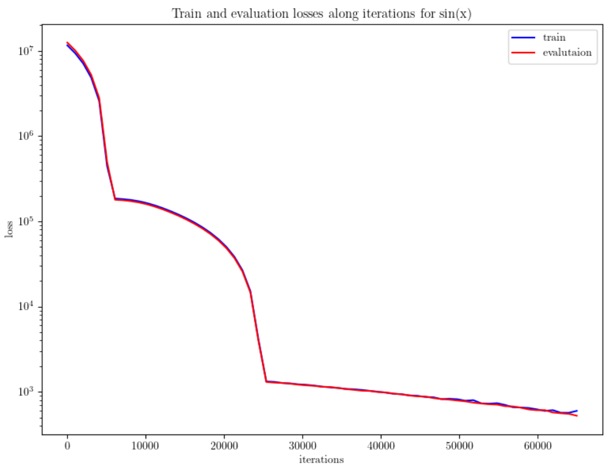


Figure 18: Train and evaluation losses of sin_model (y in log scale, a=5)

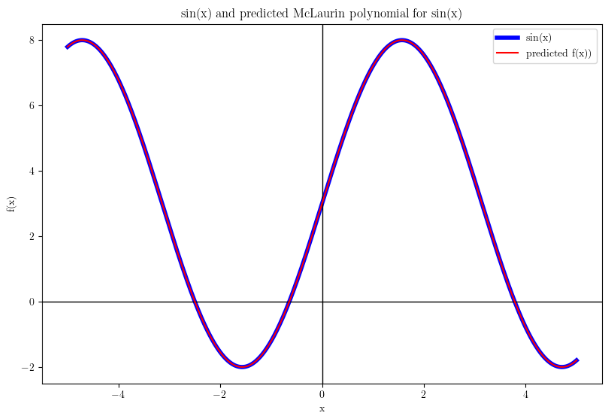


Figure 19: f(x) and sin_model function (a=5)

Questions

(a) Yes, it can be beneficial to choose different learning rates for the weights and the bias. When models run on a set of features of the data set, the features might have different scales, and therefore the gradient magnitudes for weights and bias could differ. Having different learning rates allows for adaptive learning for each of the said parameters. A good example of this stems from the previous exercise, where I included the bias in the weight vector. This is not ideal because the rate of change for the bias does not follow the rate of change for the features (e.g. $[t\ t^3\ ...\ t^{2n-1}]$).

(b) Choosing two different static learning rates means manually setting and keeping them constant throughout training. Adaptive methods, like AdaGrad, adjust the learning rates of all parameters dynamically based on the historical gradient values. AdaGrad, for instance, adapts the learning rates of all parameters by using the square root of the sum of the square past gradients. This allows parameters with frequent high gradients to have reduced learning rates and vice-versa, leading to a more nuanced and often faster convergence.