

# DLIP - Assignment 1: Training

Bradley Spronk - s2504758

February 15, 2026

## 1 Python data science refresher.

[Link of github repository to recent python projects](#). These python projects were based on two assignments for the course Computational Physics. In these projects I used packages such as pickle, numpy, numba, and matplotlib for data handling, efficient computation, and visualization. And they show my python skills in data science, including data manipulation, debugging, and visualization.

## 2 Tensorflow playground.

### Duarte Homework 1 Problem 4A

#### Linear model

It is not possible to fit the data with a linear model with only the features  $x_1$  and  $x_2$ , because the data is not linearly separable. Or in other words, we cannot draw a straight line that separates the two classes of points in the feature space defined by  $x_1$  and  $x_2$ .

However, if we add the feature  $x_1x_2$ , we can fit the data perfectly with a linear model. Because it allows us to capture the interaction between  $x_1$  and  $x_2$ , which is exactly what is necessary to separate the classes. The feature  $x_1x_2$  creates a new dimension in the feature space that allows us to draw a linear decision boundary that separates the two classes of points. [1]

#### ReLU activation model

With ReLU, a neural network with 1 hidden layer with 4 neurons is the smallest neural network that fits the training data perfectly (with a training loss of  $\leq 0.001$ ). The test loss is 0.001 and could go to 0 if we increase the number of epochs however in general we then risk overfitting. The configuration can be seen in Fig.1. In theory it should be possible to fit the data with a smaller network, but in practice it is not possible to find the right weights and biases to achieve that.[2] Additionally, the ReLU activation function can lead to dead neurons (neurons that output 0 for all inputs), which can further complicate the training process for smaller networks.

<https://playground.tensorflow.org/#activation=relu&batchSize=10&dataset=xor&regDataset=reg-plane&learningRate=0.03&regularizationRate=0&noise=0&networkShape=4&seed=0.14959&showTestData=false&discretize=false&percTrainData=50&x=true&y=true&xTimesY=false&xSquared=false&ySquared=false&cosX=false&sinX=false&cosY=false&sinY=false&collectStats=false&problem=classification&initZero=false&hideText=false>

*Note: We can't change the output neurons activation. For instance when there is no hidden layer and we choose the ReLU model ( $\max(0, x)$ ) the output still shows negative numbers. Possibly*

*a sigmoid function as the last neuron is being used to convert the output to a probability, but this is not explicitly stated in the playground.*

## Duarte Homework 1: Problem 4B

### Spiral dataset with all features

A neural network with 1 hidden layer with 6 neurons can fit the spiral dataset with all available features. This configuration can be seen in Fig.2. The training error is 0.000 and the test error is 0.010. This is a low bias/low variance model, because it fits the training data perfectly (low bias) and also generalizes well to the test data (low variance). However, as can be seen from the figure the decision boundary does not separate the classes in an intelligent way, which suggests that the model is not learning a meaningful representation of the data and is likely overfitting to the training data. Possible features to add to reduce the variance and improve the generalization of the model could be adding the features  $\sqrt{(x_1^2 + x_2^2)}$  or  $\arctan(x_2/x_1)$  to capture the radial and angular structure of the spiral dataset.

<https://playground.tensorflow.org/#activation=tanh&batchSize=10&dataset=spiral&regDataset=reg-plane&learningRate=0.03&regularizationRate=0&noise=0&networkShape=6&seed=0.78711&showTestData=false&discretize=false&percTrainData=50&x=true&y=true&xTimesY=false&xSquared=false&ySquared=false&cosX=false&sinX=true&cosY=false&sinY=true&collectStats=false&problem=classification&initZero=false&hideText=false>

### Spiral dataset with only $x_1$ and $x_2$ features

A neural network with 3 hidden layers with 8, 7, and 6 neurons respectively can fit the spiral dataset with only the features  $x_1$  and  $x_2$ . This configuration can be seen in Fig.3. The absence of additional features means that the neural network has to rely solely on the raw input features  $x_1$  and  $x_2$  to learn the decision boundaries, which can make it more challenging to fit the data and may require a larger network to achieve good performance. A spiral pattern is simply not linearly separable in the original feature space, so the network needs to have a more hidden layers to separate the classes than in the previous case. The test error is 0.036, which is even worse than before, and the decision boundary is even less intelligent with gaps in between the spiral arms and rays shooting out, which suggests that the model is likely overfitting to the training data and may not generalize well to unseen data. Large spikes in training/test error on the spiral dataset typically indicate unstable optimization, probaly due to an aggressive learning rate.

<https://playground.tensorflow.org/#activation=relu&batchSize=10&dataset=spiral&regDataset=reg-plane&learningRate=0.03&regularizationRate=0&noise=0&networkShape=8,7,6&seed=0.20569&showTestData=false&discretize=true&percTrainData=50&x=true&y=true&xTimesY=false&xSquared=false&ySquared=false&cosX=false&sinX=false&cosY=false&sinY=false&collectStats=false&problem=classification&initZero=false&hideText=false>

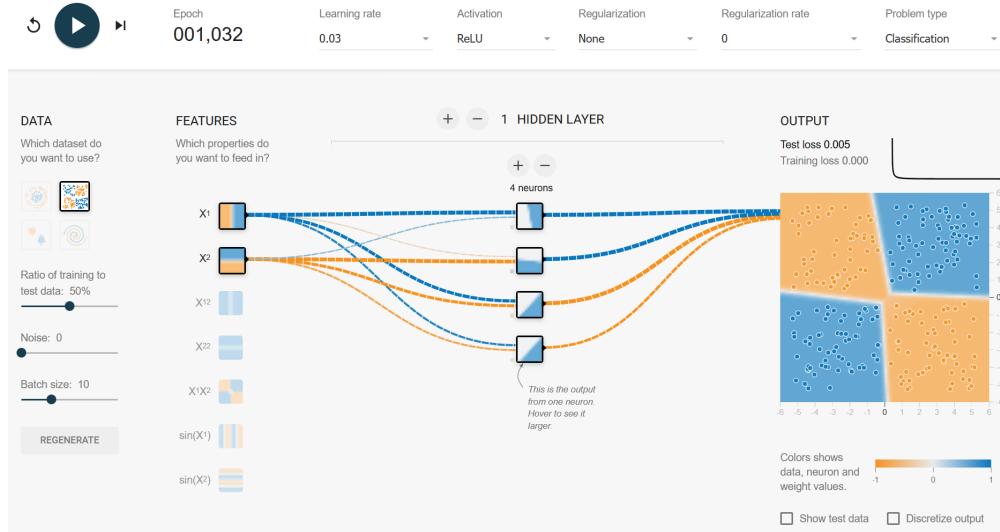


Figure 1: Screenshot[3] of the smallest neural network that fits the training data perfectly.

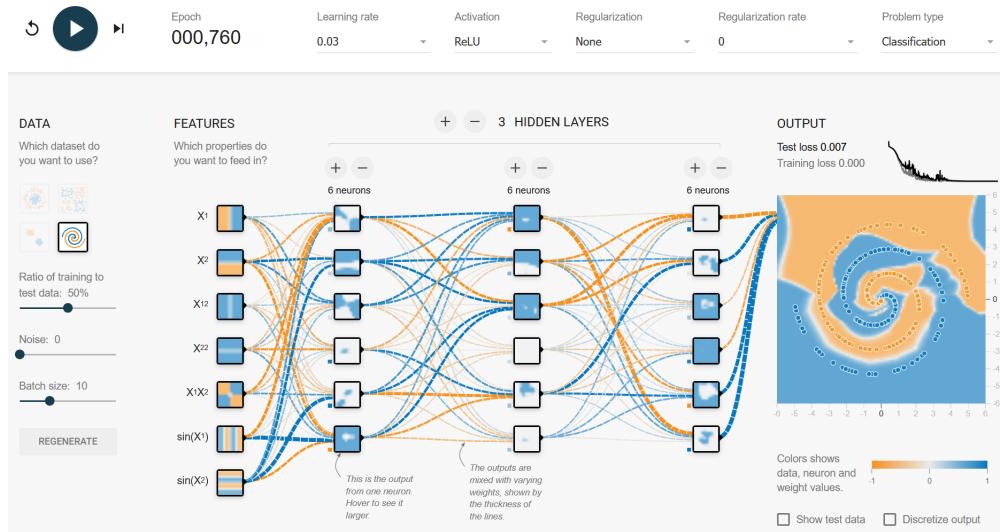


Figure 2: Screenshot[3] of a neural network that can fit a spiral dataset with all available features.

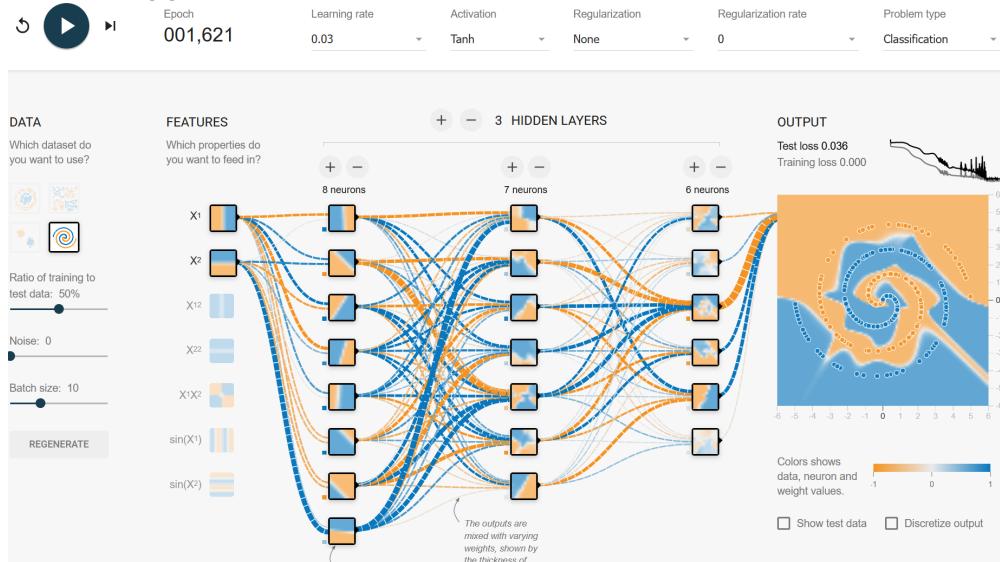


Figure 3: Screenshot[3] of a neural network that can fit a spiral dataset with only the features  $x_1$  and  $x_2$ .

### 3 Training curves for polynomial regression.

**Brightspace question about training error trends with epochs and data size**

With a small dataset the model can effectively memorize the training points, but as more data is added it must generalize across more variation, which can raise the training error while it learns a more faithful model of the data.

#### Duarte Homework 1 Problem 2

##### Problem 2 Part 1A

We can make our algebra and coding simpler by writing  $f(x_1, x_2, \dots, x_d) = \mathbf{w}^\top \mathbf{x}$  for vectors  $\mathbf{w}$  and  $\mathbf{x}$ . But at first glance, this formulation seems to be missing the bias term  $b$  from the equation above. How should we define  $\mathbf{x}$  and  $\mathbf{w}$  such that the model includes the bias term?

*Hint: Include an additional element in  $\mathbf{w}$  and  $\mathbf{x}$*

**Solution:**

We can define vectors  $\mathbf{x}$  and  $\mathbf{w}$  as:

$$\mathbf{x} = [x_1, x_2, \dots, x_d, 1], \quad \mathbf{w} = [w_1, w_2, \dots, w_d, \text{bias}] \quad (1)$$

So that the dot product gives:

$$\mathbf{w}^\top \mathbf{x} = w_1 x_1 + w_2 x_2 + \dots + w_d x_d + 1 \cdot \text{bias} \quad (2)$$

##### Problem 2 Part 1B: Gradient of the Squared Loss

For a single data point  $(\mathbf{x}, y)$ , the squared loss is:

$$L(\mathbf{w}) = (y - \mathbf{w}^\top \mathbf{x})^2 \quad (3)$$

The gradient of  $L$  with respect to  $\mathbf{w}$  is obtained using the chain rule:

$$\nabla_{\mathbf{w}} L(\mathbf{w}) = \frac{\partial}{\partial \mathbf{w}} (y - \mathbf{w}^\top \mathbf{x})^2 \quad (4)$$

$$= 2(y - \mathbf{w}^\top \mathbf{x}) \cdot \frac{\partial}{\partial \mathbf{w}} (y - \mathbf{w}^\top \mathbf{x}) \quad (5)$$

$$= 2(y - \mathbf{w}^\top \mathbf{x}) \cdot (-\mathbf{x}) \quad (6)$$

$$= -2(y - \mathbf{w}^\top \mathbf{x})\mathbf{x} \quad (7)$$

# Training curves for polynomial regression

February 15, 2026

## 1 Problem 2

```
[2]: # Setup:  
import numpy as np  
import matplotlib.pyplot as plt
```

```
[3]: from sklearn.metrics import mean_squared_error  
from sklearn.model_selection import KFold  
  
import warnings  
  
warnings.filterwarnings("ignore")  
  
# Fix random seed for reproducibility  
np.random.seed(42)
```

### 1.1 Loading the Data for Problem 2

This code loads the data from `bv_data.csv` using the `load_data` helper function. Note that `data[:, 0]` is an array of all the  $x$  values in the data and `data[:, 1]` is an array of the corresponding  $y$  values.

```
[7]: def load_data(filename):  
    """  
        Function loads data stored in the file filename and returns it as a numpy  
        ndarray.  
        Input:  
            filename: given as a string.  
        Output:  
            Data contained in the file, returned as a numpy ndarray  
    """  
    return np.loadtxt(filename, skiprows=1, delimiter=",")
```

```
[8]: data = load_data("../data/bv_data.csv")  
x = data[:, 0]  
y = data[:, 1]
```

## 1.2 Problem 2, Parts C-H

### Algorithm (Learning Curves)

1. For each degree  $d \in \{1, 2, 6, 12\}$  and each training size  $N$ , take the first  $N$  points.
2. Split those  $N$  points into 5 contiguous folds.
3. For each fold, fit a degree- $d$  polynomial on the training folds and compute training and validation MSE.
4. Average the 5 fold errors to get the mean training and validation MSE for that  $N$ .
5. Repeat across all  $N$  values to build the learning curves.

```
[17]: degrees = [1, 2, 6, 12]           # Polynomial degrees to evaluate
N_values = np.arange(20, 101, 5) # Training set sizes: 20, 25, ..., 100
num_folds = 5                   # 5-fold cross-validation

# We will store a list of training errors and validation errors for each degree.
learning_curves = {}

# Store polynomial coefficients for each degree and N (all folds).
fit_store = {degree: {} for degree in degrees}

for degree in degrees:
    # Initialize lists for this degree's learning curve
    train_errors = []
    val_errors = []

    # Loop over different training set sizes N
    for N in N_values:
        # Select only the first N points (as instructed)
        x_subset = x[:N]
        y_subset = y[:N]

        # Build a KFold splitter with contiguous blocks (no shuffle)
        kf = KFold(n_splits=num_folds)

        # Accumulators for fold errors and fitted coefficients
        fold_train_errors = []
        fold_val_errors = []
        fold_coeffs = []

        # Iterate through each fold
        for train_index, val_index in kf.split(x_subset):
            # Split the data into train/validation for this fold
            x_train, x_val = x_subset[train_index], x_subset[val_index]
            y_train, y_val = y_subset[train_index], y_subset[val_index]

            # Fit the polynomial model on the training data (degree d)
            coeffs = np.polyfit(x_train, y_train, degree)

            # Add error and coefficient to accumulators
            fold_train_errors.append(np.mean((np.polyval(coeffs, x_train) - y_train)**2))
            fold_val_errors.append(np.mean((np.polyval(coeffs, x_val) - y_val)**2))
            fold_coeffs.append(coeffs)

    # Add the accumulated errors and coefficients to the main dictionary
    fit_store[degree][N] = {'train_errors': fold_train_errors, 'val_errors': fold_val_errors, 'coeffs': fold_coeffs}
```

```

fold_coeffs.append(coeffs)

# Evaluate the fitted polynomial on train and validation inputs
y_train_pred = np.polyval(coeffs, x_train)
y_val_pred = np.polyval(coeffs, x_val)

# Compute MSE for train and validation and store them
fold_train_errors.append(mean_squared_error(y_train, y_train_pred))
fold_val_errors.append(mean_squared_error(y_val, y_val_pred))

# Average the errors across folds for this N
train_errors.append(np.mean(fold_train_errors))
val_errors.append(np.mean(fold_val_errors))

# Save fitted coefficients for this degree and N
fit_store[degree][N] = fold_coeffs

# Save this degree's learning curve results
learning_curves[degree] = {
    "N": N_values,
    "train": np.array(train_errors),
    "val": np.array(val_errors)
}

```

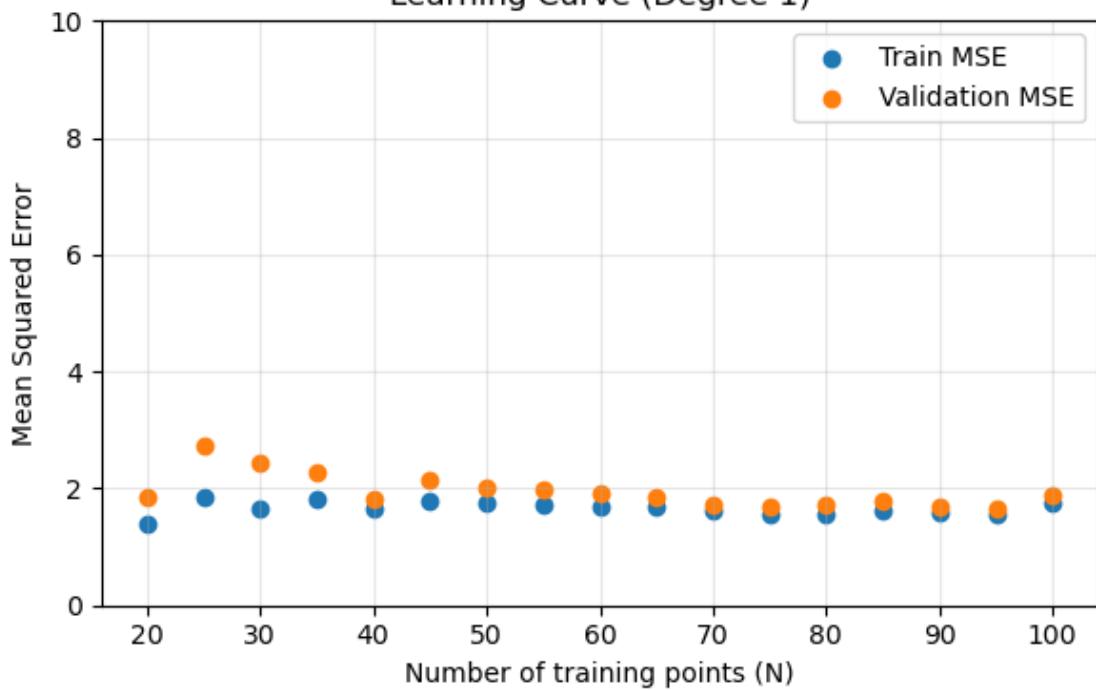
[18]: #Plot one learning curve per degree

```

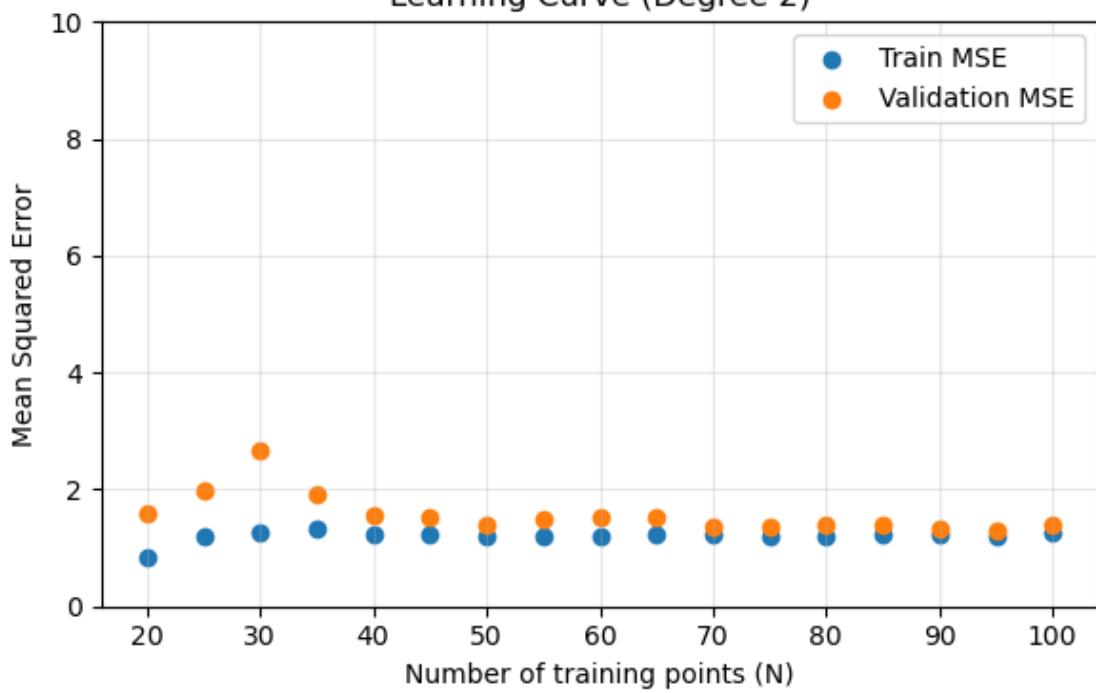
for degree in degrees:
    plt.figure(figsize=(6, 4))
    plt.scatter(learning_curves[degree]["N"], learning_curves[degree]["train"], label="Train MSE")
    plt.scatter(learning_curves[degree]["N"], learning_curves[degree]["val"], label="Validation MSE")
    plt.xlabel("Number of training points (N)")
    plt.ylabel("Mean Squared Error")
    plt.title(f"Learning Curve (Degree {degree})")
    plt.ylim(0,10)
    plt.legend()
    plt.grid(True, alpha=0.3)
    plt.tight_layout()
    plt.show()

```

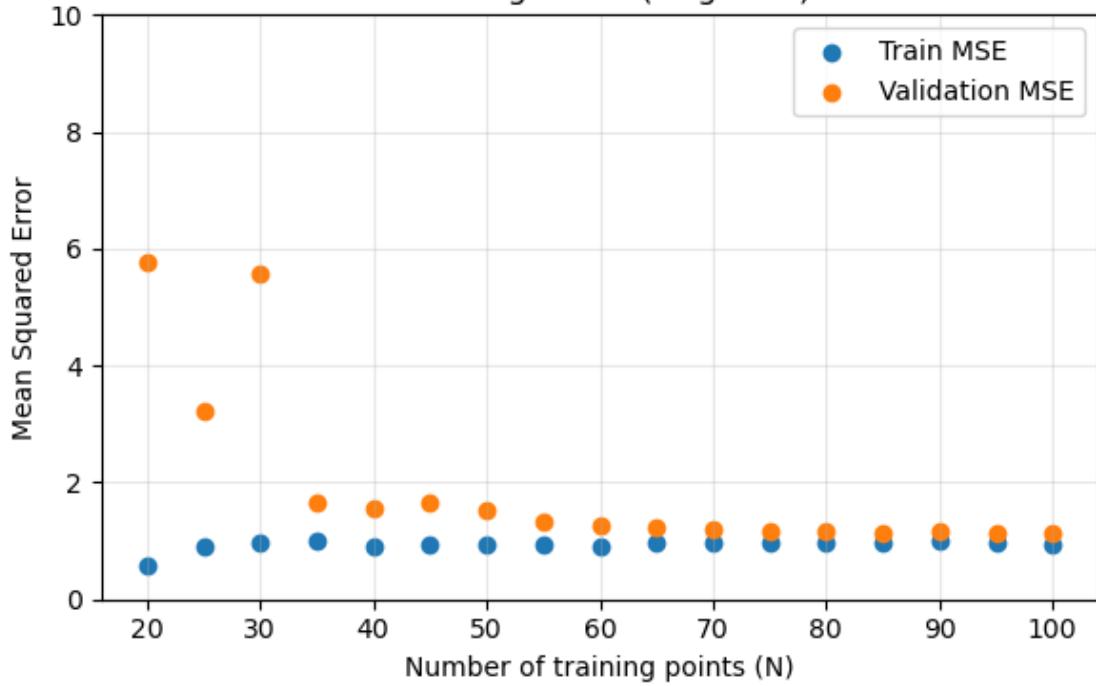
Learning Curve (Degree 1)



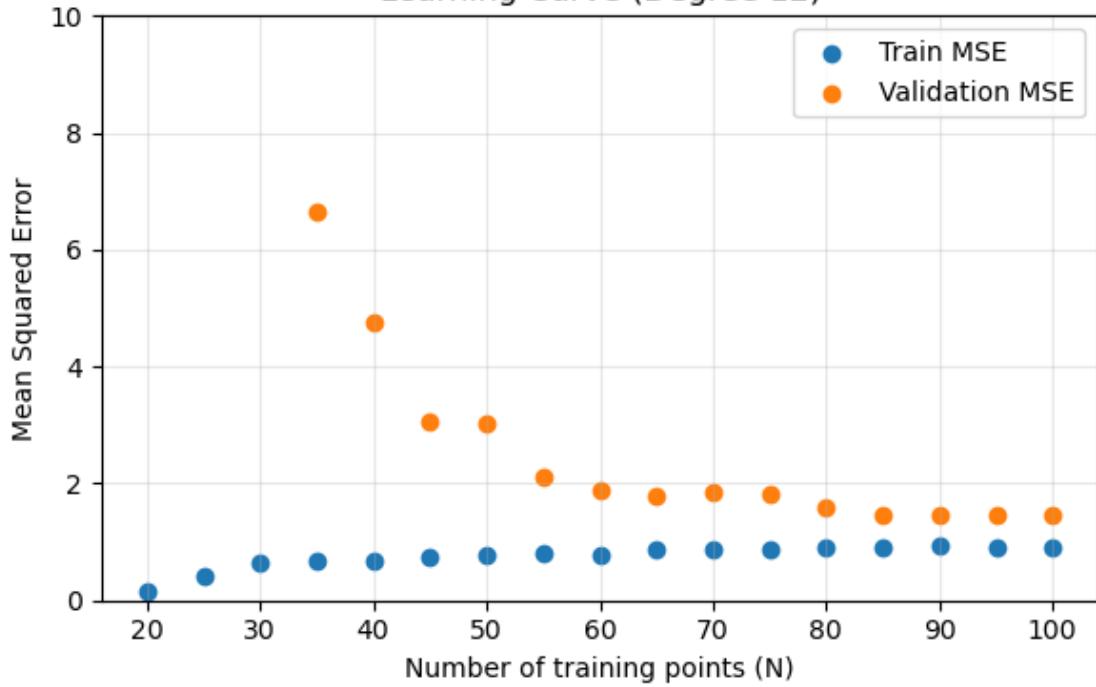
Learning Curve (Degree 2)



Learning Curve (Degree 6)



Learning Curve (Degree 12)



### 1.3 Problem D

Q: Based on the learning curves, which polynomial regression model (i.e. which degree polynomial) has the highest bias? How can you tell?

The highest bias is for the first model (1 degree) as it has the highest train MSE and it's consistent for all number of training points, in other words equally wrong for all data sets.

### 1.4 Problem E

Q: Which model has the highest variance? How can you tell?

The degree-12 model shows the highest variance. It achieves very low training MSE but much higher validation MSE, especially for small  $N$ , and the gap between the two errors is much larger than for the lower-degree models.

### 1.5 Problem F

Q: What does the learning curve of the quadratic model tell you about how much the model will improve if we had additional training points?

The quadratic model's learning curve flattens out, so adding more training points is unlikely to change the errors much. Both training and validation MSE plateau after the first part of the data.

### 1.6 Problem G

Q: Why is training error generally lower than validation error?

The training error is generally lower because that's exactly what your model is supposed to bring down. The validation error only goes down once your model actually generalizes your data well.

### 1.7 Problem H

Q: Based on the learning curves, which model would you expect to perform best on some unseen data drawn from the same distribution as the training data, and why?

I would go with degree 2. Anything higher (6 or 12) looks like overfitting given how noisy the data are, and the curves start to behave poorly outside the range we actually observe. The quadratic fit captures the main trend without getting too wiggly, while a linear model clearly underfits.

#### 1.7.1 Extra plots to visualize the models performance and under- or overfitting

```
[27]: # Plot fold fits as subplots: rows = degrees, columns = N values
plot_N_values = [20, 50, 100] # Choose N values to visualize
x_plot = np.linspace(x.min(), x.max(), 400) # Smooth x-grid for curves

fig, axes = plt.subplots(
    nrows=len(degrees),
    ncols=len(plot_N_values),
    figsize=(4.2 * len(plot_N_values), 3.2 * len(degrees)),
    sharex=True,
    sharey=True
```

```

)
for i, degree in enumerate(degrees):
    for j, plot_N in enumerate(plot_N_values):
        ax = axes[i, j]
        x_subset = x[:plot_N]
        y_subset = y[:plot_N]

        # Pull all saved fold fits for this degree and N
        fold_coeffs = fit_store[degree][plot_N]

        # Plot data and all fold fits for this degree
        ax.scatter(x, y, s=12, alpha=0.35, label="All data" if (i == 0 and j == 0) else None)
        ax.scatter(x_subset, y_subset, s=10, label=f"Training data" if i == 0 else None)

        # Plot each fold's fitted curve with low opacity
        y_folds = []
        for coeffs in fold_coeffs:
            y_curve = np.polyval(coeffs, x_plot)
            y_folds.append(y_curve)
            ax.plot(x_plot, y_curve, color="red", alpha=0.2)

        # Plot the average fitted curve across folds
        y_mean = np.mean(np.vstack(y_folds), axis=0)
        ax.plot(x_plot, y_mean, color="red", linewidth=2, label=f"Degree {degree} (mean)" if j == 0 else None)

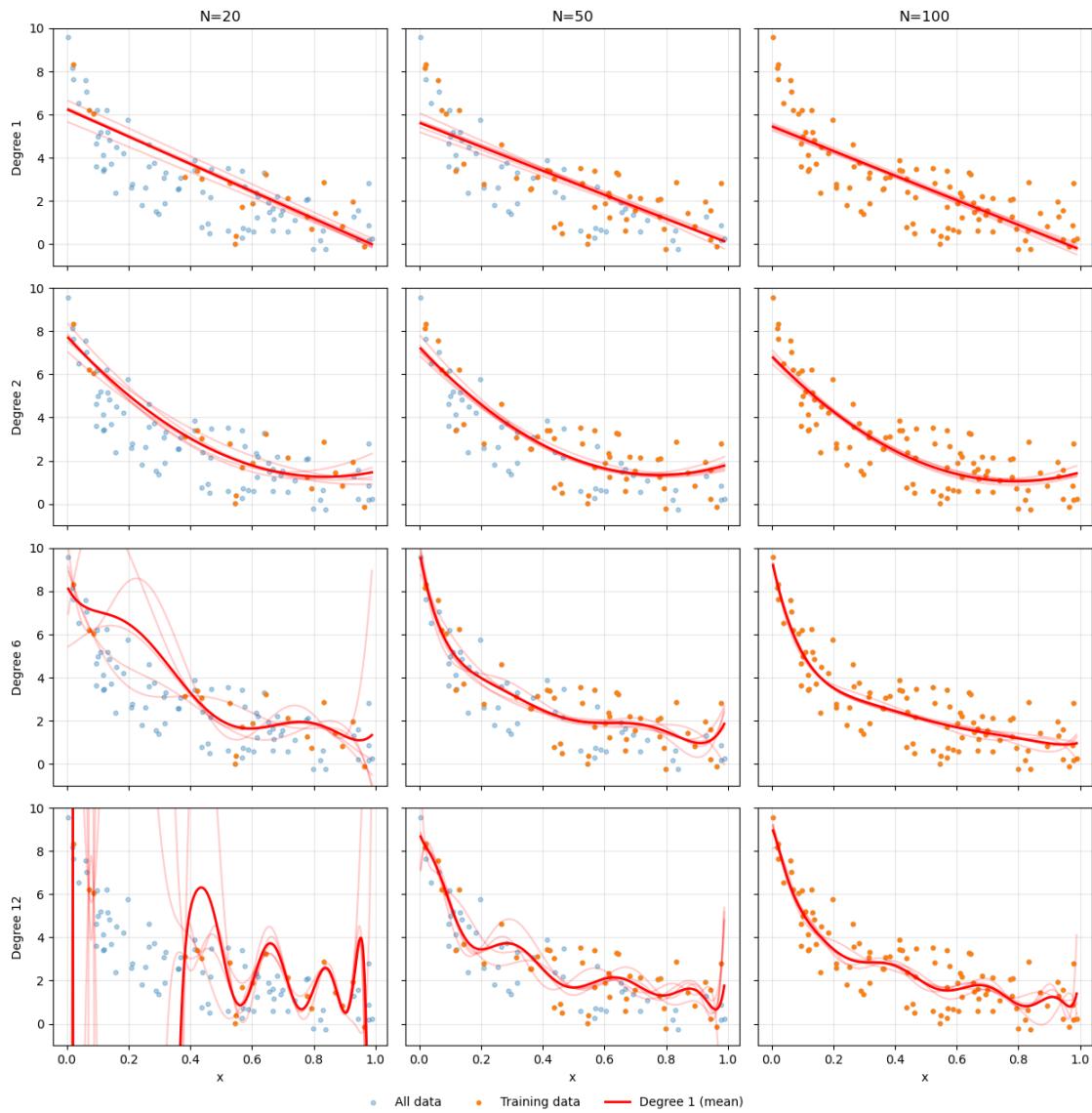
        ax.set_ylim(-1, 10)

        if i == 0:
            ax.set_title(f"N={plot_N}")
        if j == 0:
            ax.set_ylabel(f"Degree {degree}")
        if i == len(degrees) - 1:
            ax.set_xlabel("x")
        ax.grid(True, alpha=0.3)

# Build a single legend for the whole figure
handles, labels = axes[0, 0].get_legend_handles_labels()
fig.legend(handles, labels, loc="lower center", ncol=3, frameon=False, bbox_to_anchor=(0.5, -0.02))
fig.suptitle("Fold Fits Across Degrees and N", y=1.02)
fig.tight_layout()
plt.show()

```

Fold Fits Across Degrees and N



# Stochastic gradient descent for linear regression

February 15, 2026

## 1 Problem 1, Parts C-E: Stochastic Gradient Descent Visualization

```
[42]: import numpy as np
import matplotlib.pyplot as plt
from IPython.display import Image

from sgd_helper import (
    generate_dataset1,
    generate_dataset2,
    plot_dataset,
    plot_loss_function,
    animate_convergence,
    animate_sgd_suite,
)
```

### 1.1 Problem 1C: Implementation of SGD

Fill in the loss, gradient, and SGD functions according to the guidelines given in the problem statement in order to perform SGD.

```
[43]: def loss(X, Y, w):
    """
    Calculate the squared loss function.

    Inputs:
        X: A (N, D) shaped numpy array containing the data points.
        Y: A (N, ) shaped numpy array containing the (float) labels of the data points.
        w: A (D, ) shaped numpy array containing the weight vector.

    Outputs:
        The loss evaluated with respect to X, Y, and w.
    """

    Loss = 0
    N = X.shape[0]
```

```

for i in range(N):
    Loss += (Y[i] - np.dot(w, X[i])) ** 2

return Loss / N

def gradient(x, y, w):
    """
    Calculate the gradient of the loss function with respect to
    a single point (x, y), and using weight vector w.

    Inputs:
        x: A (D, ) shaped numpy array containing a single data point.
        y: The float label for the data point.
        w: A (D, ) shaped numpy array containing the weight vector.

    Output:
        The gradient of the loss with respect to x, y, and w.
    """
    return -2 * (y - np.dot(w, x)) * x

def SGD(X, Y, w_start, eta, N_epochs):
    """
    Perform SGD using dataset (X, Y), initial weight vector w_start,
    learning rate eta, and N_epochs epochs.

    Inputs:
        X: A (N, D) shaped numpy array containing the data points.
        Y: A (N, ) shaped numpy array containing the (float) labels of the data
           points.
        w_start: A (D, ) shaped numpy array containing the weight vector
           initialization like.
        eta: The step size.
        N_epochs: The number of epochs (iterations) to run SGD.

    Outputs:
        w: A (N_epochs, D) shaped array containing the weight vectors from all
           iterations.
        losses: A (N_epochs, ) shaped array containing the losses from all
           iterations.
    """
    W = np.zeros((N_epochs, len(w_start)))
    losses = np.zeros(N_epochs)

    w = w_start.copy()

```

```

for epoch in range(N_epochs):
    W[epoch] = w.copy()
    losses[epoch] = loss(X, Y, w)

    # Shuffle the data points
    indices = np.random.permutation(X.shape[0])
    X_shuffled = X[indices]
    Y_shuffled = Y[indices]

    for i in range(X.shape[0]):
        x_i = X_shuffled[i]
        y_i = Y_shuffled[i]
        grad = gradient(x_i, y_i, w)
        w -= eta * grad

return W, losses

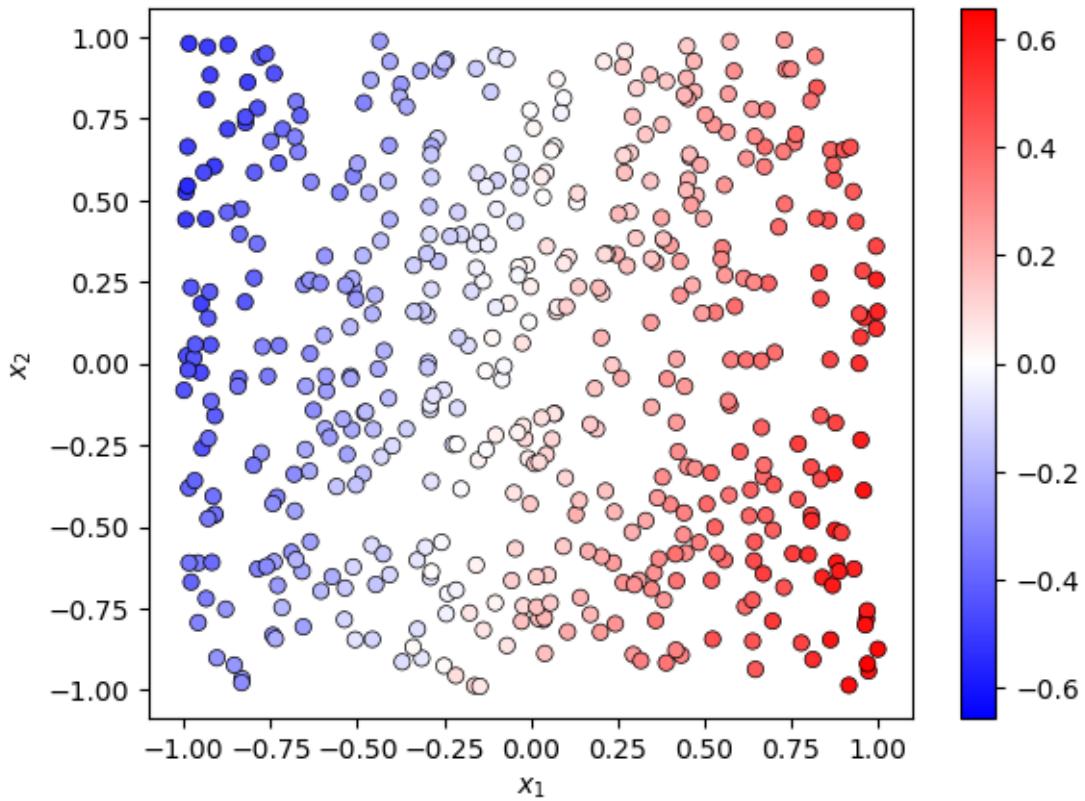
```

## 1.2 Problem 1D: Visualization

### 1.2.1 Dataset

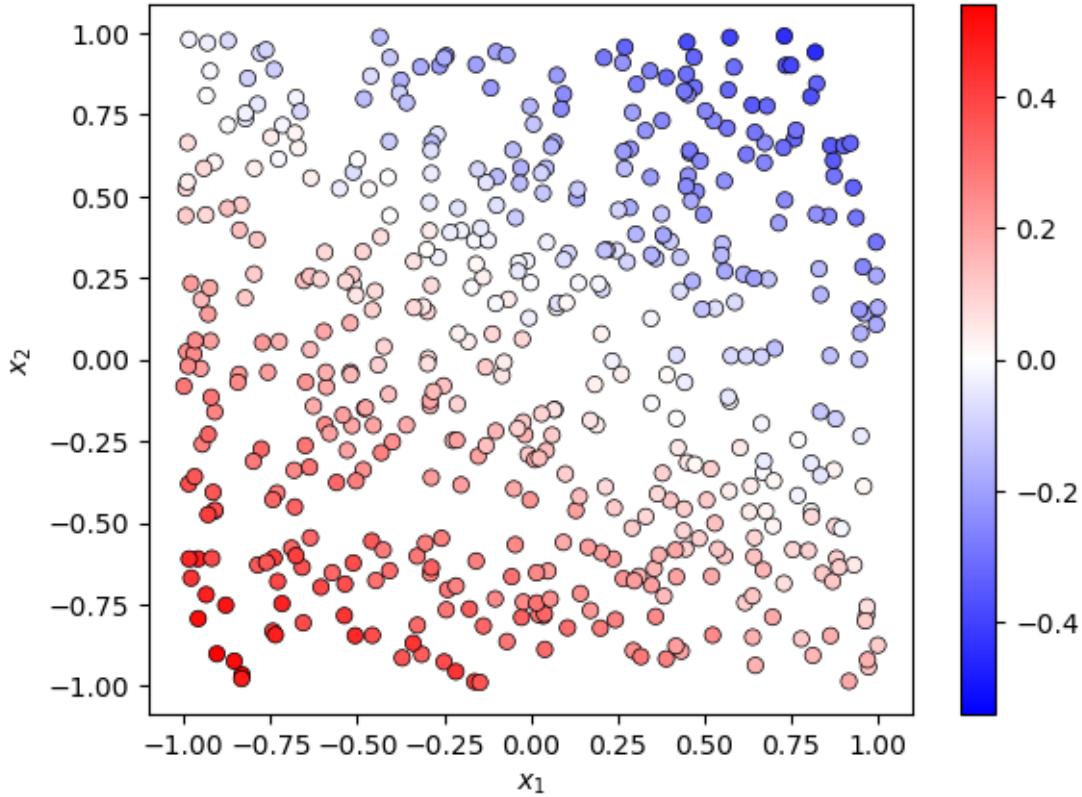
We'll start off by generating two simple 2-dimensional datasets. For simplicity, we do not consider separate training and test sets.

[44]: X1, Y1 = generate\_dataset1()  
plot\_dataset(X1, Y1)



[44]: (<Figure size 640x480 with 2 Axes>, <Axes: xlabel='\\$x\\_1\$', ylabel='\\$x\\_2\$'>)

[45]: X2, Y2 = generate\_dataset2()  
plot\_dataset(X2, Y2)



[45]: (<Figure size 640x480 with 2 Axes>, <Axes: xlabel='\\$x\\_1\$', ylabel='\\$x\\_2\$'>)

### 1.2.2 SGD from a single point

First, let's visualize SGD from a single starting point:

Let's view how the weights change as the algorithm converges:

```
[ ]: # Parameters to feed the SGD.
params = ({"w_start": [0.01, 0.01], "eta": 0.00001},)
N_epochs = 1000
FR = 20

# Let's do it!
W, _ = SGD(X1, Y1, params[0]["w_start"], params[0]["eta"], N_epochs)
anim = animate_convergence(X1, Y1, W, FR)
anim.save("animation2.gif", fps=30, writer="imagemagick")
Image(open("animation2.gif", "rb").read())
```

### 1.2.3 SGD from multiple points

Now, let's visualize SGD from multiple arbitrary starting points:

```
[ ]: # Parameters to feed the SGD.
# Here, we specify each different set of initializations as a dictionary.
params = (
    {"w_start": [-0.8, -0.3], "eta": 0.00001},
    {"w_start": [-0.9, 0.4], "eta": 0.00001},
    {"w_start": [-0.4, 0.9], "eta": 0.00001},
    {"w_start": [0.8, 0.8], "eta": 0.00001},
)
N_epochs = 1000
FR = 20

# Let's go!
anim = animate_sgd_suite(SGD, loss, X1, Y1, params, N_epochs, FR)
anim.save("animation3.gif", fps=30, writer="imagemagick")
Image(open("animation3.gif", "rb").read())
```

Let's do the same thing but with a different dataset:

```
[ ]: # Parameters to feed the SGD.
params = (
    {"w_start": [-0.8, -0.3], "eta": 0.00001},
    {"w_start": [-0.9, 0.4], "eta": 0.00001},
    {"w_start": [-0.4, 0.9], "eta": 0.00001},
    {"w_start": [0.8, 0.8], "eta": 0.00001},
)
N_epochs = 1000
FR = 20

# Animate!
anim = animate_sgd_suite(SGD, loss, X2, Y2, params, N_epochs, FR)
anim.save("animation4.gif", fps=30, writer="imagemagick")
Image(open("animation4.gif", "rb").read())
```

### 1.3 Problem 1E: SGD with different step sizes

Now, let's visualize SGD with different step sizes (eta):

(For ease of visualization: the trajectories are ordered from left to right by increasing eta value. Also, note that we use smaller values of N\_epochs and FR here for easier visualization.)

```
[ ]: # Parameters to feed the SGD.
params = (
    {"w_start": [0.7, 0.8], "eta": 0.00001},
    {"w_start": [0.2, 0.8], "eta": 0.00005},
    {"w_start": [-0.2, 0.7], "eta": 0.0001},
    {"w_start": [-0.6, 0.6], "eta": 0.0002},
)
N_epochs = 100
```

```

FR = 2

# Go!
anim = animate_sgd_suite(SGD, loss, X1, Y1, params, N_epochs, FR, ms=2)
anim.save("animation5.gif", fps=30, writer="imagemagick")
Image(open("animation5.gif", "rb").read())

```

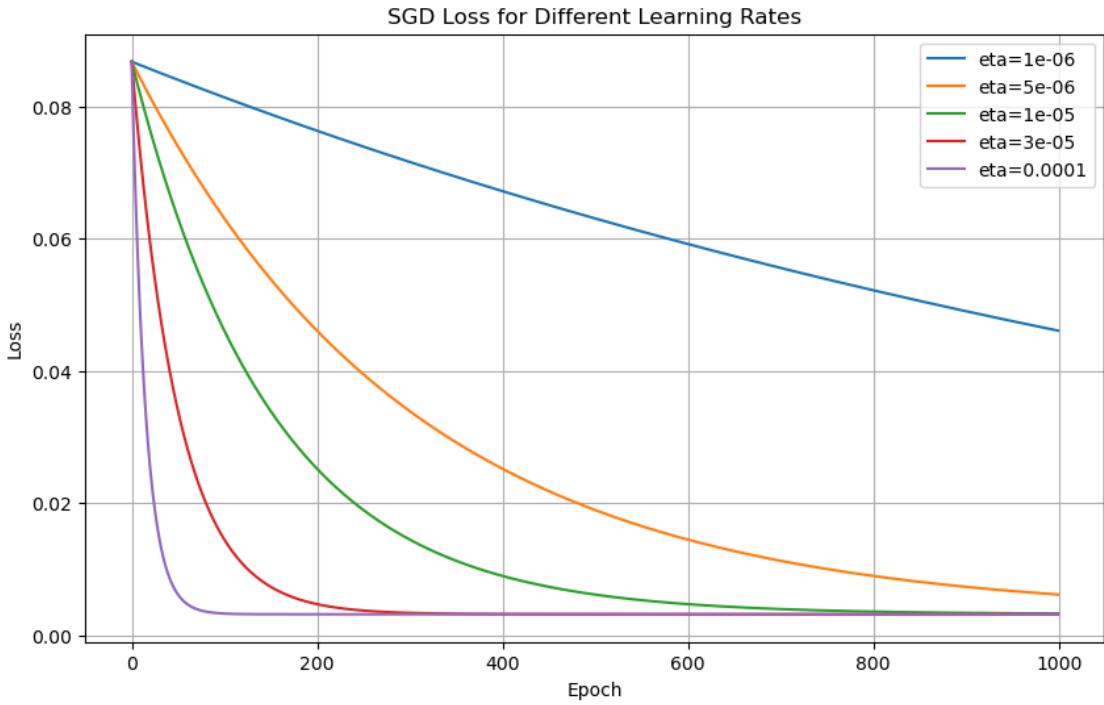
### 1.3.1 Plotting SGD Convergence

Let's visualize the difference in convergence rates a different way. Plot the loss with respect to epoch (iteration) number for each value of eta on the same graph.

```
[47]: """Plotting SGD convergence"""

eta_vals = [1e-6, 5e-6, 1e-5, 3e-5, 1e-4]
w_start = [0.01, 0.01]
N_epochs = 1000

plt.figure(figsize=(10, 6))
for eta in eta_vals:
    _, losses = SGD(X1, Y1, w_start, eta, N_epochs)
    plt.plot(losses, label=f"eta={eta}")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.title("SGD Loss for Different Learning Rates")
plt.legend()
plt.grid()
plt.show()
```



Clearly, a big step size results in fast convergence! Why don't we just set eta to a really big value, then? Say, eta=1?

(Again, note that the FR is lower for this animation.)

```
[ ]: # Parameters to feed the SGD.
params = ({"w_start": [0.01, 0.01], "eta": 1},)
N_epochs = 100
FR = 2

# Voila!
anim = animate_sgd_suite(SGD, loss, X1, Y1, params, N_epochs, FR, ms=2)
anim.save("animation6.gif", fps=30, writer="imagemagick")
Image(open("animation6.gif", "rb").read())
```

Just for fun, let's try eta=10 as well. What happens? (Hint: look at W)

```
[ ]: # Parameters to feed the SGD.
w_start = [0.01, 0.01]
eta = 10
N_epochs = 100

# Presto!
W, losses = SGD(X1, Y1, w_start, eta, N_epochs)
```

the weight explodes

## 1.4 Extra Visualization (not part of the homework problem)

One final visualization! What happens if the loss function has multiple optima?

```
[ ]: # Import different SGD & loss functions.  
# In particular, the loss function has multiple optima.  
from sgd_multiopt_helper import SGD, loss  
  
# Parameters to feed the SGD.  
params = (  
    {"w_start": [0.9, 0.9], "eta": 0.01},  
    {"w_start": [0.0, 0.0], "eta": 0.01},  
    {"w_start": [-0.8, 0.6], "eta": 0.01},  
    {"w_start": [-0.8, -0.6], "eta": 0.01},  
    {"w_start": [-0.4, -0.3], "eta": 0.01},  
)  
N_epochs = 100  
FR = 2  
  
# One more time!  
anim = animate_sgd_suite(SGD, loss, X1, Y1, params, N_epochs, FR, ms=2)  
anim.save("animation7.gif", fps=30, writer="imagemagick")  
Image(open("animation7.gif", "rb").read())
```

if the learning rate is small enough it can get stuck in a local minima

## 2 Problem 1, Parts F-H: Stochastic Gradient Descent with a Larger Dataset

Use this notebook to write your code for problem 1 parts F-H by filling in the sections marked `# TODO` and running all cells.

```
[49]: %matplotlib inline
```

### 2.1 Problem 1F: Perform SGD with the new dataset

For the functions below, you may re-use your code from parts 4C-E. Note that you can now modify your SGD function to return the final weight vector instead of the weights after every epoch.

```
[50]: def loss, gradient, SGD  
  
def loss(X, Y, w):  
    """  
    Calculate the squared loss function.  
  
    Inputs:  
        X: A (N, D) shaped numpy array containing the data points.
```

```
    Y: A (N, ) shaped numpy array containing the (float) labels of the data_  
    ↪points.  
    w: A (D, ) shaped numpy array containing the weight vector.
```

*Outputs:*

The loss evaluated with respect to X, Y, and w.

"""

```
Loss = 0
```

```
N = X.shape[0]
```

```
for i in range(N):  
    Loss += (Y[i] - np.dot(w, X[i])) ** 2  
return Loss / N
```

```
def gradient(x, y, w):
```

"""

Calculate the gradient of the loss function with respect to  
a single point (x, y), and using weight vector w.

*Inputs:*

x: A (D, ) shaped numpy array containing a single data point.

y: The float label for the data point.

w: A (D, ) shaped numpy array containing the weight vector.

*Output:*

The gradient of the loss with respect to x, y, and w.

"""

```
return -2 * (y - np.dot(w, x)) * x
```

```
def SGD(X, Y, w_start, eta, N_epochs):
```

"""

Perform SGD using dataset (X, Y), initial weight vector w\_start,  
learning rate eta, and N\_epochs epochs.

*Inputs:*

X: A (N, D) shaped numpy array containing the data points.

Y: A (N, ) shaped numpy array containing the (float) labels of the data\_  
 ↪points.

w\_start: A (D, ) shaped numpy array containing the weight vector  
 ↪initialization.

eta: The step size.

N\_epochs: The number of epochs (iterations) to run SGD.

*Outputs:*

w: A (D, ) shaped array containing the final weight vector.

```

    losses: A ( $N_{\text{epochs}}$ , ) shaped array containing the losses from all
    ↵iterations.
    """
    losses = np.zeros(N_epochs)

    w = w_start.copy()

    for epoch in range(N_epochs):
        losses[epoch] = loss(X, Y, w)

        # Shuffle the data points
        indices = np.random.permutation(X.shape[0])
        X_shuffled = X[indices]
        Y_shuffled = Y[indices]

        for i in range(X.shape[0]):
            x_i = X_shuffled[i]
            y_i = Y_shuffled[i]
            grad = gradient(x_i, y_i, w)
            w -= eta * grad

    return w, losses

```

Next, we need to load the dataset. In doing so, the following function may be helpful:

```
[51]: def load_data(filename):
    """
    Function loads data stored in the file filename and returns it as a numpy
    ↵ndarray.

    Inputs:
        filename: Given as a string.

    Outputs:
        Data contained in the file, returned as a numpy ndarray
    """
    return np.loadtxt(filename, skiprows=1, delimiter=",")
```

Now, load the dataset in `../data/sgd_data.csv` and run SGD using the given parameters; print out the final weights.

```
[52]: data = load_data("../data/sgd_data.csv")
X = data[:, :4]
Y = data[:, 4]

w_start = [0.001, 0.001, 0.001, 0.001]
eta = np.exp(-15)
```

```

N_epochs = 800

bias = 0.001

# append the bias to X and w_start
X = np.hstack((X, np.ones((X.shape[0], 1))))
w_start = np.append(w_start, bias)

# print first few data points
print(X[:5])
print(Y[:5])

w, losses = SGD(X, Y, w_start, eta, N_epochs)
print("Final weights:", w)

```

```

[[8.55834 3.08451 1.79961 6.61568 1.      ]
 [4.08455 8.66603 2.8698  1.67996 1.      ]
 [1.93701 8.9219  8.99454 0.56534 1.      ]
 [3.11286 7.36286 4.35027 7.14242 1.      ]
 [5.52064 3.46109 6.07592 2.77215 1.      ]]
[ 0.45023 -7.29982 -74.71018 22.2565 -66.01461]
Final weights: [-5.94209018  3.94392493 -11.72381711  8.78570506
 -0.22716207]

```

## 2.2 Problem 1G: Convergence of SGD

This problem examines the convergence of SGD for different learning rates. Please implement your code in the cell below:

```

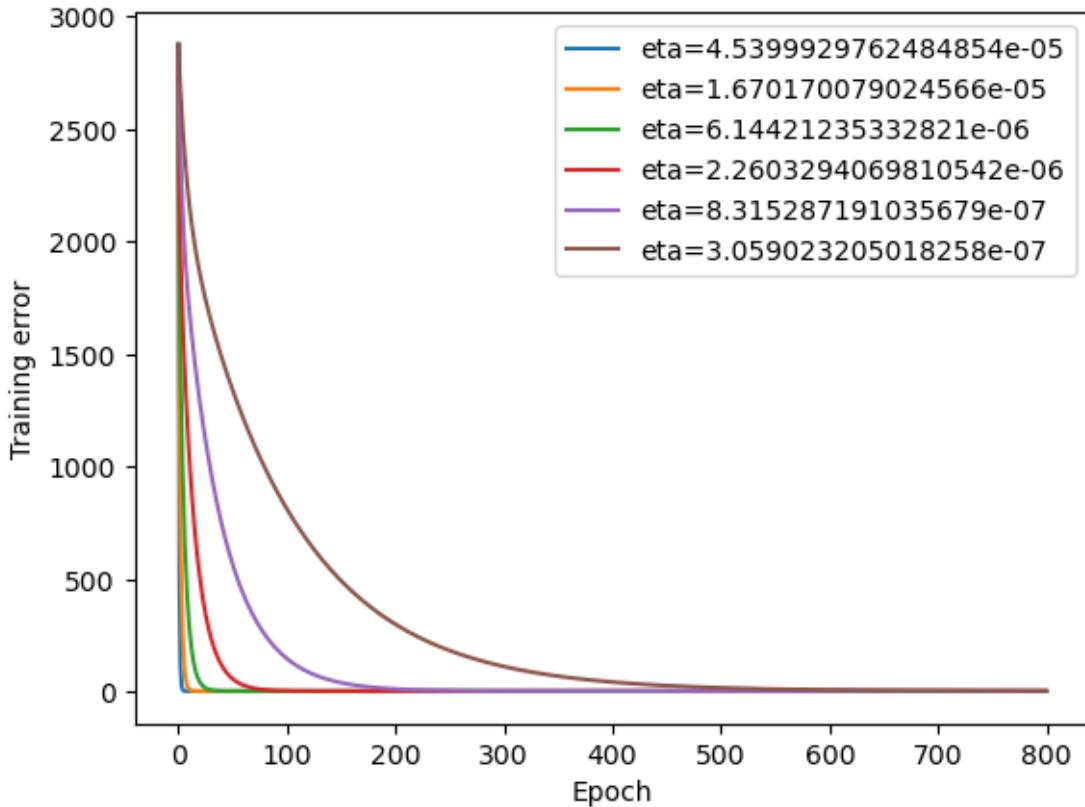
[53]: eta_vals = [np.exp(-10), np.exp(-11), np.exp(-12), np.exp(-13), np.exp(-14), np.
       ↪exp(-15)]
w_start = [0.001, 0.001, 0.001, 0.001, bias]
N_epochs = 800

plt.figure()

for eta in eta_vals:
    w_final, losses = SGD(X, Y, np.array(w_start), eta, N_epochs)
    plt.plot(losses, label=f"eta={eta}")

plt.xlabel("Epoch")
plt.ylabel("Training error")
plt.legend()
plt.show()

```



### 2.3 Problem 1H

Provide your code for computing the least-squares analytical solution below.

```
[54]: w_closed = np.linalg.inv(X.T @ X) @ (X.T @ Y)

print("SGD final weight: ", w_final)
print("Closed form: ", w_closed)
print("Difference (%): ", (w_closed - w_final) / w_final * 100)

SGD final weight:  [-5.94211258   3.94389725 -11.72384395   8.78567548
-0.22718511]
Closed form:  [-5.99157048   4.01509955 -11.93325972   8.99061096  -0.31644251]
Difference (%):  [ 0.83232869   1.80537941   1.78623808   2.3326093   39.28840273]
```

### 2.4 Problem 1I

Is there any reason to use SGD when a closed-form solution exists?

Yes, as for large datasets (high dimensionality) calculating this is very computationally heavy.

## References

- [1] M. D. Science, “The kernel trick,” 2020. [Online]. Available: <https://medium.com/data-science/the-kernel-trick-c98cdbcaeb3f>
- [2] P. with ML, “Xor problem.” [Online]. Available: <https://www.playwithml.com/projects/xor>
- [3] Google, “Tensorflow playground.” [Online]. Available: <https://playground.tensorflow.org/>