Name: Barathkrishna Satheeshkumar
Andrew ID: bsathees

Problem 1

2. Model 1 is high bias but low variance, Model 2 is high variance but low bias

Problem 2

4. K-fold cross validation partitions the data into k equal sized subsets, and trains k models, each time using one subset as the validation data and the rest as the training data

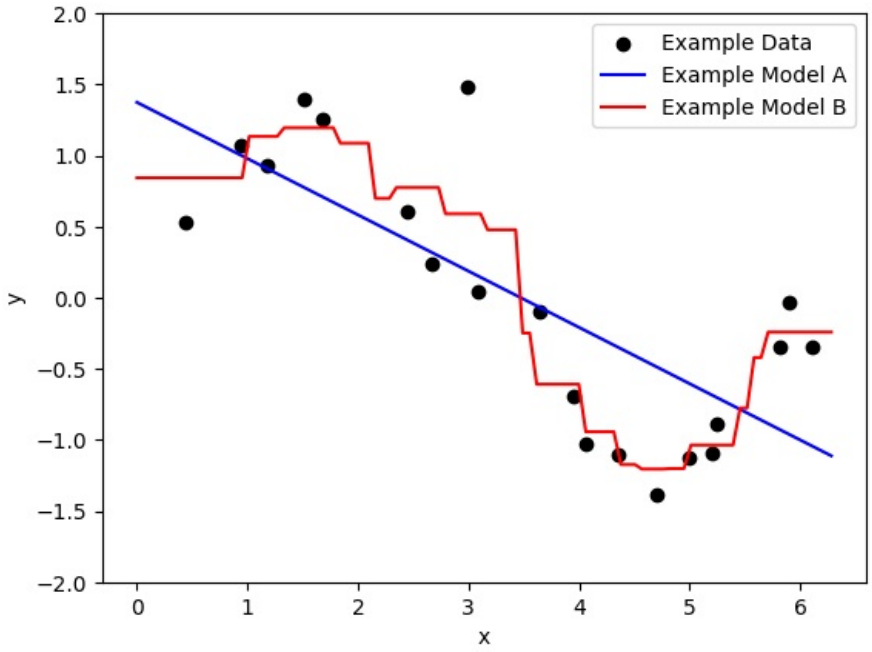
M10-L1 Problem 1

In this problem you will look compare models with lower/higher variance/bias by computing bias and variance at a single point.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.neighbors import KNeighborsRegressor

def plot_model(model,color="blue"):
    x = np.linspace(0, np.pi*2, 100)
    y = model.predict(x.reshape(-1,1))
    plt.plot(x, y, color=color)
    plt.xlabel("x")
    plt.ylabel("y")

def plot_data(x, y):
    plt.scatter(x,y,color="black")

def eval_model_at_point(model, x):
    return model.predict(np.array([[x]])).item()

def train_models():
    x = np.random.uniform(0,np.pi*2,20).reshape(-1,1)
    y = np.random.normal(np.sin(x),0.5).flatten()

    modelA = LinearRegression()
    modelB = KNeighborsRegressor(3)
    modelA.fit(x,y)
    modelB.fit(x,y)
    return modelA, modelB, x, y
```

The function `train_models` gets 20 new data points and trains two models on these data points. Model A is a linear regression model, while model B is a 3-nearest neighbor regressor.

In [2]:
```
modelA, modelB, x, y = train_models()
plt.figure()
plot_data(x,y)
plot_model(modelA,"blue")
plot_model(modelB,"red")
plt.legend(["Example Data", "Example Model A", "Example Model B"])
plt.ylim([-2,2])
plt.show()
```



Training models

First, train 50 instances of model A and 50 instances of model B. Store all 100 total models for use in the next few cells. Generate these models with the function: `modelA, modelB, x, y = train_models()`.

In [5]:
```
# YOUR CODE GOES HERE
```

```
models_A = []
models_B = []

for i in range(50):
    modelA, modelB, x, y = train_models()
    models_A.append(modelA)
    models_B.append(modelB)
```

Bias and Variance

Now we will use the definitions of bias and variance to compute the bias and variance of each type of model. You will focus on the point x = 1.57 only. First, compute the prediction for each model at x. (You can use the function `eval_model_at_point(model, x)`).

In [7]:
```
x = 1.57

# YOUR CODE GOES HERE
predictions_A = [eval_model_at_point(model, x) for model in models_A]
predictions_B = [eval_model_at_point(model, x) for model in models_B]
```

In this cell, use the values you computed above to compute and print the bias and variance of model A at the point x = 1.57. The true function value `y_GT` is given as 1 for x=1.57.

In [10]:
```
yGT = 1

# YOUR CODE GOES HERE
mean_prediction_A = np.mean(predictions_A)
bias_A = (mean_prediction_A - yGT) ** 2
var_A = np.var(predictions_A)

mean_prediction_B = np.mean(predictions_B)
bias_B = (mean_prediction_B - yGT) ** 2
var_B = np.var(predictions_B)

print(f"Model A:   Bias = {bias_A:.3f},  Variance = {var_A:.3f}")
print(f"Model B:   Bias = {bias_B:.3f},  Variance = {var_B:.3f}")
```

```
Model A:   Bias = 0.244,  Variance = 0.038
Model B:   Bias = 0.003,  Variance = 0.090
```

Questions

1. Which model has smaller bias at $x = 1.57$?

Model B

2. Which model has lower variance at $x = 1.57$?

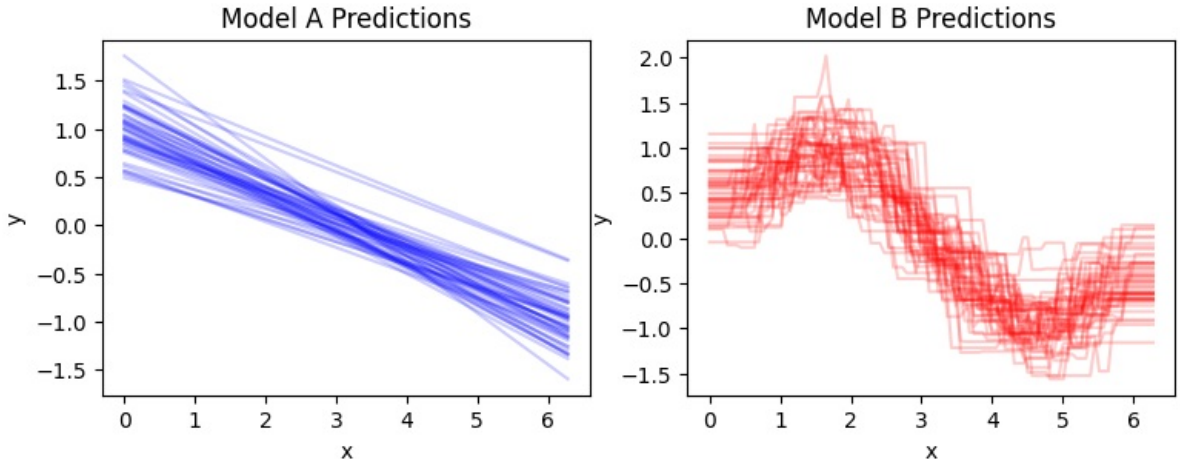
Model A

Plotting models

Now use the `plot_model` function to overlay all Model A predictions on one plot and all Model B predictions on another. Notice the spread of each model.

In [14]:
```
plt.figure(figsize=(9,3))

plt.subplot(1,2,1)
plt.title("Model A")
# YOUR CODE GOES HERE
x = np.linspace(0, np.pi * 2, 100).reshape(-1, 1)
for model in models_A:
    y_pred = model.predict(x)
    plt.plot(x, y_pred, color="blue", alpha=0.2)
plt.title("Model A Predictions")
plt.xlabel("x")
plt.ylabel("y")


plt.subplot(1,2,2)
plt.title("Model B")
# YOUR CODE GOES HERE
for model in models_B:
    y_pred = model.predict(x)
    plt.plot(x, y_pred, color="red", alpha=0.2)
plt.title("Model B Predictions")
plt.xlabel("x")
plt.ylabel("y")
```

```
plt.show()
```

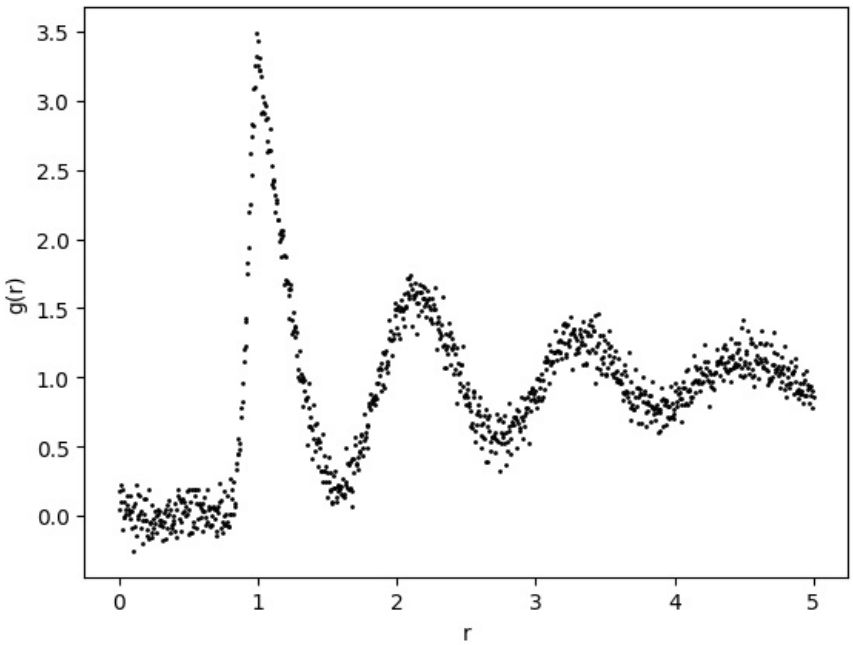


M10-L2 Problem 1

In this problem, you will perform 10-fold cross validation to find the best of 3 regression models.

You are given a dataset with testing and training data of another radial distribution function (measuring 'g(r)', the probability of a particle being a certain distance 'r' from another particle): `X_train, X_test, y_train, y_test`

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.neural_network import MLPRegressor
from sklearn.model_selection import train_test_split, KFold
from sklearn.base import clone

def get_gr(r):
    a, b, L, m, t, d = 0.54, 5.4, 1.2, 7.4, 100, 3.3
    g1 = 1 + (r+1e-9)**(-m) * (d-1-L) + (r-1+L)/(r+1e-9)*np.exp(-a*(r-1))*np.cos(b*(r-1))
    g2 = d * np.exp(-t*(r-1)**2)
    g = g1*(r>=1) + g2*(r<1)
    return g

def plot_model(model,color="blue"):
    x = np.linspace(0, 5, 1000)
    y = model.predict(x.reshape(-1,1))
    plt.plot(x, y, color=color, linewidth=2, zorder=2)
    plt.xlabel("r")
    plt.ylabel("g(r)")

def plot_data(x, y):
    plt.scatter(x,y,s=1, color="black")
    plt.xlabel("r")
    plt.ylabel("g(r)")


np.random.seed(0)
X = np.linspace(0,5,1000).reshape(-1,1)
y = np.random.normal(get_gr(X.flatten()),0.1)

X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0, train_size=800)


plt.figure()
plot_data(X,y)
plt.show()
```



Models

Below we define 3 sklearn neural network models `model1`, `model2`, and `model3`. Your goal is to find which is best using 10-fold cross-validation.

In [2]:
```
model1 = MLPRegressor([24], random_state=0, activation="tanh", max_iter=1000)
model2 = MLPRegressor([48,48], random_state=0, activation="tanh", max_iter=1000)
```

```
model3 = MLPRegressor([64,64, 64], random_state=0, activation="relu", max_iter=1000)


models = [model1, model2, model3]
for model in models:
    model.fit(X_train, y_train)
```

Cross-validation folds

This cell creates 10-fold iterator objects in sklearn. Make note of how this is done.

We also provide code for computing the cross-validation score for average R^2 over validation folds. Note that the model is retrained on each fold, and weights/biases are reset each time with `sklearn.base.clone()`

In [3]:
```
folds = KFold(n_splits=10,random_state=0,shuffle=True)


scores1 = []
for train_idx, val_idx in folds.split(X_train):
    model1 = clone(model1)
    model1.fit(X_train[train_idx,:],y_train[train_idx])
    score = model1.score(X_train[val_idx,:],y_train[val_idx])
    scores1.append(score)
    print(f"Validation score: {score}")


score1 = np.mean(np.array(scores1))
print(f"Average validation score for Model 1: {score1}")
```

```
Validation score: 0.17567883199274337
Validation score: 0.19279856417941277
Validation score: 0.277493724970579
Validation score: 0.31043523576478926
Validation score: 0.20608404129798275
Validation score: 0.0379012239544968
Validation score: 0.1676244803676995
Validation score: 0.2202500372447742
Validation score: 0.14423712046918657
Validation score: 0.19894361702001595
Average validation score for Model 1: 0.193144687726168
```

Your turn: validating models 2 and 3

Now follow the same procedure to get the average R^2 scores for `model2` and `model3` on validation folds. You can use the same KFold iterator.

In [6]:
```
# YOUR CODE GOES HERE
scores2 = []
scores3 = []

for train_idx, val_idx in folds.split(X_train):
    model2_clone = clone(model2)
    model2_clone.fit(X_train[train_idx, :], y_train[train_idx])
    score = model2_clone.score(X_train[val_idx, :], y_train[val_idx])
    scores2.append(score)
    print(f"Validation score for Model 2: {score}")

print("\n")

for train_idx, val_idx in folds.split(X_train):
    model3_clone = clone(model3)
    model3_clone.fit(X_train[train_idx, :], y_train[train_idx])
    score = model3_clone.score(X_train[val_idx, :], y_train[val_idx])
    scores3.append(score)
    print(f"Validation score for Model 3: {score}")

score2 = np.mean(np.array(scores2))
score3 = np.mean(np.array(scores3))

print("\n")

print(f"Average validation score for Model 2: {score2}")
print(f"Average validation score for Model 3: {score3}")
```

```
Validation score for Model 2: 0.9135256064394238
Validation score for Model 2: 0.92381162019413
Validation score for Model 2: 0.9109428377428712
Validation score for Model 2: 0.916683295227516
Validation score for Model 2: 0.8980936123083956
Validation score for Model 2: 0.9208009063665946
Validation score for Model 2: 0.9123834705950664
Validation score for Model 2: 0.8780032287365068
Validation score for Model 2: 0.9281564779069267
Validation score for Model 2: 0.95771300087561


Validation score for Model 3: 0.9629033605148645
Validation score for Model 3: 0.9466107686883457
Validation score for Model 3: 0.9518315048355763
Validation score for Model 3: 0.9514051770741323
Validation score for Model 3: 0.9229643307655354
Validation score for Model 3: 0.9501422202077937
Validation score for Model 3: 0.9322229519501162
Validation score for Model 3: 0.9238931238090653
Validation score for Model 3: 0.9461292855545795
Validation score for Model 3: 0.9611128180031757


Average validation score for Model 2: 0.9160114056393042
Average validation score for Model 3: 0.9449215541403184
```

Comparing models

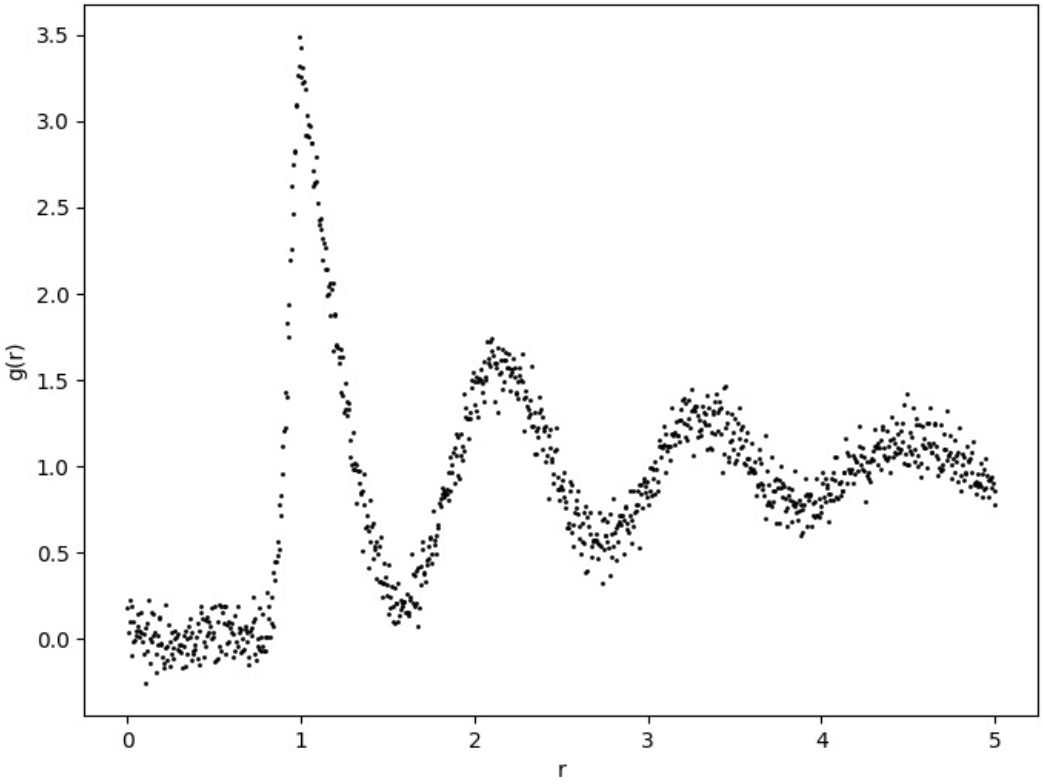
Which model had the best performance according to your validation study?

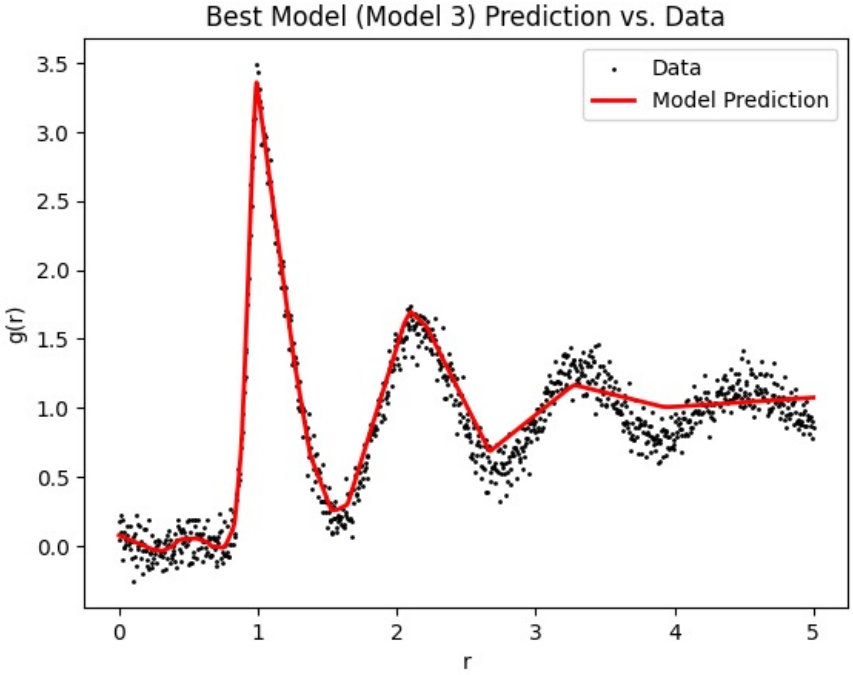
The best-performing model is Model 3 because it has the highest average validation R^2 score.

Retrain this model on the full training dataset and report the R2 score on training and testing data. Then complete the code to plot the model prediction with the data using the `plot_model` function.

In [7]:
```python
# YOUR CODE GOES HERE
best_model = clone(model3)
best_model.fit(X_train, y_train)

train_score = best_model.score(X_train, y_train)
test_score = best_model.score(X_test, y_test)

print(f"R^2 score on training data: {train_score}")
print(f"R^2 score on testing data: {test_score}")

plt.figure(figsize=(8, 6))
plot_data(X, y)

plt.figure()
plot_data(X,y)

# YOUR CODE GOES HERE
plot_model(best_model, color="red")
plt.title("Best Model (Model 3) Prediction vs. Data")
plt.legend(["Data", "Model Prediction"])

plt.show()
```

```
R^2 score on training data: 0.9547034601442719
R^2 score on testing data: 0.9371864974414209
```





Problem 1

Problem Description

In this problem you will fit a neural network to solve a simple regression problem. You will use 5 fold cross validation, plotting training and validation loss curves, as well as model predictions for each of the folds. You will compare between results for 3 neural networks, trained for 100, 500, and 2000 epochs respectively.

Fill out the notebook as instructed, making the requested plots and printing necessary values.

You are welcome to use any of the code provided in the lecture activities.

Summary of deliverables:

- Visualization of provided data
- `trainModel()` function
- 15 figures containing two subplots (loss curves and model prediction) across all 5 folds for the 3 models
- Average MSE across all folds for the 3 models
- Discussion and comparison of model performance, and the importance of cross validation for evaluating model performance.

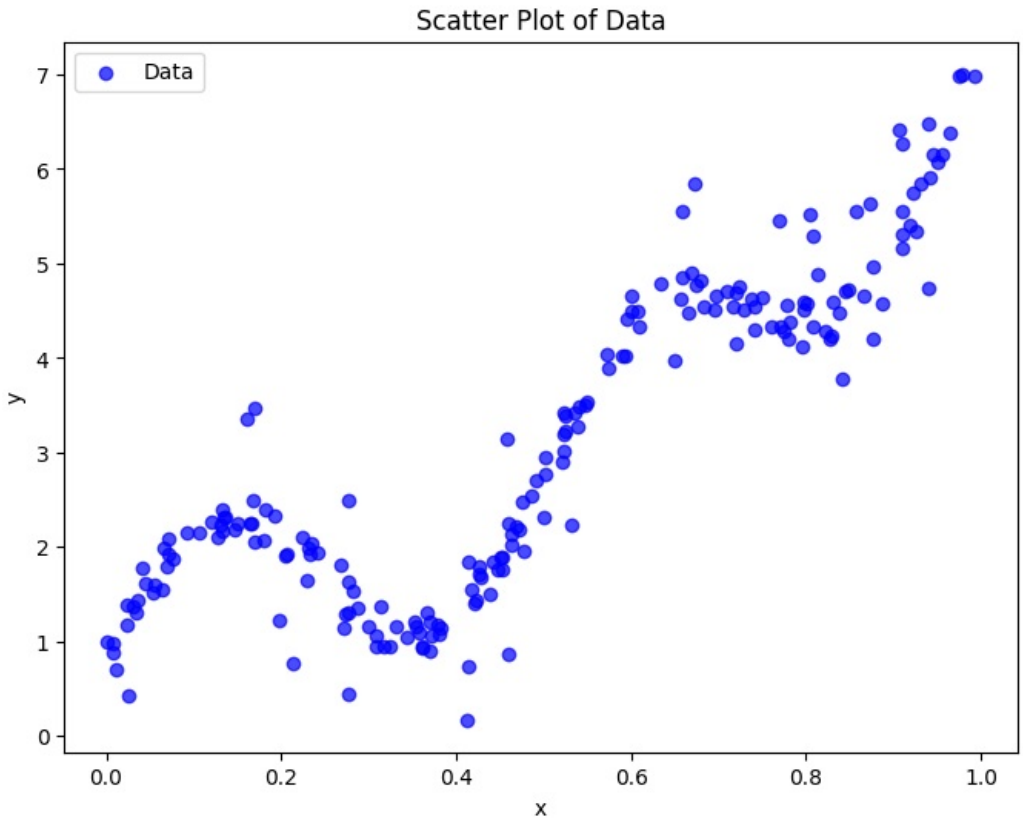
Imports and Utility Functions:

In [5]:
```
import torch.nn as nn
import torch
import torch.optim as optim

import numpy as np
import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split, KFold
from sklearn.metrics import mean_squared_error

def plotLoss(ax, train_curve, val_curve):
    ax.plot(train_curve, label = 'Training')
    ax.plot(val_curve, label = 'Validation')
    ax.set_xlabel('Epoch')
    ax.set_ylabel('Loss')
    ax.legend()

def plotModel(ax, model, x, y, idx_train, idx_test):
    xs = torch.linspace(min(x).item(), max(x).item(), 200).reshape(-1,1)
    ys = model(xs)
    ax.scatter(x[idx_train], y[idx_train], c = 'blue', alpha = 0.5, label = 'Training Data')
    ax.scatter(x[idx_test], y[idx_test], c = 'green', alpha = 0.5, label = 'Test Data')
    ax.plot(xs.detach().numpy(), ys.detach().numpy(), 'k--', label = 'Fitted Function')
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.legend()
```

Load and visualize the data

Data can be loaded from the `m10-hw1-data.txt` file using `np.loadtxt()`. The first column of the data corresponds to the x values and the second column corresponds to the y values. Visualize the data using a scatter plot.

In [4]:
```
## YOUR CODE GOES HERE
data = np.loadtxt("data/m10-hw1-data.txt")

x = data[:, 0]
y = data[:, 1]

plt.figure(figsize=(8, 6))
plt.scatter(x, y, color="blue", alpha=0.7, label="Data")
plt.xlabel("x")
plt.ylabel("y")
plt.title("Scatter Plot of Data")
plt.legend()
plt.show()
```



Create Neural Network and NN Training Function

Create a neural network to predict the underlying function of the data using fully connected layers and tanh activation functions, with no activation on the output layer. The network should have 4 hidden layers, with the following shape: $[64, 128, 128, 64]$.

Since we are going to train many models throughout k-fold cross validation, you will create a function `trainModel(x, y, n_epoch)` that returns `model,train_curve,val_curve`, where `model` is the trained PyTorch model, `train_curve` and `val_curve` are lists of the training and validation loss at each epoch throughout the training, respectively. Use `nn.MSELoss()` as the loss function. Use the `torch.optim.Adam()` optimizer with a learning rate of 0.01. You will instantiate your neural network inside of the training function, as we train a new model with each of the k folds. The x and y which we pass the model will be split into training and validation sets using `train_test_split()` from sklearn, with a `test_size` of 0.25. Note: since we already split the train/test data 80/20 with each k fold, 25% of the remaining training data will correspond to 20% of the total data. Thus for any given fold, we have 60% of the data for training, 20% for validation, and 20% for testing.

In [6]:
```
## YOUR CODE GOES HERE
class NeuralNetwork(nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__init__()
        self.network = nn.Sequential(nn.Linear(1, 64), nn.Tanh(), nn.Linear(64, 128), nn.Tanh(), nn.Linear(128,
                                     nn.Tanh(), nn.Linear(128, 64), nn.Tanh(), nn.Linear(64, 1))

    def forward(self, x):
        return self.network(x)

def trainModel(x, y, n_epoch=100):
    x_tensor = torch.tensor(x, dtype=torch.float32).reshape(-1, 1)
    y_tensor = torch.tensor(y, dtype=torch.float32).reshape(-1, 1)

    x_train, x_val, y_train, y_val = train_test_split(x_tensor, y_tensor, test_size=0.25, random_state=0)

    model = NeuralNetwork()
    loss_fn = nn.MSELoss()
    optimizer = optim.Adam(model.parameters(), lr=0.01)

    train_curve = []
    val_curve = []

    for epoch in range(n_epoch):
        model.train()
        optimizer.zero_grad()
        y_pred_train = model(x_train)
        train_loss = loss_fn(y_pred_train, y_train)
        train_loss.backward()
        optimizer.step()

        model.eval()
```

```
        with torch.no_grad():
            y_pred_val = model(x_val)
            val_loss = loss_fn(y_pred_val, y_val)

        train_curve.append(train_loss.item())
        val_curve.append(val_loss.item())

    return model, train_curve, val_curve
```

K-Fold Cross Validation

Now we will compare across three models trained for $[100, 500, 2000]$ epochs using 5-fold cross validation. We will use the `KFold()` function from sklearn to get indices of the training and test sets for the 5 folds. Then use your `trainModel()` function from the previous section to train a network for each fold.

For each fold, generate a figure with two subplots: training and validation curves on one, and the model prediction plotted with the training and test data on the other. The training and validation curves can be generated using the provided `plotLoss()` function which takes in a subplot axes handle, `ax`, and the training and validation loss lists, `train_curve` and `val_curve`. The model prediction can be plotted using the `plotModel()` function which takes in a subplot axes handle, `ax`, the trained model, `model`, the complete datasets `x` and `y`, and `idx_train` and `idx_test`, the indices of of the training and test data for that specific fold.

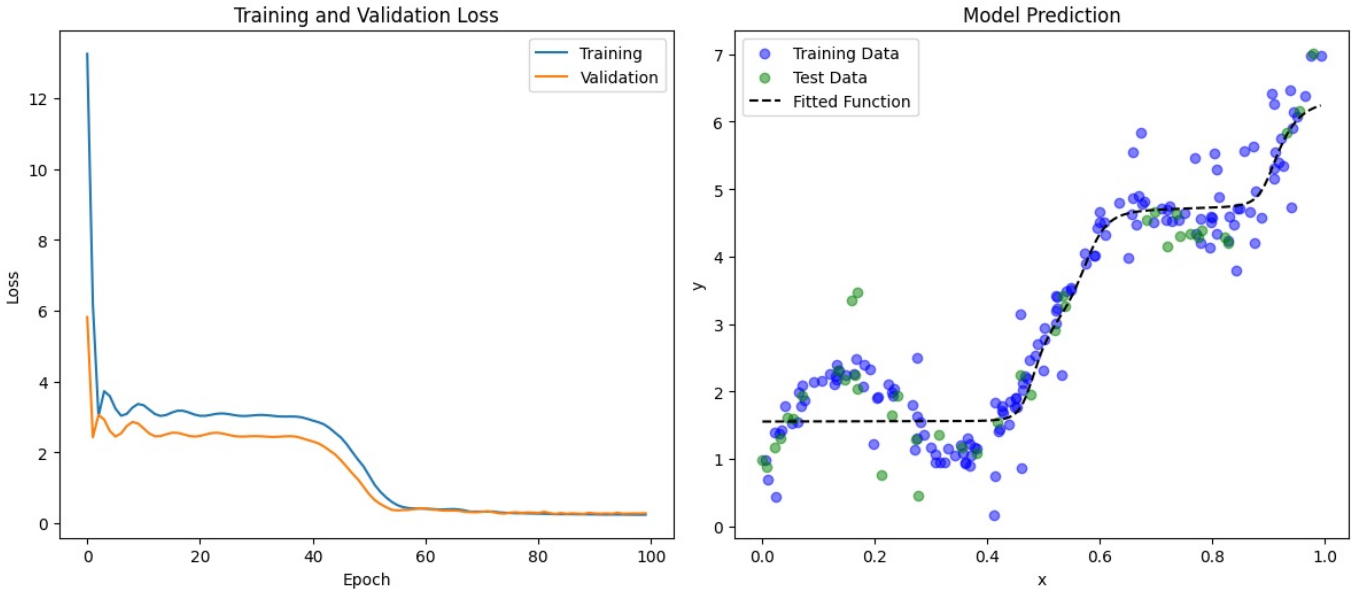
The generated figure should also be titled with the MSE of the trained model on the test data using `suptitle()` from matplotlib, such that the title is centered above the two subplots. The MSE can be computed using the `mean_squared_error` function from sklearn or `MSELoss` from PyTorch.

Average the MSE loss on the test set across the 5 folds, and report a single MSE loss for each of the three models.

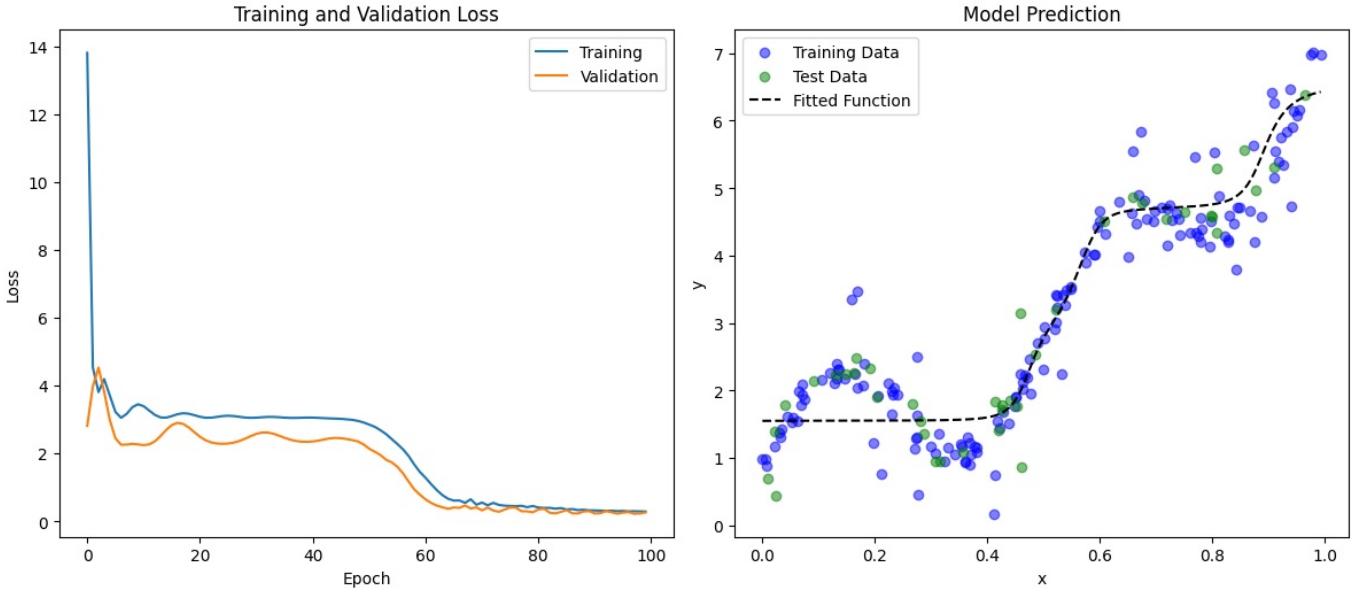
Since there are three models and we are using 5-fold cross validation, you should output 15 figures, with two subplots each.

In [7]:
```
## YOUR CODE GOES HERE
kf = KFold(n_splits=5, shuffle=True, random_state=0)
epoch_values = [100, 500, 2000]
average_mse_per_epoch = {}

for n_epoch in epoch_values:
    mse_list = []

    fold_num = 1
    for train_idx, test_idx in kf.split(x):
        x_train, x_test = x[train_idx], x[test_idx]
        y_train, y_test = y[train_idx], y[test_idx]

        model, train_curve, val_curve = trainModel(x_train, y_train, n_epoch)

        x_test_tensor = torch.tensor(x_test, dtype=torch.float32).reshape(-1, 1)
        y_test_tensor = torch.tensor(y_test, dtype=torch.float32).reshape(-1, 1)
        with torch.no_grad():
            y_pred = model(x_test_tensor)
        mse = mean_squared_error(y_test_tensor.numpy(), y_pred.numpy())
        mse_list.append(mse)

        fig, axs = plt.subplots(1, 2, figsize=(12, 6))

        plotLoss(axs[0], train_curve, val_curve)
        axs[0].set_title("Training and Validation Loss")

        plotModel(axs[1], model, x, y, train_idx, test_idx)
        axs[1].set_title("Model Prediction")

        fig.suptitle(f"Fold {fold_num}, Epochs: {n_epoch}, Test MSE: {mse:.4f}")
        plt.tight_layout(rect=[0, 0, 1, 0.95])
        plt.show()

        fold_num += 1

    avg_mse = np.mean(mse_list)
    average_mse_per_epoch[n_epoch] = avg_mse
    print(f"Average Test MSE for {n_epoch} Epochs: {avg_mse:.4f}")

print("\nAverage MSEs for All Epoch Configurations:")
for n_epoch, avg_mse in average_mse_per_epoch.items():
    print(f"Epochs: {n_epoch}, Average Test MSE: {avg_mse:.4f}")
```

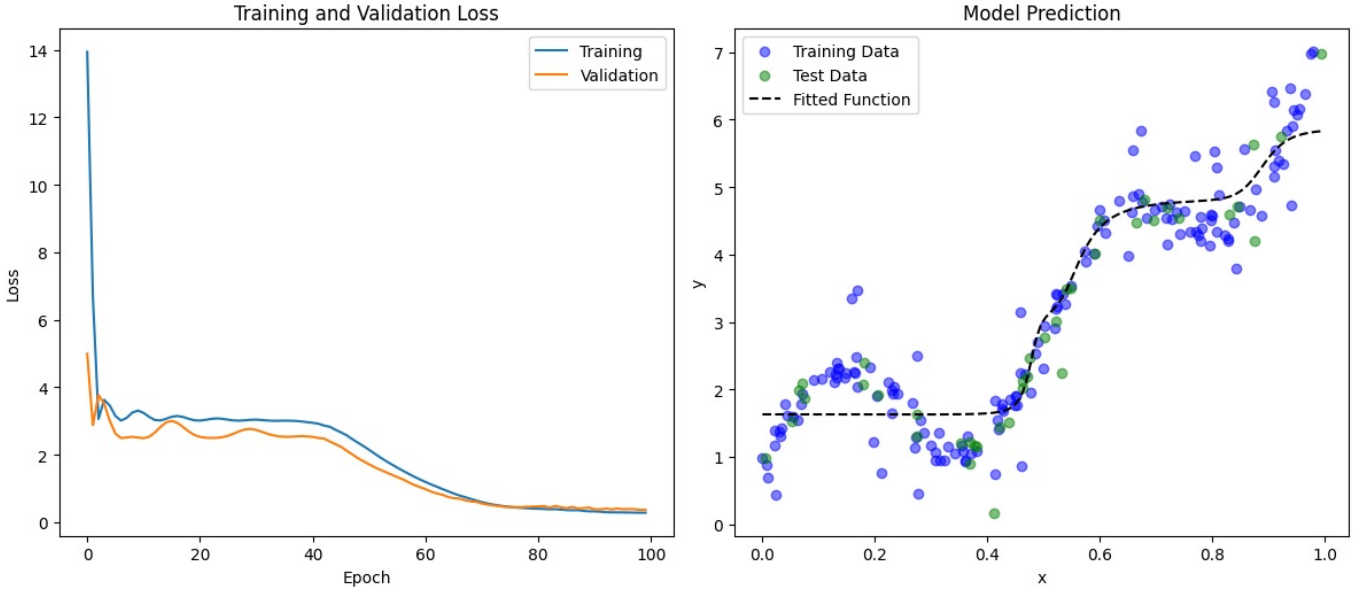
Fold 1, Epochs: 100, Test MSE: 0.3662



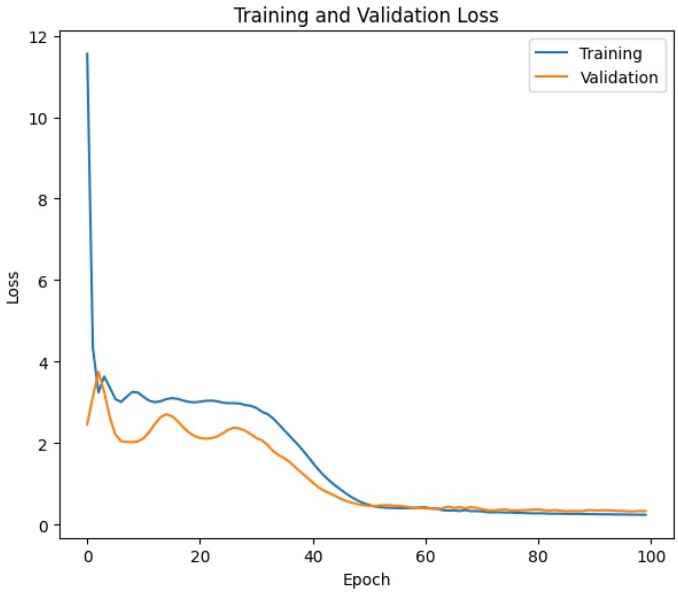
Fold 2, Epochs: 100, Test MSE: 0.2673

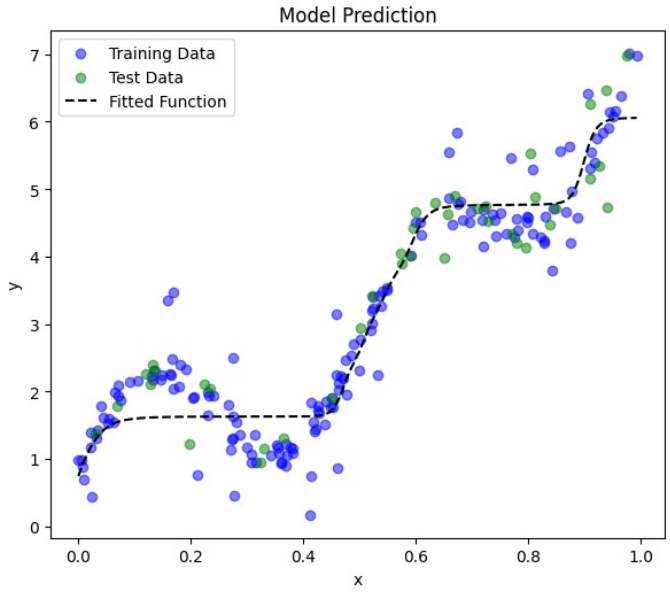


Fold 3, Epochs: 100, Test MSE: 0.2445

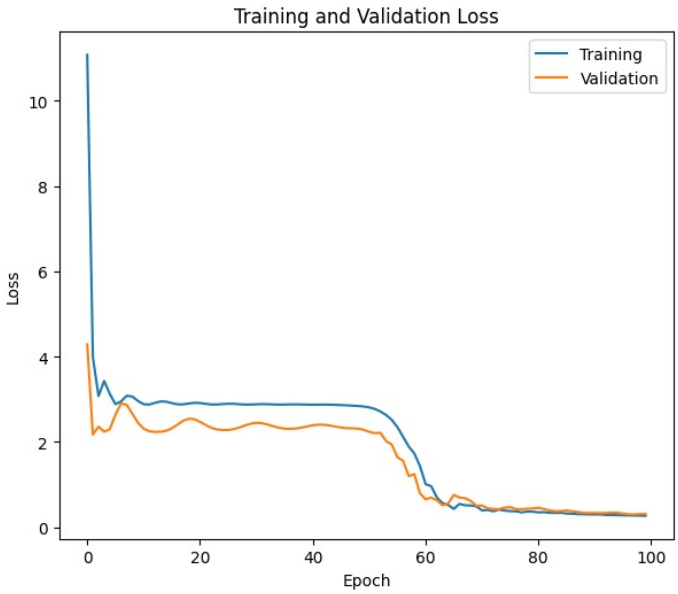


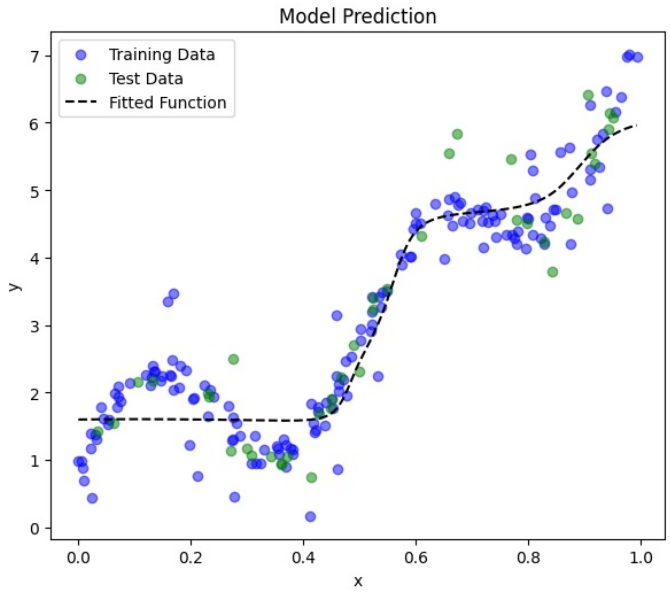
Fold 4, Epochs: 100, Test MSE: 0.2404





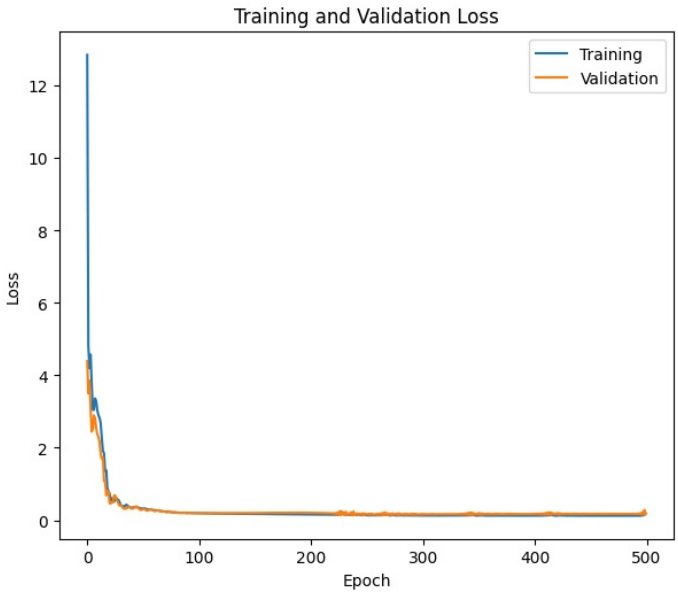
Fold 5, Epochs: 100, Test MSE: 0.2977

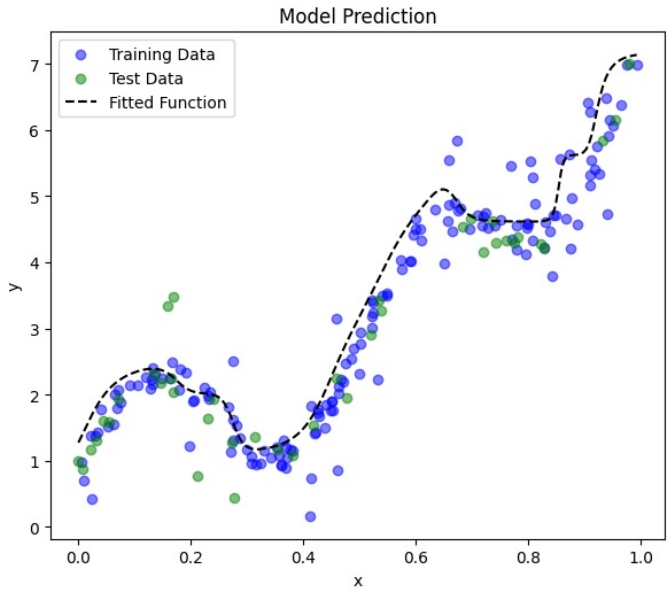




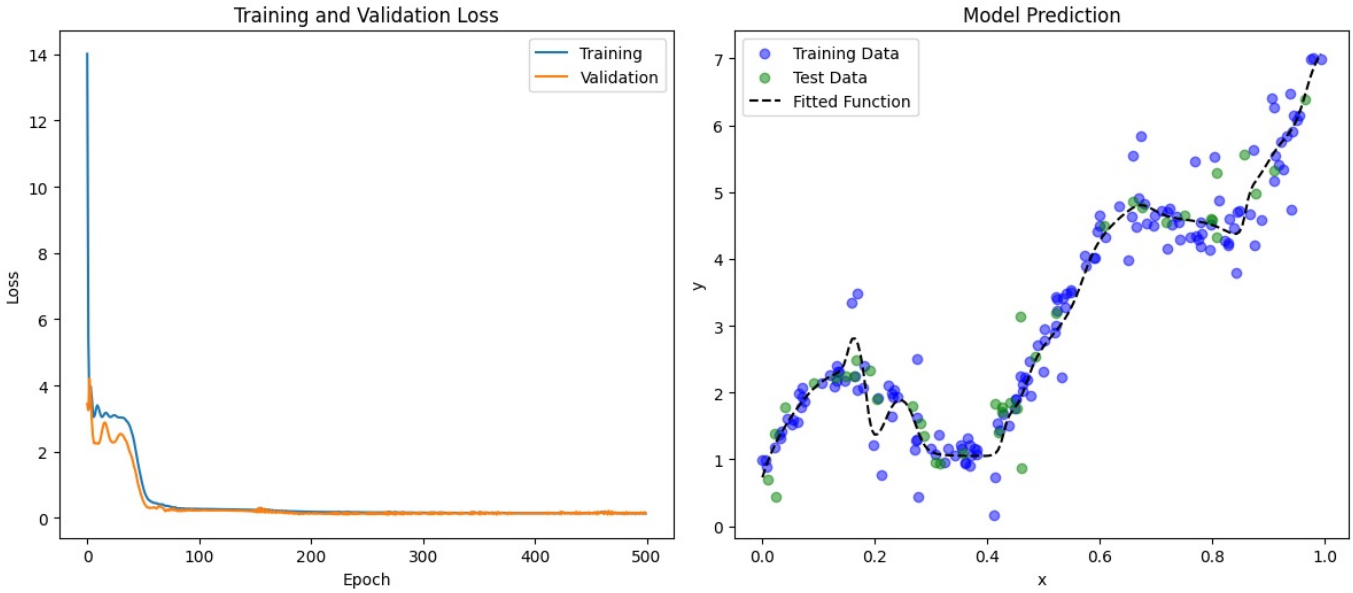

```
Average Test MSE for 100 Epochs: 0.2832
```

Fold 1, Epochs: 500, Test MSE: 0.2736

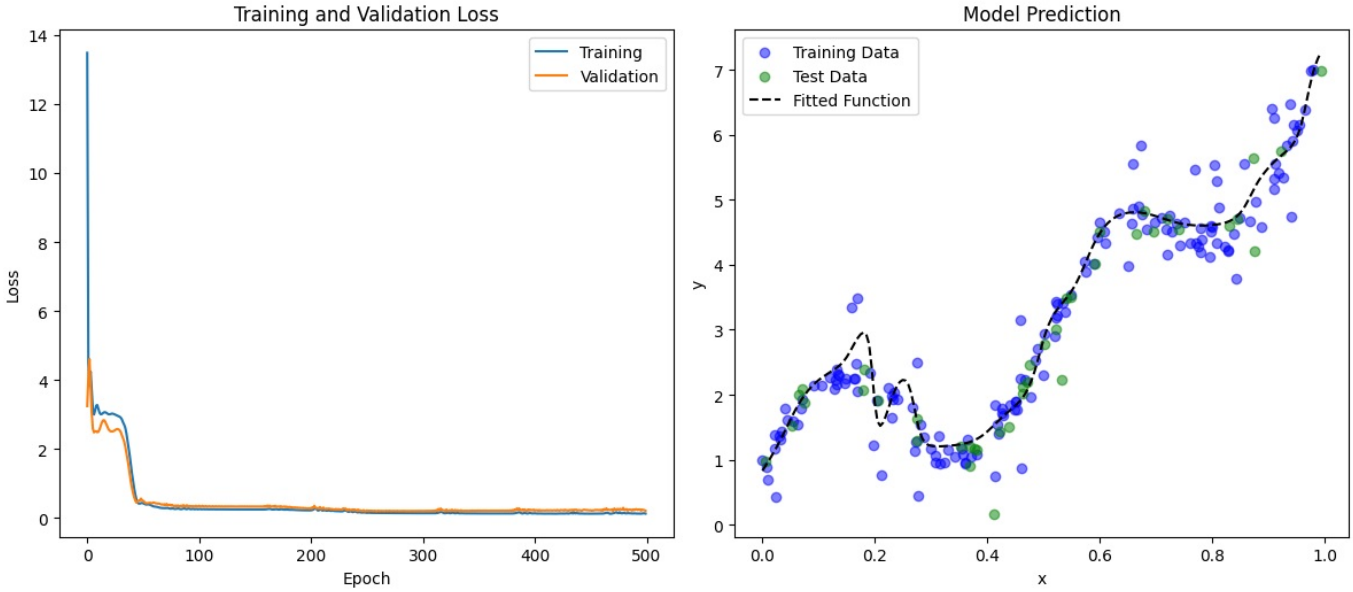




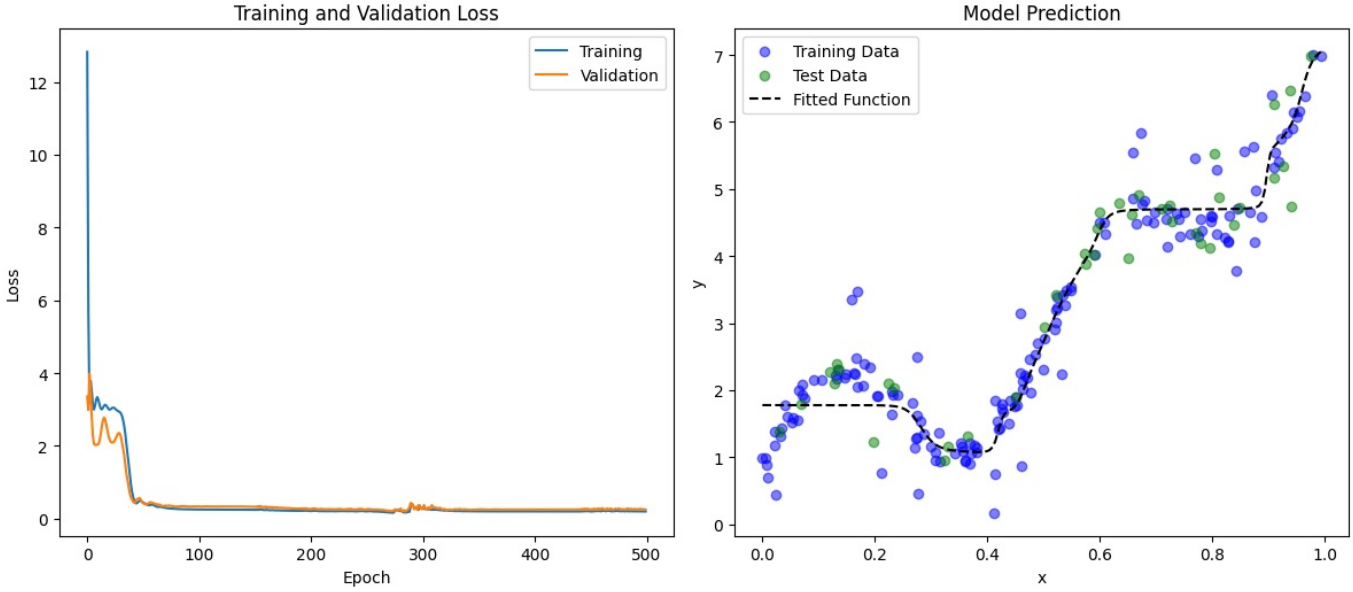
Fold 2, Epochs: 500, Test MSE: 0.1979



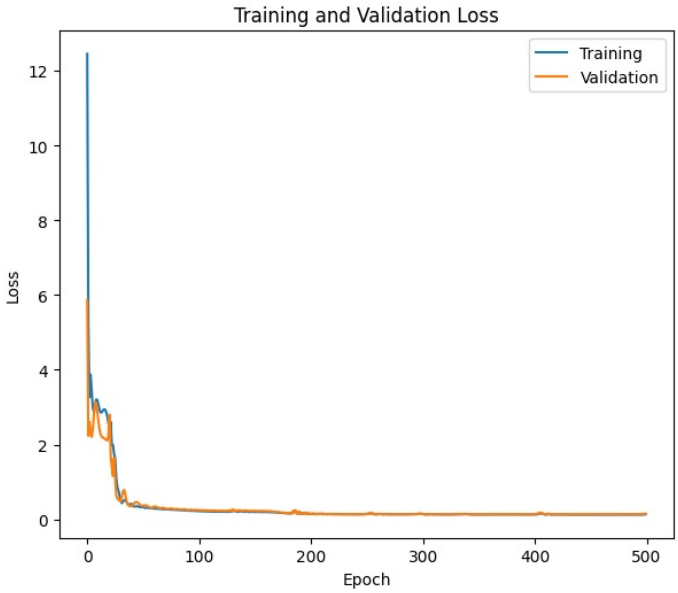
Fold 3, Epochs: 500, Test MSE: 0.1684

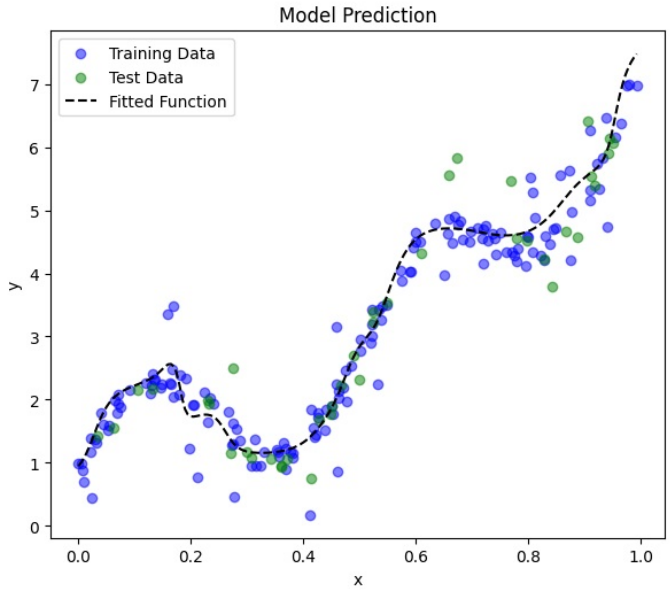


Fold 4, Epochs: 500, Test MSE: 0.1692



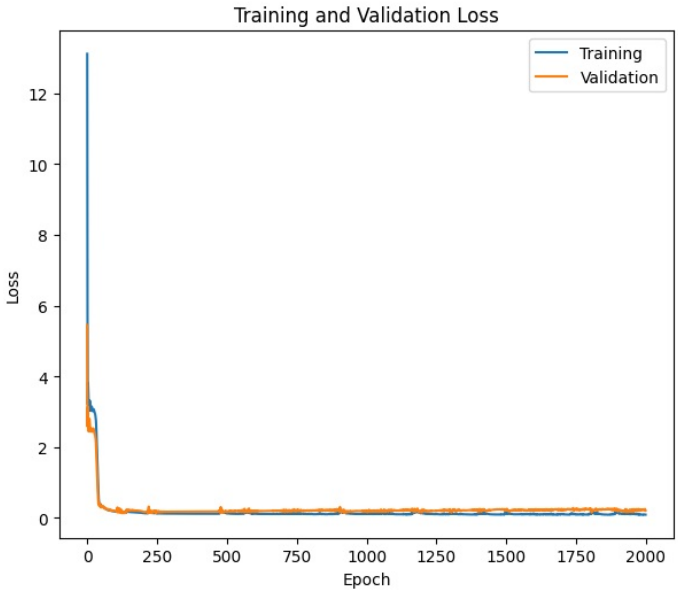
Fold 5, Epochs: 500, Test MSE: 0.2296

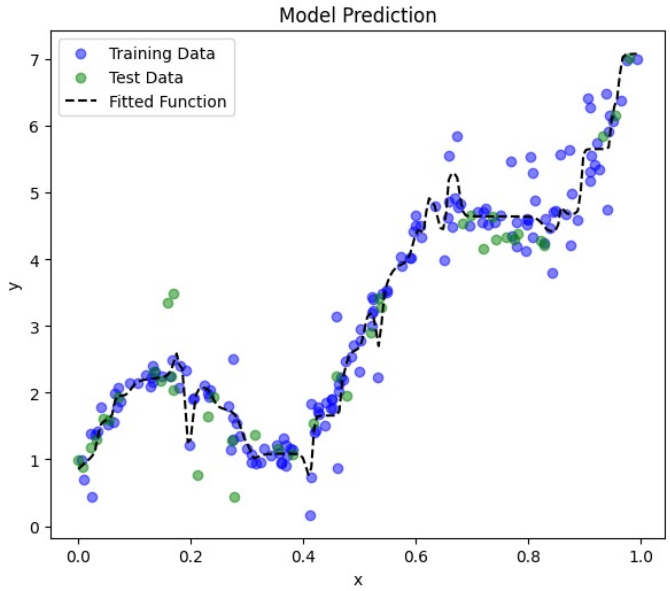




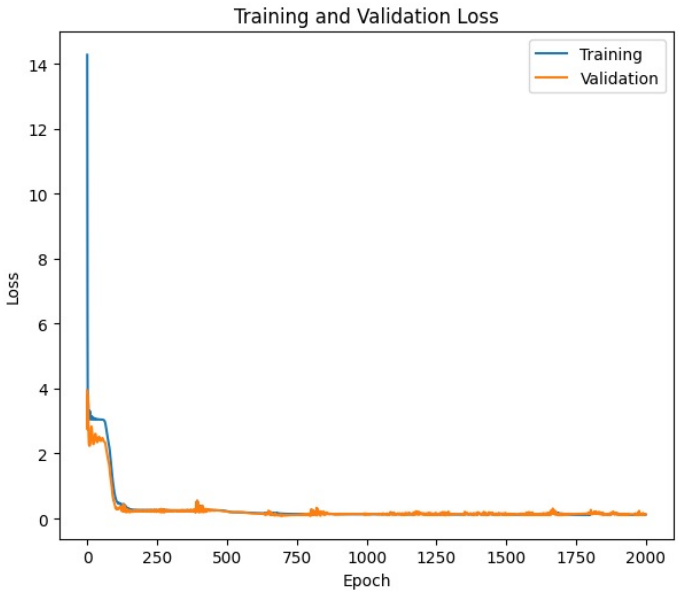
Average Test MSE for 500 Epochs: 0.2077

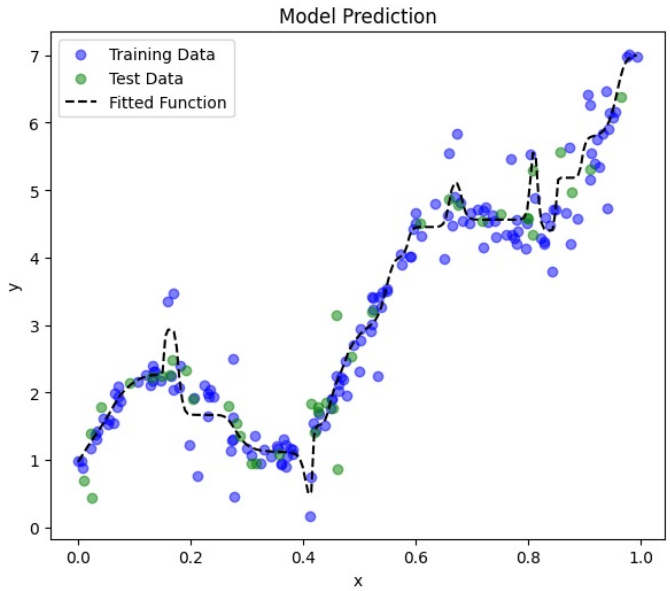
Fold 1, Epochs: 2000, Test MSE: 0.2101



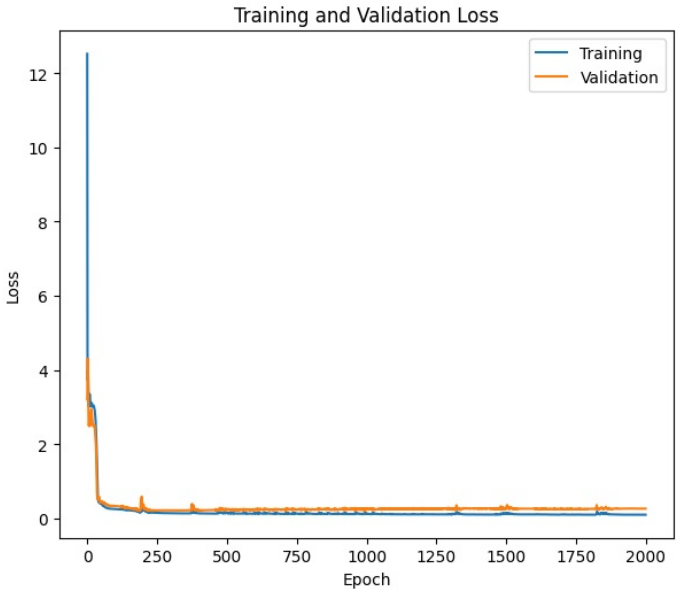


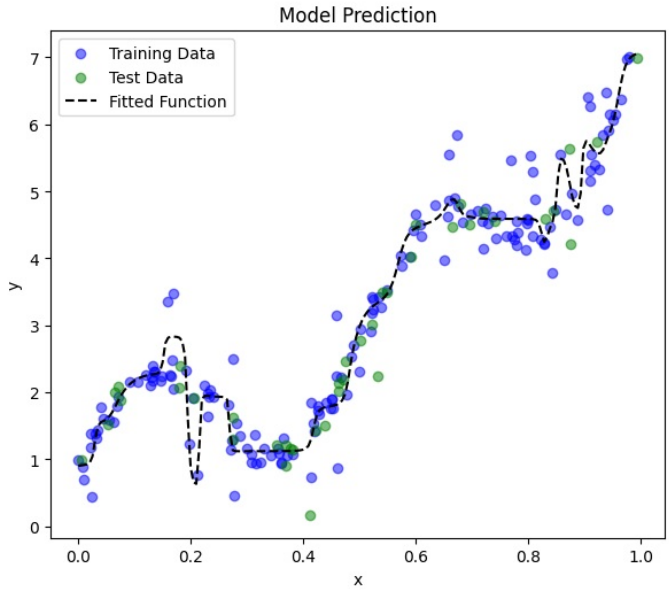
Fold 2, Epochs: 2000, Test MSE: 0.2293



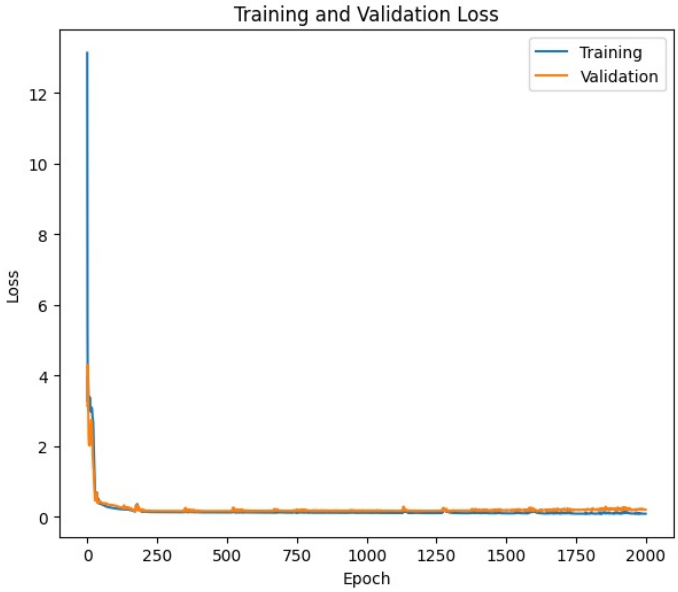


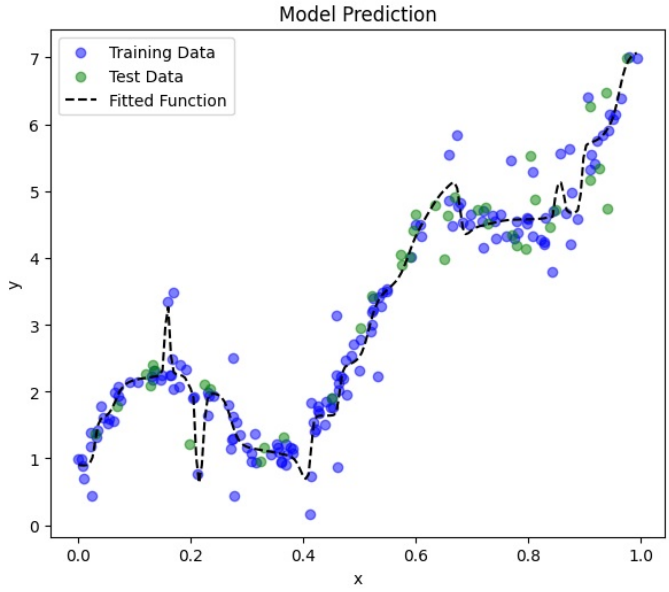
Fold 3, Epochs: 2000, Test MSE: 0.1817



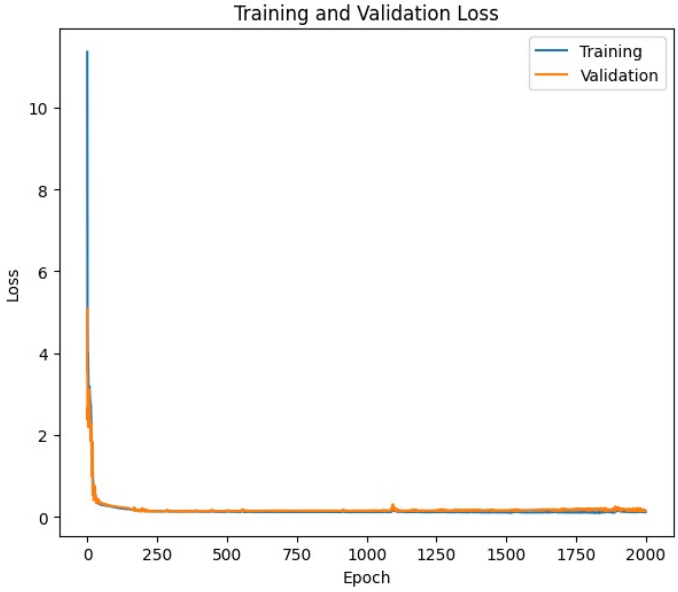


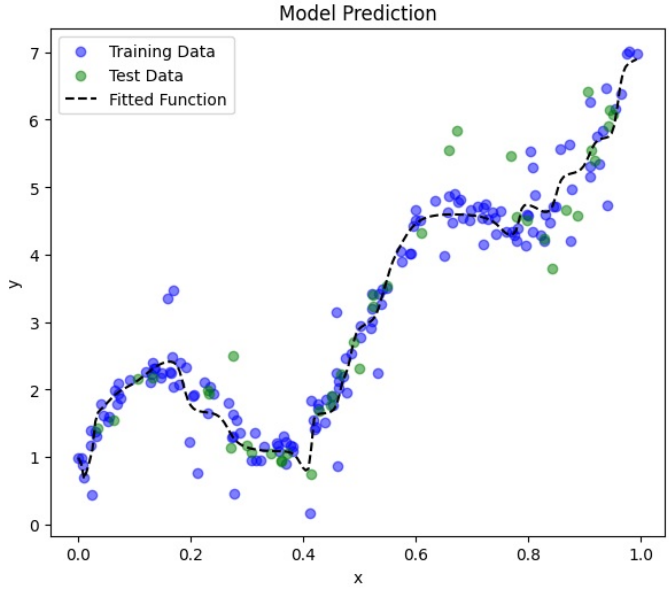
Fold 4, Epochs: 2000, Test MSE: 0.1884





Fold 5, Epochs: 2000, Test MSE: 0.2388





```
Average Test MSE for 2000 Epochs: 0.2096

Average MSEs for All Epoch Configurations:
Epochs: 100, Average Test MSE: 0.2832
Epochs: 500, Average Test MSE: 0.2077
Epochs: 2000, Average Test MSE: 0.2096
```

Discussion

Compare the averaged MSE result for the three different models, and comment on which number of epochs is most optimal. Why is it important that we perform cross validation when evaluating a model? For a given number of epochs, are all 5 of the k-fold models similar, or is there significant variation? Are some models underfit, overfit?

100 epochs: This model has the highest MSE, suggesting underfitting. The training was insufficient to fully capture the data patterns.

500 epochs: This model has the lowest MSE, indicating that it is the most optimal. It finds a balance between training long enough to model the data and avoiding overfitting.

2000 epochs: The performance is close to that of the 500-epoch model but slightly worse. This suggests slight overfitting, and the model might have started fitting noise in the training data, reducing its generalization to the test data.

With cross-validation, instead of relying on a single train-test split, it averages performance across multiple splits, reducing variability in the evaluation caused by how data is divided. It also helps assess how well the model generalizes to unseen data. Cross-validation also provides insights into whether the model is underfitting or overfitting by analyzing loss trends across folds.

For a fixed number of epochs, the 5 k-fold models are generally similar but can exhibit some variation due to data distribution and model complexity.

The 100 epochs model is underfit, the 2000 epochs model is slightly overfit, and the 500 epochs model is fit optimally, capturing the data patterns adequately while generalizing well to the test data.