ANN

April 17, 2024

1 Introduction

Name	Student ID	Email
Mahdi Asadolahzade	99442029	mahdiasadi140@gmail.com

In this project, we explore the diverse applications of neural networks across various stages of data analysis, ranging from preprocessing and model training to noise analysis and denoising techniques. The goal is to leverage neural networks to address real-world data challenges and derive meaningful insights through structured experimentation.

2 Building and Configuring a Multi-layer Neural Network

2.1 Abstract

In this phase, we will build a *multi-layer perceptron* (MLP) neural network to approximate several functions ranging from simple (like a linear equation) to complex (like a trigonometric function) within a specified domain. We will generate data points from these functions and use a portion of these points as our training set.

2.2 Set Up Environment

importing **libraries** we need

[ ]:
```
import tensorflow as tf
from tensorflow.keras.datasets import fashion_mnist
from tensorflow import keras
import matplotlib.pyplot as plt
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, accuracy_score, precision_score,␣
↪recall_score, f1_score
import seaborn as sns
from PIL import Image
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, r2_score
from tensorflow.keras.models import Model
```

```
from tensorflow.keras.layers import Input, Conv2D, MaxPooling2D, UpSampling2D
```

2.3 Define and Generate Data

Define a set of functions with inputs in one dimension (x) ranging from simple to complex. Generate data points from these functions within a specified domain

2.3.1 Functions

[ ]:
```
def linear_function(x):
    return 2 * x + 3

def sinusoidal_function(x):
    return np.sin(x)

def complicated_function(x):
    return np.log(x**2 + 1)
```

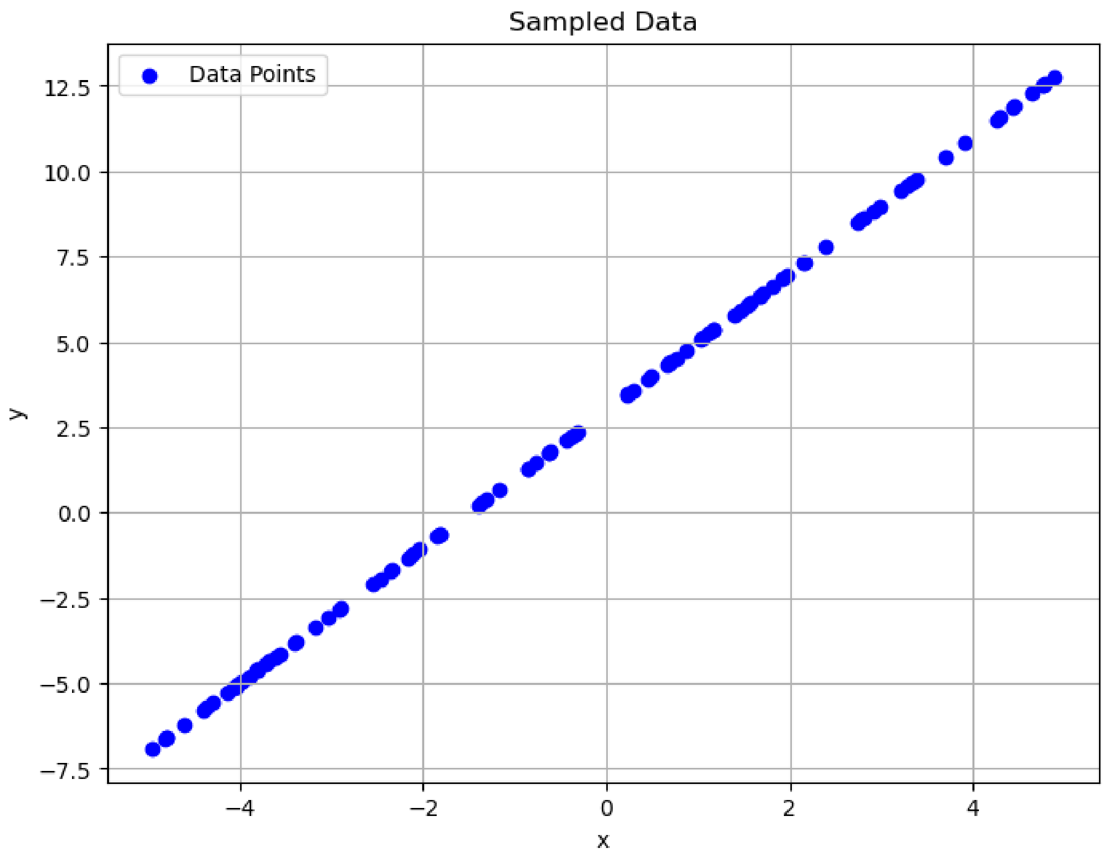
2.3.2 Define the domain and Visualize the function

[ ]:
```
np.random.seed(0)
num_points = 100
x_train = np.random.uniform(-5, 5, num_points)
```

2.3.3 linear_function

[ ]:
```
y_train_linear = linear_function(x_train)
```

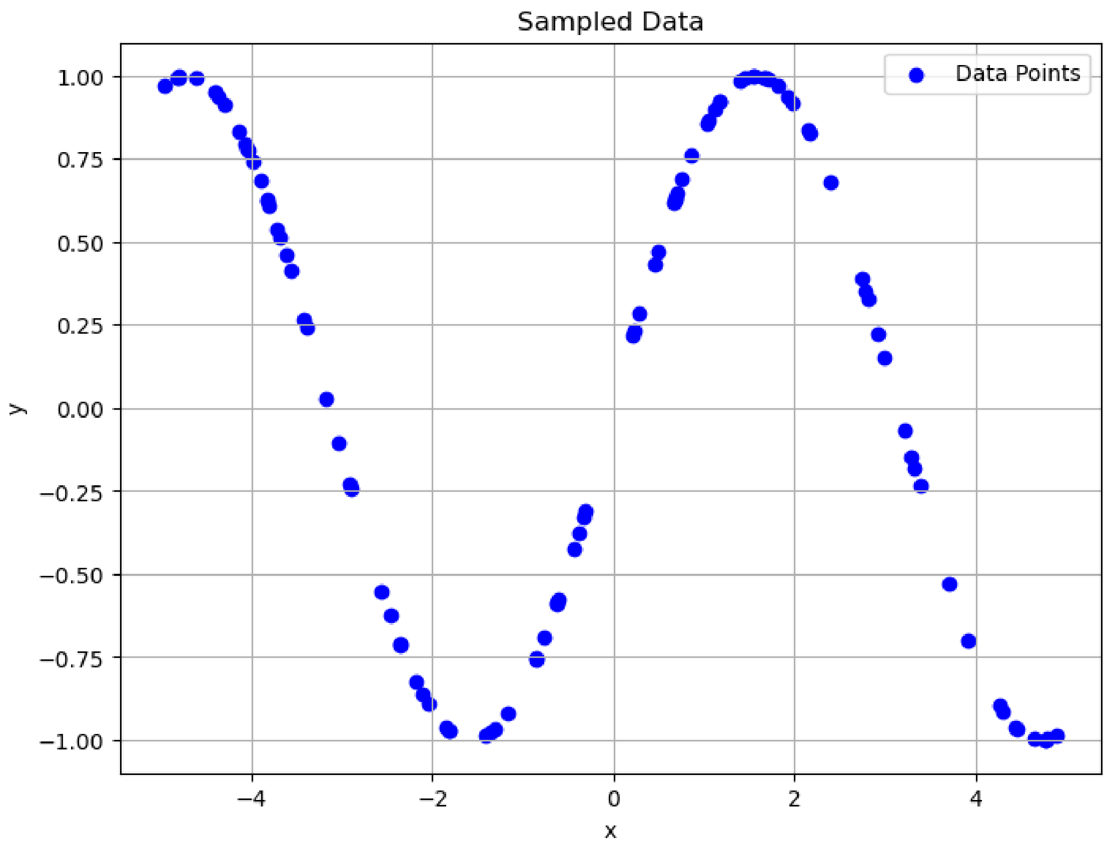
[ ]:
```
plt.figure(figsize=(8, 6))
plt.scatter(x_train, y_train_linear, color='blue', label='Data Points')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Sampled Data')
plt.legend()
plt.grid(True)
plt.show()
```



2.3.4 sinusoidal_function

[ ]:
```
y_train_sinusoidal = sinusoidal_function(x_train)
```

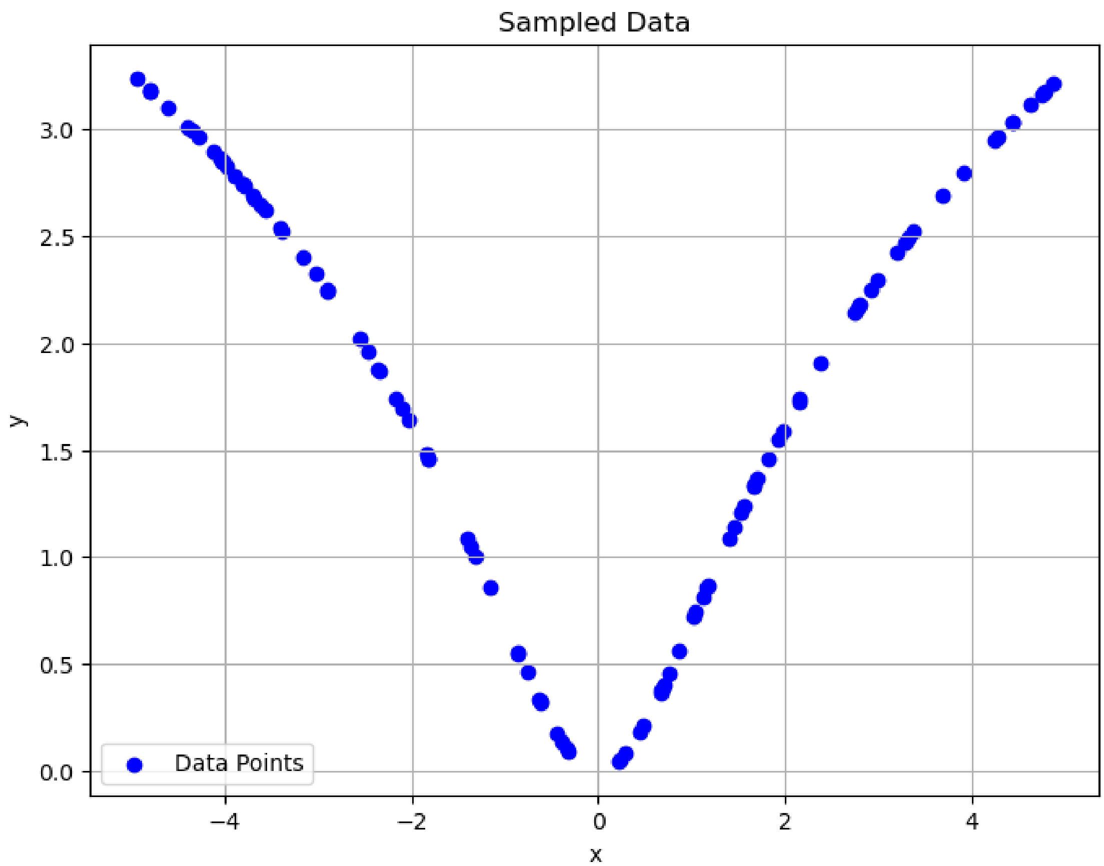
[ ]:
```
plt.figure(figsize=(8, 6))
plt.scatter(x_train, y_train_sinusoidal, color='blue', label='Data Points')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Sampled Data')
plt.legend()
plt.grid(True)
plt.show()
```



2.3.5 complicated_function

[ ]:
```
y_train_complicated = complicated_function(x_train)
```

[ ]:
```
plt.figure(figsize=(8, 6))
plt.scatter(x_train, y_train_complicated, color='blue', label='Data Points')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Sampled Data')
plt.legend()
plt.grid(True)
plt.show()
```



2.4 Split Data into Training and Testing Sets (linear)

Randomly select a portion of the generated data points as the training set.

[ ]:
```
np.random.seed(0)
num_points = 100
x_train = np.random.uniform(-5, 5, num_points)
x_train, x_test, y_train, y_test = train_test_split(x_train, y_train_linear,␣
 ↪test_size=0.2, random_state=42)
```

2.5 Build and Train the Neural Network (linear)

Construct a multi-layer perceptron (MLP) model using TensorFlow/Keras and train it using the generated training data.

[ ]:
```
model = keras.Sequential([
    keras.layers.Dense(64, activation='relu', input_shape=(1,)),
    keras.layers.Dense(64, activation='relu'),
    keras.layers.Dense(1)
])
```

```python
model.compile(optimizer='adam', loss='mse', metrics=['mae'])

history = model.fit(x_train, y_train, epochs=100, validation_data=(x_test,␣
↪y_test))
```

```
Epoch 1/100

c:\Users\Mahdi\anaconda3\Lib\site-packages\keras\src\layers\core\dense.py:86:
UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When
using Sequential models, prefer using an `Input(shape)` object as the first
layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)

3/3               1s 57ms/step - loss:
41.2417 - mae: 5.4635 - val_loss: 31.6440 - val_mae: 4.8090
Epoch 2/100
3/3               0s 8ms/step - loss:
38.0051 - mae: 5.3423 - val_loss: 29.8094 - val_mae: 4.6452
Epoch 3/100
3/3               0s 8ms/step - loss:
31.7557 - mae: 4.8023 - val_loss: 28.0839 - val_mae: 4.4869
Epoch 4/100
3/3               0s 16ms/step - loss:
31.1888 - mae: 4.8142 - val_loss: 26.4464 - val_mae: 4.3349
Epoch 5/100
3/3               0s 17ms/step - loss:
29.7433 - mae: 4.6084 - val_loss: 24.9195 - val_mae: 4.1911
Epoch 6/100
3/3               0s 16ms/step - loss:
27.1193 - mae: 4.3886 - val_loss: 23.5712 - val_mae: 4.0791
Epoch 7/100
3/3               0s 15ms/step - loss:
27.7794 - mae: 4.5093 - val_loss: 22.2420 - val_mae: 3.9645
Epoch 8/100
3/3               0s 7ms/step - loss:
22.7426 - mae: 4.0077 - val_loss: 20.9131 - val_mae: 3.8458
Epoch 9/100
3/3               0s 8ms/step - loss:
21.1803 - mae: 3.8536 - val_loss: 19.5650 - val_mae: 3.7235
Epoch 10/100
3/3               0s 9ms/step - loss:
21.2843 - mae: 3.8866 - val_loss: 18.2310 - val_mae: 3.6098
Epoch 11/100
3/3               0s 16ms/step - loss:
18.5651 - mae: 3.6051 - val_loss: 16.8922 - val_mae: 3.4959
Epoch 12/100
3/3               0s 16ms/step - loss:
17.1043 - mae: 3.4245 - val_loss: 15.5367 - val_mae: 3.3810
```

```
Epoch 13/100
3/3             0s 16ms/step - loss:
15.6545 - mae: 3.2568 - val_loss: 14.1703 - val_mae: 3.2562
Epoch 14/100
3/3             0s 6ms/step - loss:
13.4735 - mae: 3.0430 - val_loss: 12.8241 - val_mae: 3.1252
Epoch 15/100
3/3             0s 7ms/step - loss:
13.0822 - mae: 3.0015 - val_loss: 11.4951 - val_mae: 2.9877
Epoch 16/100
3/3             0s 7ms/step - loss:
10.5488 - mae: 2.6642 - val_loss: 10.2104 - val_mae: 2.8420
Epoch 17/100
3/3             0s 10ms/step - loss:
9.5677 - mae: 2.5809 - val_loss: 8.9482 - val_mae: 2.6864
Epoch 18/100
3/3             0s 8ms/step - loss:
8.6036 - mae: 2.4783 - val_loss: 7.7380 - val_mae: 2.5224
Epoch 19/100
3/3             0s 15ms/step - loss:
7.3484 - mae: 2.3145 - val_loss: 6.6068 - val_mae: 2.3545
Epoch 20/100
3/3             0s 17ms/step - loss:
5.7067 - mae: 2.0390 - val_loss: 5.5680 - val_mae: 2.1819
Epoch 21/100
3/3             0s 16ms/step - loss:
4.7083 - mae: 1.8713 - val_loss: 4.5923 - val_mae: 2.0004
Epoch 22/100
3/3             0s 15ms/step - loss:
3.5586 - mae: 1.6324 - val_loss: 3.7377 - val_mae: 1.8180
Epoch 23/100
3/3             0s 8ms/step - loss:
3.0322 - mae: 1.5393 - val_loss: 2.9896 - val_mae: 1.6338
Epoch 24/100
3/3             0s 8ms/step - loss:
2.3919 - mae: 1.3818 - val_loss: 2.3797 - val_mae: 1.4552
Epoch 25/100
3/3             0s 8ms/step - loss:
1.7866 - mae: 1.1745 - val_loss: 1.9207 - val_mae: 1.2860
Epoch 26/100
3/3             0s 9ms/step - loss:
1.4744 - mae: 1.0362 - val_loss: 1.5897 - val_mae: 1.1295
Epoch 27/100
3/3             0s 16ms/step - loss:
1.3571 - mae: 0.9613 - val_loss: 1.3629 - val_mae: 1.0357
Epoch 28/100
3/3             0s 16ms/step - loss:
1.1255 - mae: 0.8580 - val_loss: 1.2287 - val_mae: 0.9583
```

```
Epoch 29/100
3/3              0s 7ms/step - loss:
1.1538 - mae: 0.8802 - val_loss: 1.1417 - val_mae: 0.8968
Epoch 30/100
3/3              0s 8ms/step - loss:
1.1518 - mae: 0.8991 - val_loss: 1.0836 - val_mae: 0.8712
Epoch 31/100
3/3              0s 8ms/step - loss:
1.0696 - mae: 0.8579 - val_loss: 1.0440 - val_mae: 0.8610
Epoch 32/100
3/3              0s 16ms/step - loss:
1.1338 - mae: 0.9010 - val_loss: 1.0250 - val_mae: 0.8519
Epoch 33/100
3/3              0s 18ms/step - loss:
1.0852 - mae: 0.8906 - val_loss: 1.0072 - val_mae: 0.8424
Epoch 34/100
3/3              0s 16ms/step - loss:
1.0125 - mae: 0.8341 - val_loss: 0.9909 - val_mae: 0.8329
Epoch 35/100
3/3              0s 16ms/step - loss:
0.9898 - mae: 0.8460 - val_loss: 0.9708 - val_mae: 0.8226
Epoch 36/100
3/3              0s 15ms/step - loss:
0.9572 - mae: 0.8194 - val_loss: 0.9483 - val_mae: 0.8114
Epoch 37/100
3/3              0s 16ms/step - loss:
0.9493 - mae: 0.8213 - val_loss: 0.9284 - val_mae: 0.8092
Epoch 38/100
3/3              0s 15ms/step - loss:
0.9102 - mae: 0.7940 - val_loss: 0.9140 - val_mae: 0.8109
Epoch 39/100
3/3              0s 8ms/step - loss:
0.8779 - mae: 0.7876 - val_loss: 0.8974 - val_mae: 0.8122
Epoch 40/100
3/3              0s 8ms/step - loss:
0.8034 - mae: 0.7402 - val_loss: 0.8766 - val_mae: 0.8095
Epoch 41/100
3/3              0s 8ms/step - loss:
0.8465 - mae: 0.7673 - val_loss: 0.8491 - val_mae: 0.8030
Epoch 42/100
3/3              0s 8ms/step - loss:
0.7469 - mae: 0.7256 - val_loss: 0.8217 - val_mae: 0.7953
Epoch 43/100
3/3              0s 7ms/step - loss:
0.7614 - mae: 0.7254 - val_loss: 0.7912 - val_mae: 0.7836
Epoch 44/100
3/3              0s 17ms/step - loss:
0.6996 - mae: 0.6961 - val_loss: 0.7685 - val_mae: 0.7699
```

```
Epoch 45/100
3/3            0s 16ms/step - loss:
0.6749 - mae: 0.6807 - val_loss: 0.7409 - val_mae: 0.7553
Epoch 46/100
3/3            0s 16ms/step - loss:
0.6865 - mae: 0.7000 - val_loss: 0.7109 - val_mae: 0.7382
Epoch 47/100
3/3            0s 9ms/step - loss:
0.6588 - mae: 0.6795 - val_loss: 0.6841 - val_mae: 0.7240
Epoch 48/100
3/3            0s 8ms/step - loss:
0.6565 - mae: 0.6863 - val_loss: 0.6609 - val_mae: 0.7086
Epoch 49/100
3/3            0s 15ms/step - loss:
0.6200 - mae: 0.6693 - val_loss: 0.6367 - val_mae: 0.6929
Epoch 50/100
3/3            0s 16ms/step - loss:
0.5678 - mae: 0.6264 - val_loss: 0.6094 - val_mae: 0.6778
Epoch 51/100
3/3            0s 10ms/step - loss:
0.5892 - mae: 0.6455 - val_loss: 0.5820 - val_mae: 0.6634
Epoch 52/100
3/3            0s 7ms/step - loss:
0.5471 - mae: 0.6221 - val_loss: 0.5589 - val_mae: 0.6502
Epoch 53/100
3/3            0s 8ms/step - loss:
0.5594 - mae: 0.6398 - val_loss: 0.5405 - val_mae: 0.6412
Epoch 54/100
3/3            0s 9ms/step - loss:
0.4715 - mae: 0.5784 - val_loss: 0.5217 - val_mae: 0.6302
Epoch 55/100
3/3            0s 8ms/step - loss:
0.4394 - mae: 0.5520 - val_loss: 0.5024 - val_mae: 0.6173
Epoch 56/100
3/3            0s 8ms/step - loss:
0.4601 - mae: 0.5744 - val_loss: 0.4805 - val_mae: 0.6046
Epoch 57/100
3/3            0s 9ms/step - loss:
0.4287 - mae: 0.5564 - val_loss: 0.4577 - val_mae: 0.5923
Epoch 58/100
3/3            0s 8ms/step - loss:
0.4253 - mae: 0.5498 - val_loss: 0.4334 - val_mae: 0.5772
Epoch 59/100
3/3            0s 16ms/step - loss:
0.3694 - mae: 0.5089 - val_loss: 0.4100 - val_mae: 0.5612
Epoch 60/100
3/3            0s 16ms/step - loss:
0.3549 - mae: 0.5110 - val_loss: 0.3861 - val_mae: 0.5458
```

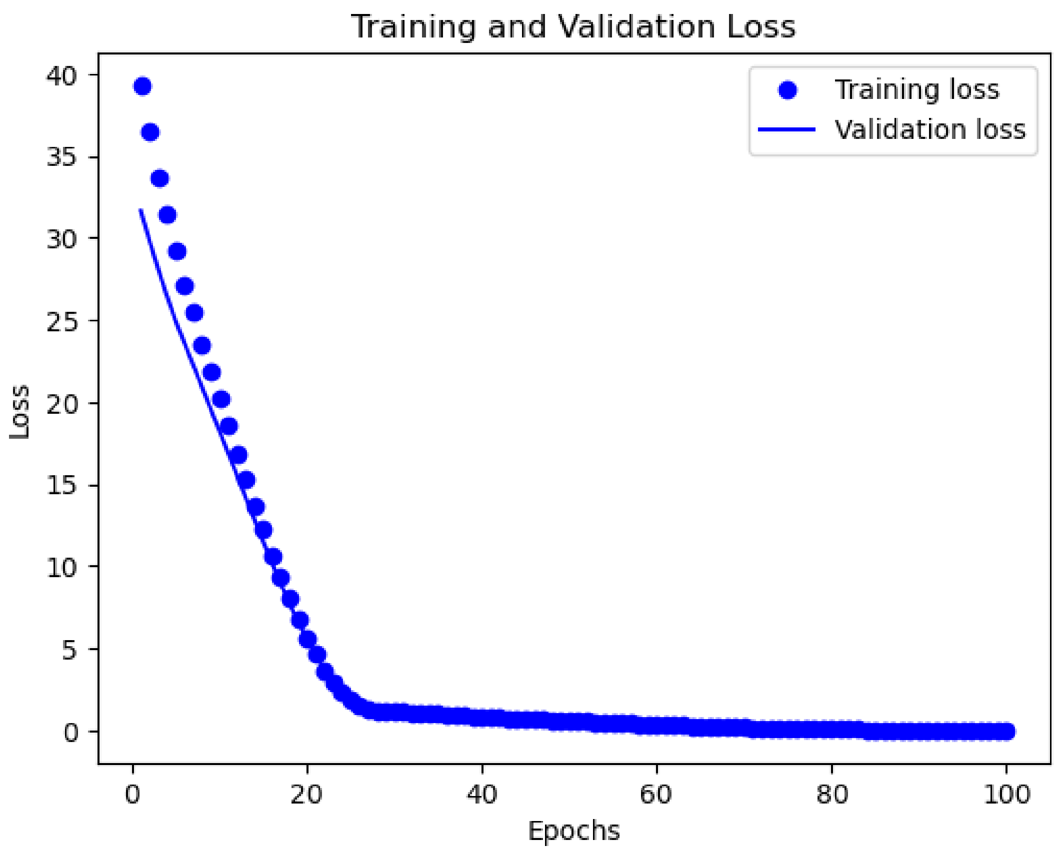
```
Epoch 61/100
3/3            0s 8ms/step - loss:
0.3414 - mae: 0.4902 - val_loss: 0.3645 - val_mae: 0.5290
Epoch 62/100
3/3            0s 8ms/step - loss:
0.3058 - mae: 0.4762 - val_loss: 0.3405 - val_mae: 0.5132
Epoch 63/100
3/3            0s 8ms/step - loss:
0.3146 - mae: 0.4799 - val_loss: 0.3146 - val_mae: 0.4944
Epoch 64/100
3/3            0s 7ms/step - loss:
0.2999 - mae: 0.4718 - val_loss: 0.2958 - val_mae: 0.4793
Epoch 65/100
3/3            0s 8ms/step - loss:
0.2752 - mae: 0.4583 - val_loss: 0.2798 - val_mae: 0.4641
Epoch 66/100
3/3            0s 9ms/step - loss:
0.2619 - mae: 0.4426 - val_loss: 0.2611 - val_mae: 0.4468
Epoch 67/100
3/3            0s 8ms/step - loss:
0.2309 - mae: 0.4128 - val_loss: 0.2423 - val_mae: 0.4294
Epoch 68/100
3/3            0s 16ms/step - loss:
0.2197 - mae: 0.4041 - val_loss: 0.2225 - val_mae: 0.4116
Epoch 69/100
3/3            0s 16ms/step - loss:
0.2058 - mae: 0.3922 - val_loss: 0.2058 - val_mae: 0.3960
Epoch 70/100
3/3            0s 16ms/step - loss:
0.1934 - mae: 0.3841 - val_loss: 0.1926 - val_mae: 0.3821
Epoch 71/100
3/3            0s 15ms/step - loss:
0.1802 - mae: 0.3624 - val_loss: 0.1801 - val_mae: 0.3680
Epoch 72/100
3/3            0s 16ms/step - loss:
0.1674 - mae: 0.3483 - val_loss: 0.1679 - val_mae: 0.3551
Epoch 73/100
3/3            0s 8ms/step - loss:
0.1509 - mae: 0.3323 - val_loss: 0.1546 - val_mae: 0.3409
Epoch 74/100
3/3            0s 8ms/step - loss:
0.1307 - mae: 0.3100 - val_loss: 0.1415 - val_mae: 0.3270
Epoch 75/100
3/3            0s 16ms/step - loss:
0.1369 - mae: 0.3220 - val_loss: 0.1310 - val_mae: 0.3131
Epoch 76/100
3/3            0s 16ms/step - loss:
0.1134 - mae: 0.2874 - val_loss: 0.1212 - val_mae: 0.3005
```

```
Epoch 77/100
3/3            0s 17ms/step - loss:
0.1088 - mae: 0.2851 - val_loss: 0.1132 - val_mae: 0.2890
Epoch 78/100
3/3            0s 16ms/step - loss:
0.1006 - mae: 0.2698 - val_loss: 0.1058 - val_mae: 0.2803
Epoch 79/100
3/3            0s 8ms/step - loss:
0.0880 - mae: 0.2537 - val_loss: 0.0985 - val_mae: 0.2714
Epoch 80/100
3/3            0s 7ms/step - loss:
0.0849 - mae: 0.2466 - val_loss: 0.0912 - val_mae: 0.2610
Epoch 81/100
3/3            0s 8ms/step - loss:
0.0729 - mae: 0.2265 - val_loss: 0.0839 - val_mae: 0.2510
Epoch 82/100
3/3            0s 7ms/step - loss:
0.0659 - mae: 0.2174 - val_loss: 0.0779 - val_mae: 0.2419
Epoch 83/100
3/3            0s 8ms/step - loss:
0.0647 - mae: 0.2126 - val_loss: 0.0733 - val_mae: 0.2337
Epoch 84/100
3/3            0s 9ms/step - loss:
0.0598 - mae: 0.2032 - val_loss: 0.0681 - val_mae: 0.2247
Epoch 85/100
3/3            0s 8ms/step - loss:
0.0527 - mae: 0.1904 - val_loss: 0.0625 - val_mae: 0.2149
Epoch 86/100
3/3            0s 8ms/step - loss:
0.0481 - mae: 0.1800 - val_loss: 0.0540 - val_mae: 0.1983
Epoch 87/100
3/3            0s 16ms/step - loss:
0.0425 - mae: 0.1695 - val_loss: 0.0470 - val_mae: 0.1819
Epoch 88/100
3/3            0s 16ms/step - loss:
0.0392 - mae: 0.1632 - val_loss: 0.0432 - val_mae: 0.1752
Epoch 89/100
3/3            0s 16ms/step - loss:
0.0326 - mae: 0.1464 - val_loss: 0.0406 - val_mae: 0.1711
Epoch 90/100
3/3            0s 16ms/step - loss:
0.0311 - mae: 0.1385 - val_loss: 0.0372 - val_mae: 0.1636
Epoch 91/100
3/3            0s 8ms/step - loss:
0.0290 - mae: 0.1370 - val_loss: 0.0331 - val_mae: 0.1548
Epoch 92/100
3/3            0s 15ms/step - loss:
0.0258 - mae: 0.1314 - val_loss: 0.0294 - val_mae: 0.1466
```

```
Epoch 93/100
3/3              0s 16ms/step - loss:
0.0239 - mae: 0.1226 - val_loss: 0.0258 - val_mae: 0.1386
Epoch 94/100
3/3              0s 8ms/step - loss:
0.0213 - mae: 0.1188 - val_loss: 0.0225 - val_mae: 0.1287
Epoch 95/100
3/3              0s 8ms/step - loss:
0.0167 - mae: 0.1012 - val_loss: 0.0200 - val_mae: 0.1212
Epoch 96/100
3/3              0s 7ms/step - loss:
0.0165 - mae: 0.1041 - val_loss: 0.0179 - val_mae: 0.1141
Epoch 97/100
3/3              0s 5ms/step - loss:
0.0142 - mae: 0.0965 - val_loss: 0.0161 - val_mae: 0.1077
Epoch 98/100
3/3              0s 8ms/step - loss:
0.0140 - mae: 0.0956 - val_loss: 0.0151 - val_mae: 0.1035
Epoch 99/100
3/3              0s 16ms/step - loss:
0.0122 - mae: 0.0879 - val_loss: 0.0139 - val_mae: 0.0992
Epoch 100/100
3/3              0s 9ms/step - loss:
0.0116 - mae: 0.0874 - val_loss: 0.0126 - val_mae: 0.0928
```

2.6 Evaluate and Visualize Model Performance (linear)

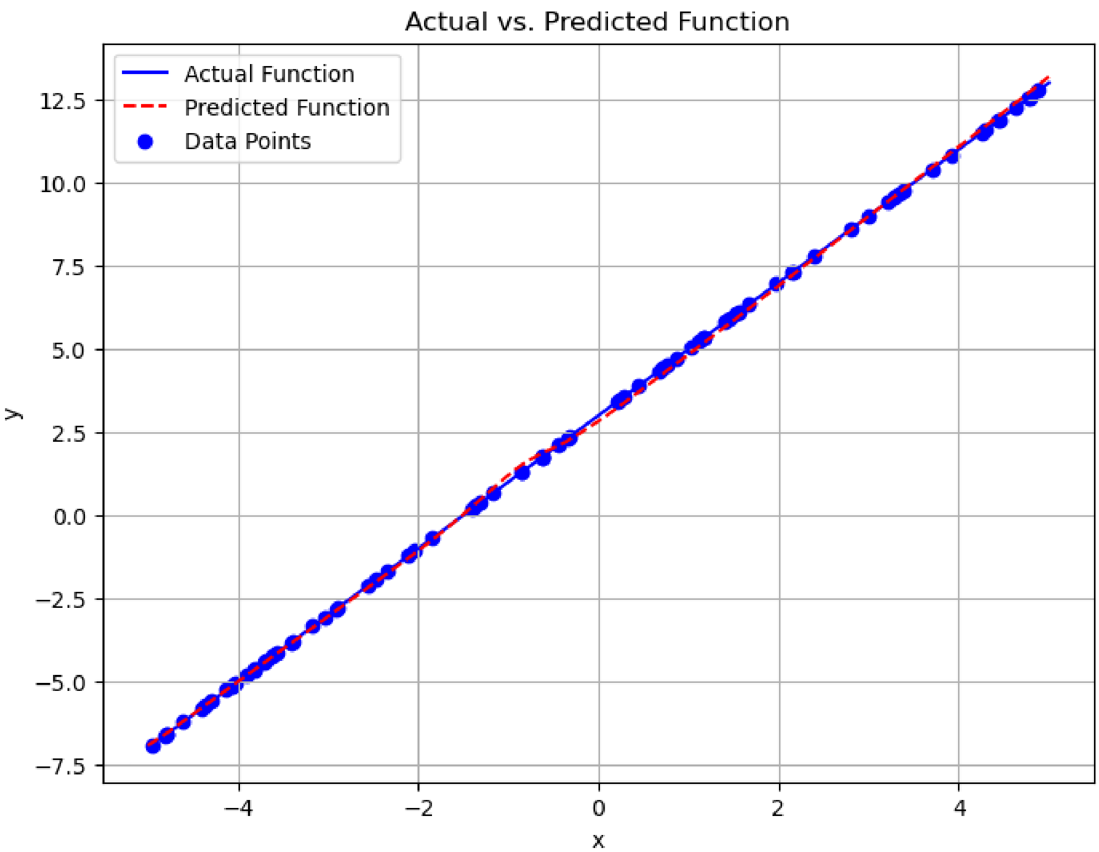
[ ]:
```
loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(1, len(loss) + 1)

plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```



2.6.1 Actual vs. Predicted Function (linear)

[ ]:
```
x_range = np.linspace(-5, 5, 100)
y_actual = linear_function(x_range)
y_predicted = model.predict(x_range)

plt.figure(figsize=(8, 6))
plt.plot(x_range, y_actual, label='Actual Function', color='blue')
plt.plot(x_range, y_predicted, label='Predicted Function', color='red',␣
↪linestyle='--')
plt.scatter(x_train, y_train, color='blue', label='Data Points')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Actual vs. Predicted Function')
plt.legend()
plt.grid(True)
plt.show()
```

```
4/4                0s 10ms/step
```



2.6.2 MAE & MSE (linear)

[ ]:
```
y_predicted = np.squeeze(y_predicted)
mae = np.mean(np.abs(y_actual - y_predicted))
print(f'Mean Absolute Error (MAE): {mae}')
```

```
Mean Absolute Error (MAE): 0.08467582718666748
```

[ ]:
```
x_test = np.linspace(0.1, 5, 50)
y_test = linear_function(x_test)
test_loss, test_mae = model.evaluate(x_test, y_test, verbose=0)
print(f'Test Loss (MSE): {test_loss}')
```

```
Test Loss (MSE): 0.014696547761559486
```

2.7 Split Data into Training and Testing Sets (sinusoidal)

[ ]:
```
np.random.seed(0)
num_points = 100
x_train = np.random.uniform(-5, 5, num_points)
```

```
x_train, x_test, y_train, y_test = train_test_split(x_train,␣
↪y_train_sinusoidal, test_size=0.2, random_state=42)
```

2.8 Build and Train the Neural Network (sinusoidal)

[ ]:
```
model = keras.Sequential([
    keras.layers.Dense(64, activation='relu', input_shape=(1,)),
    keras.layers.Dense(64, activation='relu'),
    keras.layers.Dense(1)
])

model.compile(optimizer='adam', loss='mse', metrics=['mae'])

history = model.fit(x_train, y_train, epochs=100, validation_data=(x_test,␣
↪y_test))
```

```
Epoch 1/100
3/3              1s 58ms/step - loss:
0.4773 - mae: 0.6065 - val_loss: 0.6601 - val_mae: 0.7557
Epoch 2/100
3/3              0s 9ms/step - loss:
0.5130 - mae: 0.6354 - val_loss: 0.6763 - val_mae: 0.7685
Epoch 3/100
3/3              0s 8ms/step - loss:
0.4473 - mae: 0.5782 - val_loss: 0.6809 - val_mae: 0.7686
Epoch 4/100
3/3              0s 8ms/step - loss:
0.4558 - mae: 0.5846 - val_loss: 0.6800 - val_mae: 0.7623
Epoch 5/100
3/3              0s 8ms/step - loss:
0.4743 - mae: 0.6005 - val_loss: 0.6600 - val_mae: 0.7520
Epoch 6/100
3/3              0s 8ms/step - loss:
0.4412 - mae: 0.5774 - val_loss: 0.6377 - val_mae: 0.7395
Epoch 7/100
3/3              0s 8ms/step - loss:
0.4475 - mae: 0.5826 - val_loss: 0.6028 - val_mae: 0.7238
Epoch 8/100
3/3              0s 16ms/step - loss:
0.4409 - mae: 0.5942 - val_loss: 0.5896 - val_mae: 0.7174
Epoch 9/100
3/3              0s 17ms/step - loss:
0.4032 - mae: 0.5618 - val_loss: 0.5816 - val_mae: 0.7073
Epoch 10/100
3/3              0s 8ms/step - loss:
0.4072 - mae: 0.5593 - val_loss: 0.5793 - val_mae: 0.7015
Epoch 11/100
```

```
3/3             0s 8ms/step - loss:
0.4234 - mae: 0.5704 - val_loss: 0.5750 - val_mae: 0.6976
Epoch 12/100
3/3             0s 7ms/step - loss:
0.4096 - mae: 0.5572 - val_loss: 0.5687 - val_mae: 0.6950
Epoch 13/100
3/3             0s 8ms/step - loss:
0.3858 - mae: 0.5358 - val_loss: 0.5601 - val_mae: 0.6899
Epoch 14/100
3/3             0s 17ms/step - loss:
0.3905 - mae: 0.5431 - val_loss: 0.5420 - val_mae: 0.6829
Epoch 15/100
3/3             0s 16ms/step - loss:
0.3879 - mae: 0.5473 - val_loss: 0.5296 - val_mae: 0.6753
Epoch 16/100
3/3             0s 16ms/step - loss:
0.3537 - mae: 0.5207 - val_loss: 0.5196 - val_mae: 0.6639
Epoch 17/100
3/3             0s 8ms/step - loss:
0.3394 - mae: 0.5022 - val_loss: 0.5078 - val_mae: 0.6521
Epoch 18/100
3/3             0s 8ms/step - loss:
0.3442 - mae: 0.5036 - val_loss: 0.4941 - val_mae: 0.6409
Epoch 19/100
3/3             0s 7ms/step - loss:
0.3299 - mae: 0.4951 - val_loss: 0.4800 - val_mae: 0.6297
Epoch 20/100
3/3             0s 7ms/step - loss:
0.3186 - mae: 0.4759 - val_loss: 0.4663 - val_mae: 0.6220
Epoch 21/100
3/3             0s 9ms/step - loss:
0.3097 - mae: 0.4813 - val_loss: 0.4417 - val_mae: 0.6113
Epoch 22/100
3/3             0s 10ms/step - loss:
0.3028 - mae: 0.4811 - val_loss: 0.4311 - val_mae: 0.6027
Epoch 23/100
3/3             0s 16ms/step - loss:
0.2830 - mae: 0.4654 - val_loss: 0.4187 - val_mae: 0.5879
Epoch 24/100
3/3             0s 8ms/step - loss:
0.2851 - mae: 0.4626 - val_loss: 0.4009 - val_mae: 0.5695
Epoch 25/100
3/3             0s 8ms/step - loss:
0.2465 - mae: 0.4204 - val_loss: 0.3886 - val_mae: 0.5548
Epoch 26/100
3/3             0s 16ms/step - loss:
0.2374 - mae: 0.4013 - val_loss: 0.3575 - val_mae: 0.5423
Epoch 27/100
```

```
3/3            0s 16ms/step - loss:
0.2265 - mae: 0.4128 - val_loss: 0.3281 - val_mae: 0.5245
Epoch 28/100
3/3            0s 15ms/step - loss:
0.2117 - mae: 0.4052 - val_loss: 0.3224 - val_mae: 0.5195
Epoch 29/100
3/3            0s 16ms/step - loss:
0.2070 - mae: 0.3962 - val_loss: 0.3067 - val_mae: 0.5020
Epoch 30/100
3/3            0s 17ms/step - loss:
0.2038 - mae: 0.3927 - val_loss: 0.3031 - val_mae: 0.4901
Epoch 31/100
3/3            0s 16ms/step - loss:
0.1794 - mae: 0.3519 - val_loss: 0.2893 - val_mae: 0.4741
Epoch 32/100
3/3            0s 16ms/step - loss:
0.1769 - mae: 0.3502 - val_loss: 0.2628 - val_mae: 0.4550
Epoch 33/100
3/3            0s 13ms/step - loss:
0.1524 - mae: 0.3314 - val_loss: 0.2318 - val_mae: 0.4304
Epoch 34/100
3/3            0s 8ms/step - loss:
0.1476 - mae: 0.3324 - val_loss: 0.2150 - val_mae: 0.4143
Epoch 35/100
3/3            0s 8ms/step - loss:
0.1365 - mae: 0.3206 - val_loss: 0.2029 - val_mae: 0.3941
Epoch 36/100
3/3            0s 8ms/step - loss:
0.1239 - mae: 0.2888 - val_loss: 0.1905 - val_mae: 0.3779
Epoch 37/100
3/3            0s 17ms/step - loss:
0.0988 - mae: 0.2541 - val_loss: 0.1702 - val_mae: 0.3581
Epoch 38/100
3/3            0s 16ms/step - loss:
0.0946 - mae: 0.2526 - val_loss: 0.1531 - val_mae: 0.3425
Epoch 39/100
3/3            0s 8ms/step - loss:
0.0904 - mae: 0.2584 - val_loss: 0.1403 - val_mae: 0.3220
Epoch 40/100
3/3            0s 7ms/step - loss:
0.0779 - mae: 0.2306 - val_loss: 0.1270 - val_mae: 0.2993
Epoch 41/100
3/3            0s 8ms/step - loss:
0.0726 - mae: 0.2124 - val_loss: 0.1121 - val_mae: 0.2758
Epoch 42/100
3/3            0s 17ms/step - loss:
0.0621 - mae: 0.1951 - val_loss: 0.0936 - val_mae: 0.2553
Epoch 43/100
```

```
3/3            0s 16ms/step - loss:
0.0548 - mae: 0.1901 - val_loss: 0.0844 - val_mae: 0.2443
Epoch 44/100
3/3            0s 12ms/step - loss:
0.0504 - mae: 0.1829 - val_loss: 0.0789 - val_mae: 0.2334
Epoch 45/100
3/3            0s 16ms/step - loss:
0.0433 - mae: 0.1603 - val_loss: 0.0725 - val_mae: 0.2189
Epoch 46/100
3/3            0s 8ms/step - loss:
0.0423 - mae: 0.1577 - val_loss: 0.0601 - val_mae: 0.2023
Epoch 47/100
3/3            0s 7ms/step - loss:
0.0365 - mae: 0.1521 - val_loss: 0.0500 - val_mae: 0.1876
Epoch 48/100
3/3            0s 8ms/step - loss:
0.0313 - mae: 0.1437 - val_loss: 0.0440 - val_mae: 0.1759
Epoch 49/100
3/3            0s 9ms/step - loss:
0.0268 - mae: 0.1323 - val_loss: 0.0384 - val_mae: 0.1615
Epoch 50/100
3/3            0s 16ms/step - loss:
0.0232 - mae: 0.1209 - val_loss: 0.0328 - val_mae: 0.1499
Epoch 51/100
3/3            0s 16ms/step - loss:
0.0217 - mae: 0.1182 - val_loss: 0.0304 - val_mae: 0.1475
Epoch 52/100
3/3            0s 15ms/step - loss:
0.0188 - mae: 0.1107 - val_loss: 0.0256 - val_mae: 0.1359
Epoch 53/100
3/3            0s 8ms/step - loss:
0.0172 - mae: 0.1055 - val_loss: 0.0207 - val_mae: 0.1181
Epoch 54/100
3/3            0s 16ms/step - loss:
0.0159 - mae: 0.0991 - val_loss: 0.0193 - val_mae: 0.1108
Epoch 55/100
3/3            0s 8ms/step - loss:
0.0147 - mae: 0.0927 - val_loss: 0.0191 - val_mae: 0.1150
Epoch 56/100
3/3            0s 17ms/step - loss:
0.0161 - mae: 0.0992 - val_loss: 0.0144 - val_mae: 0.0977
Epoch 57/100
3/3            0s 16ms/step - loss:
0.0130 - mae: 0.0885 - val_loss: 0.0118 - val_mae: 0.0823
Epoch 58/100
3/3            0s 8ms/step - loss:
0.0136 - mae: 0.0817 - val_loss: 0.0121 - val_mae: 0.0823
Epoch 59/100
```

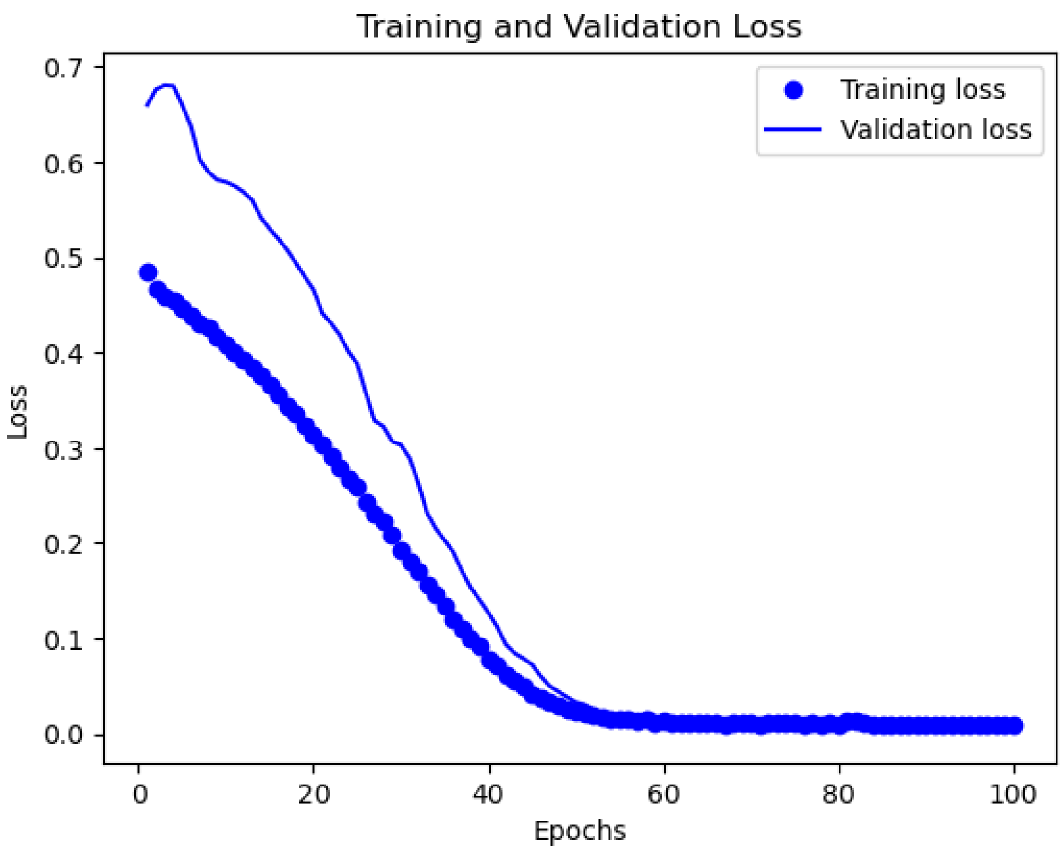
```
3/3            0s 8ms/step - loss:
0.0122 - mae: 0.0823 - val_loss: 0.0114 - val_mae: 0.0840
Epoch 60/100
3/3            0s 16ms/step - loss:
0.0125 - mae: 0.0839 - val_loss: 0.0102 - val_mae: 0.0795
Epoch 61/100
3/3            0s 9ms/step - loss:
0.0109 - mae: 0.0750 - val_loss: 0.0087 - val_mae: 0.0677
Epoch 62/100
3/3            0s 16ms/step - loss:
0.0122 - mae: 0.0800 - val_loss: 0.0078 - val_mae: 0.0654
Epoch 63/100
3/3            0s 16ms/step - loss:
0.0116 - mae: 0.0799 - val_loss: 0.0076 - val_mae: 0.0636
Epoch 64/100
3/3            0s 8ms/step - loss:
0.0106 - mae: 0.0770 - val_loss: 0.0065 - val_mae: 0.0603
Epoch 65/100
3/3            0s 8ms/step - loss:
0.0105 - mae: 0.0730 - val_loss: 0.0062 - val_mae: 0.0585
Epoch 66/100
3/3            0s 9ms/step - loss:
0.0100 - mae: 0.0714 - val_loss: 0.0063 - val_mae: 0.0572
Epoch 67/100
3/3            0s 16ms/step - loss:
0.0095 - mae: 0.0733 - val_loss: 0.0072 - val_mae: 0.0644
Epoch 68/100
3/3            0s 8ms/step - loss:
0.0109 - mae: 0.0732 - val_loss: 0.0067 - val_mae: 0.0585
Epoch 69/100
3/3            0s 8ms/step - loss:
0.0096 - mae: 0.0656 - val_loss: 0.0065 - val_mae: 0.0545
Epoch 70/100
3/3            0s 8ms/step - loss:
0.0113 - mae: 0.0788 - val_loss: 0.0049 - val_mae: 0.0431
Epoch 71/100
3/3            0s 7ms/step - loss:
0.0099 - mae: 0.0735 - val_loss: 0.0051 - val_mae: 0.0505
Epoch 72/100
3/3            0s 16ms/step - loss:
0.0099 - mae: 0.0666 - val_loss: 0.0064 - val_mae: 0.0578
Epoch 73/100
3/3            0s 17ms/step - loss:
0.0133 - mae: 0.0805 - val_loss: 0.0061 - val_mae: 0.0532
Epoch 74/100
3/3            0s 11ms/step - loss:
0.0113 - mae: 0.0780 - val_loss: 0.0037 - val_mae: 0.0390
Epoch 75/100
```

```
3/3              0s 8ms/step - loss:
0.0102 - mae: 0.0711 - val_loss: 0.0042 - val_mae: 0.0437
Epoch 76/100
3/3              0s 16ms/step - loss:
0.0086 - mae: 0.0638 - val_loss: 0.0081 - val_mae: 0.0661
Epoch 77/100
3/3              0s 16ms/step - loss:
0.0112 - mae: 0.0741 - val_loss: 0.0072 - val_mae: 0.0602
Epoch 78/100
3/3              0s 7ms/step - loss:
0.0092 - mae: 0.0677 - val_loss: 0.0036 - val_mae: 0.0399
Epoch 79/100
3/3              0s 8ms/step - loss:
0.0103 - mae: 0.0698 - val_loss: 0.0036 - val_mae: 0.0392
Epoch 80/100
3/3              0s 8ms/step - loss:
0.0083 - mae: 0.0650 - val_loss: 0.0060 - val_mae: 0.0519
Epoch 81/100
3/3              0s 16ms/step - loss:
0.0118 - mae: 0.0764 - val_loss: 0.0081 - val_mae: 0.0659
Epoch 82/100
3/3              0s 16ms/step - loss:
0.0126 - mae: 0.0784 - val_loss: 0.0066 - val_mae: 0.0557
Epoch 83/100
3/3              0s 15ms/step - loss:
0.0136 - mae: 0.0723 - val_loss: 0.0049 - val_mae: 0.0470
Epoch 84/100
3/3              0s 8ms/step - loss:
0.0093 - mae: 0.0719 - val_loss: 0.0050 - val_mae: 0.0451
Epoch 85/100
3/3              0s 9ms/step - loss:
0.0102 - mae: 0.0726 - val_loss: 0.0040 - val_mae: 0.0414
Epoch 86/100
3/3              0s 8ms/step - loss:
0.0097 - mae: 0.0679 - val_loss: 0.0038 - val_mae: 0.0412
Epoch 87/100
3/3              0s 8ms/step - loss:
0.0115 - mae: 0.0710 - val_loss: 0.0054 - val_mae: 0.0522
Epoch 88/100
3/3              0s 16ms/step - loss:
0.0099 - mae: 0.0733 - val_loss: 0.0053 - val_mae: 0.0508
Epoch 89/100
3/3              0s 15ms/step - loss:
0.0090 - mae: 0.0667 - val_loss: 0.0041 - val_mae: 0.0430
Epoch 90/100
3/3              0s 16ms/step - loss:
0.0084 - mae: 0.0642 - val_loss: 0.0043 - val_mae: 0.0436
Epoch 91/100
```

```
3/3               0s 8ms/step - loss:
0.0078 - mae: 0.0629 - val_loss: 0.0050 - val_mae: 0.0479
Epoch 92/100
3/3               0s 15ms/step - loss:
0.0087 - mae: 0.0664 - val_loss: 0.0048 - val_mae: 0.0462
Epoch 93/100
3/3               0s 8ms/step - loss:
0.0093 - mae: 0.0704 - val_loss: 0.0038 - val_mae: 0.0402
Epoch 94/100
3/3               0s 8ms/step - loss:
0.0101 - mae: 0.0717 - val_loss: 0.0037 - val_mae: 0.0395
Epoch 95/100
3/3               0s 7ms/step - loss:
0.0098 - mae: 0.0662 - val_loss: 0.0059 - val_mae: 0.0554
Epoch 96/100
3/3               0s 12ms/step - loss:
0.0106 - mae: 0.0726 - val_loss: 0.0047 - val_mae: 0.0466
Epoch 97/100
3/3               0s 9ms/step - loss:
0.0078 - mae: 0.0622 - val_loss: 0.0042 - val_mae: 0.0437
Epoch 98/100
3/3               0s 8ms/step - loss:
0.0104 - mae: 0.0739 - val_loss: 0.0034 - val_mae: 0.0376
Epoch 99/100
3/3               0s 8ms/step - loss:
0.0098 - mae: 0.0689 - val_loss: 0.0042 - val_mae: 0.0433
Epoch 100/100
3/3               0s 7ms/step - loss:
0.0096 - mae: 0.0651 - val_loss: 0.0048 - val_mae: 0.0469
```

2.9 Evaluate and Visualize Model Performance (sinusoidal)

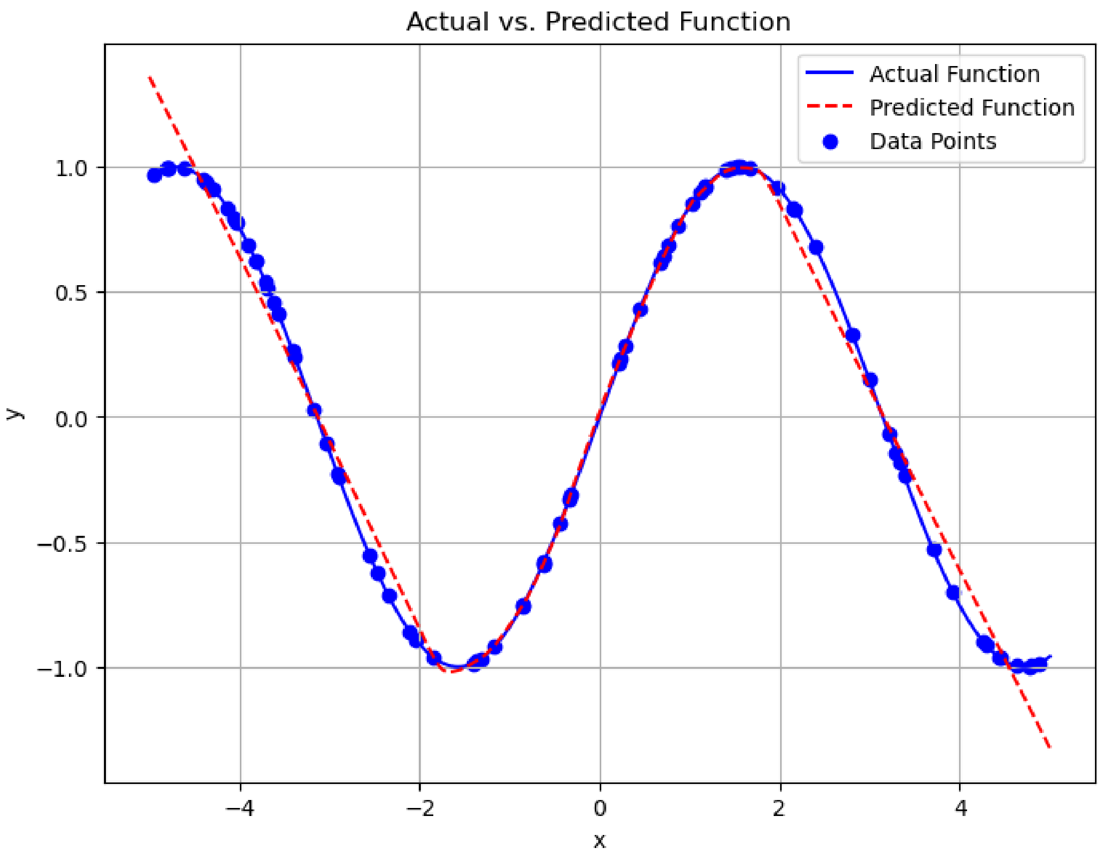
[ ]:
```
loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(1, len(loss) + 1)

plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```



2.9.1 Actual vs. Predicted Function (sinusoidal)

[ ]:
```
x_range = np.linspace(-5, 5, 1000)
y_actual = sinusoidal_function(x_range)
y_predicted = model.predict(x_range)

plt.figure(figsize=(8, 6))
plt.plot(x_range, y_actual, label='Actual Function', color='blue')
plt.plot(x_range, y_predicted, label='Predicted Function', color='red',␣
 ↪linestyle='--')
plt.scatter(x_train, y_train, color='blue', label='Data Points')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Actual vs. Predicted Function')
plt.legend()
plt.grid(True)
plt.show()
```

```
32/32                 0s 1ms/step
```



2.9.2 MAE , MSE (sinusoidal)

[ ]:
```
y_predicted = np.squeeze(y_predicted)
mae = np.mean(np.abs(y_actual - y_predicted))
print(f'Mean Absolute Error (MAE): {mae}')
```

```
Mean Absolute Error (MAE): 0.06452057732094311
```

[ ]:
```
x_test = np.linspace(0.1, 5, 50)
y_test = sinusoidal_function(x_test)
test_loss, test_mae = model.evaluate(x_test, y_test, verbose=0)
print(f'Test Loss (MSE): {test_loss}')
```

```
Test Loss (MSE): 0.010068106465041637
```

2.10 Split Data into Training and Testing Sets (complicated)

[ ]:
```
np.random.seed(0)
num_points = 100
x_train = np.random.uniform(-5, 5, num_points)
```

```
x_train, x_test, y_train, y_test = train_test_split(x_train,␣
↪y_train_complicated, test_size=0.2, random_state=42)
```

2.11 Build and Train the Neural Network (complicated)

```
[ ]: model = keras.Sequential([
         keras.layers.Dense(64, activation='relu', input_shape=(1,)),
         keras.layers.Dense(64, activation='relu'),
         keras.layers.Dense(1)
     ])

     model.compile(optimizer='adam', loss='mse', metrics=['mae'])

     history = model.fit(x_train, y_train, epochs=100, validation_data=(x_test,␣
     ↪y_test))
```

```
Epoch 1/100
3/3              1s 55ms/step - loss:
5.8194 - mae: 2.0323 - val_loss: 2.8355 - val_mae: 1.4073
Epoch 2/100
3/3              0s 8ms/step - loss:
4.7536 - mae: 1.8538 - val_loss: 2.3674 - val_mae: 1.2768
Epoch 3/100
3/3              0s 16ms/step - loss:
3.8200 - mae: 1.6262 - val_loss: 1.9591 - val_mae: 1.1531
Epoch 4/100
3/3              0s 8ms/step - loss:
3.0626 - mae: 1.4513 - val_loss: 1.6009 - val_mae: 1.0327
Epoch 5/100
3/3              0s 8ms/step - loss:
2.9274 - mae: 1.4608 - val_loss: 1.2779 - val_mae: 0.9144
Epoch 6/100
3/3              0s 16ms/step - loss:
2.2280 - mae: 1.2522 - val_loss: 0.9850 - val_mae: 0.7988
Epoch 7/100
3/3              0s 16ms/step - loss:
1.7074 - mae: 1.0892 - val_loss: 0.7275 - val_mae: 0.6861
Epoch 8/100
3/3              0s 8ms/step - loss:
1.2578 - mae: 0.9352 - val_loss: 0.5094 - val_mae: 0.5776
Epoch 9/100
3/3              0s 7ms/step - loss:
0.8319 - mae: 0.7578 - val_loss: 0.3394 - val_mae: 0.4743
Epoch 10/100
3/3              0s 16ms/step - loss:
0.5337 - mae: 0.5917 - val_loss: 0.2143 - val_mae: 0.3826
Epoch 11/100
```

```
3/3            0s 8ms/step - loss:
0.3433 - mae: 0.4821 - val_loss: 0.1268 - val_mae: 0.3021
Epoch 12/100
3/3            0s 8ms/step - loss:
0.1902 - mae: 0.3598 - val_loss: 0.0720 - val_mae: 0.2362
Epoch 13/100
3/3            0s 16ms/step - loss:
0.1027 - mae: 0.2760 - val_loss: 0.0441 - val_mae: 0.1868
Epoch 14/100
3/3            0s 8ms/step - loss:
0.0421 - mae: 0.1790 - val_loss: 0.0344 - val_mae: 0.1489
Epoch 15/100
3/3            0s 8ms/step - loss:
0.0340 - mae: 0.1533 - val_loss: 0.0340 - val_mae: 0.1253
Epoch 16/100
3/3            0s 9ms/step - loss:
0.0378 - mae: 0.1513 - val_loss: 0.0359 - val_mae: 0.1281
Epoch 17/100
3/3            0s 16ms/step - loss:
0.0439 - mae: 0.1668 - val_loss: 0.0357 - val_mae: 0.1297
Epoch 18/100
3/3            0s 8ms/step - loss:
0.0453 - mae: 0.1727 - val_loss: 0.0328 - val_mae: 0.1257
Epoch 19/100
3/3            0s 16ms/step - loss:
0.0374 - mae: 0.1528 - val_loss: 0.0288 - val_mae: 0.1235
Epoch 20/100
3/3            0s 12ms/step - loss:
0.0311 - mae: 0.1408 - val_loss: 0.0264 - val_mae: 0.1288
Epoch 21/100
3/3            0s 9ms/step - loss:
0.0253 - mae: 0.1276 - val_loss: 0.0273 - val_mae: 0.1432
Epoch 22/100
3/3            0s 8ms/step - loss:
0.0229 - mae: 0.1272 - val_loss: 0.0295 - val_mae: 0.1553
Epoch 23/100
3/3            0s 12ms/step - loss:
0.0256 - mae: 0.1369 - val_loss: 0.0308 - val_mae: 0.1604
Epoch 24/100
3/3            0s 16ms/step - loss:
0.0247 - mae: 0.1334 - val_loss: 0.0301 - val_mae: 0.1584
Epoch 25/100
3/3            0s 9ms/step - loss:
0.0249 - mae: 0.1357 - val_loss: 0.0290 - val_mae: 0.1540
Epoch 26/100
3/3            0s 8ms/step - loss:
0.0233 - mae: 0.1281 - val_loss: 0.0273 - val_mae: 0.1473
Epoch 27/100
```

```
3/3            0s 8ms/step - loss:
0.0216 - mae: 0.1238 - val_loss: 0.0259 - val_mae: 0.1408
Epoch 28/100
3/3            0s 9ms/step - loss:
0.0217 - mae: 0.1237 - val_loss: 0.0250 - val_mae: 0.1360
Epoch 29/100
3/3            0s 16ms/step - loss:
0.0224 - mae: 0.1226 - val_loss: 0.0243 - val_mae: 0.1330
Epoch 30/100
3/3            0s 8ms/step - loss:
0.0220 - mae: 0.1244 - val_loss: 0.0240 - val_mae: 0.1322
Epoch 31/100
3/3            0s 6ms/step - loss:
0.0232 - mae: 0.1278 - val_loss: 0.0239 - val_mae: 0.1336
Epoch 32/100
3/3            0s 16ms/step - loss:
0.0224 - mae: 0.1273 - val_loss: 0.0240 - val_mae: 0.1367
Epoch 33/100
3/3            0s 8ms/step - loss:
0.0222 - mae: 0.1265 - val_loss: 0.0241 - val_mae: 0.1387
Epoch 34/100
3/3            0s 13ms/step - loss:
0.0208 - mae: 0.1203 - val_loss: 0.0242 - val_mae: 0.1406
Epoch 35/100
3/3            0s 8ms/step - loss:
0.0201 - mae: 0.1196 - val_loss: 0.0242 - val_mae: 0.1415
Epoch 36/100
3/3            0s 10ms/step - loss:
0.0209 - mae: 0.1242 - val_loss: 0.0242 - val_mae: 0.1417
Epoch 37/100
3/3            0s 16ms/step - loss:
0.0201 - mae: 0.1218 - val_loss: 0.0239 - val_mae: 0.1405
Epoch 38/100
3/3            0s 16ms/step - loss:
0.0193 - mae: 0.1200 - val_loss: 0.0236 - val_mae: 0.1394
Epoch 39/100
3/3            0s 11ms/step - loss:
0.0189 - mae: 0.1169 - val_loss: 0.0234 - val_mae: 0.1392
Epoch 40/100
3/3            0s 8ms/step - loss:
0.0186 - mae: 0.1186 - val_loss: 0.0232 - val_mae: 0.1389
Epoch 41/100
3/3            0s 16ms/step - loss:
0.0193 - mae: 0.1206 - val_loss: 0.0231 - val_mae: 0.1389
Epoch 42/100
3/3            0s 8ms/step - loss:
0.0209 - mae: 0.1256 - val_loss: 0.0228 - val_mae: 0.1383
Epoch 43/100
```

```
3/3            0s 6ms/step - loss:
0.0186 - mae: 0.1182 - val_loss: 0.0224 - val_mae: 0.1367
Epoch 44/100
3/3            0s 8ms/step - loss:
0.0190 - mae: 0.1189 - val_loss: 0.0220 - val_mae: 0.1352
Epoch 45/100
3/3            0s 9ms/step - loss:
0.0195 - mae: 0.1196 - val_loss: 0.0218 - val_mae: 0.1346
Epoch 46/100
3/3            0s 16ms/step - loss:
0.0191 - mae: 0.1175 - val_loss: 0.0216 - val_mae: 0.1340
Epoch 47/100
3/3            0s 16ms/step - loss:
0.0187 - mae: 0.1178 - val_loss: 0.0215 - val_mae: 0.1343
Epoch 48/100
3/3            0s 7ms/step - loss:
0.0188 - mae: 0.1192 - val_loss: 0.0216 - val_mae: 0.1356
Epoch 49/100
3/3            0s 8ms/step - loss:
0.0197 - mae: 0.1224 - val_loss: 0.0216 - val_mae: 0.1357
Epoch 50/100
3/3            0s 17ms/step - loss:
0.0178 - mae: 0.1147 - val_loss: 0.0214 - val_mae: 0.1351
Epoch 51/100
3/3            0s 16ms/step - loss:
0.0191 - mae: 0.1210 - val_loss: 0.0213 - val_mae: 0.1349
Epoch 52/100
3/3            0s 16ms/step - loss:
0.0189 - mae: 0.1200 - val_loss: 0.0213 - val_mae: 0.1352
Epoch 53/100
3/3            0s 7ms/step - loss:
0.0180 - mae: 0.1161 - val_loss: 0.0213 - val_mae: 0.1353
Epoch 54/100
3/3            0s 8ms/step - loss:
0.0190 - mae: 0.1197 - val_loss: 0.0212 - val_mae: 0.1353
Epoch 55/100
3/3            0s 17ms/step - loss:
0.0181 - mae: 0.1185 - val_loss: 0.0210 - val_mae: 0.1346
Epoch 56/100
3/3            0s 8ms/step - loss:
0.0184 - mae: 0.1180 - val_loss: 0.0207 - val_mae: 0.1332
Epoch 57/100
3/3            0s 8ms/step - loss:
0.0177 - mae: 0.1160 - val_loss: 0.0205 - val_mae: 0.1321
Epoch 58/100
3/3            0s 16ms/step - loss:
0.0167 - mae: 0.1127 - val_loss: 0.0203 - val_mae: 0.1312
Epoch 59/100
```

```
3/3              0s 10ms/step - loss:
0.0181 - mae: 0.1161 - val_loss: 0.0201 - val_mae: 0.1309
Epoch 60/100
3/3              0s 8ms/step - loss:
0.0184 - mae: 0.1185 - val_loss: 0.0204 - val_mae: 0.1335
Epoch 61/100
3/3              0s 16ms/step - loss:
0.0171 - mae: 0.1135 - val_loss: 0.0205 - val_mae: 0.1340
Epoch 62/100
3/3              0s 13ms/step - loss:
0.0173 - mae: 0.1142 - val_loss: 0.0206 - val_mae: 0.1347
Epoch 63/100
3/3              0s 8ms/step - loss:
0.0169 - mae: 0.1130 - val_loss: 0.0203 - val_mae: 0.1329
Epoch 64/100
3/3              0s 8ms/step - loss:
0.0173 - mae: 0.1158 - val_loss: 0.0198 - val_mae: 0.1305
Epoch 65/100
3/3              0s 10ms/step - loss:
0.0162 - mae: 0.1103 - val_loss: 0.0197 - val_mae: 0.1303
Epoch 66/100
3/3              0s 16ms/step - loss:
0.0169 - mae: 0.1136 - val_loss: 0.0198 - val_mae: 0.1311
Epoch 67/100
3/3              0s 16ms/step - loss:
0.0170 - mae: 0.1133 - val_loss: 0.0198 - val_mae: 0.1314
Epoch 68/100
3/3              0s 16ms/step - loss:
0.0173 - mae: 0.1144 - val_loss: 0.0195 - val_mae: 0.1301
Epoch 69/100
3/3              0s 16ms/step - loss:
0.0173 - mae: 0.1158 - val_loss: 0.0193 - val_mae: 0.1291
Epoch 70/100
3/3              0s 8ms/step - loss:
0.0159 - mae: 0.1092 - val_loss: 0.0192 - val_mae: 0.1280
Epoch 71/100
3/3              0s 8ms/step - loss:
0.0148 - mae: 0.1044 - val_loss: 0.0189 - val_mae: 0.1272
Epoch 72/100
3/3              0s 16ms/step - loss:
0.0166 - mae: 0.1114 - val_loss: 0.0190 - val_mae: 0.1286
Epoch 73/100
3/3              0s 8ms/step - loss:
0.0158 - mae: 0.1089 - val_loss: 0.0190 - val_mae: 0.1290
Epoch 74/100
3/3              0s 8ms/step - loss:
0.0159 - mae: 0.1098 - val_loss: 0.0190 - val_mae: 0.1294
Epoch 75/100
```

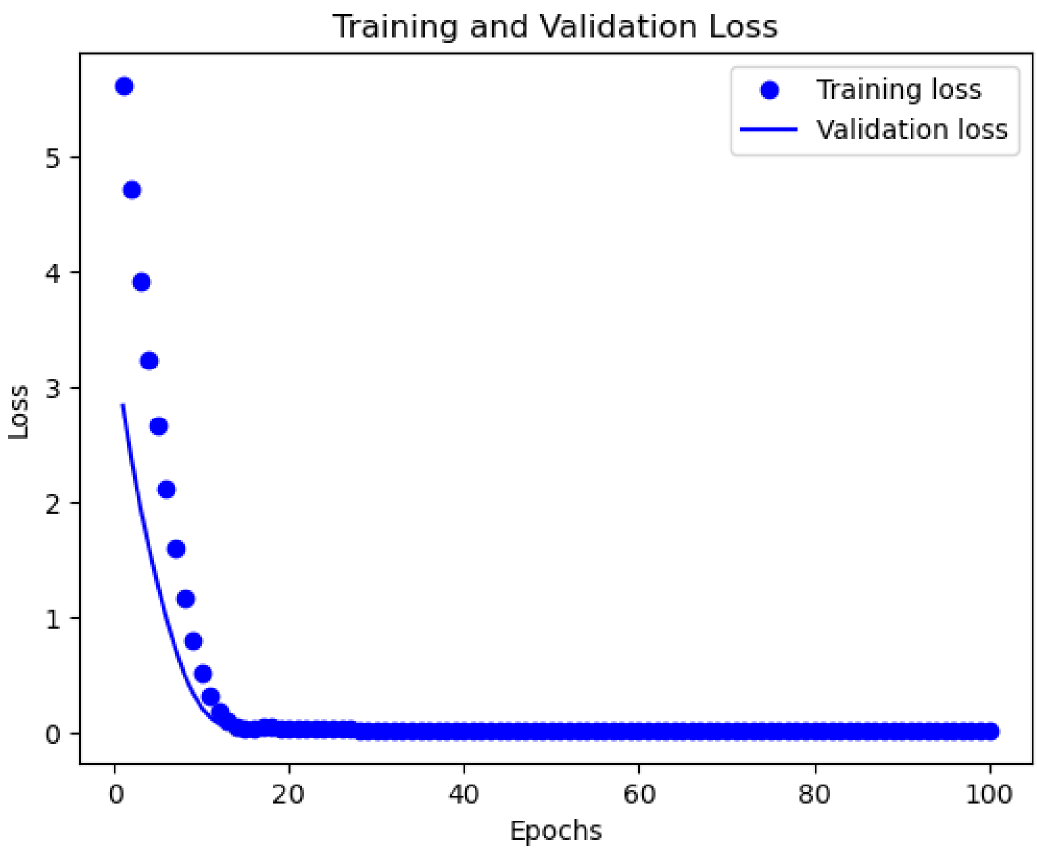
```
3/3            0s 8ms/step - loss:
0.0167 - mae: 0.1138 - val_loss: 0.0187 - val_mae: 0.1276
Epoch 76/100
3/3            0s 16ms/step - loss:
0.0155 - mae: 0.1080 - val_loss: 0.0184 - val_mae: 0.1253
Epoch 77/100
3/3            0s 9ms/step - loss:
0.0160 - mae: 0.1100 - val_loss: 0.0183 - val_mae: 0.1248
Epoch 78/100
3/3            0s 8ms/step - loss:
0.0164 - mae: 0.1118 - val_loss: 0.0185 - val_mae: 0.1251
Epoch 79/100
3/3            0s 17ms/step - loss:
0.0152 - mae: 0.1064 - val_loss: 0.0187 - val_mae: 0.1261
Epoch 80/100
3/3            0s 8ms/step - loss:
0.0169 - mae: 0.1145 - val_loss: 0.0192 - val_mae: 0.1287
Epoch 81/100
3/3            0s 8ms/step - loss:
0.0159 - mae: 0.1098 - val_loss: 0.0194 - val_mae: 0.1299
Epoch 82/100
3/3            0s 16ms/step - loss:
0.0164 - mae: 0.1122 - val_loss: 0.0191 - val_mae: 0.1288
Epoch 83/100
3/3            0s 16ms/step - loss:
0.0159 - mae: 0.1109 - val_loss: 0.0183 - val_mae: 0.1250
Epoch 84/100
3/3            0s 8ms/step - loss:
0.0155 - mae: 0.1087 - val_loss: 0.0177 - val_mae: 0.1226
Epoch 85/100
3/3            0s 8ms/step - loss:
0.0150 - mae: 0.1056 - val_loss: 0.0179 - val_mae: 0.1239
Epoch 86/100
3/3            0s 15ms/step - loss:
0.0150 - mae: 0.1068 - val_loss: 0.0184 - val_mae: 0.1266
Epoch 87/100
3/3            0s 16ms/step - loss:
0.0162 - mae: 0.1123 - val_loss: 0.0186 - val_mae: 0.1271
Epoch 88/100
3/3            0s 8ms/step - loss:
0.0154 - mae: 0.1087 - val_loss: 0.0183 - val_mae: 0.1254
Epoch 89/100
3/3            0s 11ms/step - loss:
0.0155 - mae: 0.1080 - val_loss: 0.0180 - val_mae: 0.1242
Epoch 90/100
3/3            0s 8ms/step - loss:
0.0161 - mae: 0.1095 - val_loss: 0.0177 - val_mae: 0.1223
Epoch 91/100
```

```
3/3              0s 8ms/step - loss:
0.0153 - mae: 0.1084 - val_loss: 0.0172 - val_mae: 0.1205
Epoch 92/100
3/3              0s 16ms/step - loss:
0.0156 - mae: 0.1073 - val_loss: 0.0171 - val_mae: 0.1203
Epoch 93/100
3/3              0s 16ms/step - loss:
0.0156 - mae: 0.1076 - val_loss: 0.0171 - val_mae: 0.1201
Epoch 94/100
3/3              0s 8ms/step - loss:
0.0149 - mae: 0.1054 - val_loss: 0.0171 - val_mae: 0.1204
Epoch 95/100
3/3              0s 8ms/step - loss:
0.0148 - mae: 0.1066 - val_loss: 0.0169 - val_mae: 0.1195
Epoch 96/100
3/3              0s 9ms/step - loss:
0.0145 - mae: 0.1046 - val_loss: 0.0169 - val_mae: 0.1194
Epoch 97/100
3/3              0s 16ms/step - loss:
0.0150 - mae: 0.1057 - val_loss: 0.0171 - val_mae: 0.1199
Epoch 98/100
3/3              0s 8ms/step - loss:
0.0147 - mae: 0.1037 - val_loss: 0.0177 - val_mae: 0.1230
Epoch 99/100
3/3              0s 7ms/step - loss:
0.0147 - mae: 0.1047 - val_loss: 0.0177 - val_mae: 0.1232
Epoch 100/100
3/3              0s 16ms/step - loss:
0.0143 - mae: 0.1039 - val_loss: 0.0172 - val_mae: 0.1206
```

2.12 Evaluate and Visualize Model Performance (complicated)

[ ]:

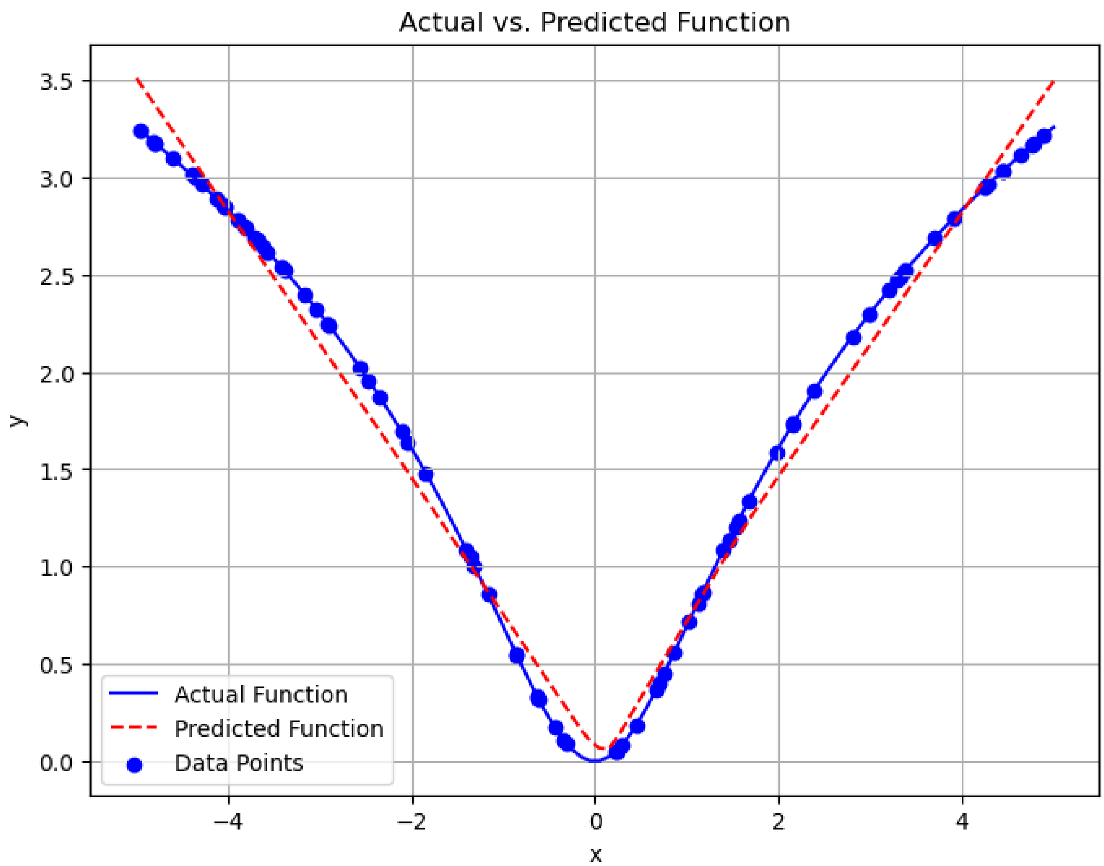
```
loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(1, len(loss) + 1)

plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```



2.12.1 Actual vs. Predicted Function (complicated)

[ ]:
```
x_range = np.linspace(-5, 5, 100)
y_actual = complicated_function(x_range)
y_predicted = model.predict(x_range)

plt.figure(figsize=(8, 6))
plt.plot(x_range, y_actual, label='Actual Function', color='blue')
plt.plot(x_range, y_predicted, label='Predicted Function', color='red',␣
↪linestyle='--')
plt.scatter(x_train, y_train, color='blue', label='Data Points')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Actual vs. Predicted Function')
plt.legend()
plt.grid(True)
plt.show()
```

```
4/4              0s 5ms/step
```



2.12.2 MAE , MSE (complicated)

[ ]:
```
y_predicted = np.squeeze(y_predicted)
mae = np.mean(np.abs(y_actual - y_predicted))
print(f'Mean Absolute Error (MAE): {mae}')
```

```
Mean Absolute Error (MAE): 0.11301847990242524
```

[ ]:
```
x_test = np.linspace(0.1, 5, 50)
y_test = complicated_function(x_test)
test_loss, test_mae = model.evaluate(x_test, y_test, verbose=0)
print(f'Test Loss (MSE): {test_loss}')
```

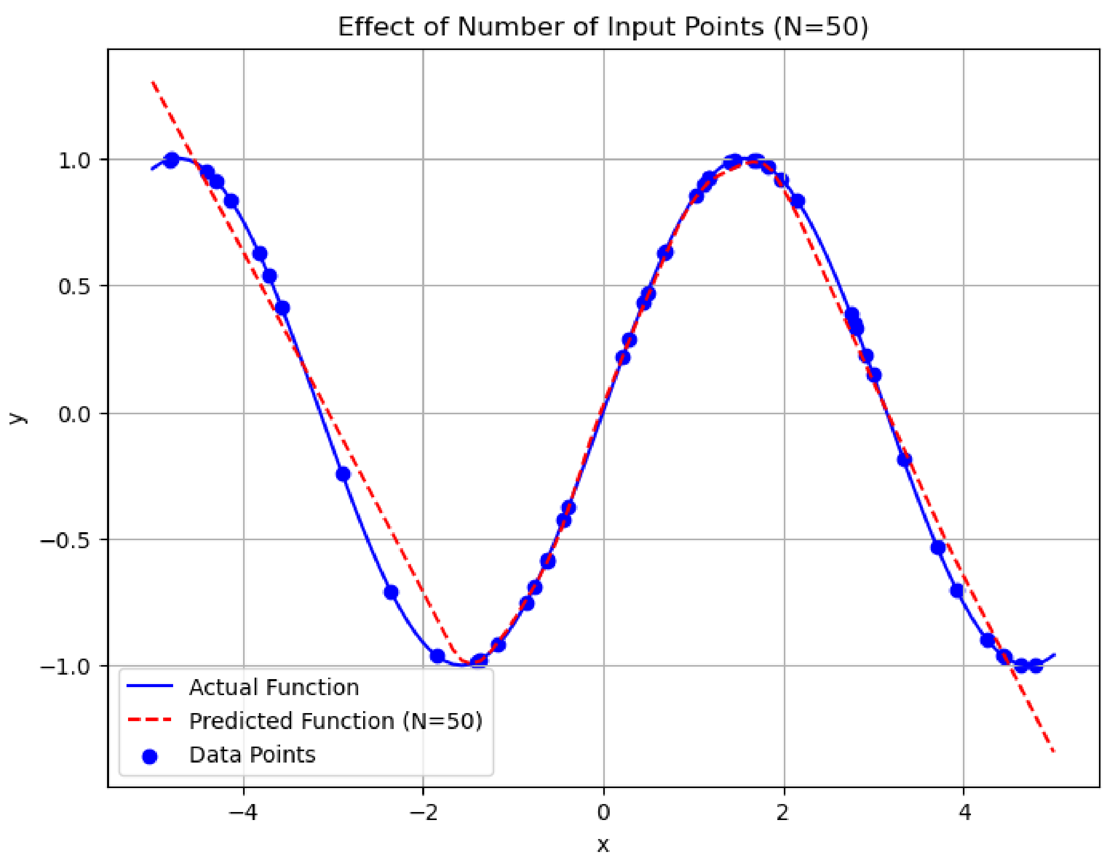
```
Test Loss (MSE): 0.014408214017748833
```

2.13 Network function

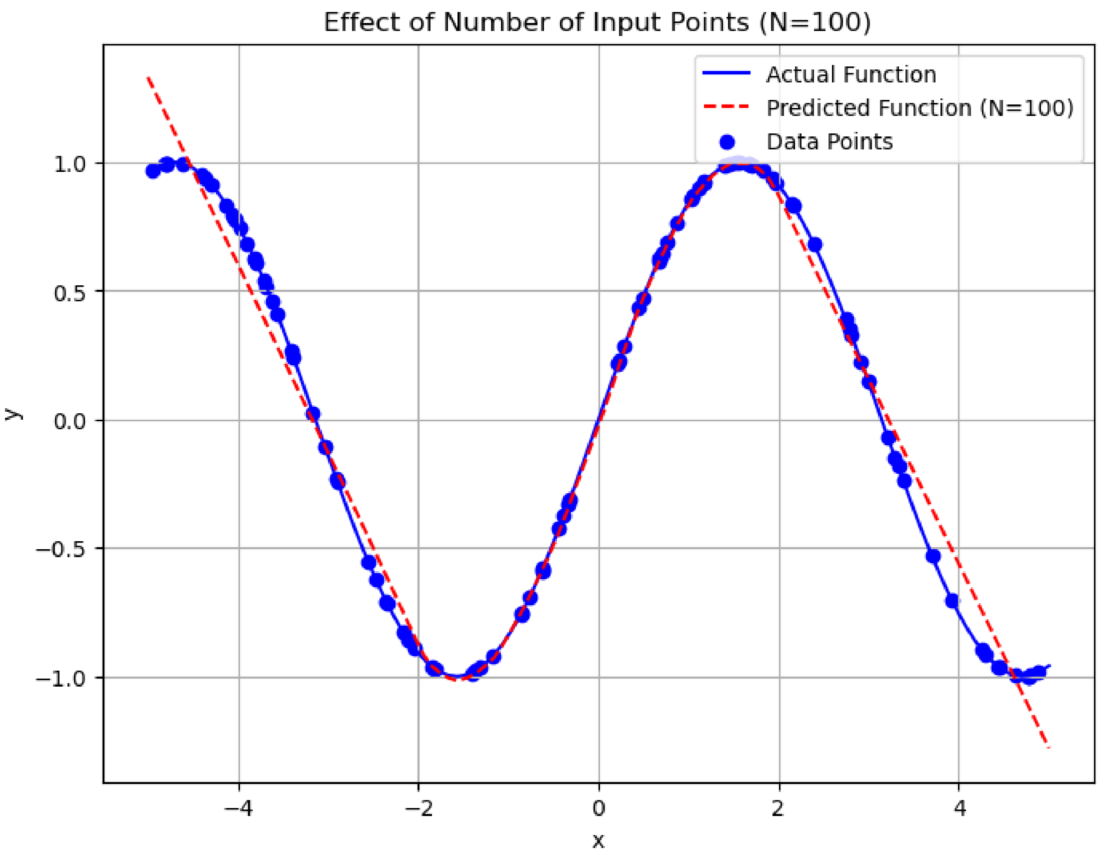
2.13.1 Varying Number of Input Points

[ ]:
```
# Define a sample function (e.g., sinusoidal)
def cosusoidal_function(x):
    return np.cos(x)

# Generate different numbers of input points
num_points_list = [50, 100, 200]

for num_points in num_points_list:
    # Generate training data
    np.random.seed(0)
    x_train = np.random.uniform(-5, 5, num_points)
    y_train = sinusoidal_function(x_train)

    # Build and train the model
    model = keras.Sequential([
        keras.layers.Dense(64, activation='relu', input_shape=(1,)),
        keras.layers.Dense(64, activation='relu'),
        keras.layers.Dense(1)
    ])
    model.compile(optimizer='adam', loss='mse', metrics=['mae'])
    history = model.fit(x_train, y_train, epochs=100, verbose=0)

    # Evaluate and plot results
    x_range = np.linspace(-5, 5, 100)
    y_actual = sinusoidal_function(x_range)
    y_predicted = model.predict(x_range)

    plt.figure(figsize=(8, 6))
    plt.plot(x_range, y_actual, label='Actual Function', color='blue')
    plt.plot(x_range, y_predicted, label=f'Predicted Function␣
↪(N={num_points})', color='red', linestyle='--')
    plt.scatter(x_train, y_train, color='blue', label='Data Points')
    plt.xlabel('x')
    plt.ylabel('y')
    plt.title(f'Effect of Number of Input Points (N={num_points})')
    plt.legend()
    plt.grid(True)
    plt.show()

    x_test = np.linspace(0.1, 5, 50)
    y_test = sinusoidal_function(x_test)
    test_loss, test_mae = model.evaluate(x_test, y_test, verbose=0)
    print(f'Test Loss (MSE): {test_loss}')
```

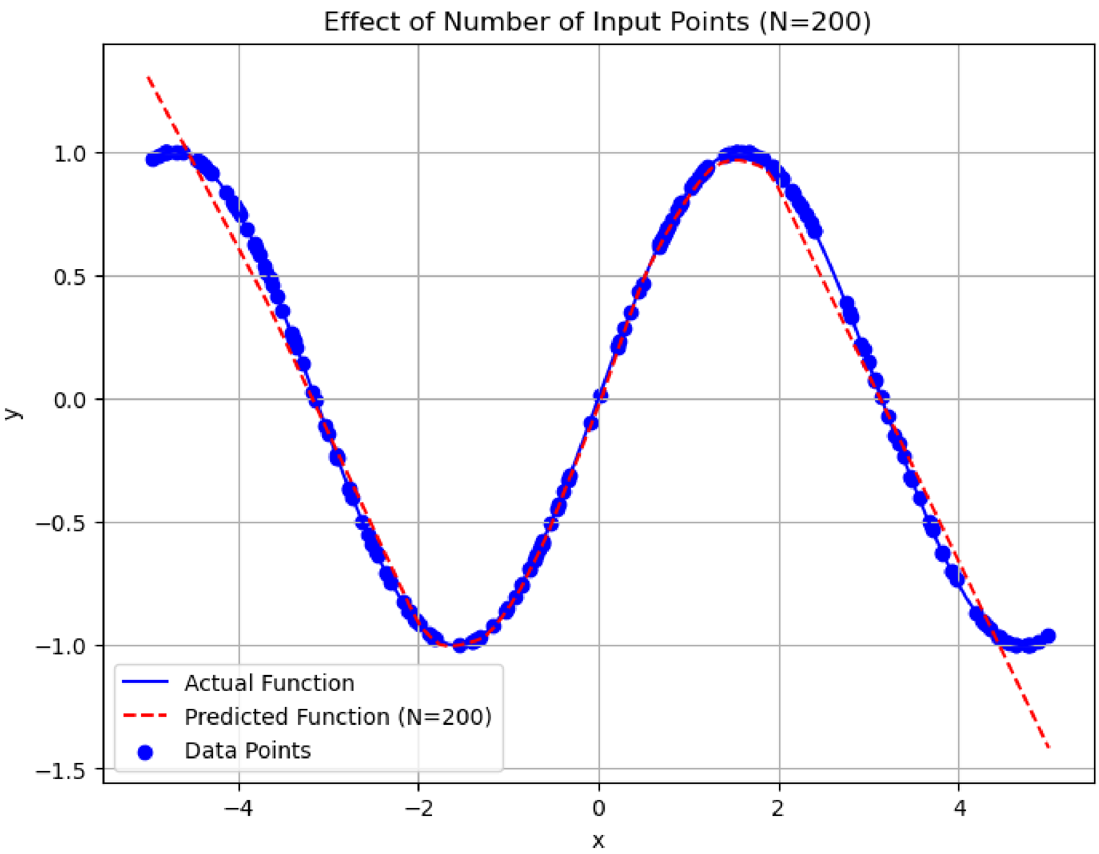
```
WARNING:tensorflow:5 out of the last 37 calls to <function
TensorFlowTrainer.make_predict_function.<locals>.one_step_on_data_distributed at
0x000001B1CEDE6F20> triggered tf.function retracing. Tracing is expensive and
the excessive number of tracings could be due to (1) creating @tf.function
repeatedly in a loop, (2) passing tensors with different shapes, (3) passing
Python objects instead of tensors. For (1), please define your @tf.function
outside of the loop. For (2), @tf.function has reduce_retracing=True option that
can avoid unnecessary retracing. For (3), please refer to
https://www.tensorflow.org/guide/function#controlling_retracing and
https://www.tensorflow.org/api_docs/python/tf/function for  more details.
1/4             0s
24ms/stepWARNING:tensorflow:6 out of the last 40 calls to <function
TensorFlowTrainer.make_predict_function.<locals>.one_step_on_data_distributed at
0x000001B1CEDE6F20> triggered tf.function retracing. Tracing is expensive and
the excessive number of tracings could be due to (1) creating @tf.function
repeatedly in a loop, (2) passing tensors with different shapes, (3) passing
Python objects instead of tensors. For (1), please define your @tf.function
outside of the loop. For (2), @tf.function has reduce_retracing=True option that
can avoid unnecessary retracing. For (3), please refer to
https://www.tensorflow.org/guide/function#controlling_retracing and
https://www.tensorflow.org/api_docs/python/tf/function for  more details.
4/4             0s 10ms/step
```



```
Test Loss (MSE): 0.008837077766656876
4/4              0s 5ms/step
```




```
Test Loss (MSE): 0.010954419150948524
4/4              0s 5ms/step
```




```
Test Loss (MSE): 0.013040501624345
```

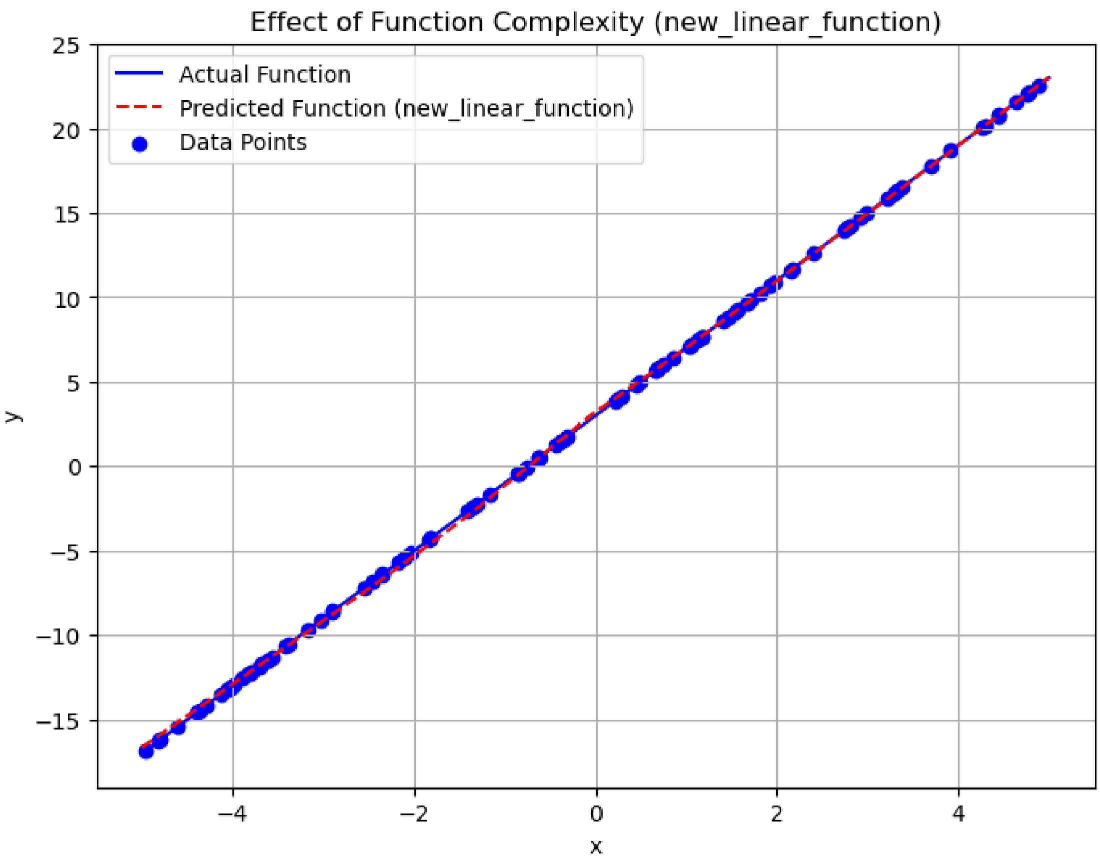
Number of Input Points: As we increase the number of input **points** (N), the model tends to fit the data more closely, capturing the underlying function more accurately

2.13.2 Varying Complexity of Target Function

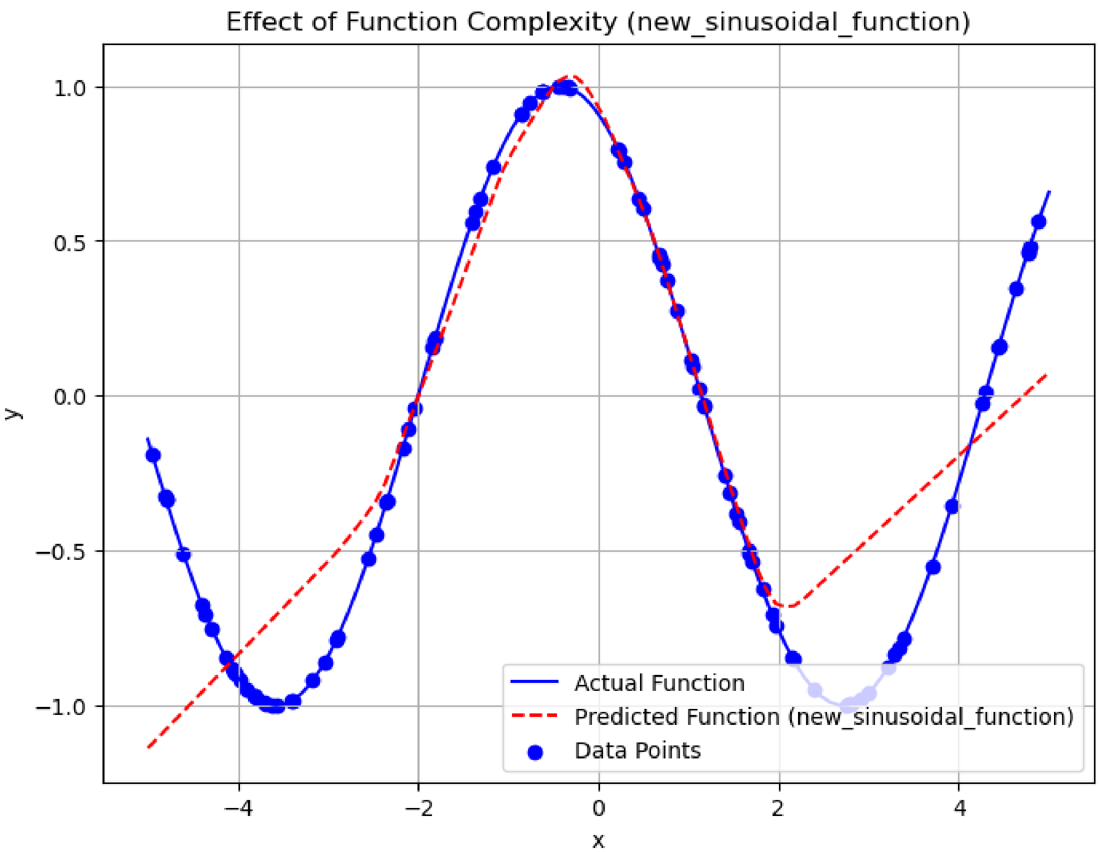
```
[ ]: # Define simple and complex functions
def new_linear_function(x):
    return 4 * x + 3

def new_sinusoidal_function(x):
    return np.sin(x+2)


def new_complicated_function(x):
    return np.log(x**2 + 2*x+1)

# Generate training data for different functions
functions = [new_linear_function, new_sinusoidal_function,
 ↪new_complicated_function]
```

```python
for func in functions:
    # Generate training data
    np.random.seed(0)
    x_train = np.random.uniform(-5, 5, 100)
    y_train = func(x_train)

    # Build and train the model
    model = keras.Sequential([
        keras.layers.Dense(64, activation='relu', input_shape=(1,)),
        keras.layers.Dense(64, activation='relu'),
        keras.layers.Dense(1)
    ])
    model.compile(optimizer='adam', loss='mse', metrics=['mae'])
    history = model.fit(x_train, y_train, epochs=100, verbose=0)

    # Evaluate and plot results
    x_range = np.linspace(-5, 5, 100)
    y_actual = func(x_range)
    y_predicted = model.predict(x_range)

    plt.figure(figsize=(8, 6))
    plt.plot(x_range, y_actual, label='Actual Function', color='blue')
    plt.plot(x_range, y_predicted, label=f'Predicted Function ({func.
 ↪__name__})', color='red', linestyle='--')
    plt.scatter(x_train, y_train, color='blue', label='Data Points')
    plt.xlabel('x')
    plt.ylabel('y')
    plt.title(f'Effect of Function Complexity ({func.__name__})')
    plt.legend()
    plt.grid(True)
    plt.show()

    x_test = np.linspace(0.1, 5, 50)
    y_test = func(x_test)
    test_loss, test_mae = model.evaluate(x_test, y_test, verbose=0)
    print(f'Test Loss (MSE): {test_loss}')
```

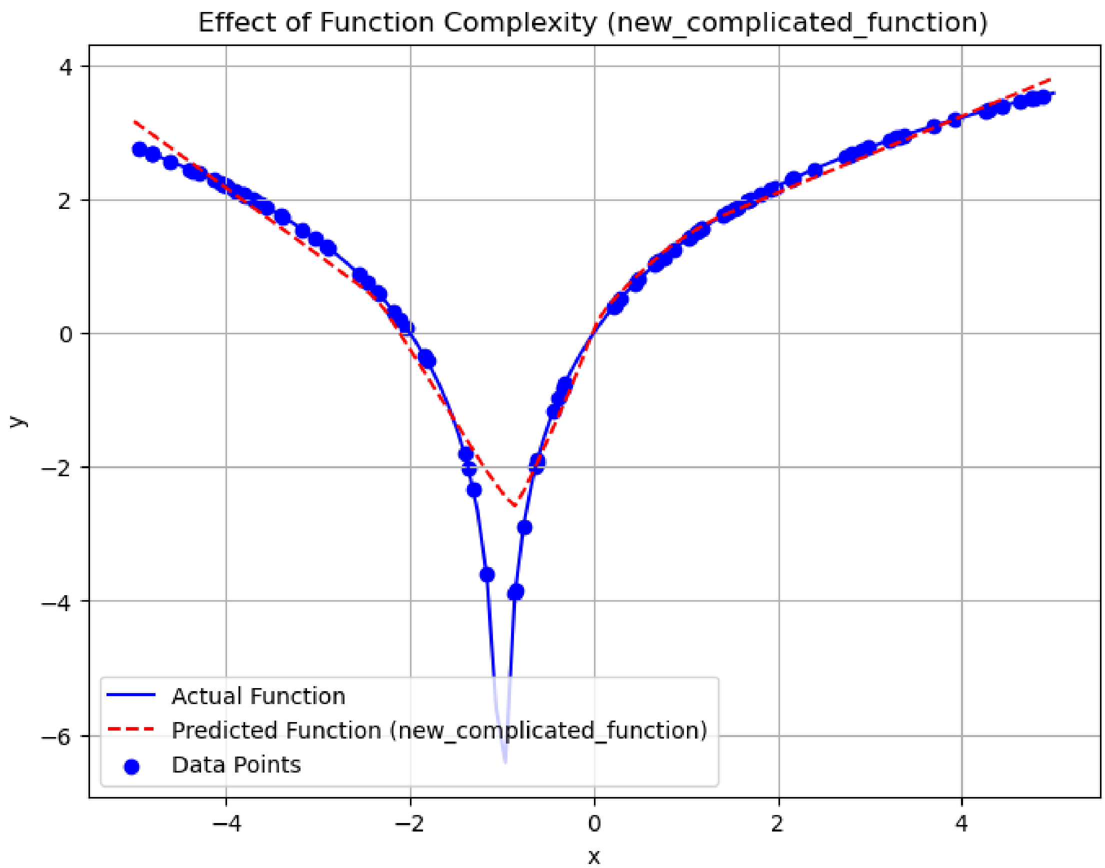
```
4/4              0s 11ms/step
```




```
WARNING:tensorflow:5 out of the last 109 calls to <function
TensorFlowTrainer.make_test_function.<locals>.one_step_on_iterator at
0x000001B1C796B420> triggered tf.function retracing. Tracing is expensive and
the excessive number of tracings could be due to (1) creating @tf.function
repeatedly in a loop, (2) passing tensors with different shapes, (3) passing
Python objects instead of tensors. For (1), please define your @tf.function
outside of the loop. For (2), @tf.function has reduce_retracing=True option that
can avoid unnecessary retracing. For (3), please refer to
https://www.tensorflow.org/guide/function#controlling_retracing and
https://www.tensorflow.org/api_docs/python/tf/function for  more details.
Test Loss (MSE): 0.002753741806373
4/4             0s 11ms/step
```




```
WARNING:tensorflow:6 out of the last 111 calls to <function
TensorFlowTrainer.make_test_function.<locals>.one_step_on_iterator at
0x000001B1CEFD96C0> triggered tf.function retracing. Tracing is expensive and
the excessive number of tracings could be due to (1) creating @tf.function
repeatedly in a loop, (2) passing tensors with different shapes, (3) passing
Python objects instead of tensors. For (1), please define your @tf.function
outside of the loop. For (2), @tf.function has reduce_retracing=True option that
can avoid unnecessary retracing. For (3), please refer to
https://www.tensorflow.org/guide/function#controlling_retracing and
https://www.tensorflow.org/api_docs/python/tf/function for  more details.
Test Loss (MSE): 0.08018937706947327
4/4            0s 5ms/step
```




```
Test Loss (MSE): 0.009606406092643738
```

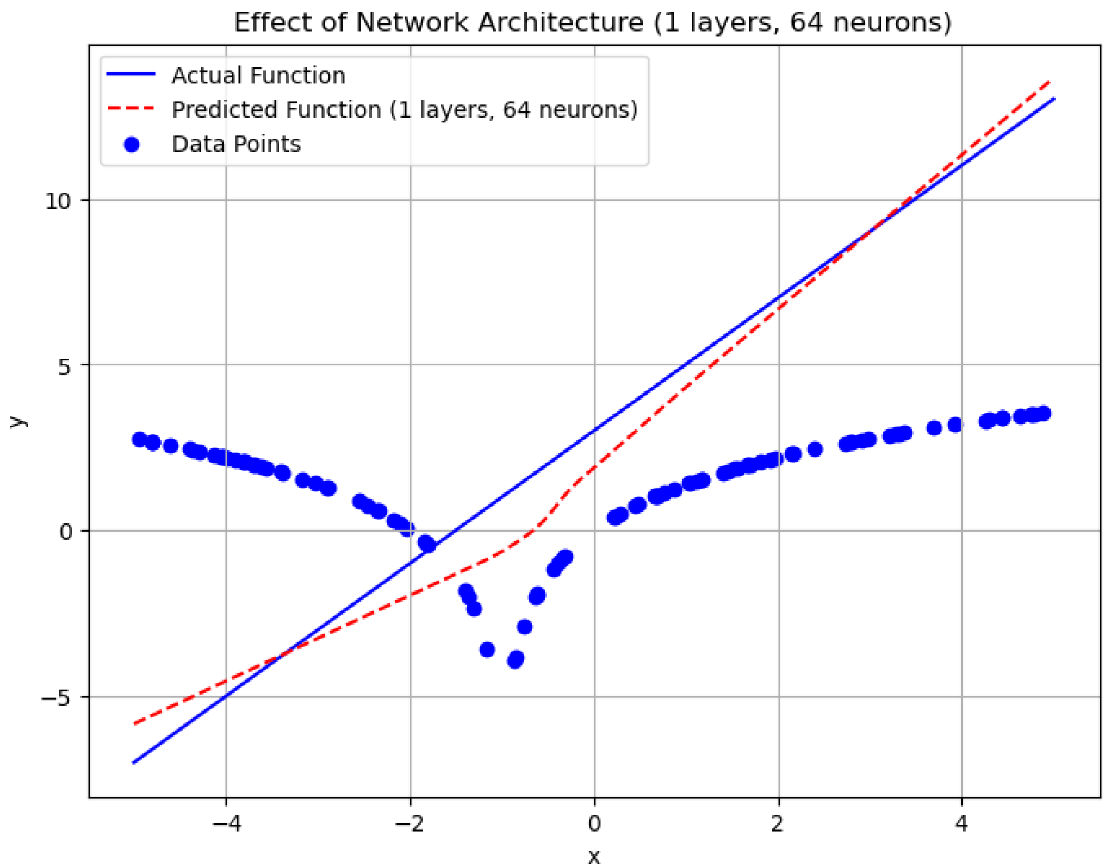
Complexity of Target Function: More complex functions (e.g., complicated_function) may require deeper or differently structured networks to accurately capture their behavior.

2.13.3 Varying Number of Layers and Neurons (linear)

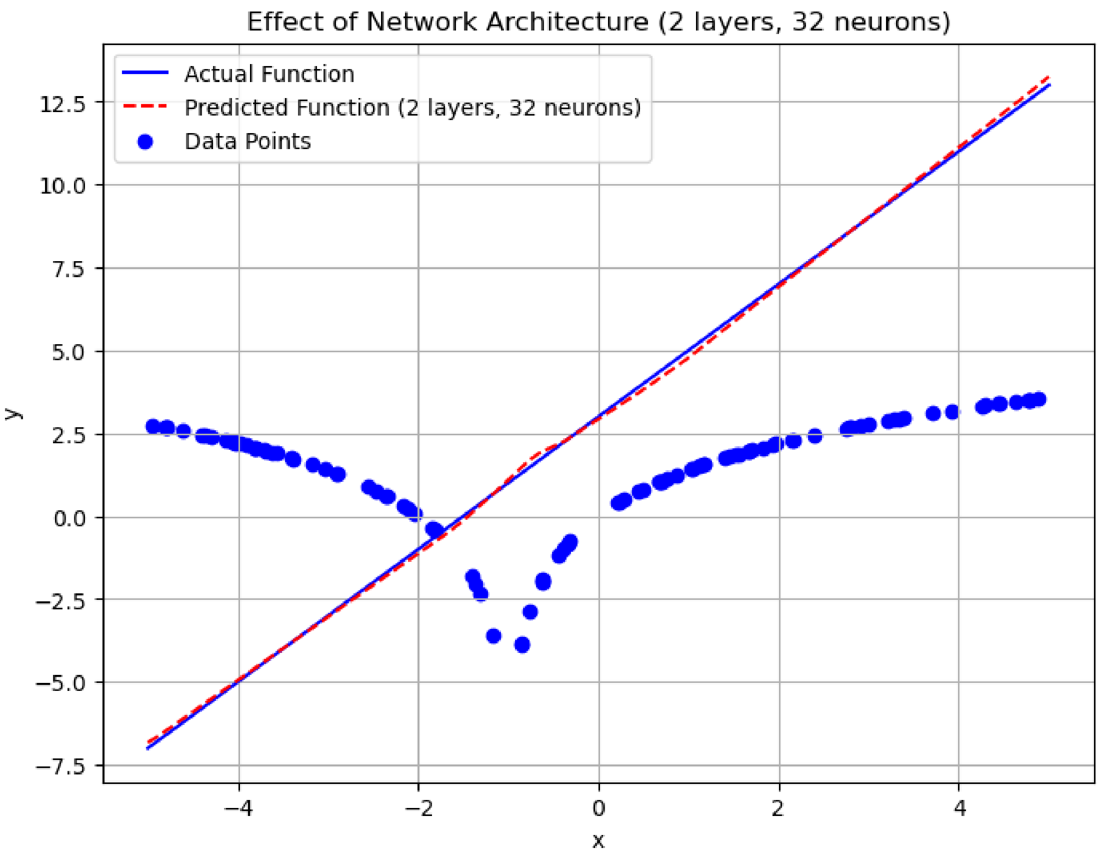
[ ]:
```
# Define a function to create and train models with different architectures
def train_model(num_layers, num_neurons):
    # Generate training data
    np.random.seed(0)
    x_train = np.random.uniform(-5, 5, 100)
    y_train = linear_function(x_train)

    # Build and train the model
    model = keras.Sequential()
    model.add(keras.layers.Dense(num_neurons, activation='relu',␣
 ↪input_shape=(1,)))

    for _ in range(num_layers - 1):
```

```
        model.add(keras.layers.Dense(num_neurons, activation='relu'))

    model.add(keras.layers.Dense(1))

    model.compile(optimizer='adam', loss='mse', metrics=['mae'])
    history = model.fit(x_train, y_train, epochs=100, verbose=0)

    return model

# Test different architectures
architectures = [(1, 64), (2, 32),(3,20), (3, 16),(4,8)]

for layers, neurons in architectures:
    model = train_model(layers, neurons)

    # Evaluate and plot results
    x_range = np.linspace(-5, 5, 100)
    y_actual = linear_function(x_range)
    y_predicted = model.predict(x_range)

    plt.figure(figsize=(8, 6))
    plt.plot(x_range, y_actual, label='Actual Function', color='blue')
    plt.plot(x_range, y_predicted, label=f'Predicted Function ({layers} layers,␣
 ↪{neurons} neurons)', color='red', linestyle='--')
    plt.scatter(x_train, y_train, color='blue', label='Data Points')
    plt.xlabel('x')
    plt.ylabel('y')
    plt.title(f'Effect of Network Architecture ({layers} layers, {neurons}␣
 ↪neurons)')
    plt.legend()
    plt.grid(True)
    plt.show()

    x_test = np.linspace(0.1, 5, 50)
    y_test = linear_function(x_test)
    test_loss, test_mae = model.evaluate(x_test, y_test, verbose=0)
    print(f'Test Loss (MSE): {test_loss}')
```

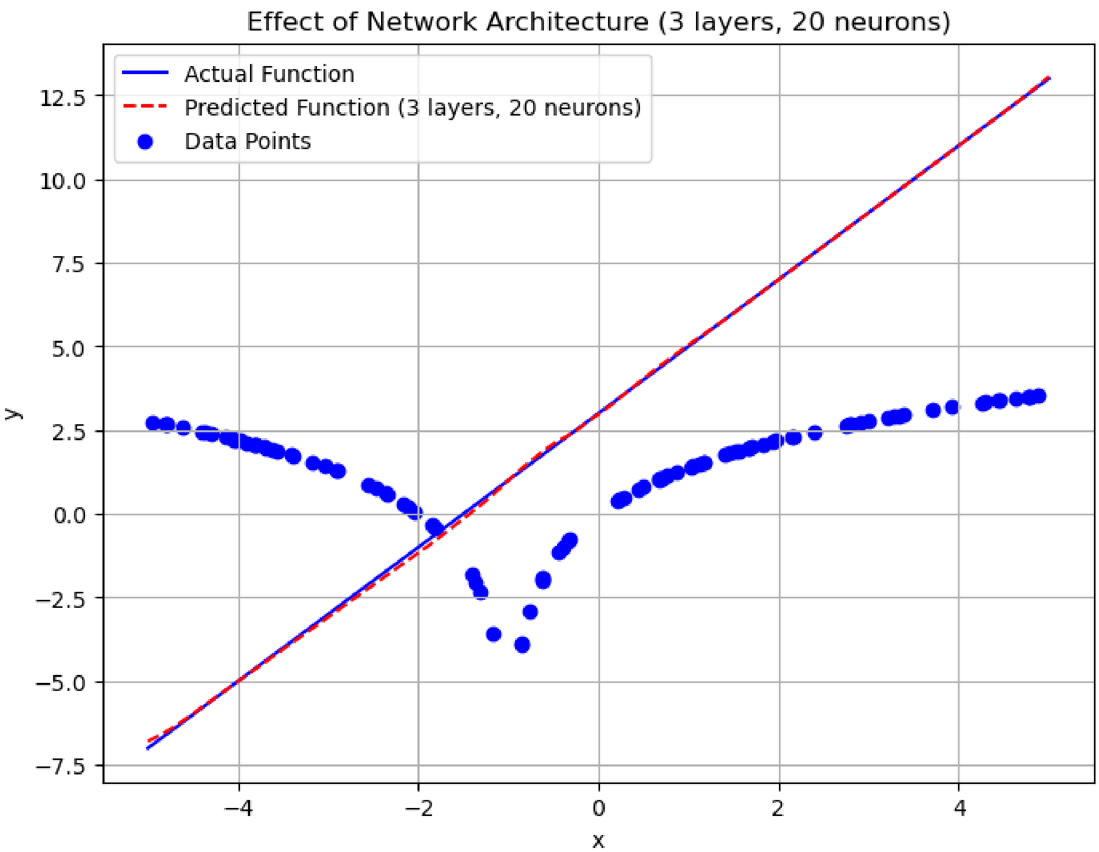
```
4/4              0s 5ms/step
```



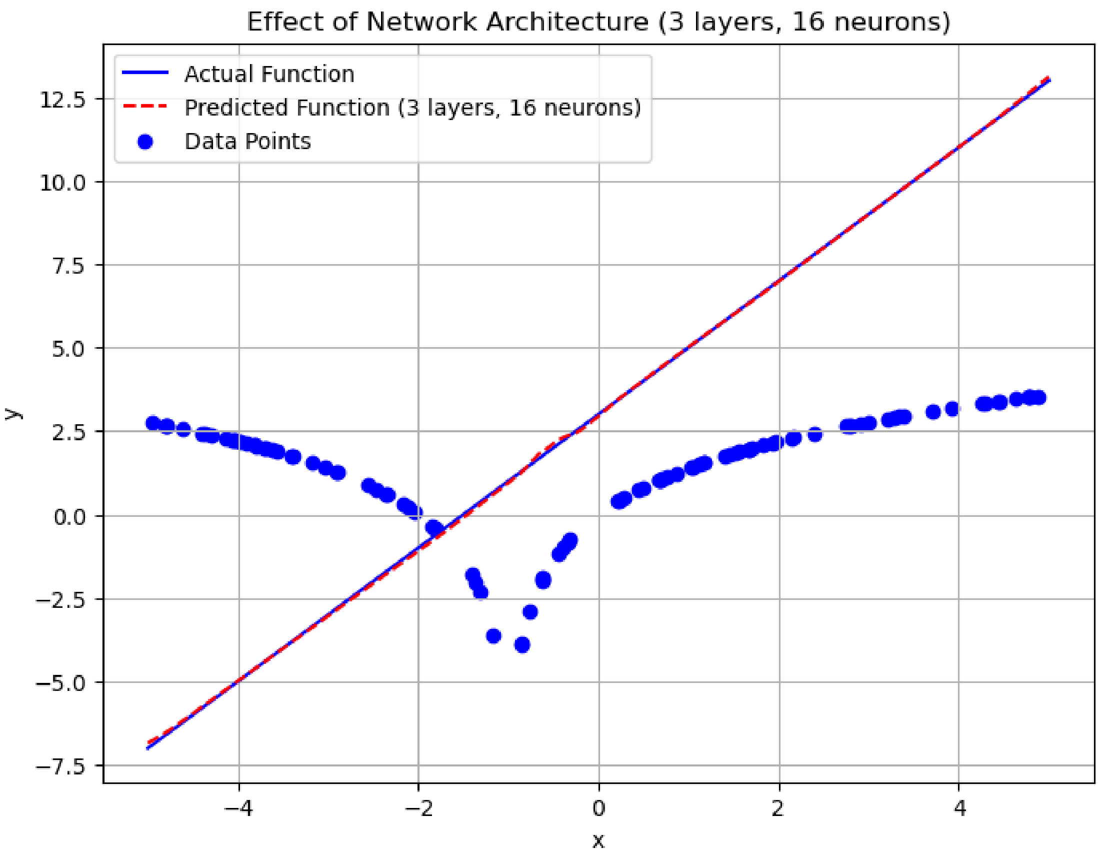

```
Test Loss (MSE): 0.27449265122413635
4/4              0s 11ms/step
```



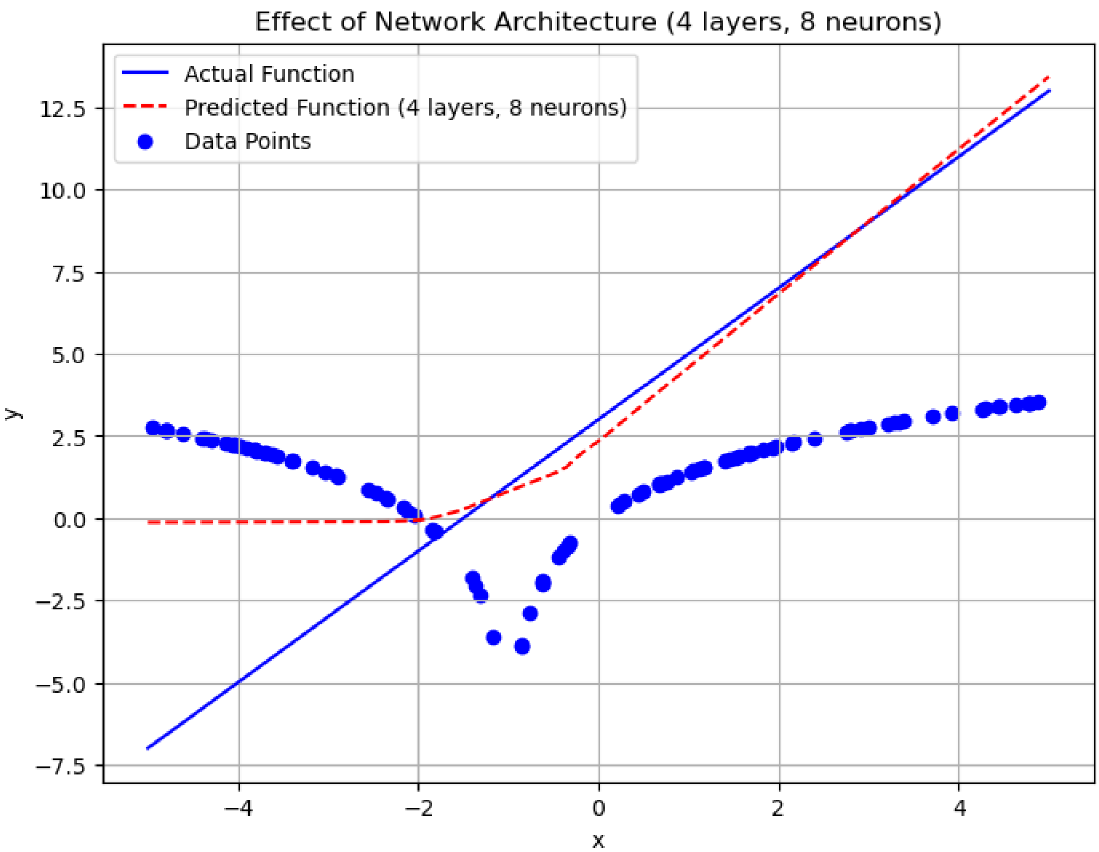

```
Test Loss (MSE): 0.019765440374612808
4/4              0s 10ms/step
```




```
Test Loss (MSE): 0.0005236889119260013
4/4              0s 12ms/step
```




```
Test Loss (MSE): 0.0009406635072082281
4/4              0s 12ms/step
```



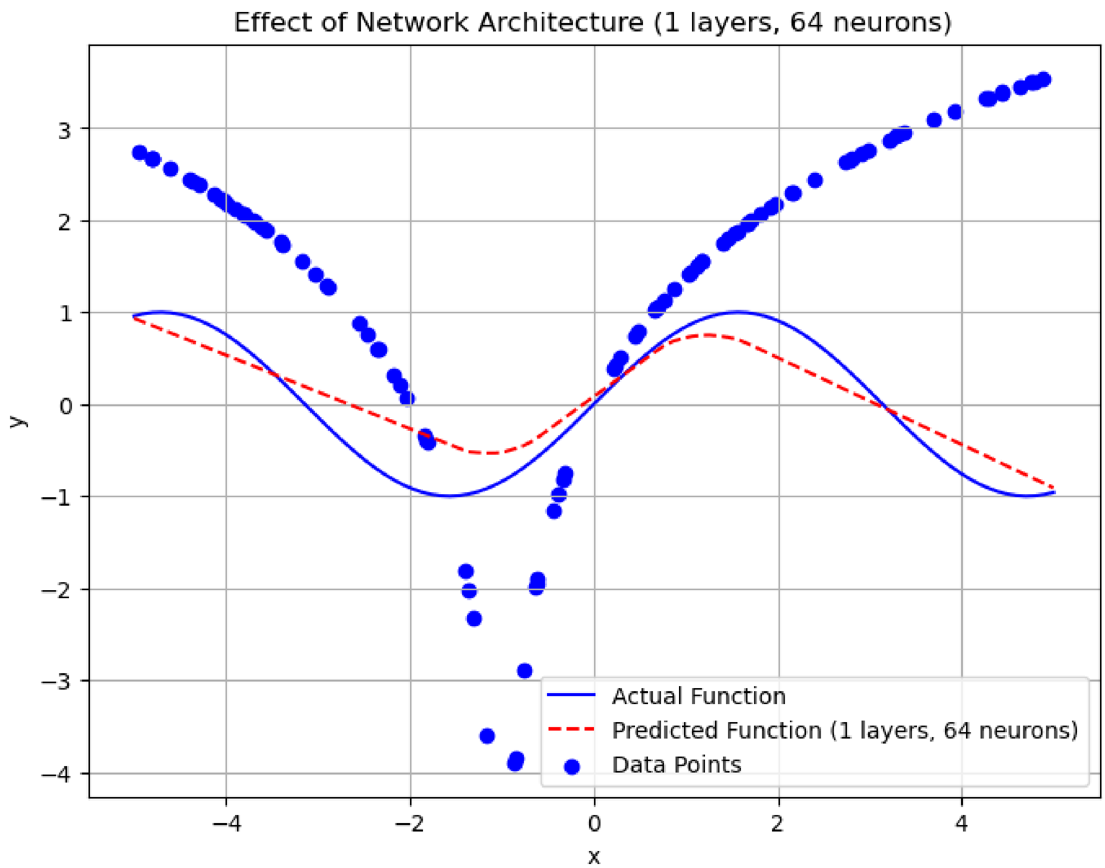

```
Test Loss (MSE): 0.10314904898405075
```

2.13.4 Varying Number of Layers and Neurons (sinusoidal)

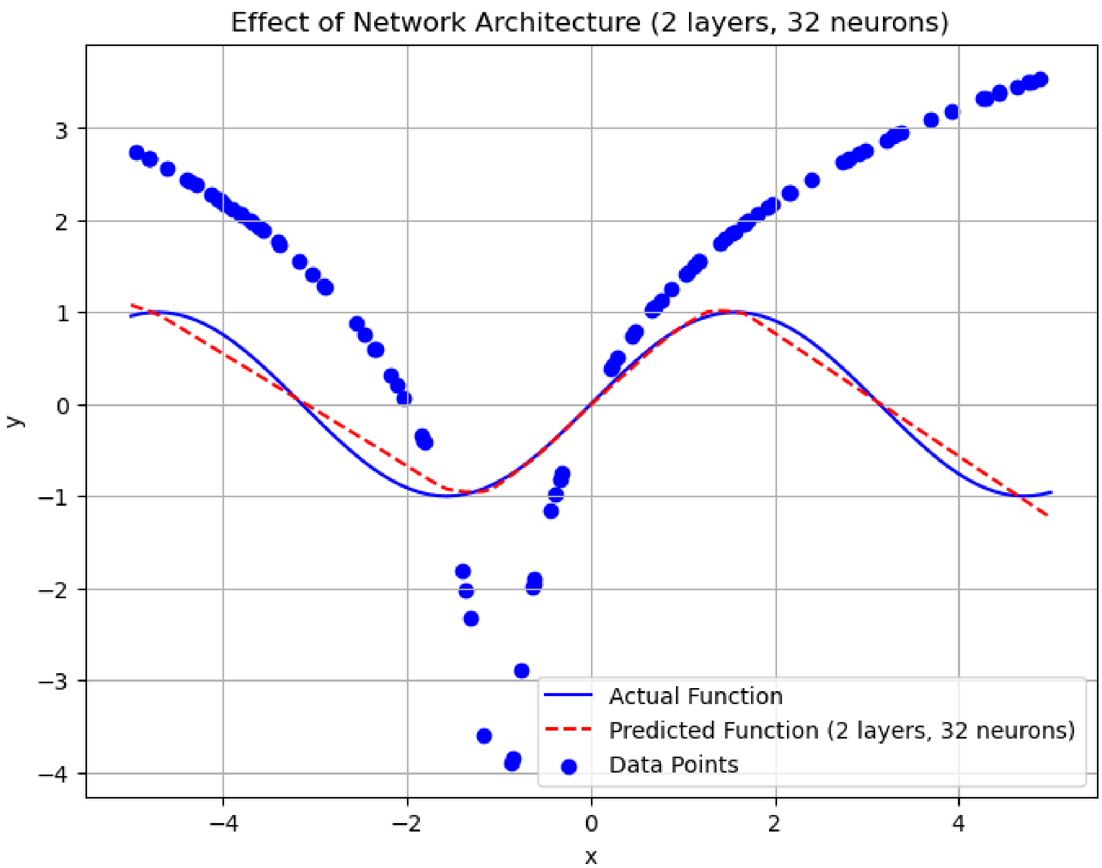
[ ]:
```python
# Define a function to create and train models with different architectures
def train_model(num_layers, num_neurons):
    # Generate training data
    np.random.seed(0)
    x_train = np.random.uniform(-5, 5, 100)
    y_train = sinusoidal_function(x_train)

    # Build and train the model
    model = keras.Sequential()
    model.add(keras.layers.Dense(num_neurons, activation='relu',␣
 ↪input_shape=(1,)))

    for _ in range(num_layers - 1):
        model.add(keras.layers.Dense(num_neurons, activation='relu'))

    model.add(keras.layers.Dense(1))
```

```
    model.compile(optimizer='adam', loss='mse', metrics=['mae'])
    history = model.fit(x_train, y_train, epochs=100, verbose=0)

    return model

# Test different architectures
architectures = [(1, 64), (2, 32),(3,20), (3, 16),(4,8)]

for layers, neurons in architectures:
    model = train_model(layers, neurons)

    # Evaluate and plot results
    x_range = np.linspace(-5, 5, 100)
    y_actual = sinusoidal_function(x_range)
    y_predicted = model.predict(x_range)

    plt.figure(figsize=(8, 6))
    plt.plot(x_range, y_actual, label='Actual Function', color='blue')
    plt.plot(x_range, y_predicted, label=f'Predicted Function ({layers} layers,␣
 ↪{neurons} neurons)', color='red', linestyle='--')
    plt.scatter(x_train, y_train, color='blue', label='Data Points')
    plt.xlabel('x')
    plt.ylabel('y')
    plt.title(f'Effect of Network Architecture ({layers} layers, {neurons}␣
 ↪neurons)')
    plt.legend()
    plt.grid(True)
    plt.show()

    x_test = np.linspace(0.1, 5, 50)
    y_test = sinusoidal_function(x_test)
    test_loss, test_mae = model.evaluate(x_test, y_test, verbose=0)
    print(f'Test Loss (MSE): {test_loss}')
```

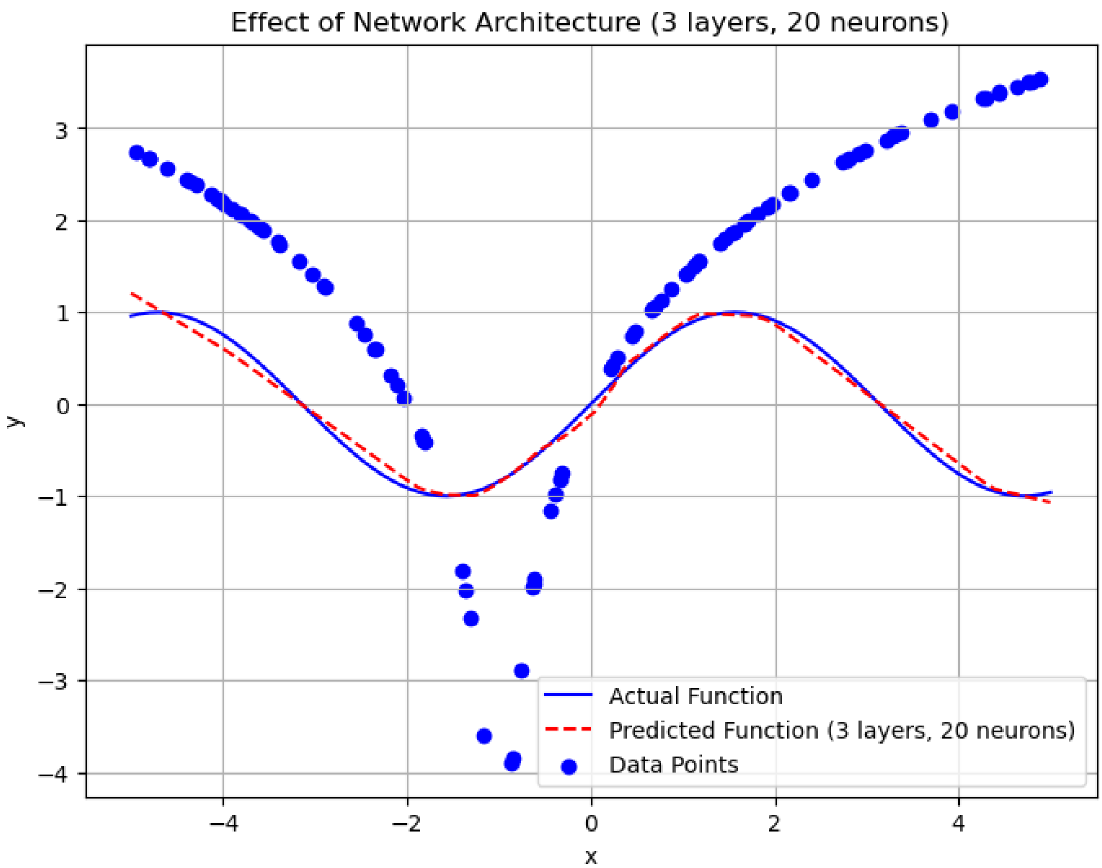
```
4/4              0s 6ms/step
```



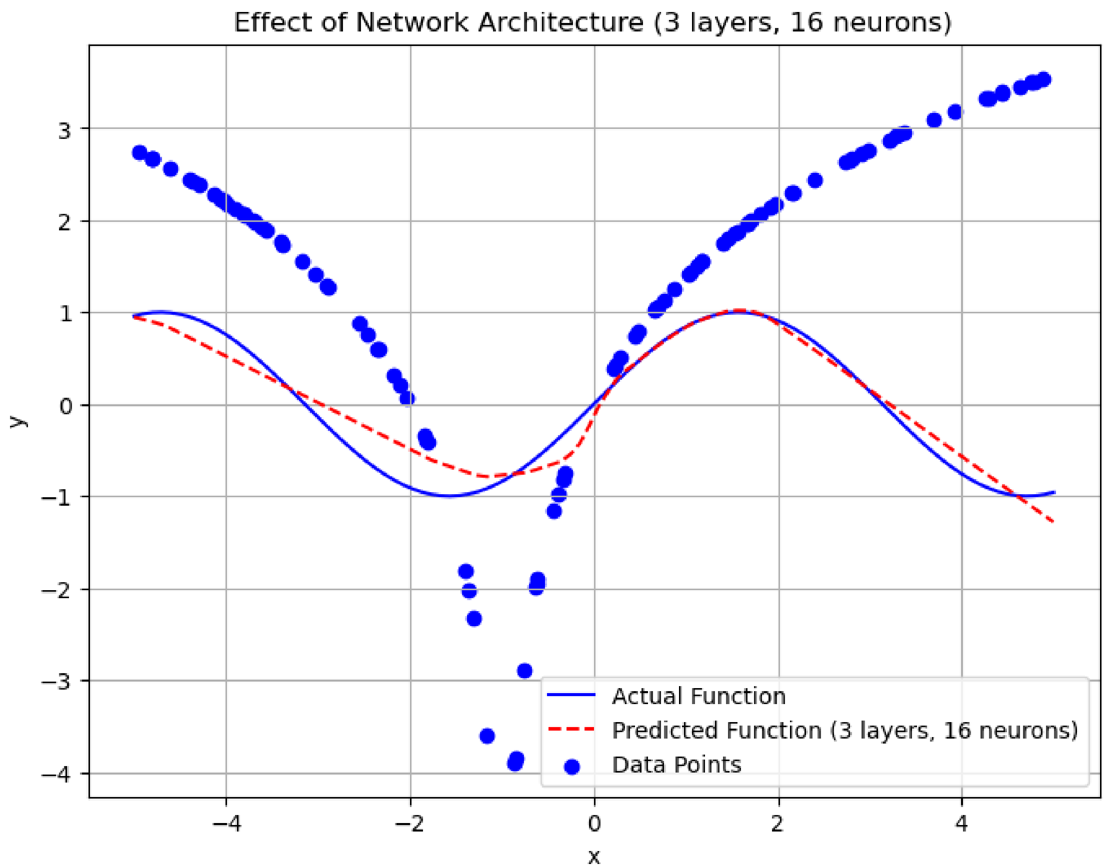

```
Test Loss (MSE): 0.05968279764056206
4/4              0s 6ms/step
```



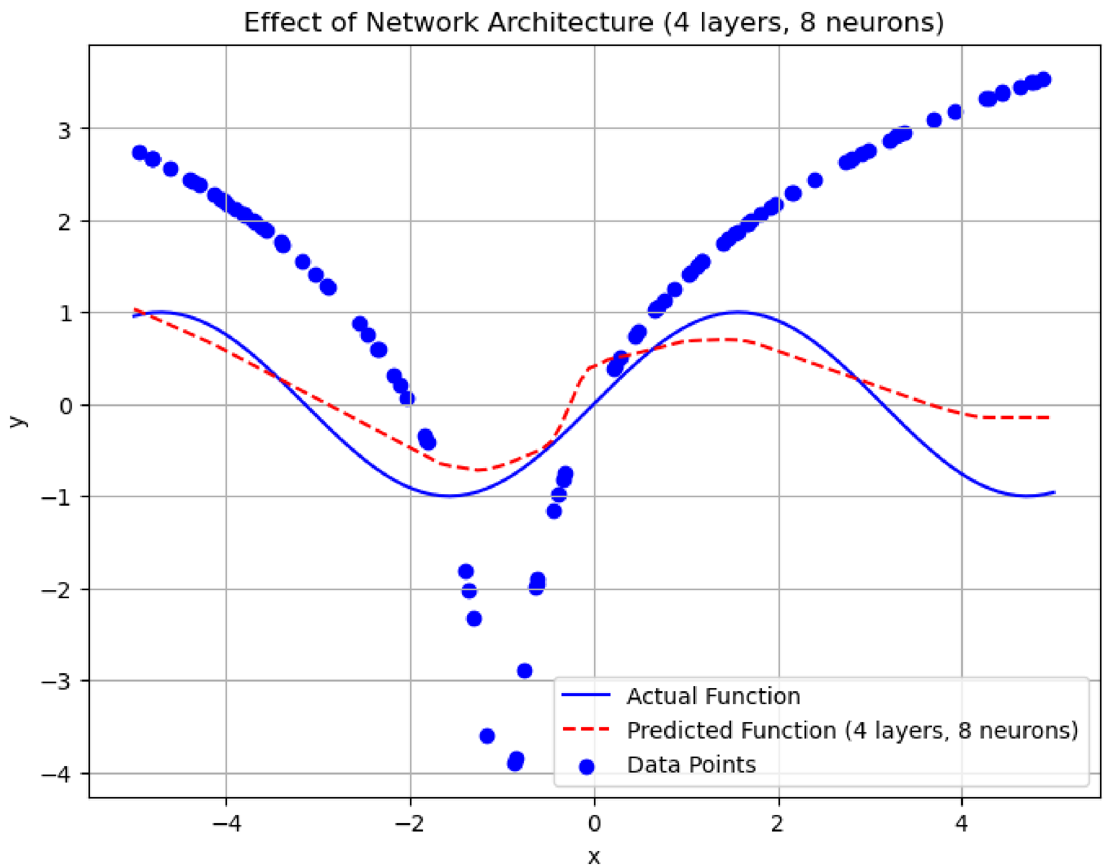

```
Test Loss (MSE): 0.012987806461751461
4/4              0s 11ms/step
```




```
Test Loss (MSE): 0.004392016679048538
4/4              0s 11ms/step
```




```
Test Loss (MSE): 0.010759213007986546
4/4              0s 11ms/step
```



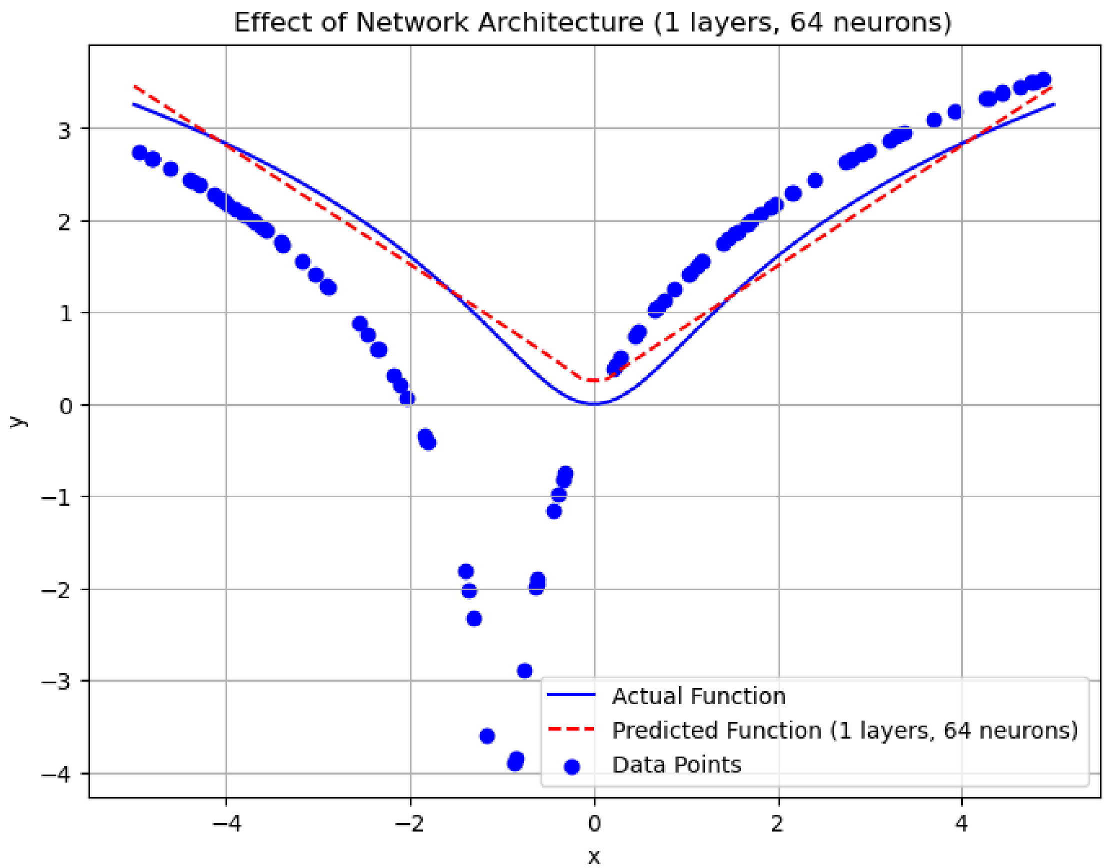

```
Test Loss (MSE): 0.2022128850221634
```

2.13.5 Varying Number of Layers and Neurons (complicated)

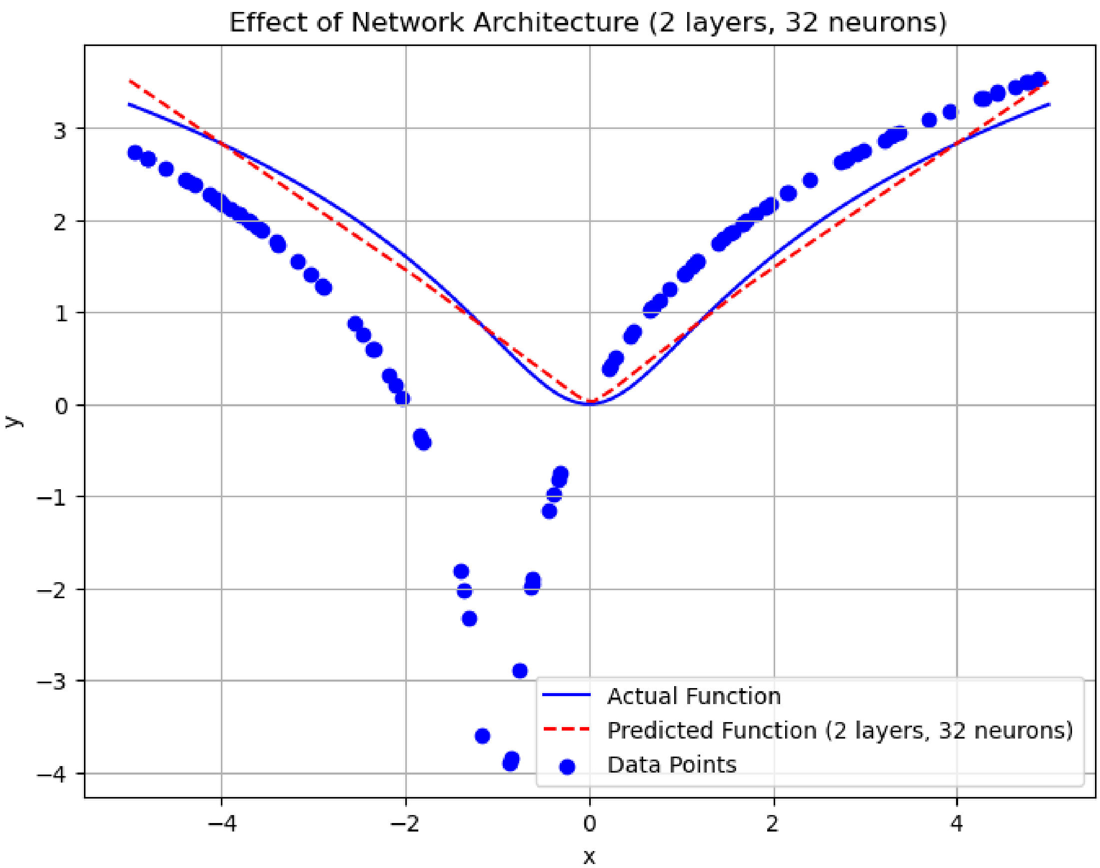
```
[ ]: # Define a function to create and train models with different architectures
def train_model(num_layers, num_neurons):
    # Generate training data
    np.random.seed(0)
    x_train = np.random.uniform(-5, 5, 100)
    y_train = complicated_function(x_train)

    # Build and train the model
    model = keras.Sequential()
    model.add(keras.layers.Dense(num_neurons, activation='relu',␣
 ↪input_shape=(1,)))

    for _ in range(num_layers - 1):
        model.add(keras.layers.Dense(num_neurons, activation='relu'))
```

```
    model.add(keras.layers.Dense(1))

    model.compile(optimizer='adam', loss='mse', metrics=['mae'])
    history = model.fit(x_train, y_train, epochs=100, verbose=0)

    return model

# Test different architectures
architectures = [(1, 64), (2, 32),(3,20), (3, 16),(4,8)]

for layers, neurons in architectures:
    model = train_model(layers, neurons)

    # Evaluate and plot results
    x_range = np.linspace(-5, 5, 100)
    y_actual = complicated_function(x_range)
    y_predicted = model.predict(x_range)

    plt.figure(figsize=(8, 6))
    plt.plot(x_range, y_actual, label='Actual Function', color='blue')
    plt.plot(x_range, y_predicted, label=f'Predicted Function ({layers} layers,␣
↪{neurons} neurons)', color='red', linestyle='--')
    plt.scatter(x_train, y_train, color='blue', label='Data Points')
    plt.xlabel('x')
    plt.ylabel('y')
    plt.title(f'Effect of Network Architecture ({layers} layers, {neurons}␣
↪neurons)')
    plt.legend()
    plt.grid(True)
    plt.show()

    x_test = np.linspace(0.1, 5, 50)
    y_test = complicated_function(x_test)
    test_loss, test_mae = model.evaluate(x_test, y_test, verbose=0)
    print(f'Test Loss (MSE): {test_loss}')
```

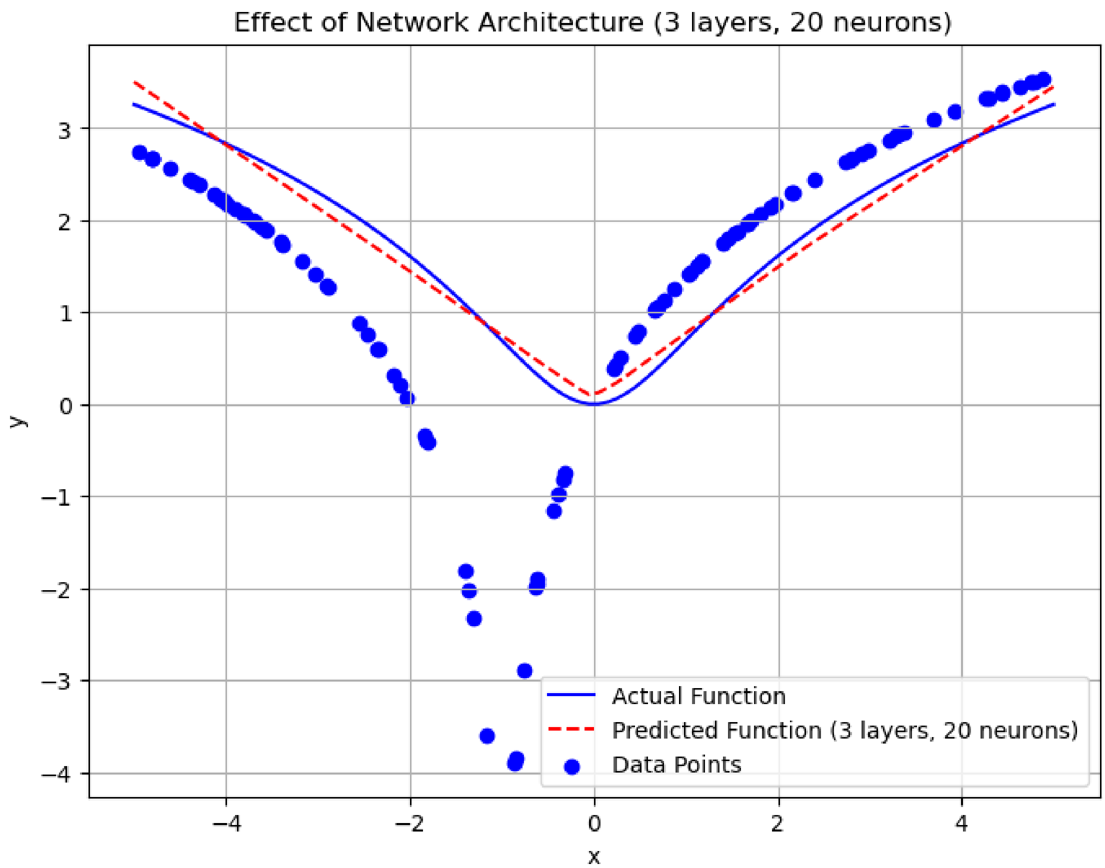
```
4/4              0s 5ms/step
```



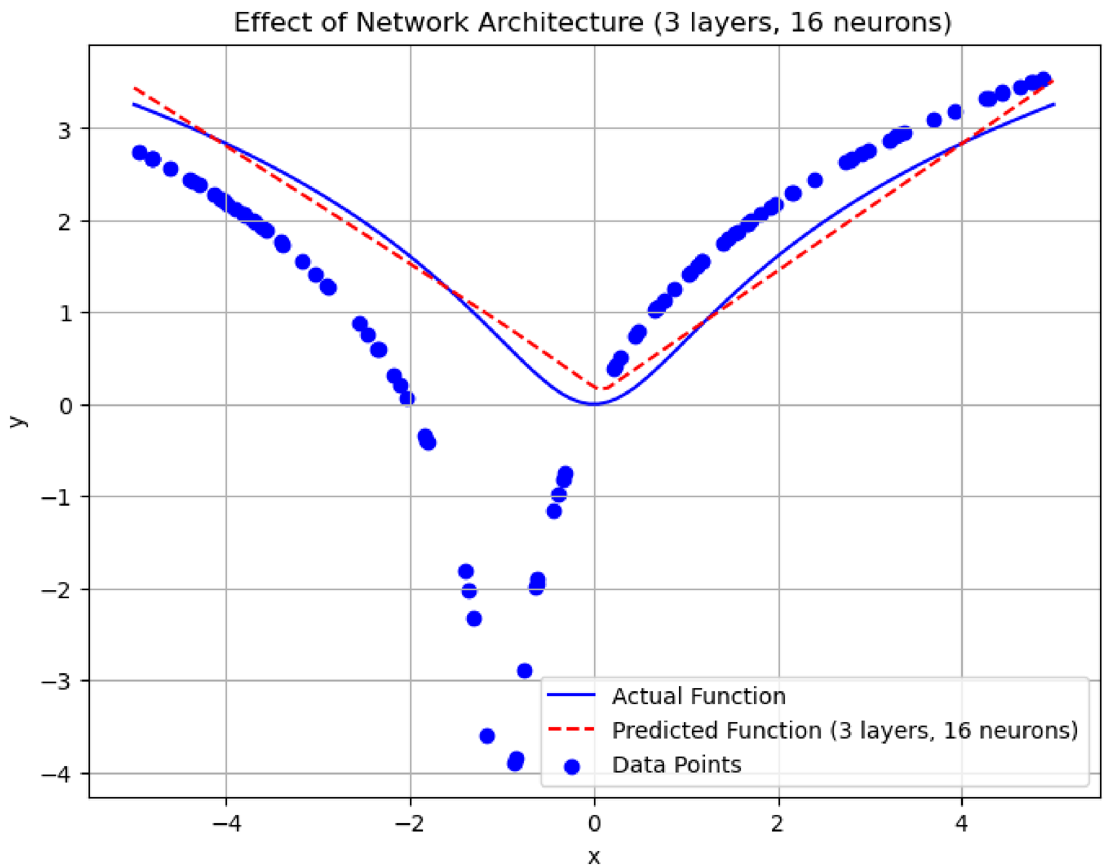

```
Test Loss (MSE): 0.022424202412366867
4/4              0s 5ms/step
```



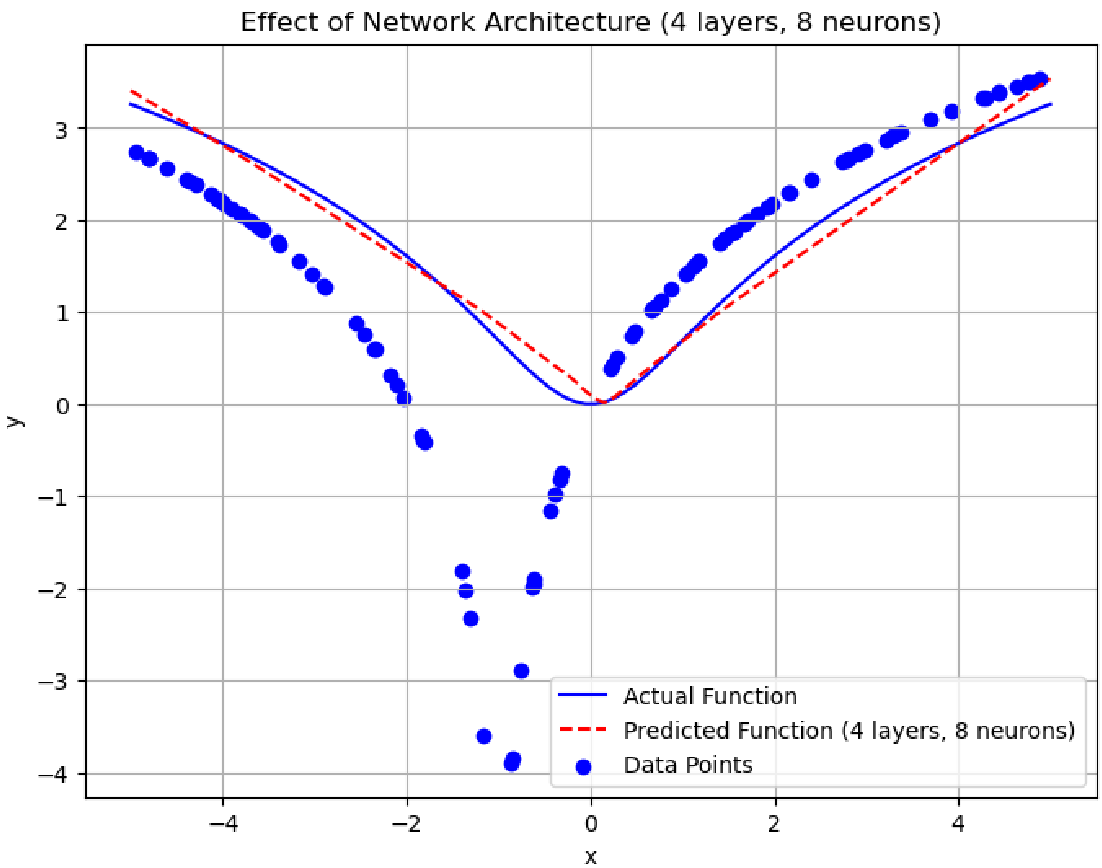

```
Test Loss (MSE): 0.014030925929546356
4/4              0s 10ms/step
```




```
Test Loss (MSE): 0.014306829310953617
4/4             0s 11ms/step
```




```
Test Loss (MSE): 0.020017467439174652
4/4              0s 12ms/step
```




```
Test Loss (MSE): 0.018877642592191696
```

Number of Layers and Neurons: Deeper networks (more layers) or wider networks (more neurons per layer) can potentially capture more complex patterns in the data but may also lead to overfitting if not properly regularized.

3 Investigating Noise in Neural Network Performance

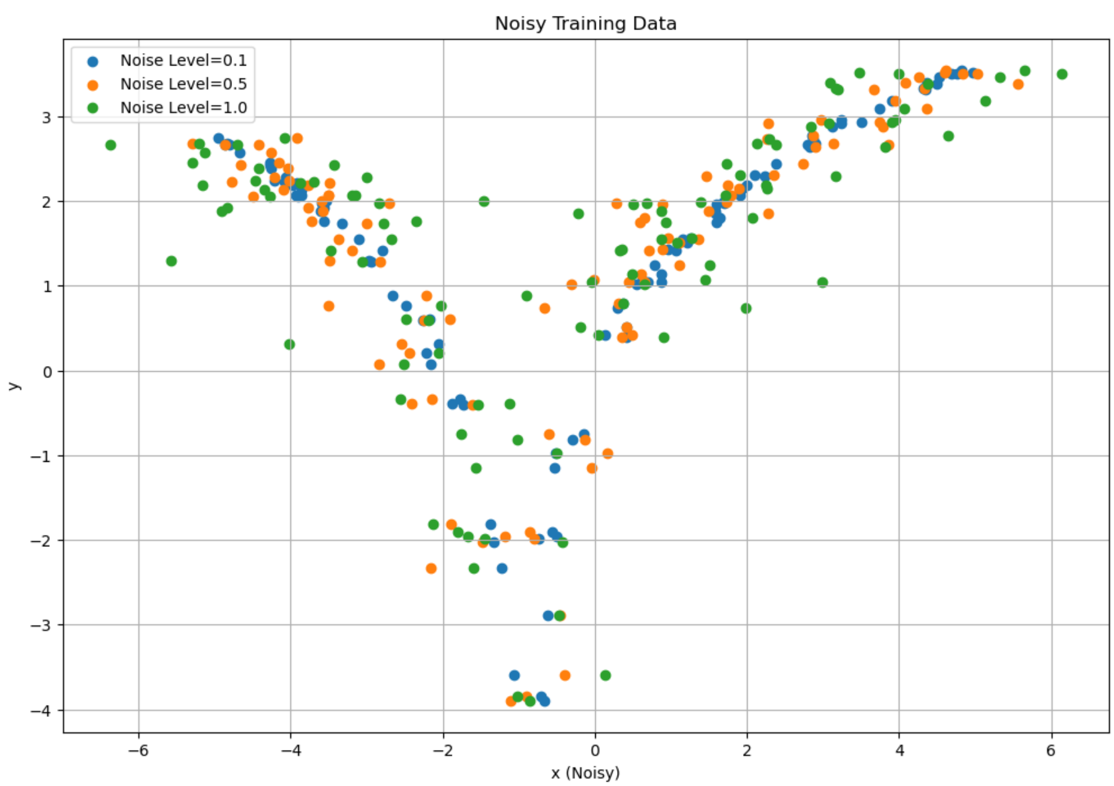
In this phase, we'll add various levels of noise to the data points generated in the previous section and observe how the neural network model handles this noise. Our goal is to report the network's performance across different levels of noise, ranging from minimal to substantial, and analyze its ability to maintain accuracy in the presence of noise.

3.1 Adding Noise to Training Data

First, we will add different levels of noise to the training data points. We will use a normal distribution to generate the noise, ranging from values close to zero (low noise) to larger values (high noise).

[ ]:

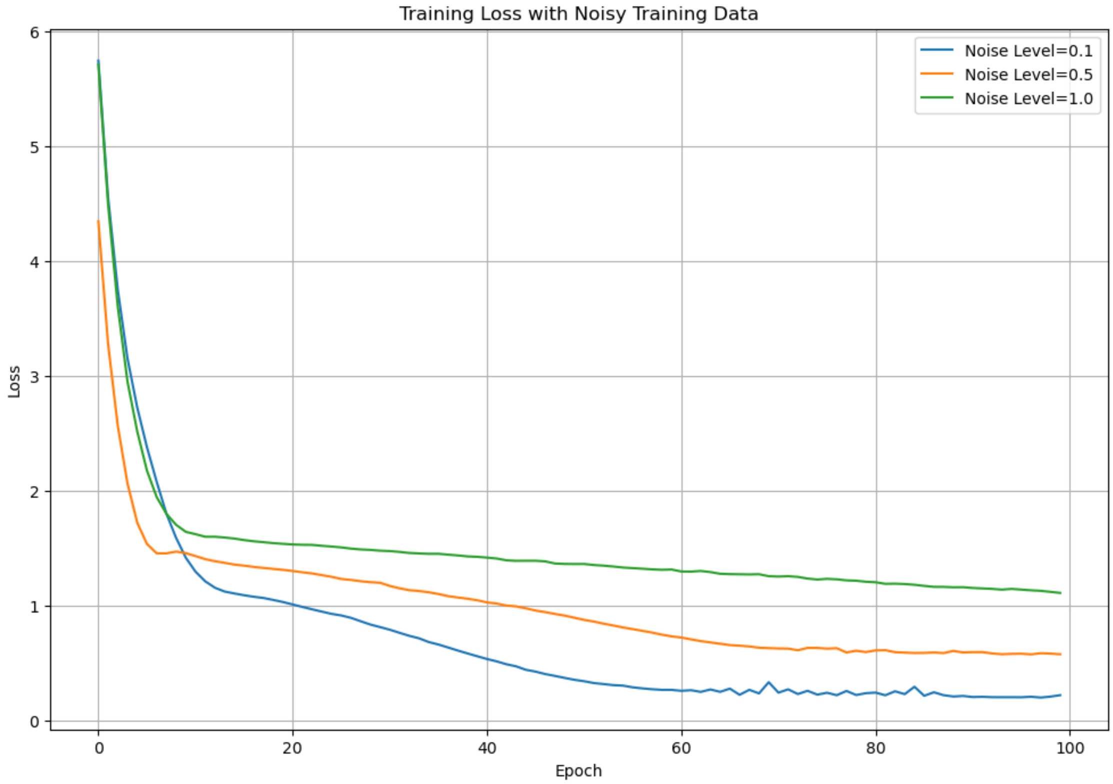
```python
# Define a function to add noise to data
def add_noise(data, noise_level):
    noisy_data = data + np.random.normal(0, noise_level, size=data.shape)
    return noisy_data

# Generate noisy training data
noise_levels = [0.1, 0.5, 1.0]  # Different noise levels
x_noisy_train_list = []

for noise_level in noise_levels:
    x_noisy_train = add_noise(x_train, noise_level)
    x_noisy_train_list.append(x_noisy_train)

# Visualize training data with noise
plt.figure(figsize=(12, 8))
for i, x_noisy_train in enumerate(x_noisy_train_list):
    plt.scatter(x_noisy_train, y_train, label=f'Noise Level={noise_levels[i]}')
plt.xlabel('x (Noisy)')
plt.ylabel('y')
plt.title('Noisy Training Data')
plt.legend()
plt.grid(True)
plt.show()
```



3.2 Training Neural Network with Noisy Training Data

Next, we will train neural network models using the noisy training data and examine their performance.

[ ]:
```
# Train models with noisy training data
models = []
histories = []

for i, x_noisy_train in enumerate(x_noisy_train_list):
    model = keras.Sequential([
        keras.layers.Dense(64, activation='relu', input_shape=(1,)),
        keras.layers.Dense(64, activation='relu'),
        keras.layers.Dense(1)
    ])
    model.compile(optimizer='adam', loss='mse', metrics=['mae'])
    history = model.fit(x_noisy_train, y_train, epochs=100, verbose=0)
    models.append(model)
    histories.append(history)

# Plot training loss for each noise level
plt.figure(figsize=(12, 8))
for i, history in enumerate(histories):
    plt.plot(history.history['loss'], label=f'Noise Level={noise_levels[i]}')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training Loss with Noisy Training Data')
plt.legend()
plt.grid(True)
plt.show()
```



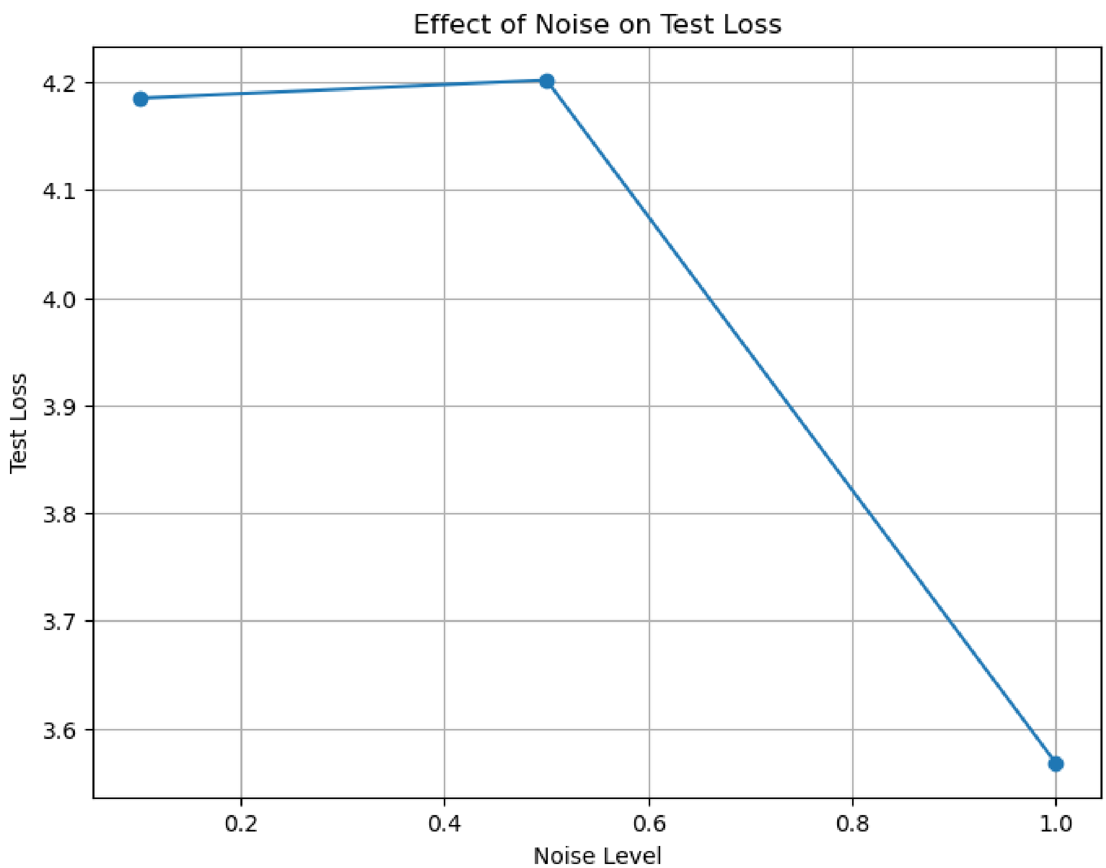
3.3 Evaluating Network Performance with Test Data

3.3.1 sinusoidal

[ ]:
```
# Define a function to evaluate model performance on test data
def evaluate_model(model, x_test, y_test):
    loss, _ = model.evaluate(x_test, y_test)
    return loss

# Generate clean test data
x_test = np.linspace(-5, 5, 100)
y_test = sinusoidal_function(x_test)

# Evaluate models with clean test data
test_losses = []

for i, model in enumerate(models):
    test_loss = evaluate_model(model, x_test, y_test)
    test_losses.append(test_loss)

# Plot test loss vs. noise level
plt.figure(figsize=(8, 6))
```

```
plt.plot(noise_levels, test_losses, marker='o')
plt.xlabel('Noise Level')
plt.ylabel('Test Loss')
plt.title('Effect of Noise on Test Loss')
plt.grid(True)
plt.show()
```

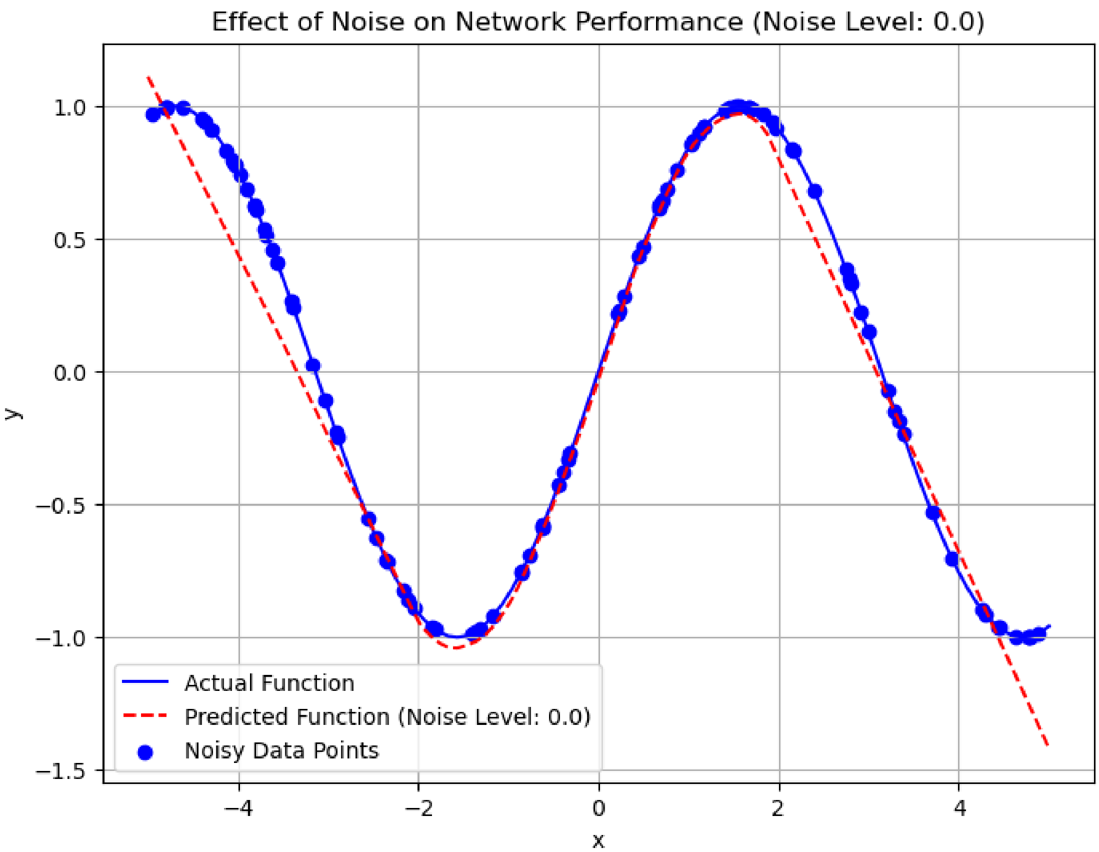
```
4/4              0s 5ms/step - loss:
2.9238 - mae: 1.3374
4/4              0s 0s/step - loss:
3.0630 - mae: 1.4109
4/4              0s 5ms/step - loss:
2.2823 - mae: 1.1124
```



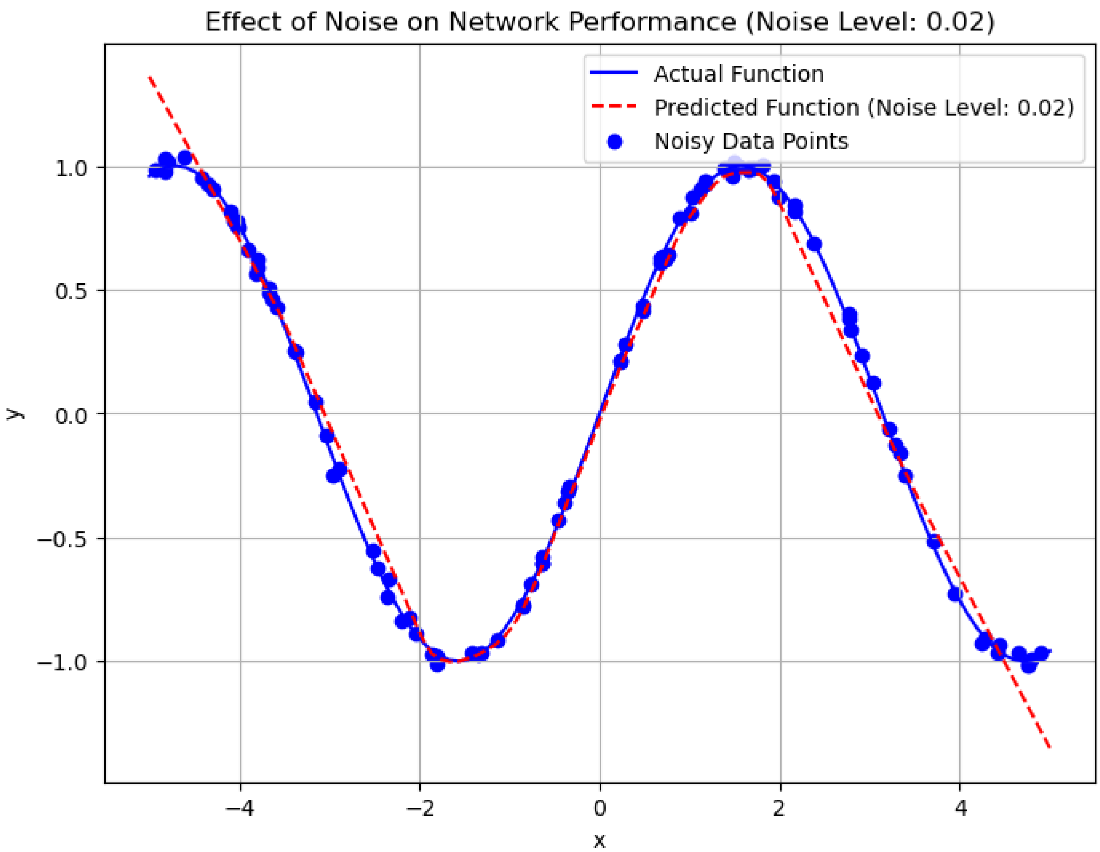
[ ]:
```
# Generate training data
np.random.seed(0)
num_points = 100
x_train = np.random.uniform(-5, 5, num_points)
y_train = sinusoidal_function(x_train)
```

```
# Function to add noise to data points
def add_noise(data, noise_level):
    noisy_data = data + np.random.normal(0, noise_level, len(data))
    return noisy_data

# Test different noise levels
noise_levels = [0.0,0.02, 0.11, 0.21,0.45, 0.89]

for noise_level in noise_levels:
    # Add noise to data points
    x_noisy = add_noise(x_train, noise_level)
    y_noisy = add_noise(y_train, noise_level)

    # Build and train the model with noisy data
    model = keras.Sequential([
        keras.layers.Dense(100, activation='relu', input_shape=(1,)),
        keras.layers.Dense(100, activation='relu'),
        keras.layers.Dense(1)
    ])
    model.compile(optimizer='adam', loss='mse', metrics=['mae'])
    history = model.fit(x_noisy, y_noisy, epochs=100, verbose=0)

    # Evaluate and plot results
    x_range = np.linspace(-5, 5, 100)
    y_actual = sinusoidal_function(x_range)
    y_predicted = model.predict(x_range)

    plt.figure(figsize=(8, 6))
    plt.plot(x_range, y_actual, label='Actual Function', color='blue')
    plt.plot(x_range, y_predicted, label=f'Predicted Function (Noise Level:␣
 ↪{noise_level})', color='red', linestyle='--')
    plt.scatter(x_noisy, y_noisy, color='blue', label='Noisy Data Points')
    plt.xlabel('x')
    plt.ylabel('y')
    plt.title(f'Effect of Noise on Network Performance (Noise Level:␣
 ↪{noise_level})')
    plt.legend()
    plt.grid(True)
    plt.show()

    x_test = np.linspace(0.1, 5, 50)
    y_test = sinusoidal_function(x_test)
    test_loss, test_mae = model.evaluate(x_test, y_test, verbose=0)
    print(f'Test Loss (MSE): {test_loss}')
```

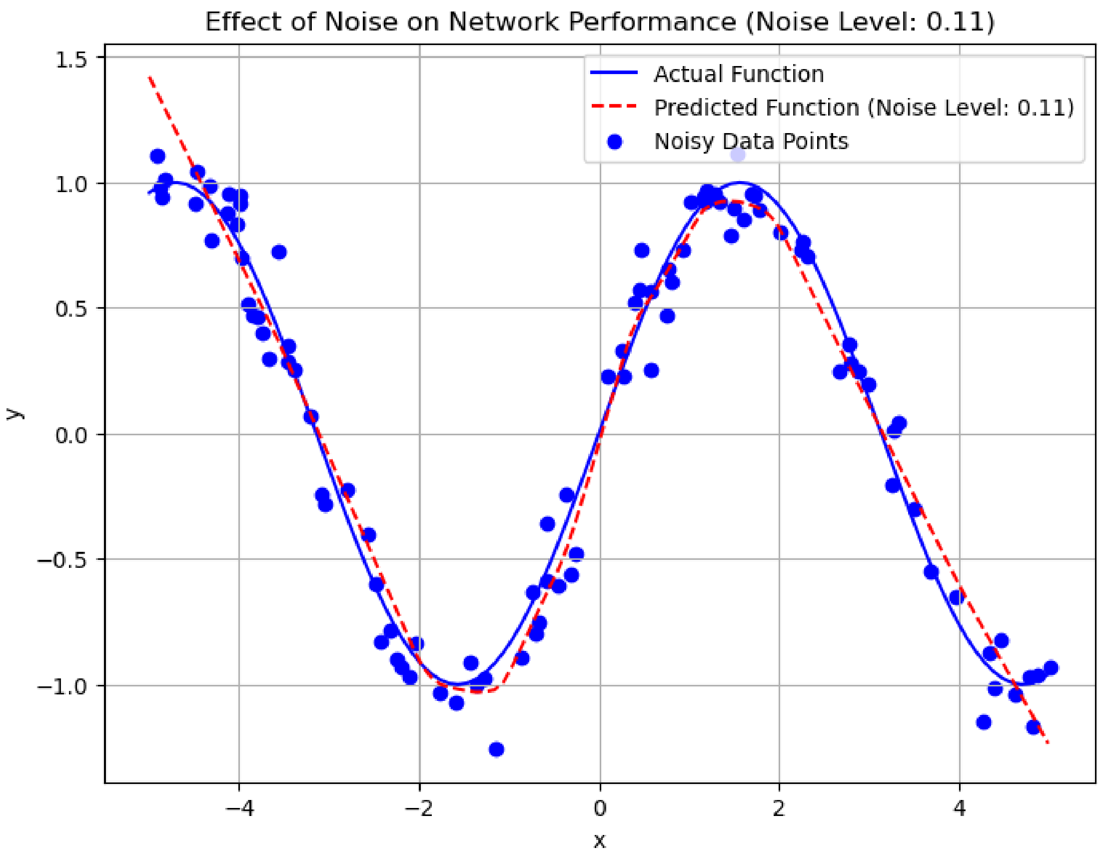
```
4/4              0s 5ms/step
```



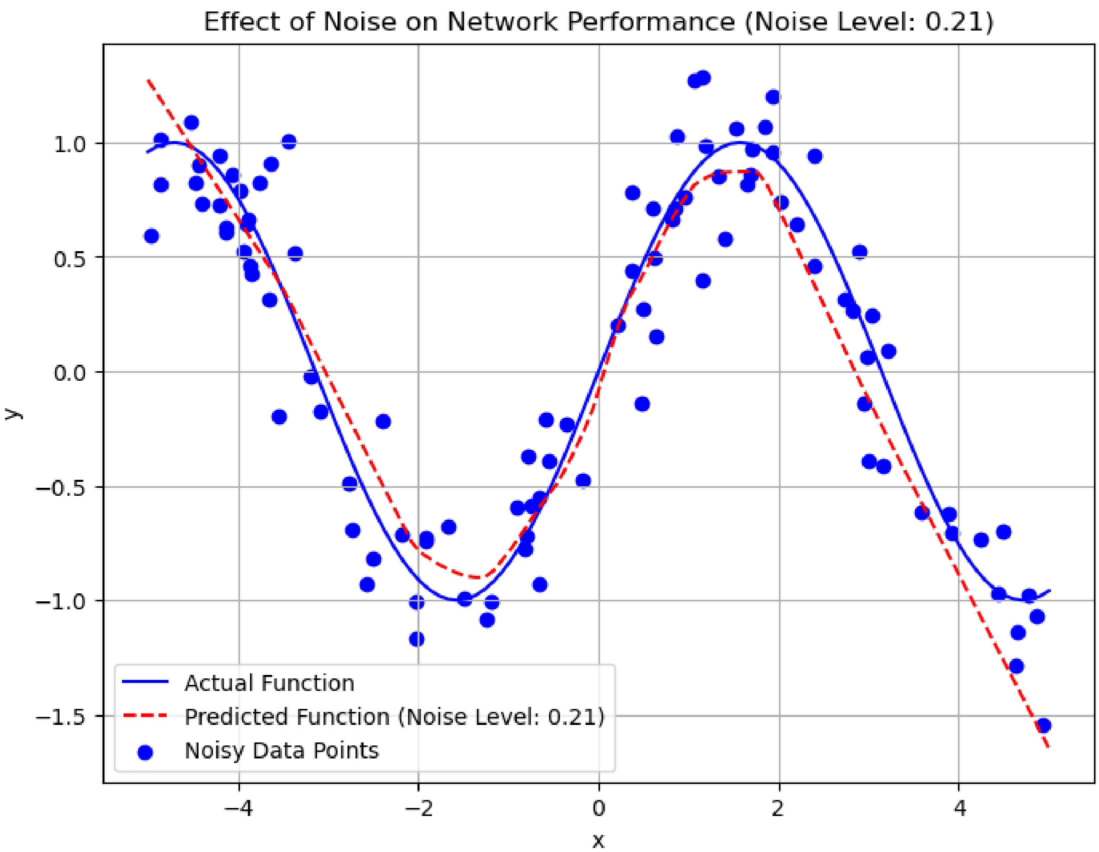

```
Test Loss (MSE): 0.015041690319776535
4/4              0s 5ms/step
```



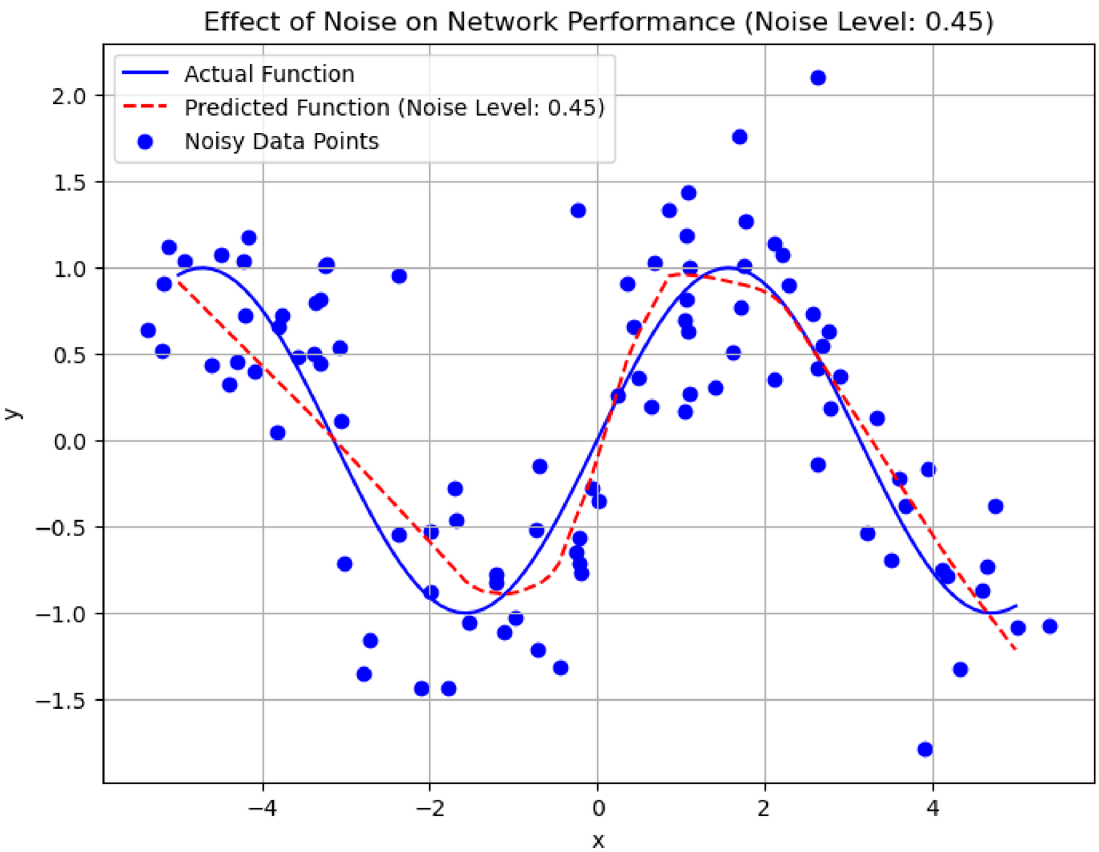

```
Test Loss (MSE): 0.011117160320281982
4/4              0s 5ms/step
```



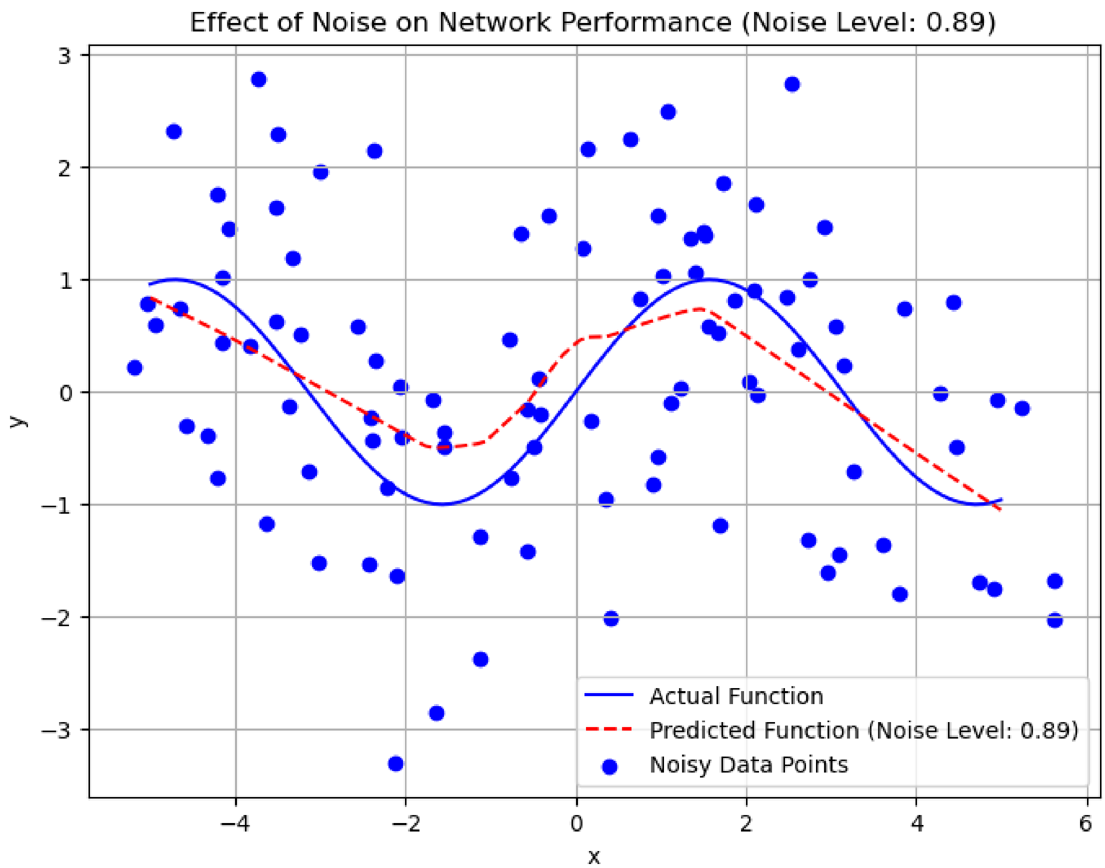

```
Test Loss (MSE): 0.010097881779074669
4/4              0s 7ms/step
```




```
Test Loss (MSE): 0.05769989639520645
4/4              0s 6ms/step
```




```
Test Loss (MSE): 0.015739614143967 63
4/4             0s 6ms/step
```



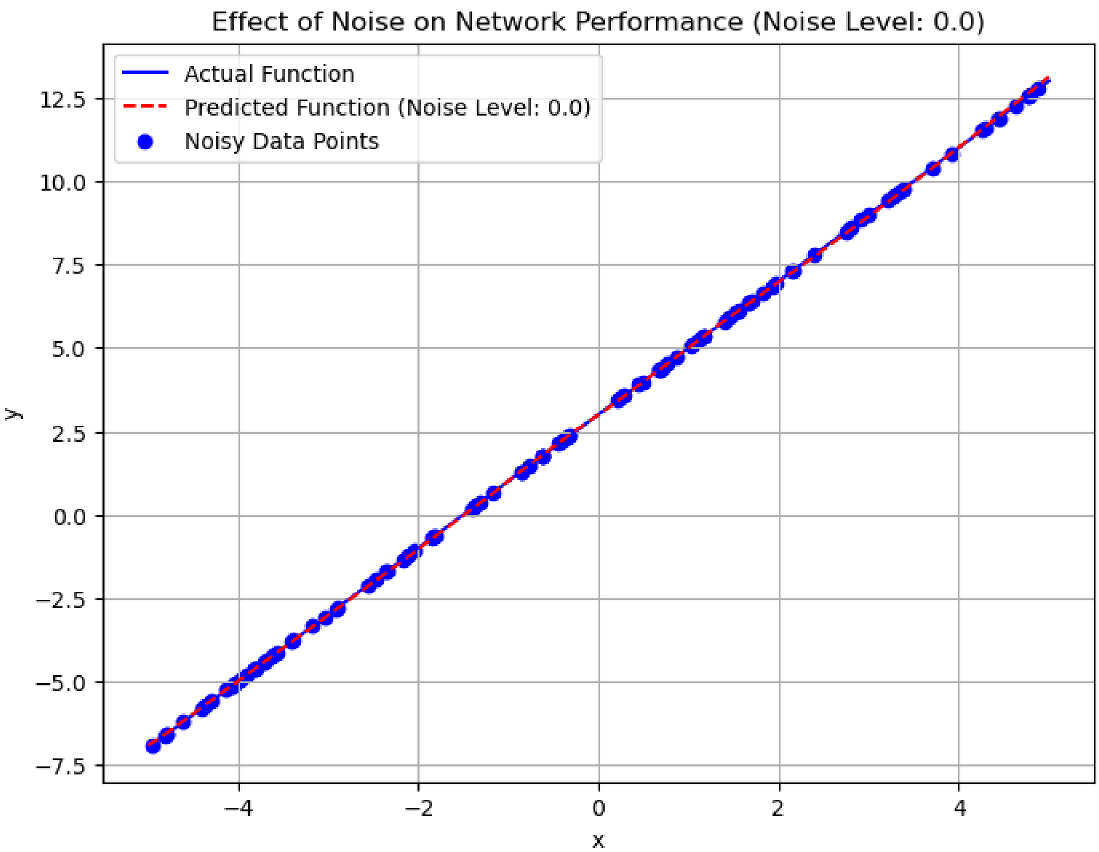

```
Test Loss (MSE): 0.056920234113931656
```

3.3.2 linear

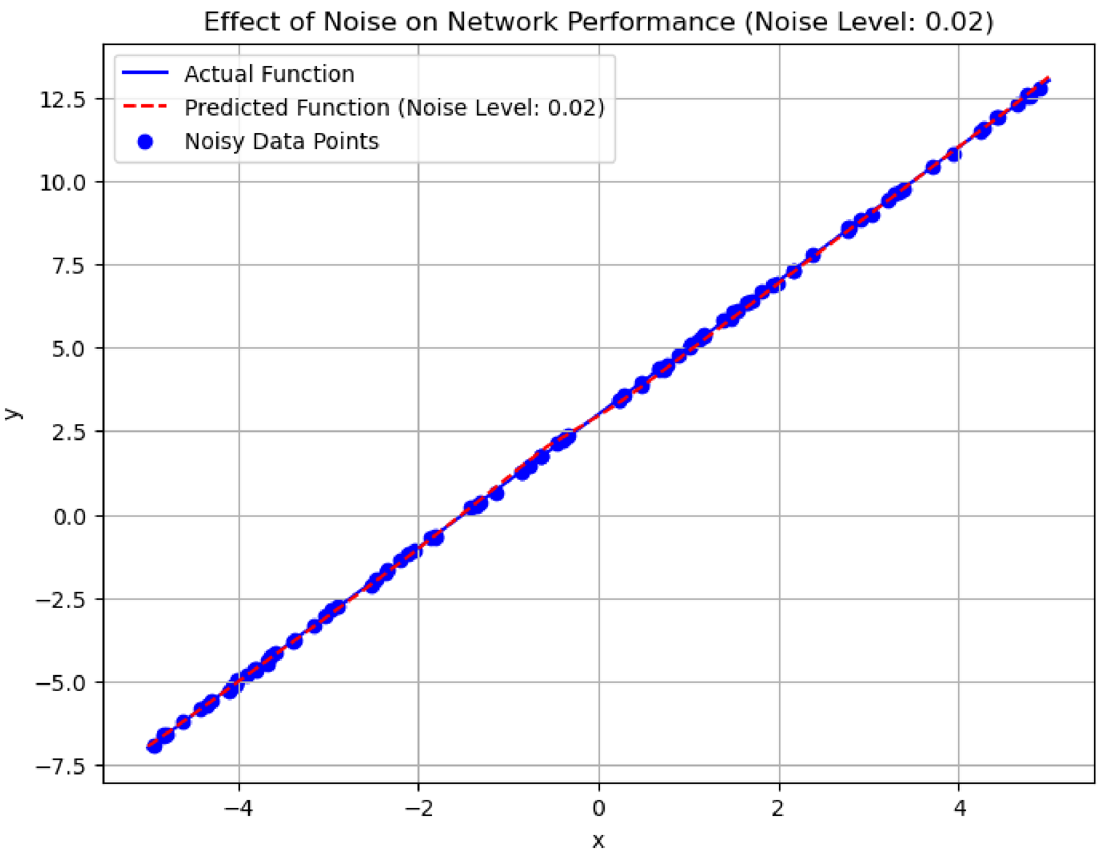
[ ]:
```python
# Generate training data
np.random.seed(0)
num_points = 100
x_train = np.random.uniform(-5, 5, num_points)
y_train = linear_function(x_train)

# Function to add noise to data points
def add_noise(data, noise_level):
    noisy_data = data + np.random.normal(0, noise_level, len(data))
    return noisy_data

# Test different noise levels
noise_levels = [0.0,0.02, 0.11, 0.21,0.45, 0.89]

for noise_level in noise_levels:
    # Add noise to data points
```

```
    x_noisy = add_noise(x_train, noise_level)
    y_noisy = add_noise(y_train, noise_level)

    # Build and train the model with noisy data
    model = keras.Sequential([
        keras.layers.Dense(64, activation='relu', input_shape=(1,)),
        keras.layers.Dense(64, activation='relu'),
        keras.layers.Dense(1)
    ])
    model.compile(optimizer='adam', loss='mse', metrics=['mae'])
    history = model.fit(x_noisy, y_noisy, epochs=100, verbose=0)

    # Evaluate and plot results
    x_range = np.linspace(-5, 5, 100)
    y_actual = linear_function(x_range)
    y_predicted = model.predict(x_range)

    plt.figure(figsize=(8, 6))
    plt.plot(x_range, y_actual, label='Actual Function', color='blue')
    plt.plot(x_range, y_predicted, label=f'Predicted Function (Noise Level:␣
↪{noise_level})', color='red', linestyle='--')
    plt.scatter(x_noisy, y_noisy, color='blue', label='Noisy Data Points')
    plt.xlabel('x')
    plt.ylabel('y')
    plt.title(f'Effect of Noise on Network Performance (Noise Level:␣
↪{noise_level})')
    plt.legend()
    plt.grid(True)
    plt.show()

    x_test = np.linspace(0.1, 5, 50)
    y_test = linear_function(x_test)
    test_loss, test_mae = model.evaluate(x_test, y_test, verbose=0)
    print(f'Test Loss (MSE): {test_loss}')
```

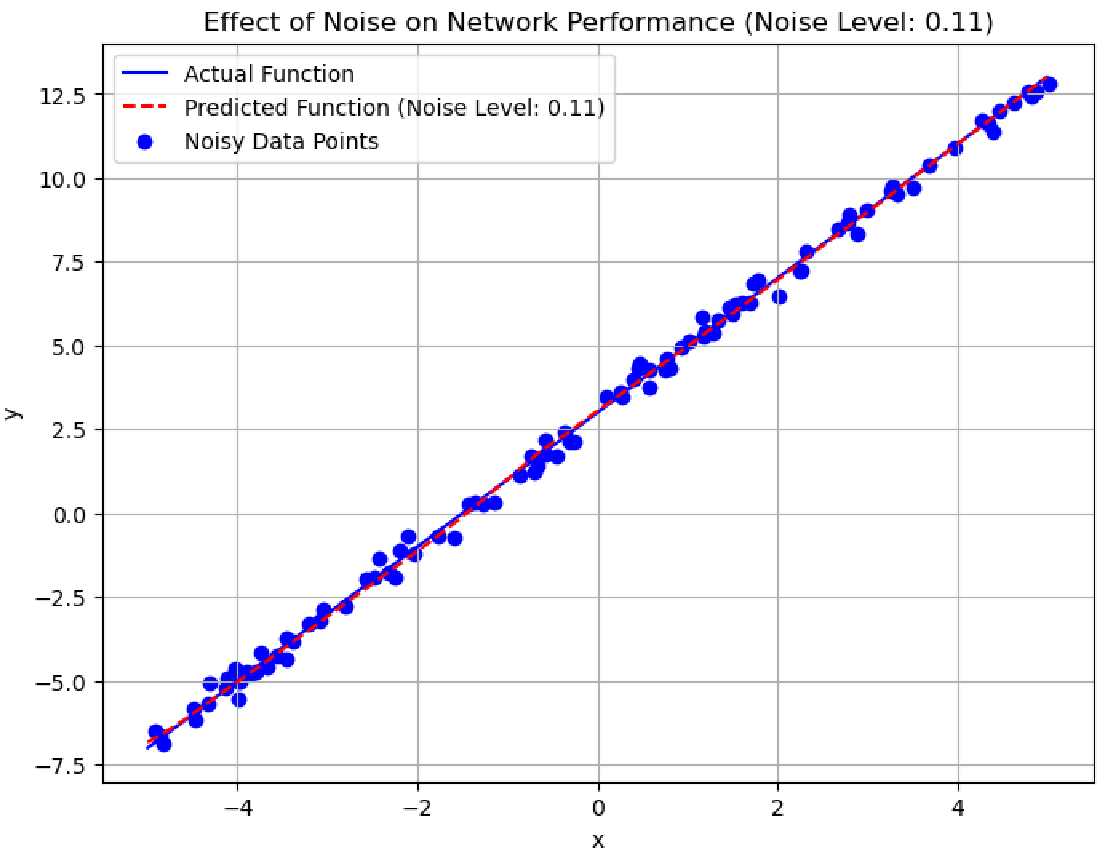
```
4/4              0s 8ms/step
```



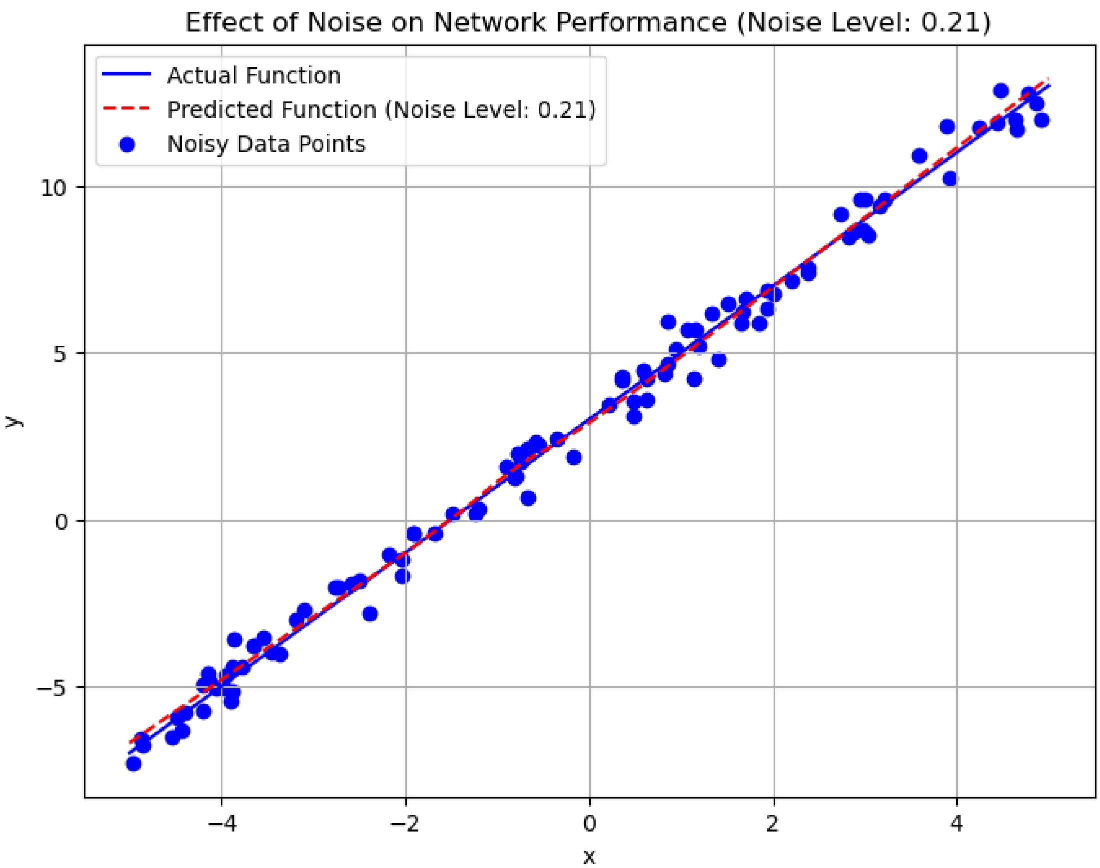

```
Test Loss (MSE): 0.0020937405060976744
4/4              0s 6ms/step
```



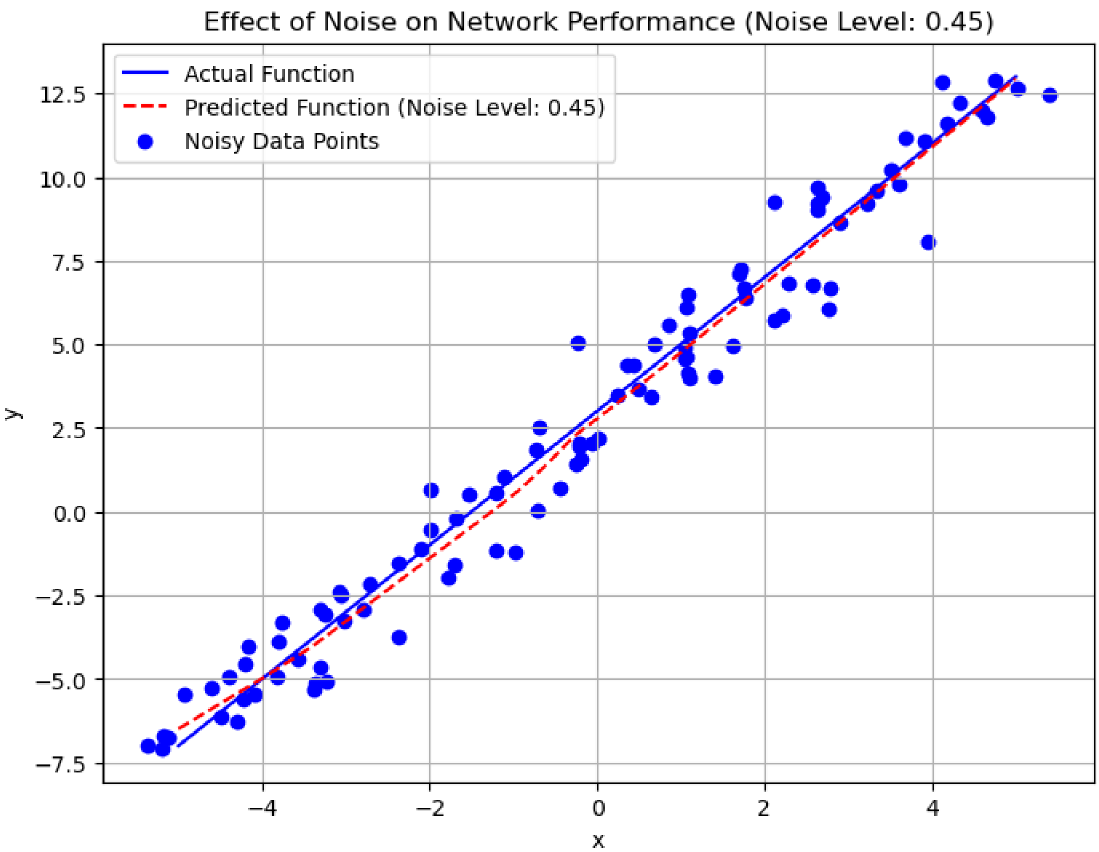

```
Test Loss (MSE): 0.0056772357784211636
4/4              0s 6ms/step
```



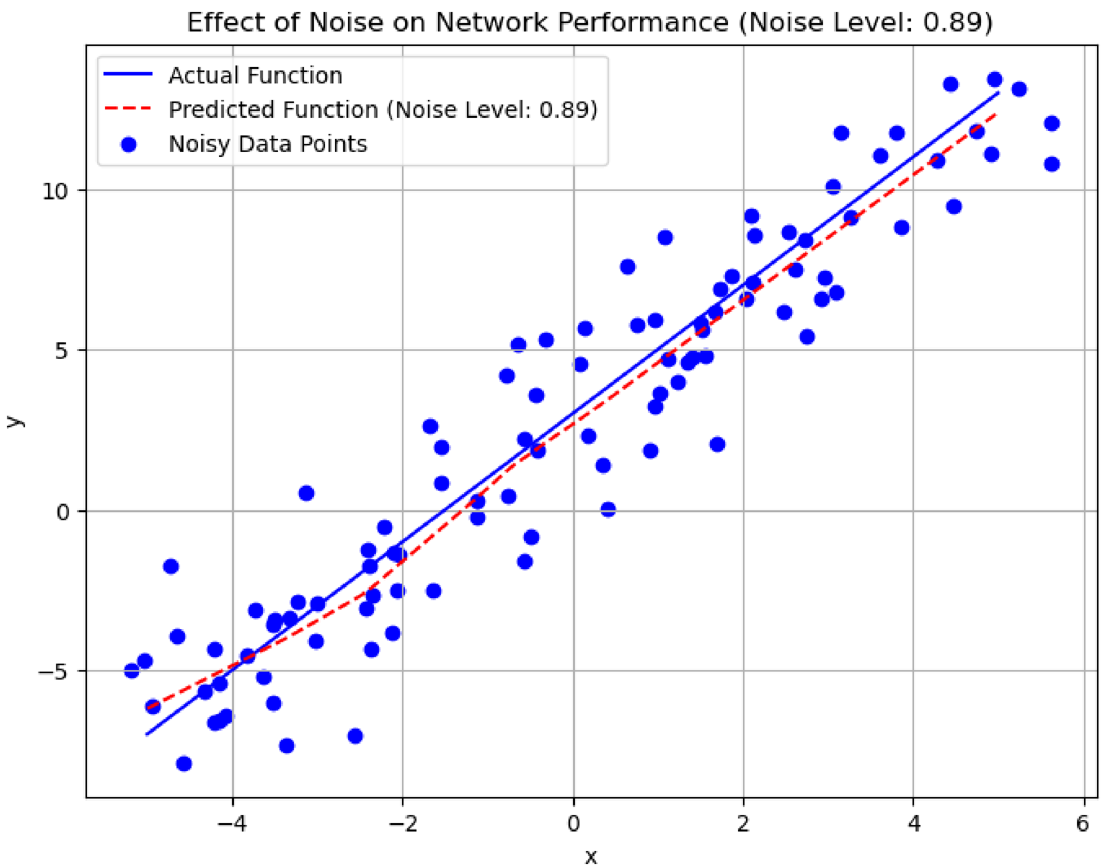

```
Test Loss (MSE): 0.0008531596977263689
4/4              0s 5ms/step
```




```
Test Loss (MSE): 0.013926164247095585
4/4              0s 11ms/step
```




```
Test Loss (MSE): 0.03280578553676605
4/4              0s 11ms/step
```



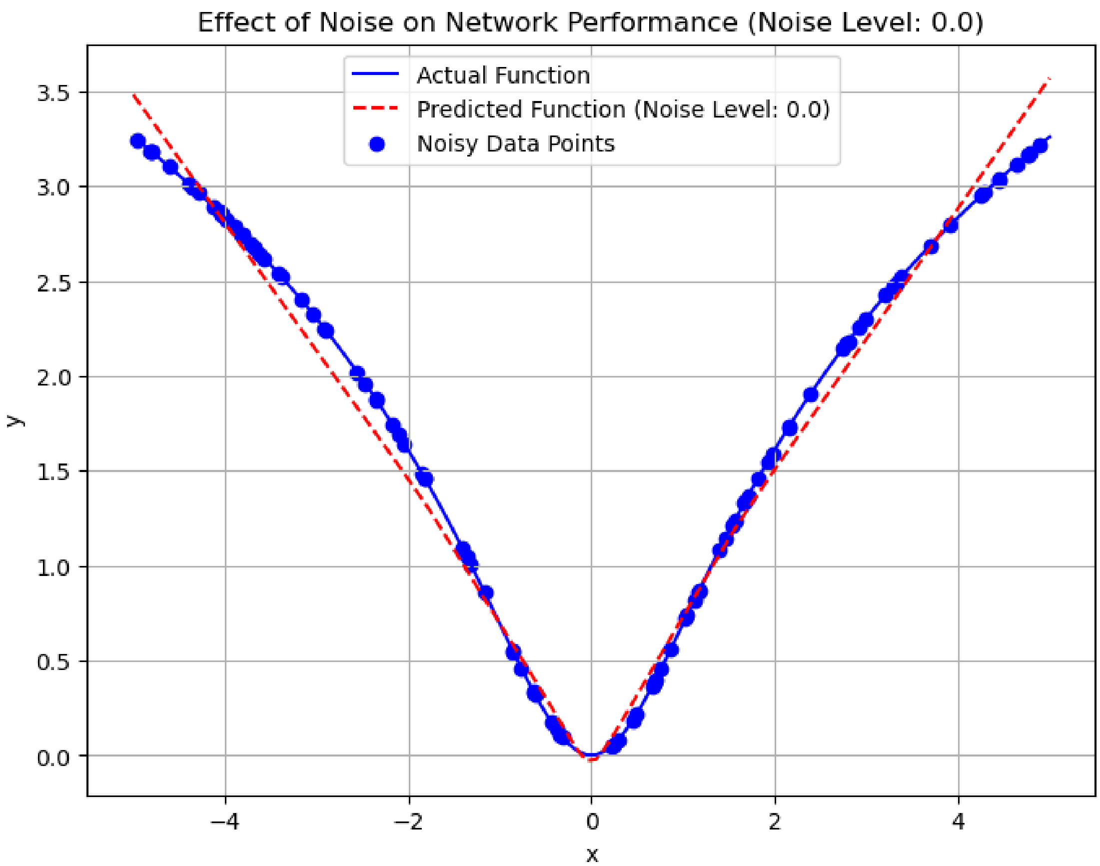
Test Loss (MSE): 0.2457078993320465

3.3.3 complicated

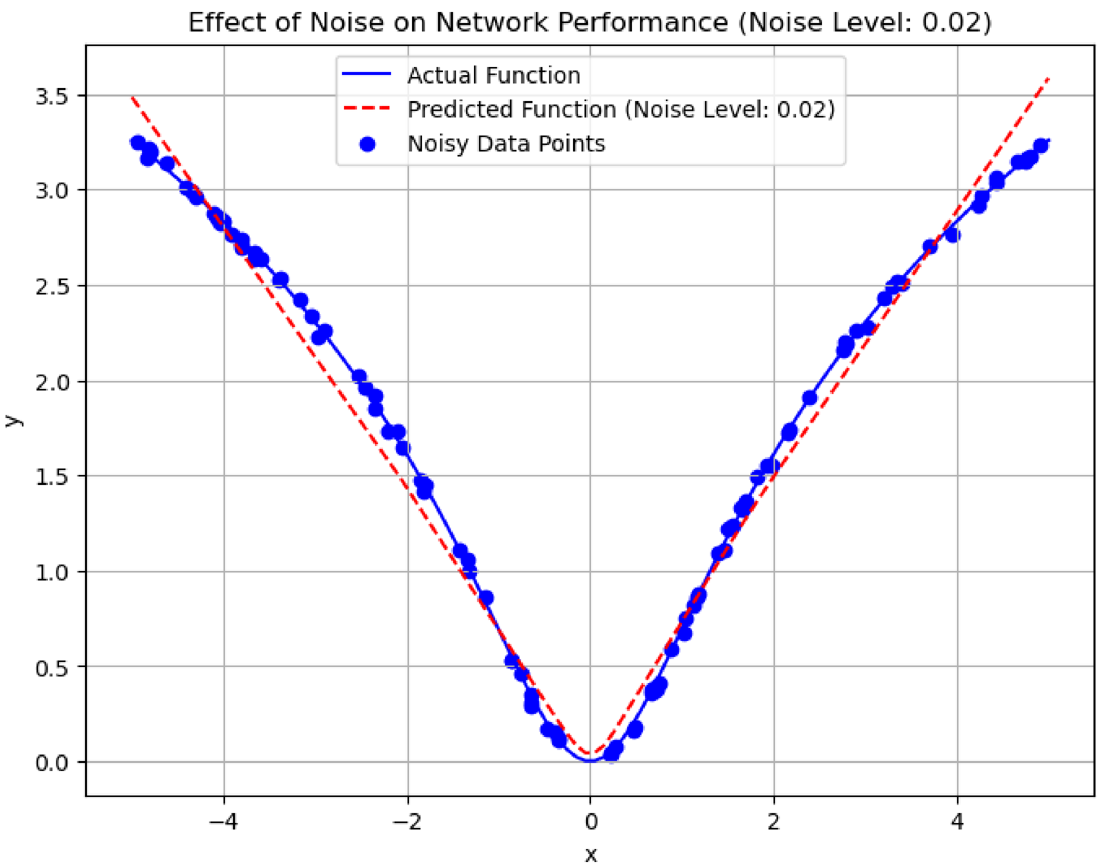
[ ]:
```
# Generate training data
np.random.seed(0)
num_points = 100
x_train = np.random.uniform(-5, 5, num_points)
y_train = complicated_function(x_train)

# Function to add noise to data points
def add_noise(data, noise_level):
    noisy_data = data + np.random.normal(0, noise_level, len(data))
    return noisy_data

# Test different noise levels
noise_levels = [0.0,0.02, 0.11, 0.21,0.45, 0.89]

for noise_level in noise_levels:
    # Add noise to data points
```

```
    x_noisy = add_noise(x_train, noise_level)
    y_noisy = add_noise(y_train, noise_level)

    # Build and train the model with noisy data
    model = keras.Sequential([
        keras.layers.Dense(64, activation='relu', input_shape=(1,)),
        keras.layers.Dense(64, activation='relu'),
        keras.layers.Dense(1)
    ])
    model.compile(optimizer='adam', loss='mse', metrics=['mae'])
    history = model.fit(x_noisy, y_noisy, epochs=100, verbose=0)

    # Evaluate and plot results
    x_range = np.linspace(-5, 5, 100)
    y_actual = complicated_function(x_range)
    y_predicted = model.predict(x_range)

    plt.figure(figsize=(8, 6))
    plt.plot(x_range, y_actual, label='Actual Function', color='blue')
    plt.plot(x_range, y_predicted, label=f'Predicted Function (Noise Level:␣
↪{noise_level})', color='red', linestyle='--')
    plt.scatter(x_noisy, y_noisy, color='blue', label='Noisy Data Points')
    plt.xlabel('x')
    plt.ylabel('y')
    plt.title(f'Effect of Noise on Network Performance (Noise Level:␣
↪{noise_level})')
    plt.legend()
    plt.grid(True)
    plt.show()

    x_test = np.linspace(0.1, 5, 50)
    y_test = complicated_function(x_test)
    test_loss, test_mae = model.evaluate(x_test, y_test, verbose=0)
    print(f'Test Loss (MSE): {test_loss}')
```

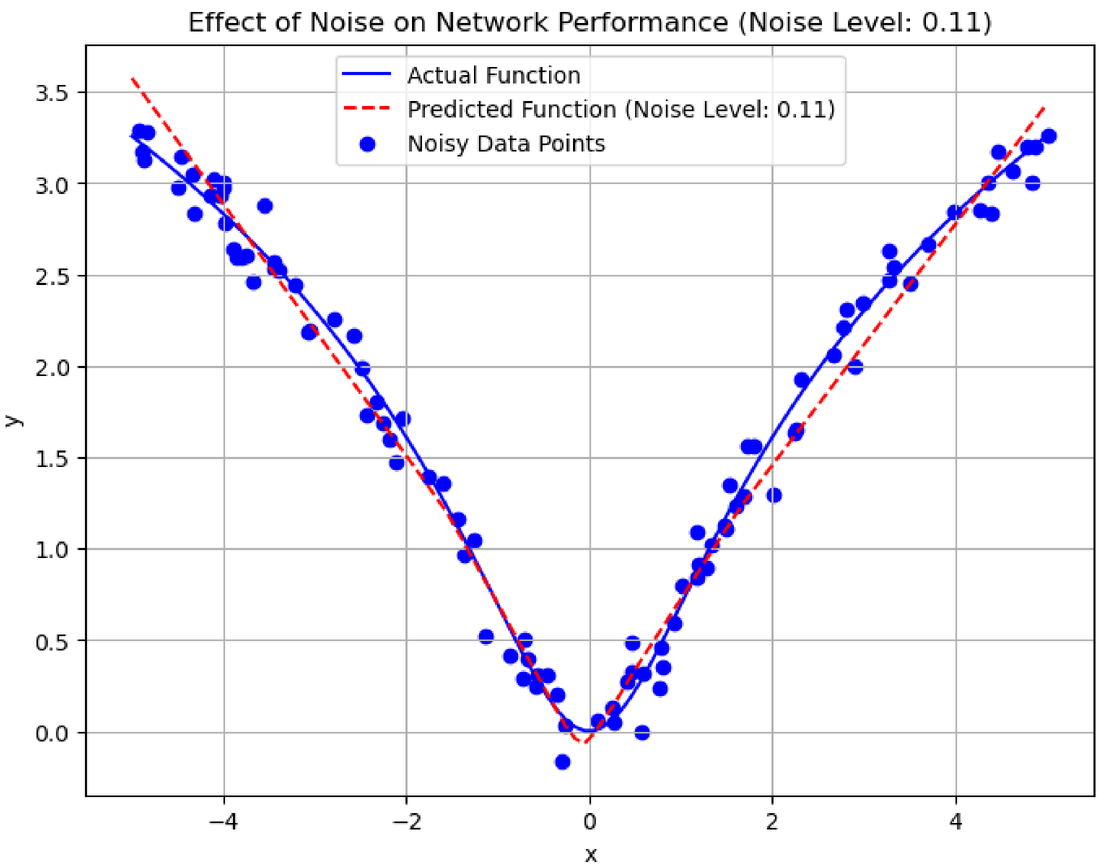
```
4/4              0s 11ms/step
```



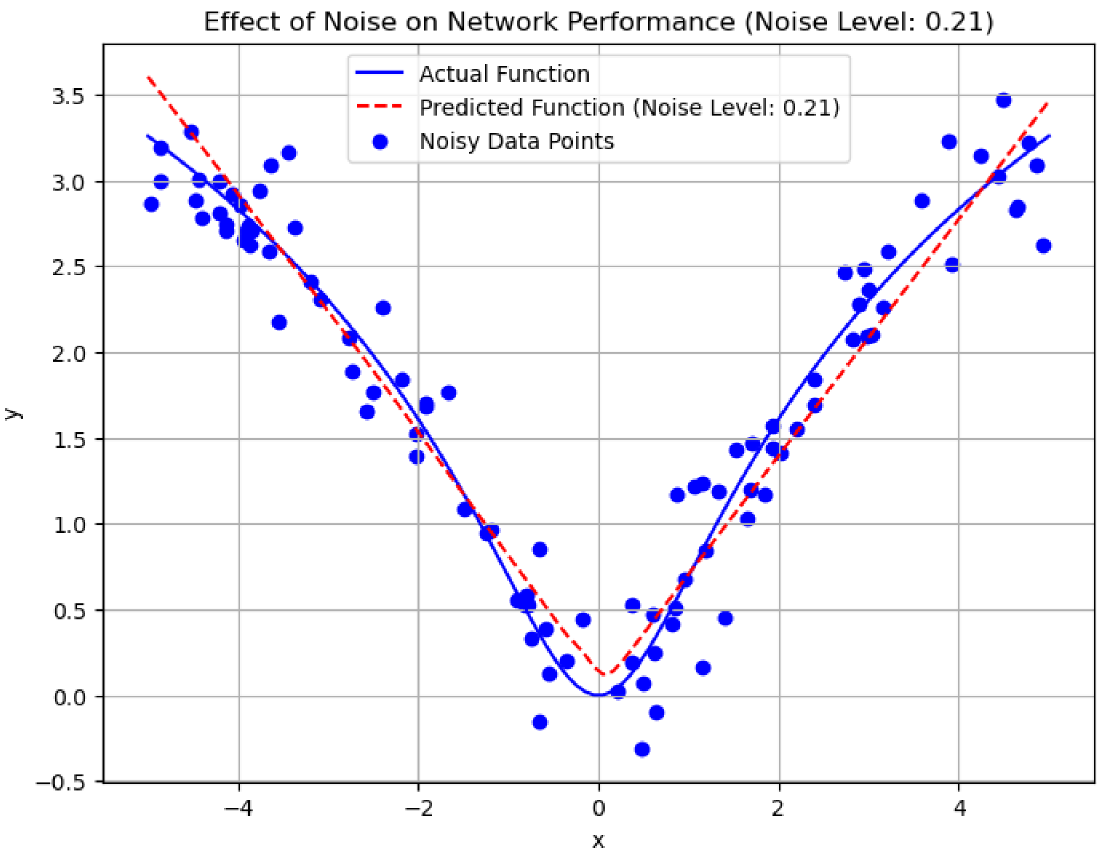

```
Test Loss (MSE): 0.013466911390423775
4/4              0s 6ms/step
```



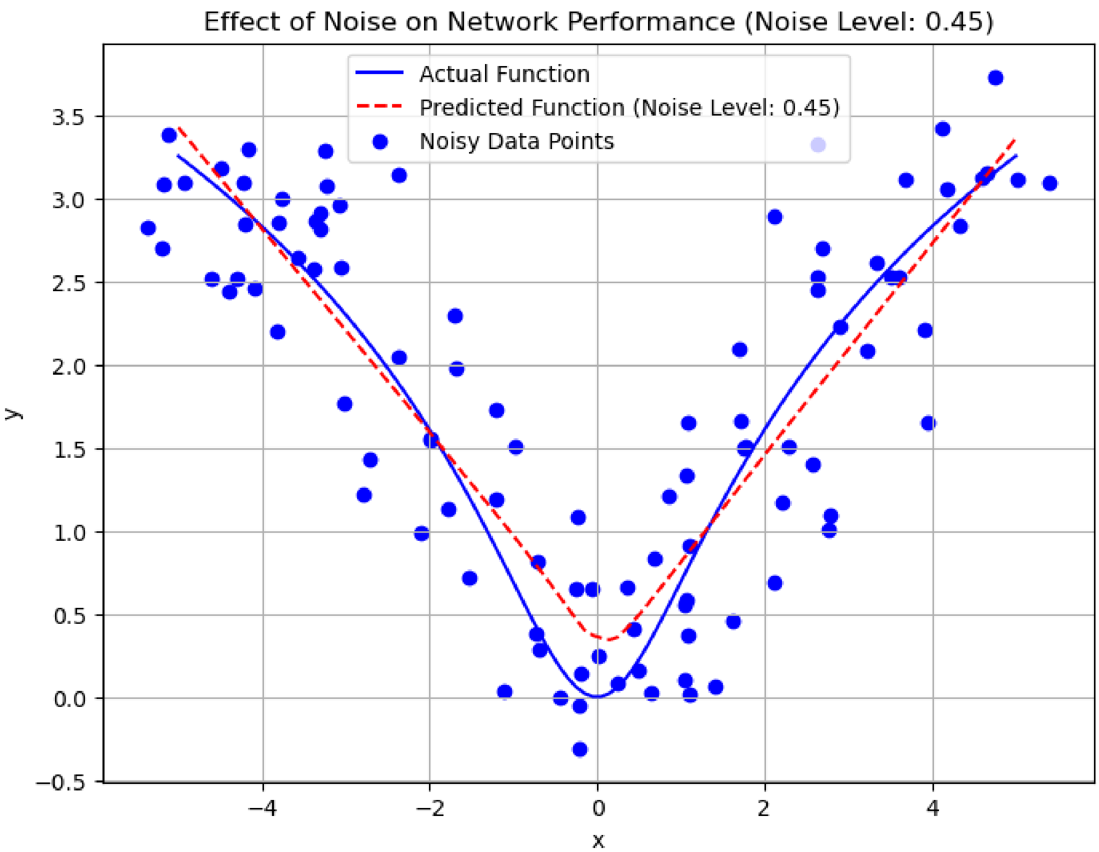

```
Test Loss (MSE): 0.015841234475374222
4/4              0s 5ms/step
```



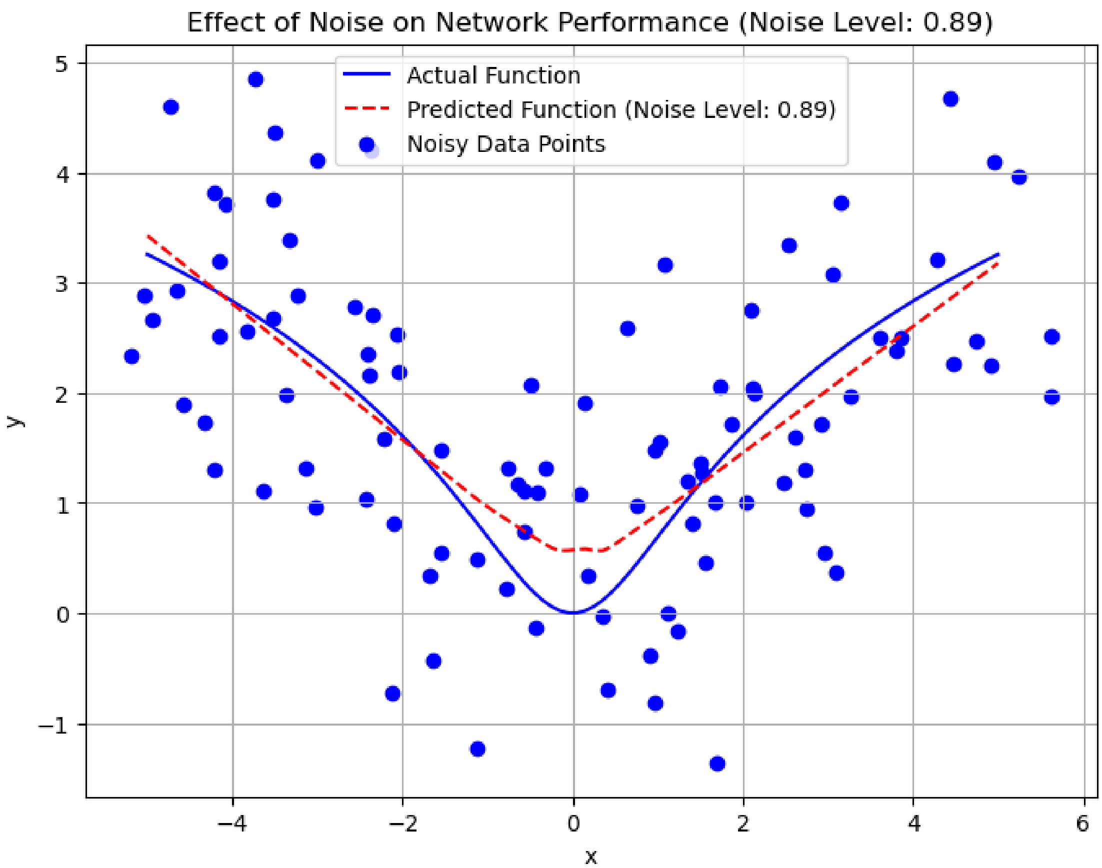

```
Test Loss (MSE): 0.015294580720365047
4/4              0s 10ms/step
```




```
Test Loss (MSE): 0.024073727428913116
4/4            0s 6ms/step
```




```
Test Loss (MSE): 0.028606681153178215
4/4              0s 11ms/step
```




```
Test Loss (MSE): 0.06482572853565216
```

3.4 Analysis

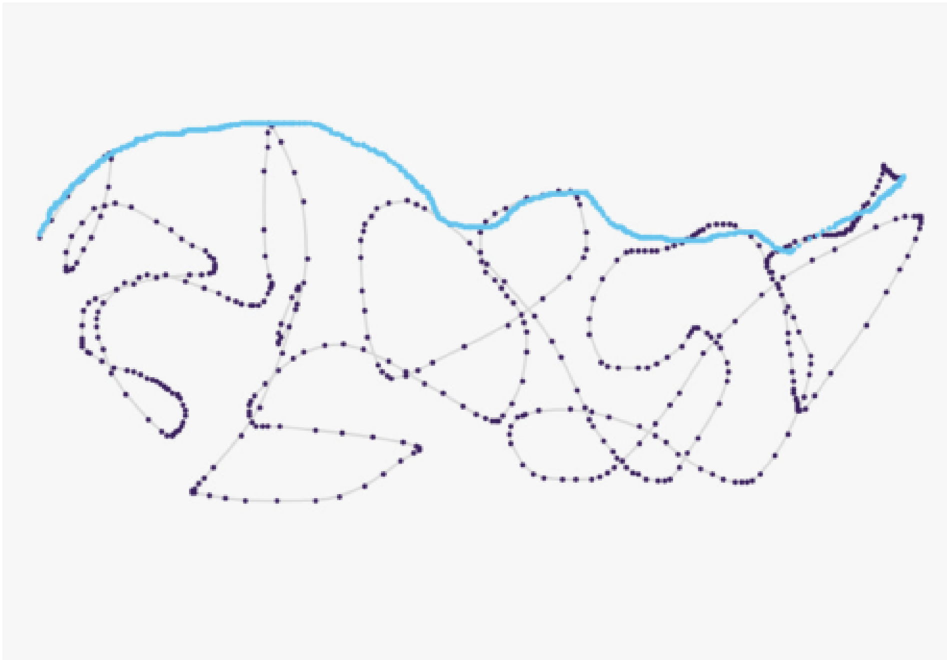
- By adding noise to the training data, we observe how the neural network models perform under different levels of noise.
- The training loss graphs show how each model adapts to the noisy training data over epochs.
- Evaluating the test loss with clean data helps us understand how well the models generalize and handle noise during inference.

4 Function estimation using input points

4.1 Data Frame

[ ]:
```
image_path = 'Screen.jpg'


img = Image.open(image_path)
```

```
plt.imshow(img)
plt.axis('off')
plt.show()
```



[ ]:

```
csv_file_path = 'data.csv'

dataFrame = pd.read_csv(csv_file_path)

print("show data frame Information")
print(dataFrame.head())

print(dataFrame.columns.tolist())

print(dataFrame.describe())
```

```
show data frame Information
    x           y  z
0  31  322.600006  a
1  40  335.600006  a
2  52  353.600006  a
3  63  366.600006  a
4  71  375.600006  a
['x', 'y', 'z']
                 x            y
count  1087.000000  1087.000000
```

```
mean    353.307268  316.775719
std     200.069894   71.809046
min      31.000000  124.600006
25%     167.500000  287.100006
50%     363.000000  331.600006
75%     528.500000  364.100006
max     695.000000  408.600006
```

[ ]:
```
df = dataFrame.copy()
```

[ ]:
```
df = df[df['z'] != 'a']
```

[ ]:
```
X = df['x']
Y = df['y']
```

[ ]:
```
print(df.describe())
```

```
                x           y
count  616.000000  616.000000
mean   339.581169  359.520461
std    197.863071   30.714820
min     31.000000  311.600006
25%    159.750000  330.600006
50%    339.500000  354.600006
75%    507.250000  390.600006
max    682.000000  408.600006
```

4.2 Neural Network Training and Evaluation

The provided code trains a deep neural network using a Sequential model architecture with multiple layers of 100 neurons each. The model is trained on input data `X` and target data `Y` for 1000 epochs using the Adam optimizer and Mean Squared Error (MSE) loss function.

4.2.1 Model Architecture

- **Input Layer**:
  - Shape: 1 (input feature)
- **Hidden Layers** (14 layers):
  - Dense layers with 100 neurons each and ReLU activation function
- **Output Layer**:
  - Single neuron for regression output (continuous prediction)

4.2.2 Model Compilation

- **Optimizer**: Adam optimizer
- **Loss Function**: Mean Squared Error (MSE)
- **Metrics**: Mean Absolute Error (MAE)

4.2.3 Training Process

The model is trained on the entire dataset (X and Y) for 1000 epochs. The training process is conducted silently (verbose=0).

4.2.4 Evaluation and Analysis

After training, the model's predictions are generated over a range of input values (x_range). The actual function values (y_actual) and predicted values (y_predicted) are plotted to visualize the model's performance.

Performance Metrics:

- **Mean Squared Error (MSE)**:
  - Value: 118.93
  - Indicates the average squared error between actual and predicted values.
- **R-squared (R2)**:
  - Value: 0.874
  - Indicates the percentage of variance in the target variable explained by the input variables.
- **Percentage Match (Accuracy)**:
  - Value: 87.37%
  - Represents the percentage of agreement (or match) between actual and predicted values based on R-squared.

4.2.5 Visualization

The plot displays the actual function (y_actual) in blue and the predicted function (y_predicted) in red dashed lines over the input range (x_range). This visualization helps in assessing how well the model captures the underlying function.

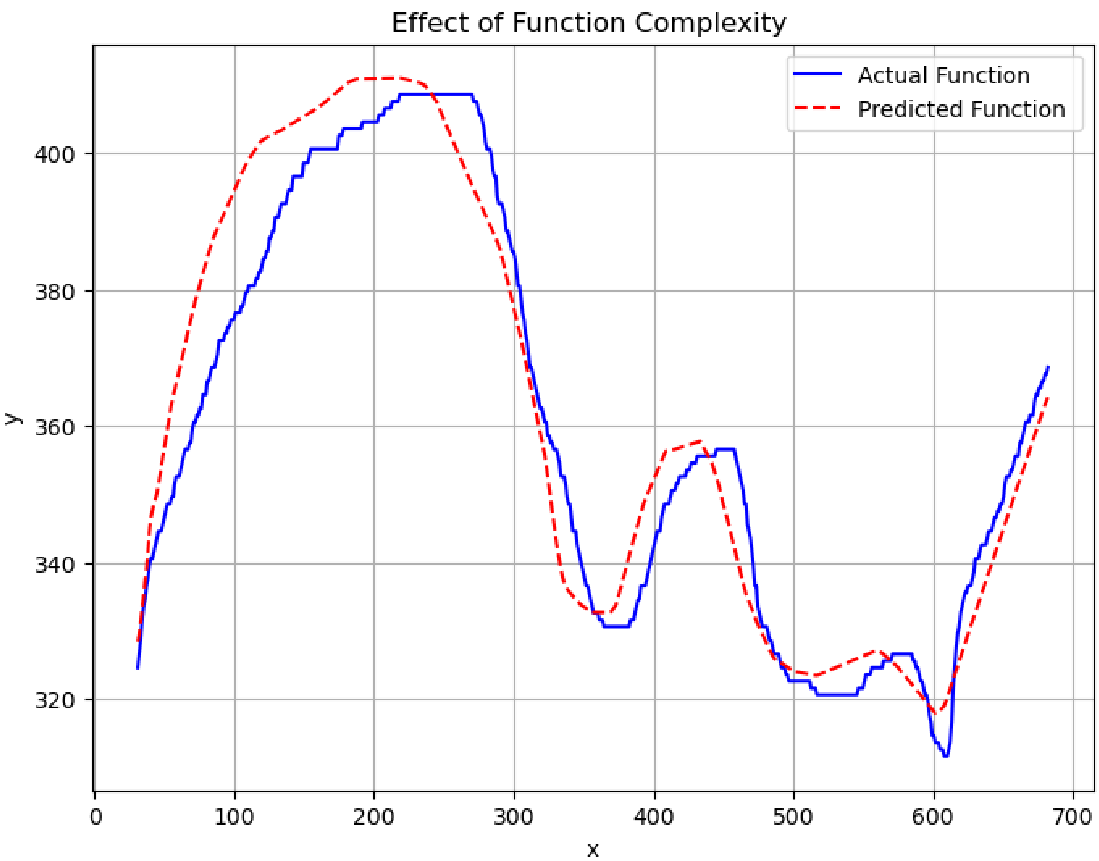
Overall, the model demonstrates strong predictive performance with a high R-squared value of 0.874, indicating that 87.37% of the variance in the target variable is explained by the input variables.

[ ]:
```
model = keras.Sequential([
    keras.layers.Dense(100, activation='relu', input_shape=(1,)),
    keras.layers.Dense(100, activation='relu'),
    keras.layers.Dense(100, activation='relu'),
    keras.layers.Dense(100, activation='relu'),
    keras.layers.Dense(100, activation='relu'),
    keras.layers.Dense(100, activation='relu'),
    keras.layers.Dense(100, activation='relu'),
    keras.layers.Dense(100, activation='relu'),
    keras.layers.Dense(100, activation='relu'),
    keras.layers.Dense(100, activation='relu'),
    keras.layers.Dense(100, activation='relu'),
    keras.layers.Dense(100, activation='relu'),
    keras.layers.Dense(100, activation='relu'),
    keras.layers.Dense(100, activation='relu'),
```

```
    keras.layers.Dense(100, activation='relu'),
    keras.layers.Dense(100, activation='relu'),
    keras.layers.Dense(100, activation='relu'),
    keras.layers.Dense(100, activation='relu'),
    keras.layers.Dense(1)
])
model.compile(optimizer='adam', loss='mse', metrics=['mae'])
history = model.fit(X, Y, epochs=1000, verbose=0)

# Evaluate and plot results
x_range = np.linspace(X.min(), X.max(), 616)
y_actual =  Y
y_predicted = model.predict(x_range)


y_pred = model.predict(X)
mse = mean_squared_error(y_actual, y_predicted)
r2 = r2_score(y_actual, y_predicted)

print(f'Test Mean Squared Error (MSE): {mse}')
print(f'Test R-squared (R2): {r2}')

# Calculate percentage match (accuracy) using R-squared
percentage_match = r2 * 100
print(f'Percentage Match (R-squared): {percentage_match:.2f}%')


plt.figure(figsize=(8, 6))
plt.plot(x_range, y_actual, label='Actual Function', color='blue')
plt.plot(x_range, y_predicted, label=f'Predicted Function ', color='red',␣
 ↪linestyle='--')

plt.xlabel('x')
plt.ylabel('y')
plt.title(f'Effect of Function Complexity ')
plt.legend()
plt.grid(True)
plt.show()
```

```
c:\Users\Mahdi\anaconda3\Lib\site-packages\keras\src\layers\core\dense.py:86:
UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When
using Sequential models, prefer using an `Input(shape)` object as the first
layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)

20/20              0s 5ms/step
20/20              0s 5ms/step
Test Mean Squared Error (MSE): 75.32767578331881
```

```
Test R-squared (R2): 0.9200231646077321
Percentage Match (R-squared): 92.00%
```



5 Neural Network for Image Classification using Fashion-MNIST Dataset

In this phase of the project, we will use a neural network for image classification using the Fashion-MNIST dataset. The Fashion-MNIST dataset consists of grayscale images of fashion items categorized into 10 classes, such as shirts, shoes, bags, etc. Each image has dimensions of 28x28 pixels.

Steps: 1. Research the extensive use of neural networks in data classification across various domains. 2. Select a dataset suitable for classification tasks, preferably from image or sound databases available in frameworks like Keras. 3. Implement a classification model using TensorFlow/Keras to categorize data into more than two classes. 4. Train the classification model and aim to optimize its accuracy. 5. Evaluate the model's performance using metrics such as accuracy, confusion matrix, and visualization techniques.

5.1 Load and Preprocess the Dataset

[ ]:
```
# Load the Fashion-MNIST dataset
(train_images, train_labels), (test_images, test_labels) = fashion_mnist.
↪load_data()

# Normalize the pixel values to be between 0 and 1
train_images = train_images.astype('float32') / 255.0
test_images = test_images.astype('float32') / 255.0

# Reshape the images to have a single channel (grayscale)
train_images = np.expand_dims(train_images, axis=-1)
test_images = np.expand_dims(test_images, axis=-1)

# Print the shapes of train and test datasets
print("Train Images Shape:", train_images.shape)
print("Train Labels Shape:", train_labels.shape)
print("Test Images Shape:", test_images.shape)
print("Test Labels Shape:", test_labels.shape)
```

```
Train Images Shape: (60000, 28, 28, 1)
Train Labels Shape: (60000,)
Test Images Shape: (10000, 28, 28, 1)
Test Labels Shape: (10000,)
```

5.2 Build the Neural Network Model

Now, build a neural network model for image classification. We'll use a simple Convolutional Neural Network (CNN) architecture for this task.

[ ]:
```
model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28,␣
↪1)),
    tf.keras.layers.MaxPooling2D((2, 2)),
    tf.keras.layers.Conv2D(64, (3, 3), activation='relu'),
    tf.keras.layers.MaxPooling2D((2, 2)),
    tf.keras.layers.Conv2D(64, (3, 3), activation='relu'),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(10)  # Output layer with 10 units (one for each class)
])

# Compile the model
model.compile(optimizer='adam',
              loss=tf.keras.losses.
↪SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

```
# Display the model summary
model.summary()
```

```
c:\Users\Mahdi\anaconda3\Lib\site-
packages\keras\src\layers\convolutional\base_conv.py:99: UserWarning: Do not
pass an `input_shape`/`input_dim` argument to a layer. When using Sequential
models, prefer using an `Input(shape)` object as the first layer in the model
instead.
  super().__init__(
```

Model: "sequential_46"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 26, 26, 32)	320
max_pooling2d (MaxPooling2D)	(None, 13, 13, 32)	0
conv2d_1 (Conv2D)	(None, 11, 11, 64)	18,496
max_pooling2d_1 (MaxPooling2D)	(None, 5, 5, 64)	0
conv2d_2 (Conv2D)	(None, 3, 3, 64)	36,928
flatten (Flatten)	(None, 576)	0
dense_163 (Dense)	(None, 64)	36,928
dense_164 (Dense)	(None, 10)	650

Total params: 93,322 (364.54 KB)

Trainable params: 93,322 (364.54 KB)

Non-trainable params: 0 (0.00 B)

5.3 Train the Model

Train the neural network model using the preprocessed training images and labels.

[ ]:
```
# Define the number of epochs and batch size
epochs = 10
batch_size = 32
```

```
# Train the model
history = model.fit(train_images, train_labels, epochs=epochs,␣
 ↪batch_size=batch_size,
                    validation_split=0.1)  # Use 10% of training data for␣
 ↪validation
```

```
Epoch 1/10
1688/1688               6s 3ms/step -
accuracy: 0.7392 - loss: 0.7131 - val_accuracy: 0.8613 - val_loss: 0.3704
Epoch 2/10
1688/1688               5s 3ms/step -
accuracy: 0.8751 - loss: 0.3471 - val_accuracy: 0.8817 - val_loss: 0.3288
Epoch 3/10
1688/1688               5s 3ms/step -
accuracy: 0.8951 - loss: 0.2868 - val_accuracy: 0.8980 - val_loss: 0.2848
Epoch 4/10
1688/1688               5s 3ms/step -
accuracy: 0.9079 - loss: 0.2516 - val_accuracy: 0.9047 - val_loss: 0.2735
Epoch 5/10
1688/1688               5s 3ms/step -
accuracy: 0.9181 - loss: 0.2243 - val_accuracy: 0.9090 - val_loss: 0.2574
Epoch 6/10
1688/1688               5s 3ms/step -
accuracy: 0.9225 - loss: 0.2066 - val_accuracy: 0.9090 - val_loss: 0.2576
Epoch 7/10
1688/1688               5s 3ms/step -
accuracy: 0.9290 - loss: 0.1871 - val_accuracy: 0.9102 - val_loss: 0.2503
Epoch 8/10
1688/1688               6s 3ms/step -
accuracy: 0.9384 - loss: 0.1686 - val_accuracy: 0.9120 - val_loss: 0.2545
Epoch 9/10
1688/1688               5s 3ms/step -
accuracy: 0.9416 - loss: 0.1580 - val_accuracy: 0.9090 - val_loss: 0.2641
Epoch 10/10
1688/1688               5s 3ms/step -
accuracy: 0.9481 - loss: 0.1420 - val_accuracy: 0.9130 - val_loss: 0.2576
```

5.4 Evaluate the Model

Evaluate the trained model on the test dataset to measure its performance

[ ]:
```
# Evaluate the model on the test dataset
test_loss, test_accuracy = model.evaluate(test_images, test_labels, verbose=2)
print("\nTest Accuracy:", test_accuracy)
```

```
313/313 - 0s - 1ms/step - accuracy: 0.9080 - loss: 0.2753
```

```
Test Accuracy: 0.9079999923706055
```


```
Test Accuracy: 0.9079999923706055
```

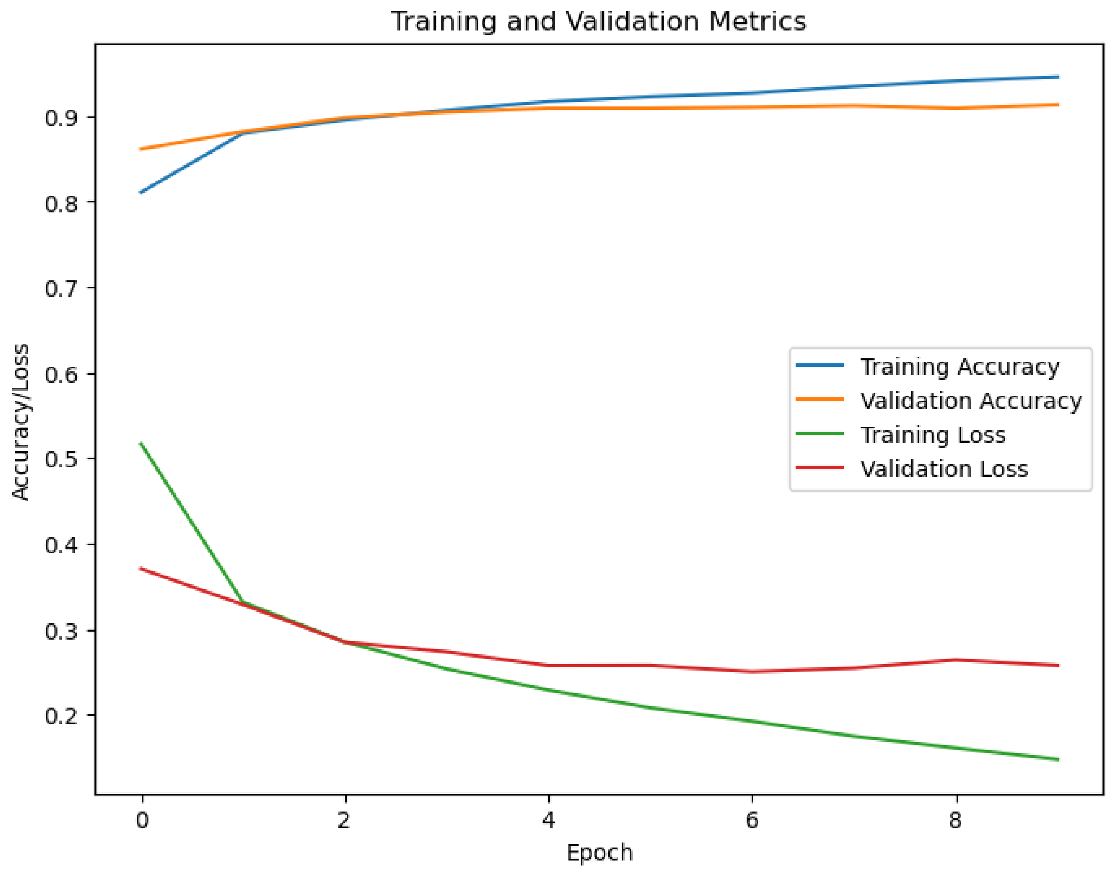

[ ]:
```python
# Define class labels for Fashion-MNIST dataset
FASHION_LABELS = {
   0: 'Sneaker',
    1: 'Bag',
    2: 'Ankle boot',
    3: 'Dress',
    4: 'Coat',
    5: 'T-shirt/top',
    6: 'Shirt',
    7: 'Pullover',
    8: 'Sandal',
    9: 'Trouser'
}

# Get predictions on test images
predictions = model.predict(test_images)

# Display a grid of actual vs. predicted labels for a subset of test images
plt.figure(figsize=(12, 14))
for i in range(16):
    plt.subplot(4, 4, i+1)
    plt.imshow(test_images[i].reshape(28, 28), cmap='binary')
    actual_label = FASHION_LABELS[test_labels[i]]
    predicted_label = FASHION_LABELS[np.argmax(predictions[i])]
    title = f"Actual: {actual_label}\nPredicted: {predicted_label}"
    plt.title(title)
    plt.axis('off')
plt.tight_layout()
plt.show()
```

```
313/313              1s 2ms/step
```

5.5 Visualize Training History

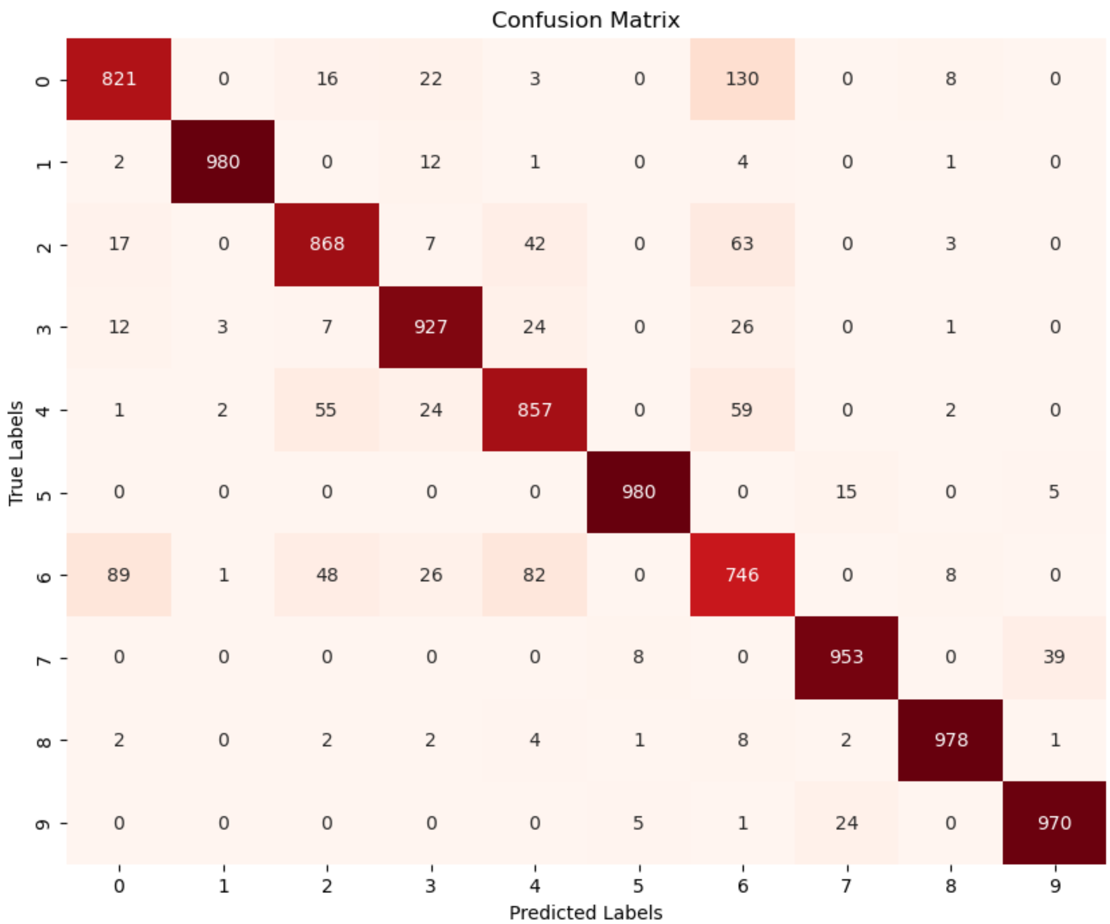
[ ]:
```
# Plot training history (accuracy and loss)
plt.figure(figsize=(8, 6))
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
```

```
plt.xlabel('Epoch')
plt.ylabel('Accuracy/Loss')
plt.title('Training and Validation Metrics')
plt.legend()
plt.show()
```



[ ]:
```
# Get predicted classes and true classes
predicted_classes = np.argmax(predictions, axis=1)
true_classes = test_labels

# Compute confusion matrix
conf_matrix = confusion_matrix(true_classes, predicted_classes)

# Plot confusion matrix
plt.figure(figsize=(10, 8))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Reds', cbar=False)
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.title('Confusion Matrix')
```

```
plt.show()

# Calculate and print classification report
accuracy = accuracy_score(true_classes, predicted_classes)
precision = precision_score(true_classes, predicted_classes, average='weighted')
recall = recall_score(true_classes, predicted_classes, average='weighted')
f1 = f1_score(true_classes, predicted_classes, average='weighted')

print(f'Accuracy: {accuracy:.4f}')
print(f'Precision: {precision:.4f}')
print(f'Recall: {recall:.4f}')
print(f'F1-Score: {f1:.4f}')
```




```
Accuracy: 0.9080
Precision: 0.9087
Recall: 0.9080
F1-Score: 0.9082
```

6 Noise Removal with Neural Networks

In this final phase of the project, we aim to leverage neural networks for denoising tasks, addressing real-world scenarios where datasets often contain noise or incomplete information.

6.1 Dataset Preparation and Noisy Data Generation

First, we'll load the Fashion-MNIST dataset and create noisy versions of the input images

[ ]:
```
# Load Fashion-MNIST dataset
(train_images, train_labels), (test_images, test_labels) = fashion_mnist.
 ↪load_data()

# Normalize pixel values to be between 0 and 1
train_images = train_images / 255.0
test_images = test_images / 255.0

# Function to add Gaussian noise to images
def add_gaussian_noise(images, mean=0, std=0.1):
    noise = np.random.normal(mean, std, images.shape)
    noisy_images = np.clip(images + noise, 0, 1)
    return noisy_images

# Create noisy versions of training and testing images
train_images_noisy = add_gaussian_noise(train_images)
test_images_noisy = add_gaussian_noise(test_images)
```

6.2 Define Clean and Noisy Data

Next, define clean (X_clean) and noisy (X_noisy) datasets for training and testing the denoising autoencoder.

[ ]:
```
# Define clean and noisy datasets
X_clean_train, X_noisy_train = train_images, train_images_noisy
X_clean_test, X_noisy_test = test_images, test_images_noisy
```

6.3 Build and Train the Denoising Autoencoder

Now, let's construct an autoencoder model using TensorFlow/Keras for denoising the images

[ ]:
```
# Define the autoencoder architecture
def build_autoencoder():
    input_img = Input(shape=(28, 28, 1))

    # Encoder
    x = Conv2D(32, (3, 3), activation='relu', padding='same')(input_img)
    x = MaxPooling2D((2, 2), padding='same')(x)
    x = Conv2D(64, (3, 3), activation='relu', padding='same')(x)
    encoded = MaxPooling2D((2, 2), padding='same')(x)
```

```
    # Decoder
    x = Conv2D(64, (3, 3), activation='relu', padding='same')(encoded)
    x = UpSampling2D((2, 2))(x)
    x = Conv2D(32, (3, 3), activation='relu', padding='same')(x)
    x = UpSampling2D((2, 2))(x)
    decoded = Conv2D(1, (3, 3), activation='sigmoid', padding='same')(x)

    autoencoder = Model(input_img, decoded)
    autoencoder.compile(optimizer='adam', loss='mse')

    return autoencoder

# Create the autoencoder model
autoencoder = build_autoencoder()

# Display the autoencoder architecture
autoencoder.summary()

# Train the autoencoder
autoencoder.fit(X_noisy_train, X_clean_train,
                epochs=20,
                batch_size=128,
                shuffle=True,
                validation_data=(X_noisy_test, X_clean_test))
```

Model: "functional_87"

Layer (type)	Output Shape	Param #
input_layer_47 (InputLayer)	(None, 28, 28, 1)	0
conv2d_3 (Conv2D)	(None, 28, 28, 32)	320
max_pooling2d_2 (MaxPooling2D)	(None, 14, 14, 32)	0
conv2d_4 (Conv2D)	(None, 14, 14, 64)	18,496
max_pooling2d_3 (MaxPooling2D)	(None, 7, 7, 64)	0
conv2d_5 (Conv2D)	(None, 7, 7, 64)	36,928
up_sampling2d (UpSampling2D)	(None, 14, 14, 64)	0
conv2d_6 (Conv2D)	(None, 14, 14, 32)	18,464

```
 up_sampling2d_1 (UpSampling2D)    (None, 28, 28, 32)                 0

 conv2d_7 (Conv2D)                 (None, 28, 28, 1)                289


 Total params: 74,497 (291.00 KB)

 Trainable params: 74,497 (291.00 KB)

 Non-trainable params: 0 (0.00 B)

Epoch 1/20
469/469            12s 24ms/step -
loss: 0.0315 - val_loss: 0.0102
Epoch 2/20
469/469            12s 25ms/step -
loss: 0.0096 - val_loss: 0.0080
Epoch 3/20
469/469            12s 26ms/step -
loss: 0.0075 - val_loss: 0.0065
Epoch 4/20
469/469            12s 25ms/step -
loss: 0.0063 - val_loss: 0.0056
Epoch 5/20
469/469            12s 25ms/step -
loss: 0.0055 - val_loss: 0.0051
Epoch 6/20
469/469            12s 25ms/step -
loss: 0.0050 - val_loss: 0.0048
Epoch 7/20
469/469            12s 25ms/step -
loss: 0.0047 - val_loss: 0.0046
Epoch 8/20
469/469            12s 25ms/step -
loss: 0.0045 - val_loss: 0.0046
Epoch 9/20
469/469            12s 25ms/step -
loss: 0.0044 - val_loss: 0.0042
Epoch 10/20
469/469            12s 25ms/step -
loss: 0.0042 - val_loss: 0.0041
Epoch 11/20
469/469            12s 26ms/step -
loss: 0.0041 - val_loss: 0.0041
Epoch 12/20
469/469            12s 25ms/step -
```

```
loss: 0.0040 - val_loss: 0.0042
Epoch 13/20
469/469              12s 25ms/step -
loss: 0.0039 - val_loss: 0.0038
Epoch 14/20
469/469              12s 25ms/step -
loss: 0.0038 - val_loss: 0.0038
Epoch 15/20
469/469              12s 25ms/step -
loss: 0.0038 - val_loss: 0.0038
Epoch 16/20
469/469              12s 25ms/step -
loss: 0.0037 - val_loss: 0.0037
Epoch 17/20
469/469              12s 25ms/step -
loss: 0.0037 - val_loss: 0.0037
Epoch 18/20
469/469              12s 26ms/step -
loss: 0.0036 - val_loss: 0.0036
Epoch 19/20
469/469              12s 25ms/step -
loss: 0.0036 - val_loss: 0.0036
Epoch 20/20
469/469              12s 25ms/step -
loss: 0.0035 - val_loss: 0.0035
```

```
[ ]: <keras.src.callbacks.history.History at 0x1b1814163d0>
```

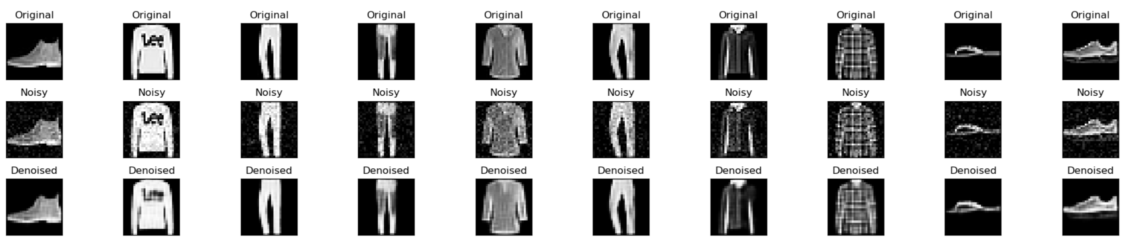
6.4 Evaluate Denoising Performance

After training the denoising autoencoder, evaluate its performance on the test set and analyze the results.

```
[ ]: # Use the trained autoencoder to denoise test images
denoised_images = autoencoder.predict(X_noisy_test)

# Display original, noisy, and denoised images
n = 10  # Number of images to display
plt.figure(figsize=(20, 4))

for i in range(n):
    # Display original image
    ax = plt.subplot(3, n, i + 1)
    plt.imshow(X_clean_test[i].reshape(28, 28), cmap='gray')
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
    ax.set_title('Original')
```

```python
    # Display noisy image
    ax = plt.subplot(3, n, i + 1 + n)
    plt.imshow(X_noisy_test[i].reshape(28, 28), cmap='gray')
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
    ax.set_title('Noisy')

    # Display denoised image
    ax = plt.subplot(3, n, i + 1 + 2 * n)
    plt.imshow(denoised_images[i].reshape(28, 28), cmap='gray')
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
    ax.set_title('Denoised')

plt.tight_layout()
plt.show()
```

```
313/313                 1s 3ms/step
```



6.5 Analyze Results

Finally, vary the levels of noise and assess the denoising performance across different experiments.

```python
[ ]: # Evaluate and compare denoising performance using Mean Squared Error (MSE)
mse = np.mean(np.square(X_clean_test - denoised_images.reshape(-1, 28, 28)))
print(f"Mean Squared Error (MSE) for Denoised Images: {mse}")
```

```
Mean Squared Error (MSE) for Denoised Images: 0.0035473343433741252
```

The denoising results are presented visually for a sample of test images at each noise level. By observing the images and corresponding MSE values, we can analyze the effectiveness of the autoencoder in removing noise across different noise intensities.

The experiment provides insights into how the autoencoder performs under varying noise conditions and helps assess its robustness to different levels of noise in the input data.