Neural Network

About me

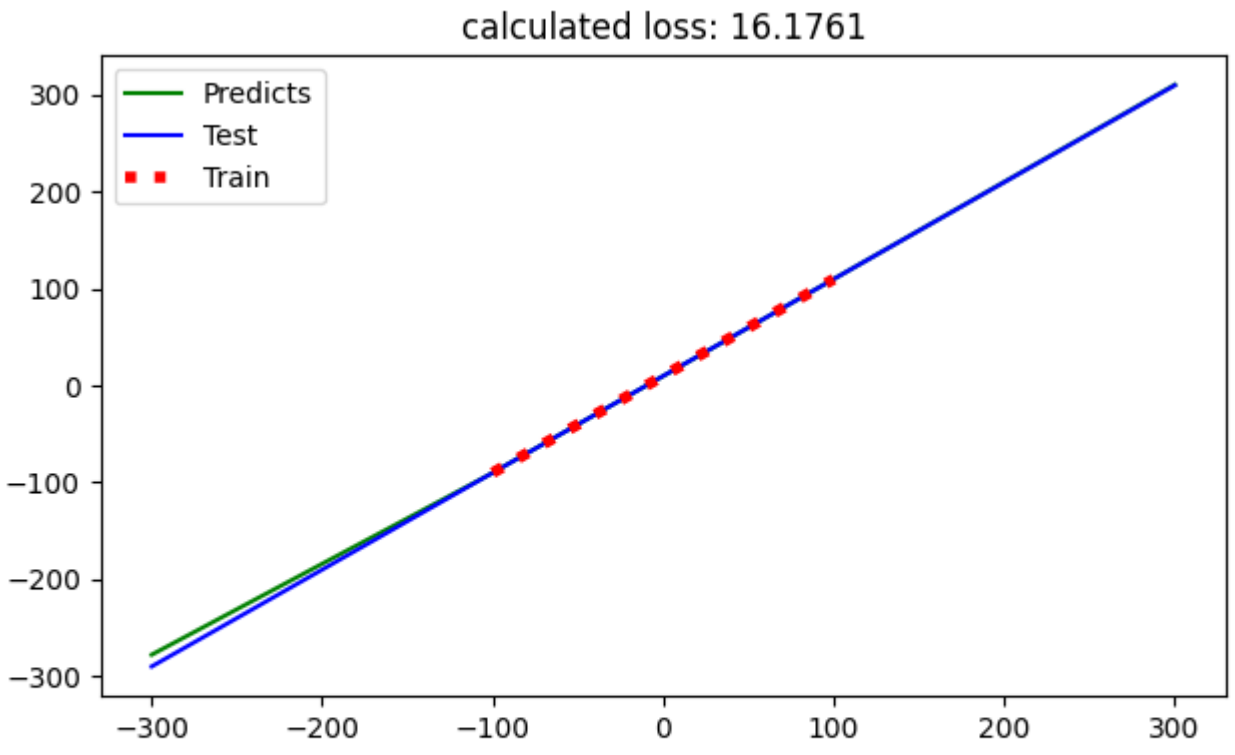
- Full name → Pooria Rahimi
- Student number → 99521289

▼ Question 1

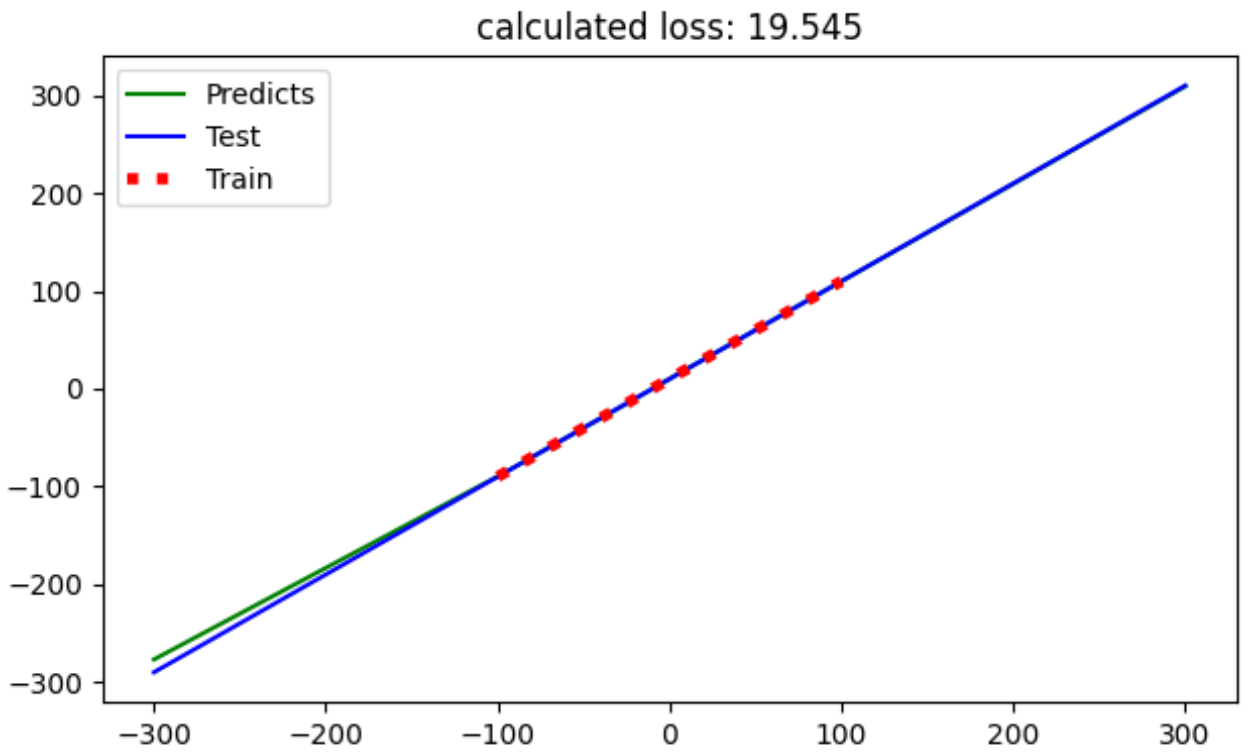
1. Function (x+10) :

> 💡 In this following section i cover results of model in different metrics

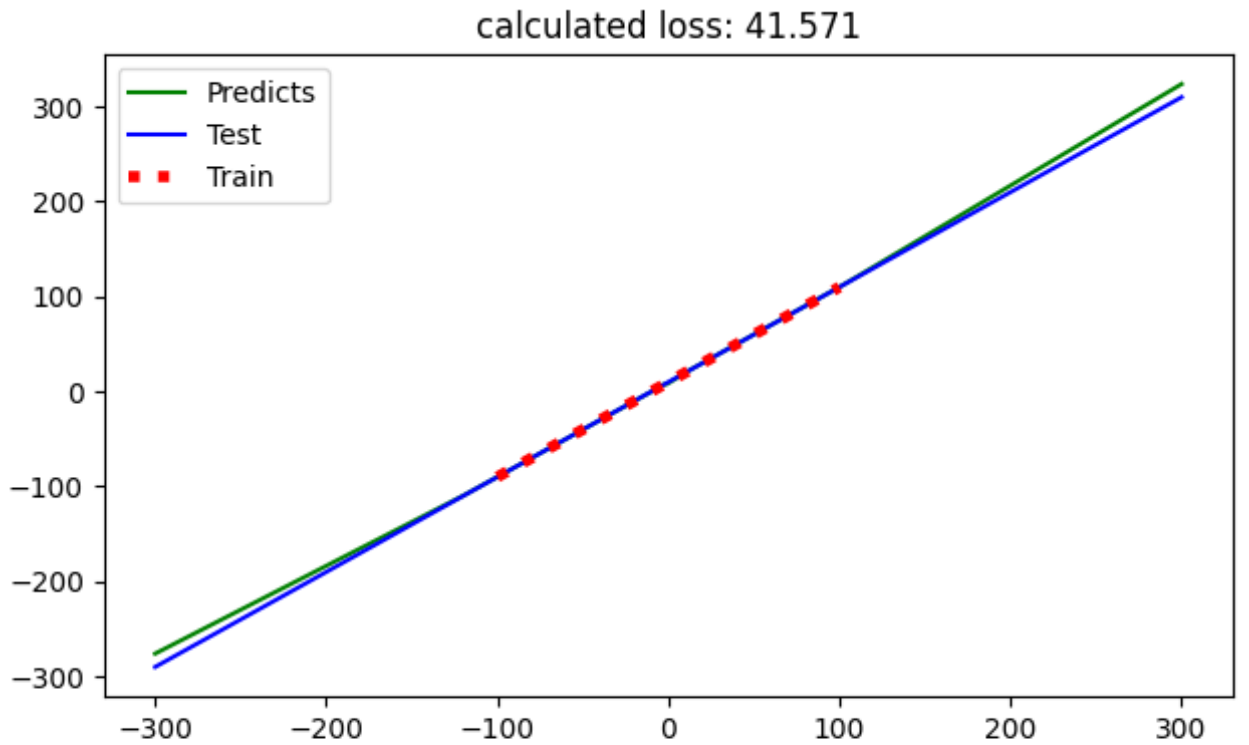
▼ Number of hidden layers



Train Dataset:300 points range from -100 to 100
Test Dataset :600 points range from -300 to 300
Number of Hidden layers : 1
Epochs : 3000



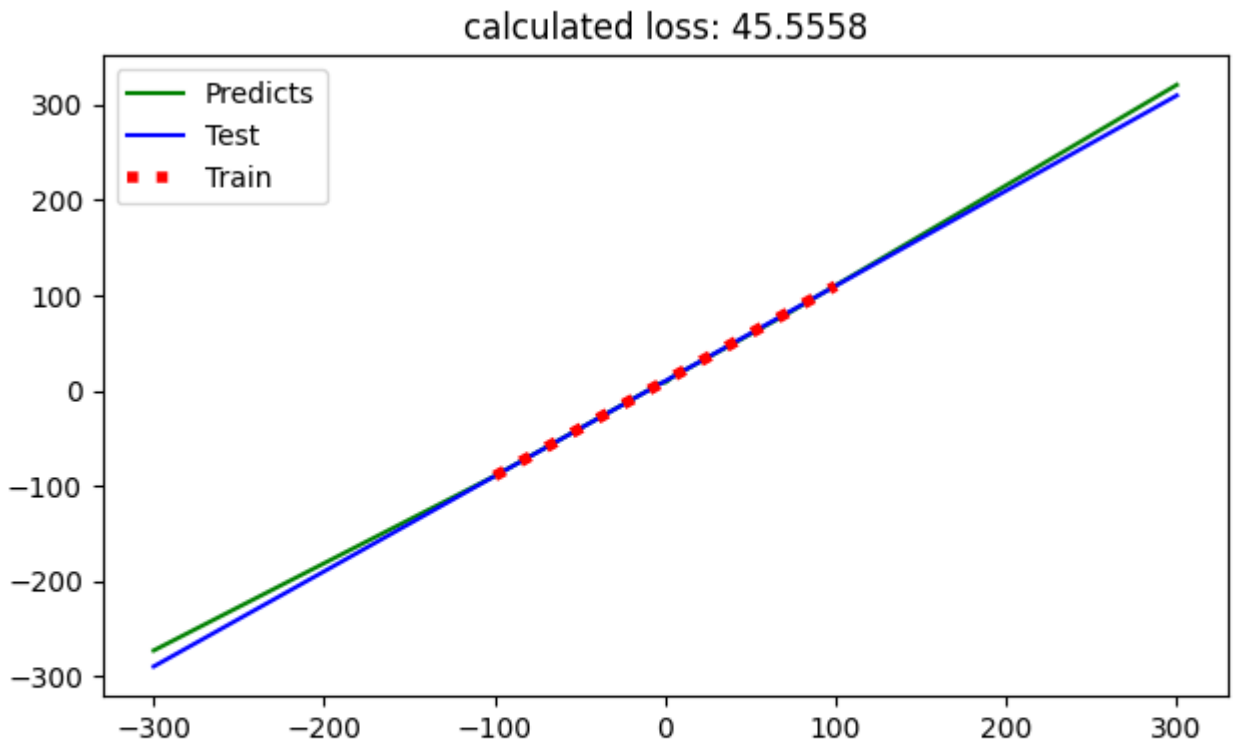
Train Dataset:300 points range from -100 to 100
Test Dataset :600 points range from -300 to 300
Number of Hidden layers : 2
Epochs : 3000



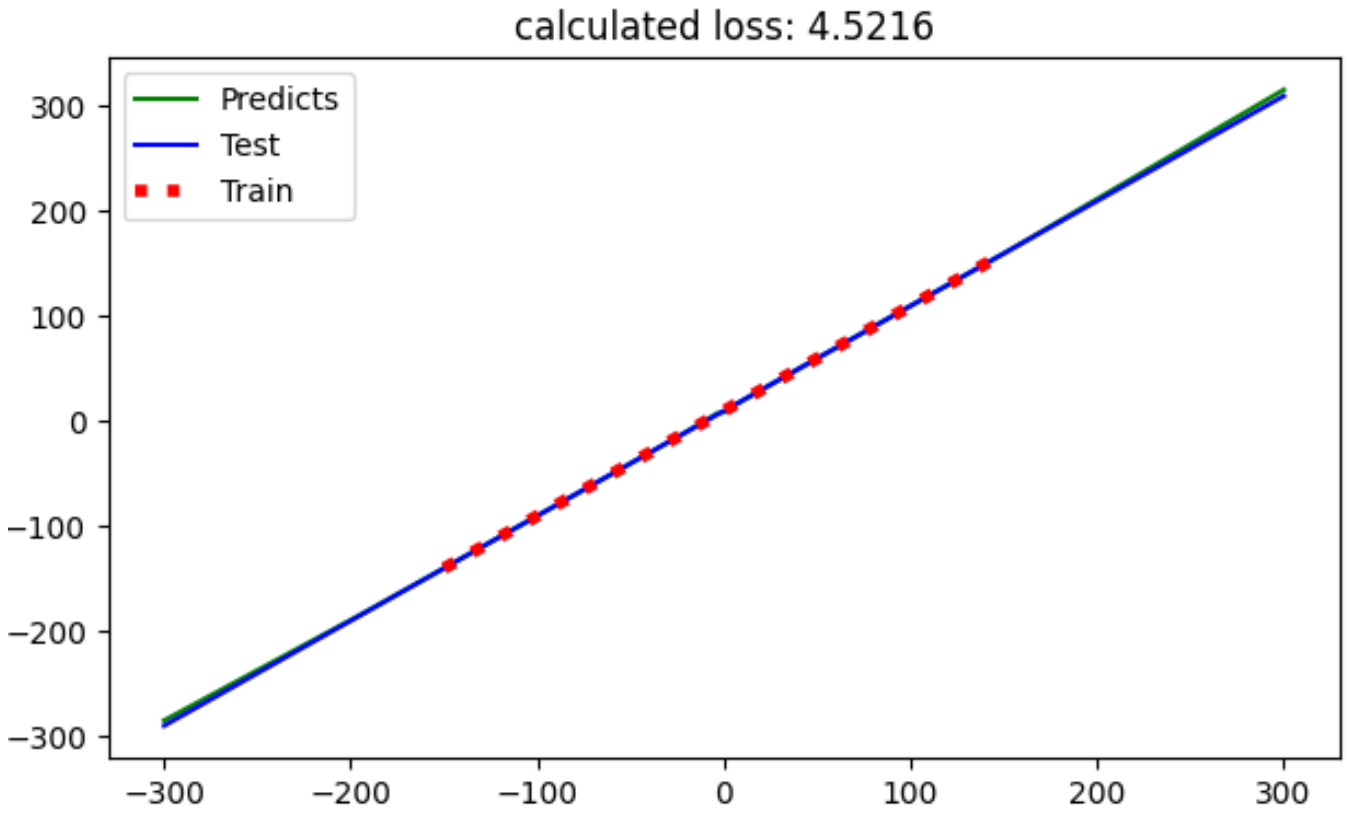
Train Dataset:300 points range from -100 to 100
Test Dataset :600 points range from -300 to 300
Number of Hidden layers : 3
Epochs : 3000

💡 increasing the number of layers with same configuration will result in higher loss and worsens the productivity of the
model

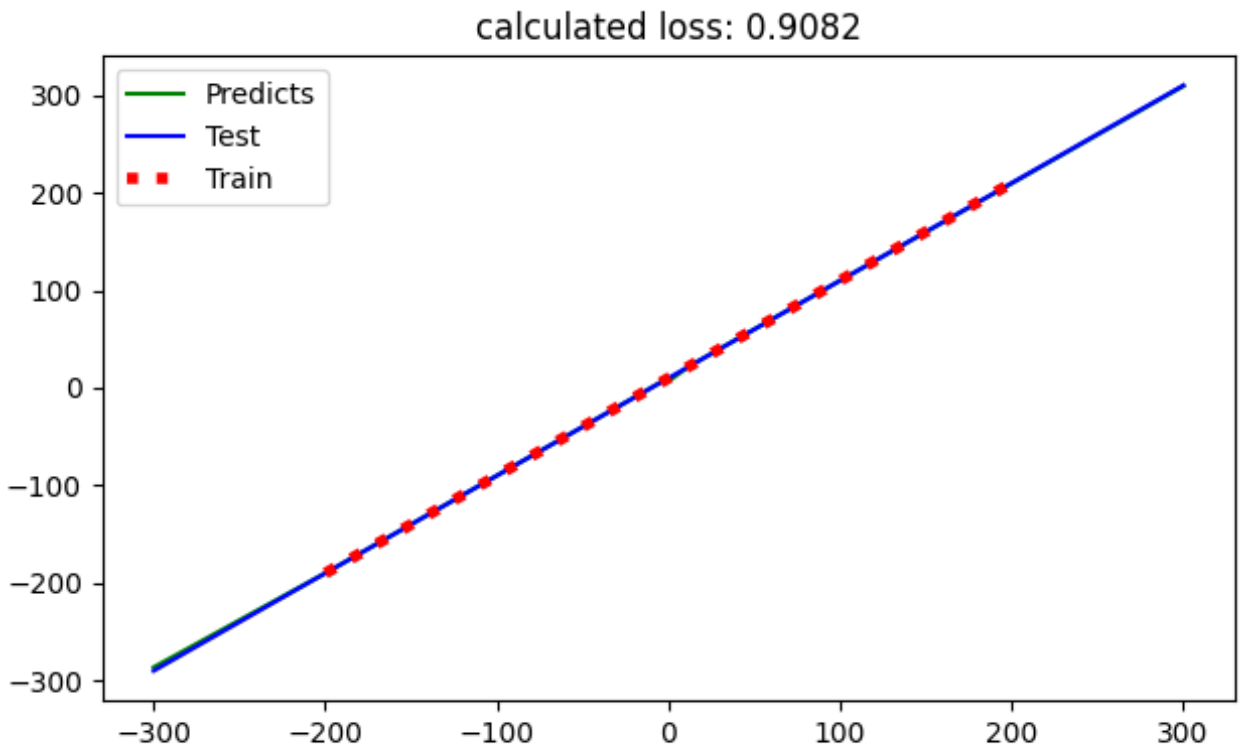
▼ Training range



Train Dataset:200 points range from -100 to 100
Test Dataset :600 points range from -300 to 300
Number of Hidden layers : 3
Epochs : 3000



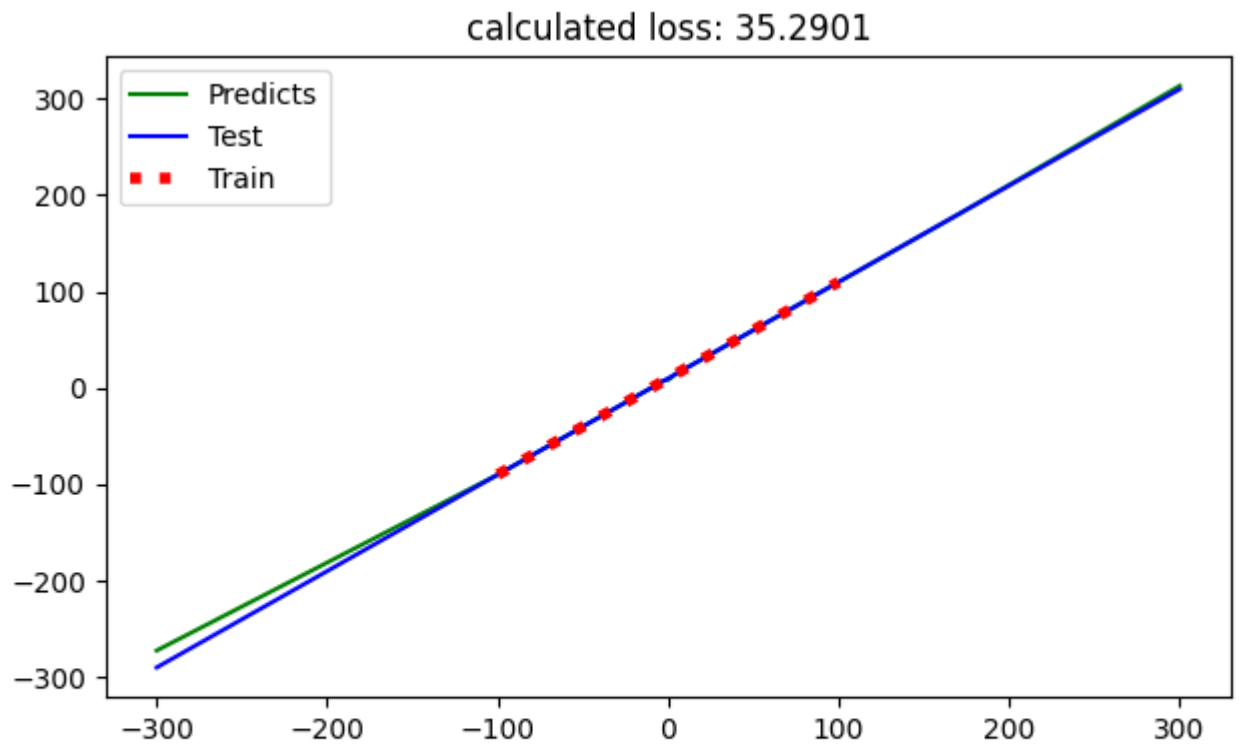
Train Dataset:200 points range from -150 to 150
Test Dataset :600 points range from -300 to 300
Number of Hidden layers : 3
Epochs : 3000



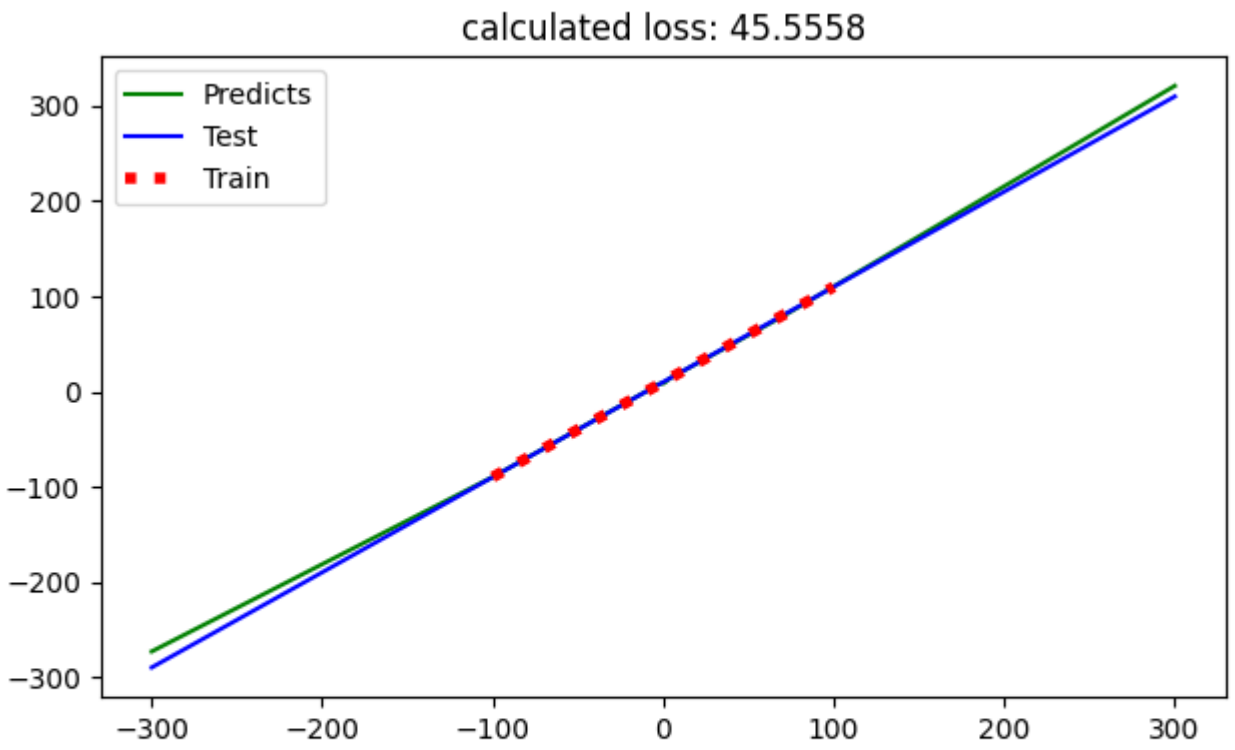
Train Dataset:200 points range from -200 to 200
Test Dataset :600 points range from -300 to 300
Number of Hidden layers : 3
Epochs : 3000

💡 while we increase the training range,the loss will decrease

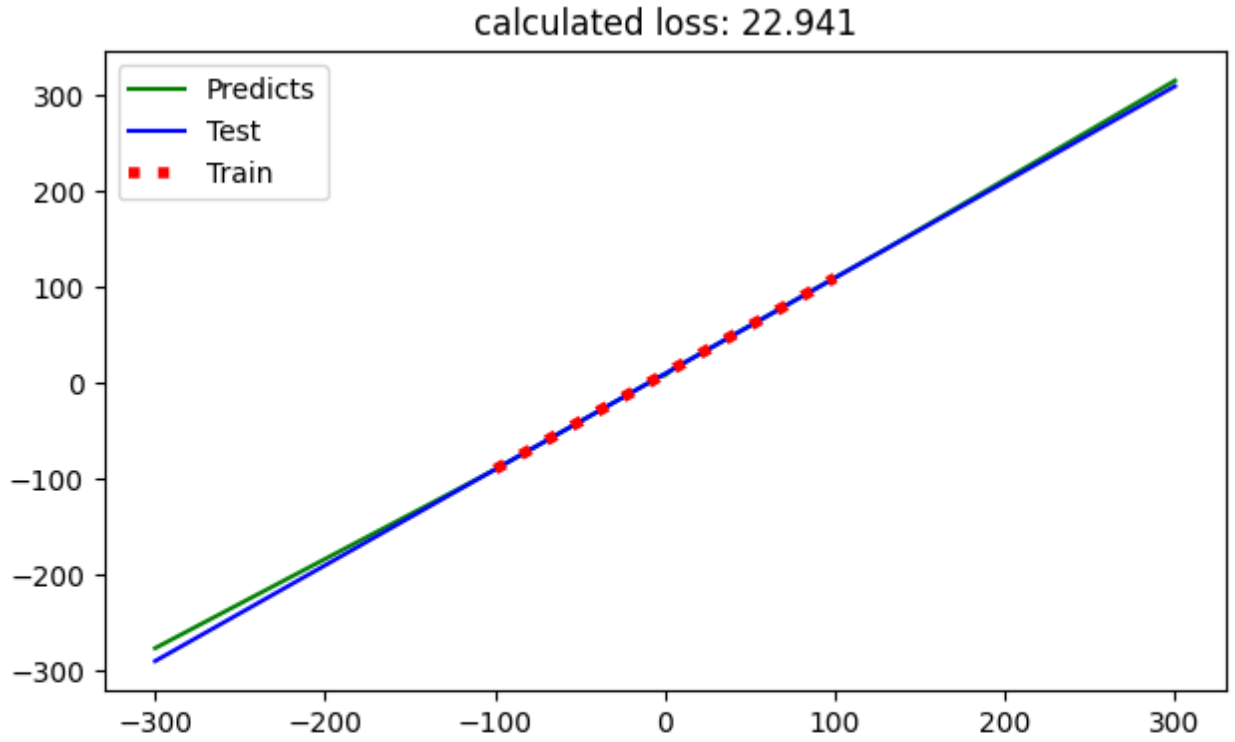
▼ Train Dataset



Train Dataset:150 points range from -100 to 100
Test Dataset :600 points range from -300 to 300
Number of Hidden layers : 3
Epochs : 2000



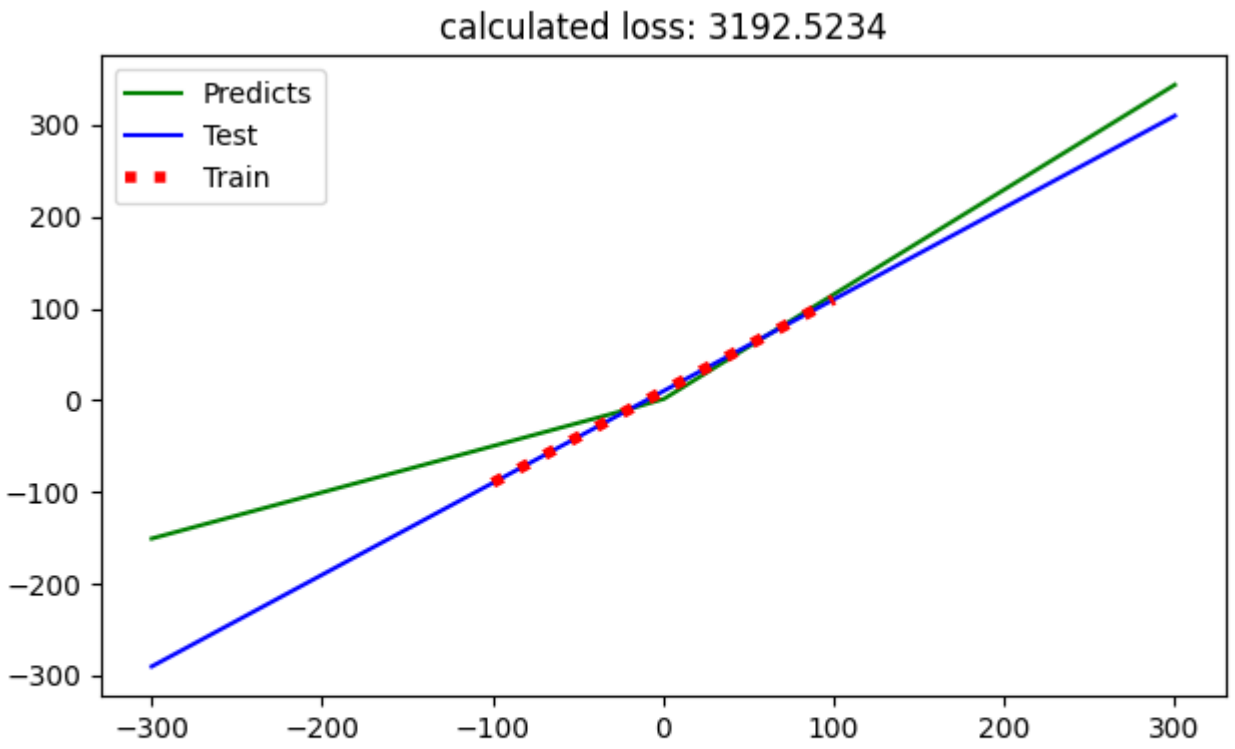
Train Dataset:200 points range from -100 to 100
Test Dataset :600 points range from -300 to 300
Number of Hidden layers : 3
Epochs : 2000



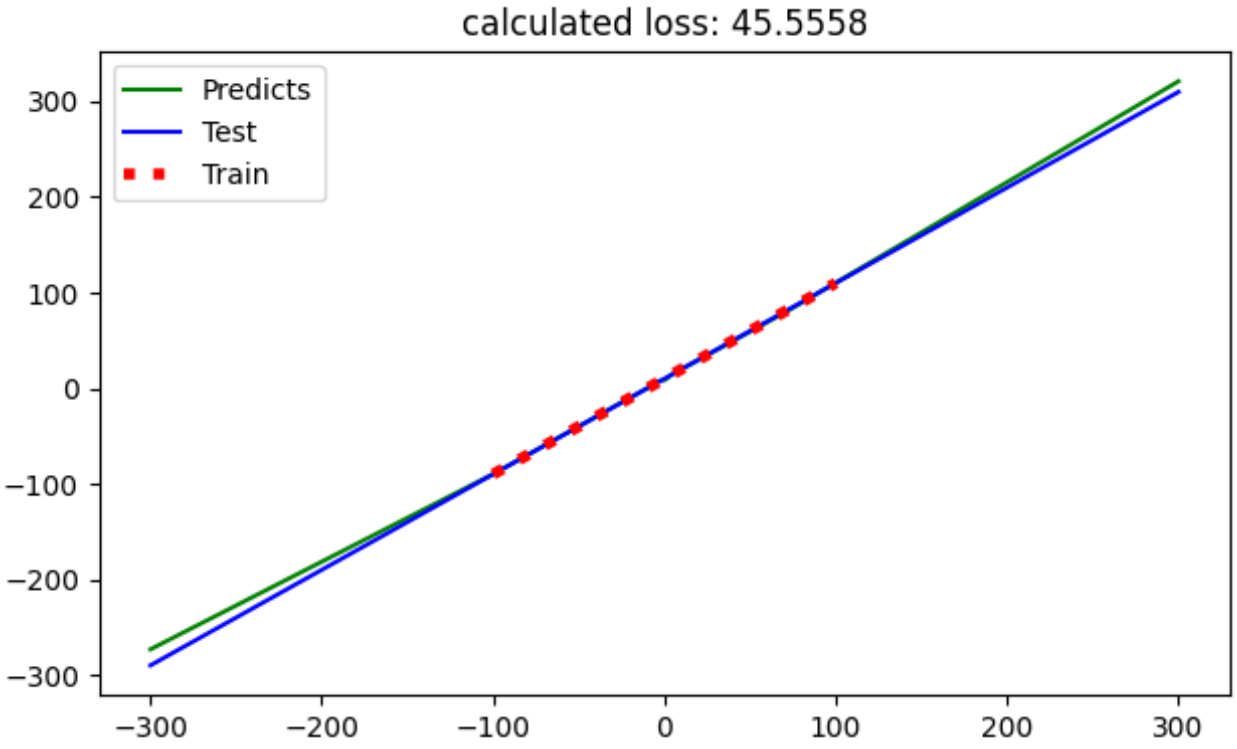
Train Dataset:250 points range from -100 to 100
Test Dataset :600 points range from -300 to 300
Number of Hidden layers : 3
Epochs : 2000

💡 increasing the number of training inputs will result in better performance

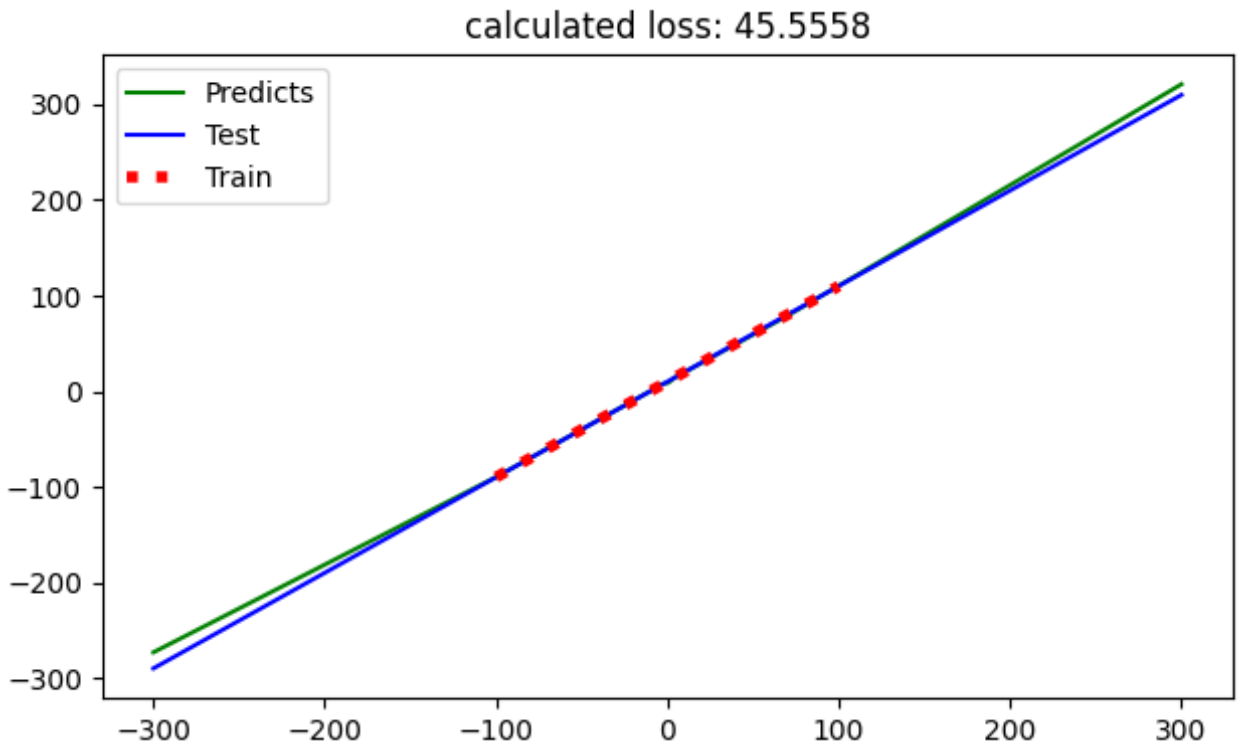
▼ Epochs



Train Dataset:200 points range from -100 to 100
Test Dataset :600 points range from -300 to 300
Number of Hidden layers : 3
Epochs : 100



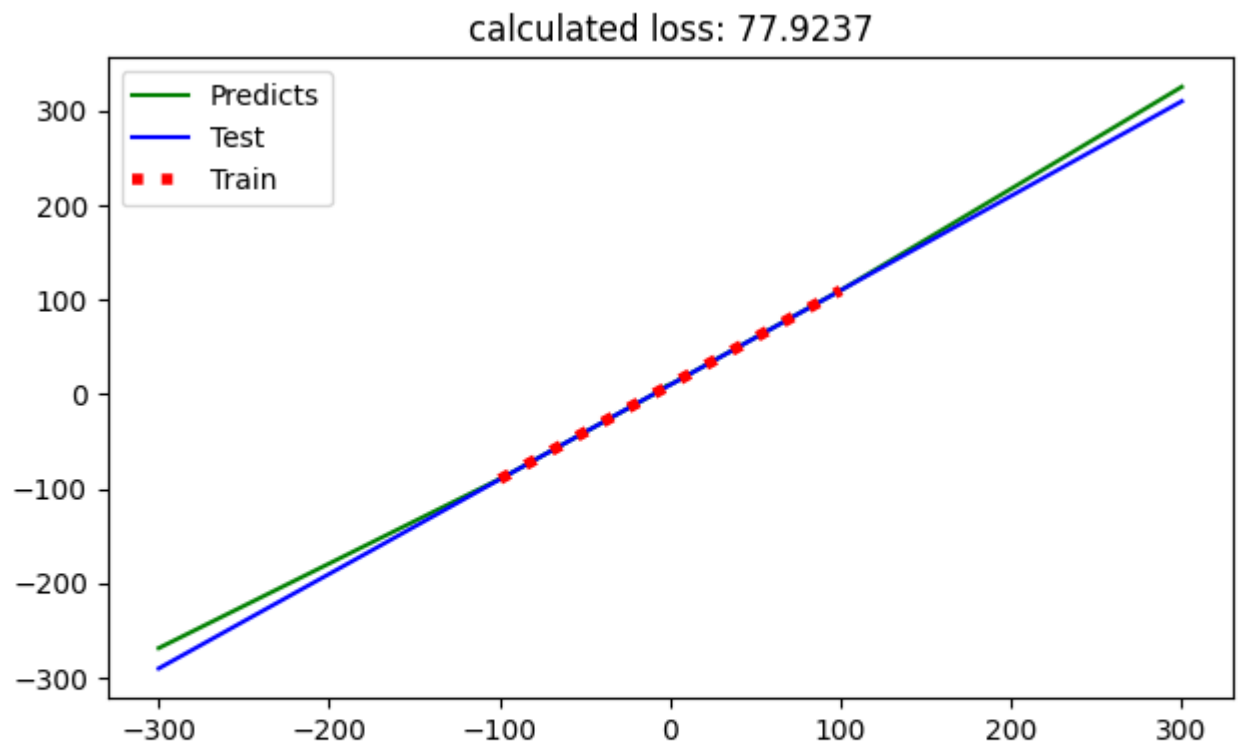
Train Dataset:200 points range from -100 to 100
Test Dataset :600 points range from -300 to 300
Number of Hidden layers : 3
Epochs : 2000



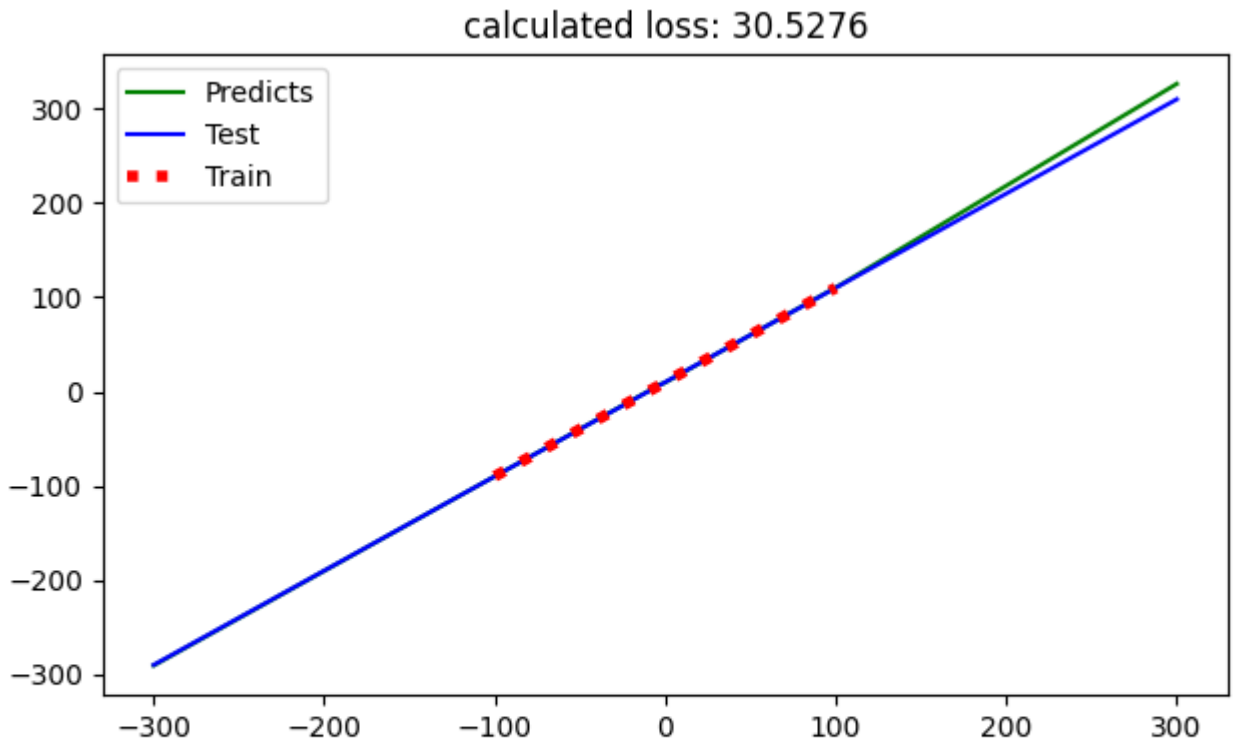
Train Dataset:200 points range from -100 to 100
Test Dataset :600 points range from -300 to 300
Number of Hidden layers : 3
Epochs : 2500

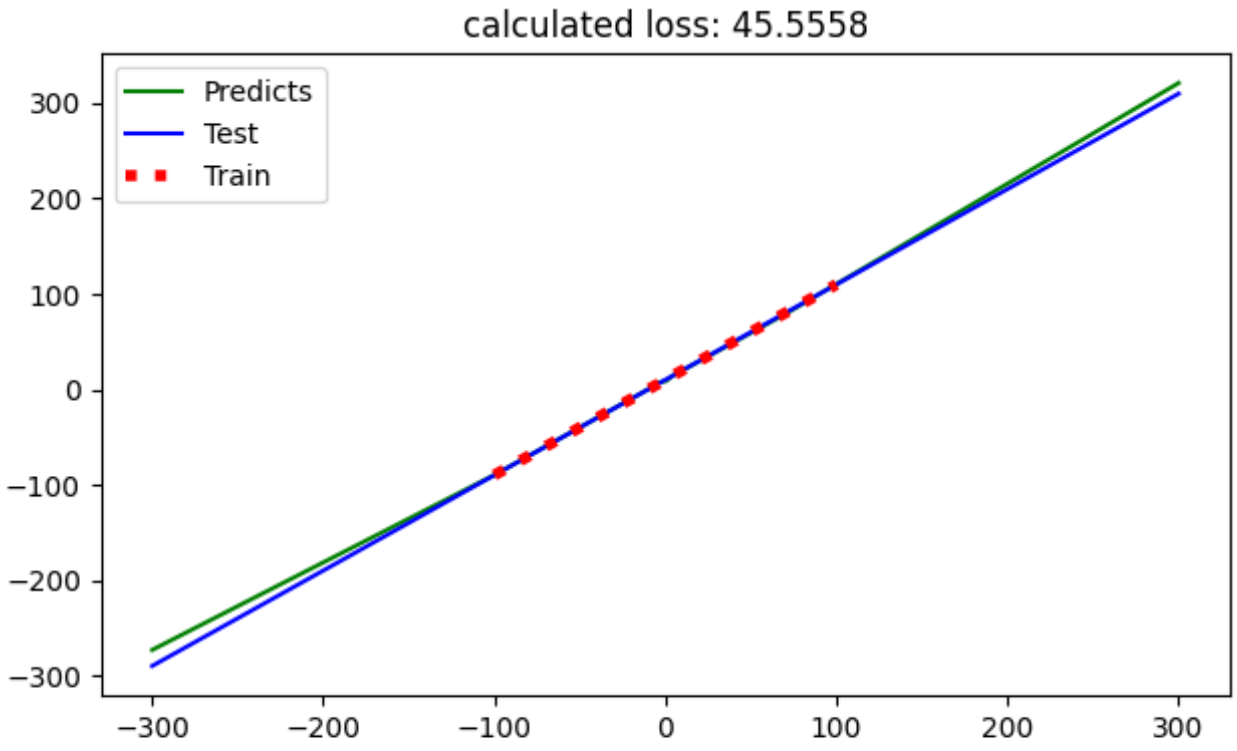
💡 as we see the number of iterations is not going to make any big difference in the result if the model meets the required
number of epochs for convergence

▼ Number of neurons



Train Dataset:200 points range from -100 to 100
Test Dataset :600 points range from -300 to 300
Number of Hidden layers : 3
Epochs : 2500

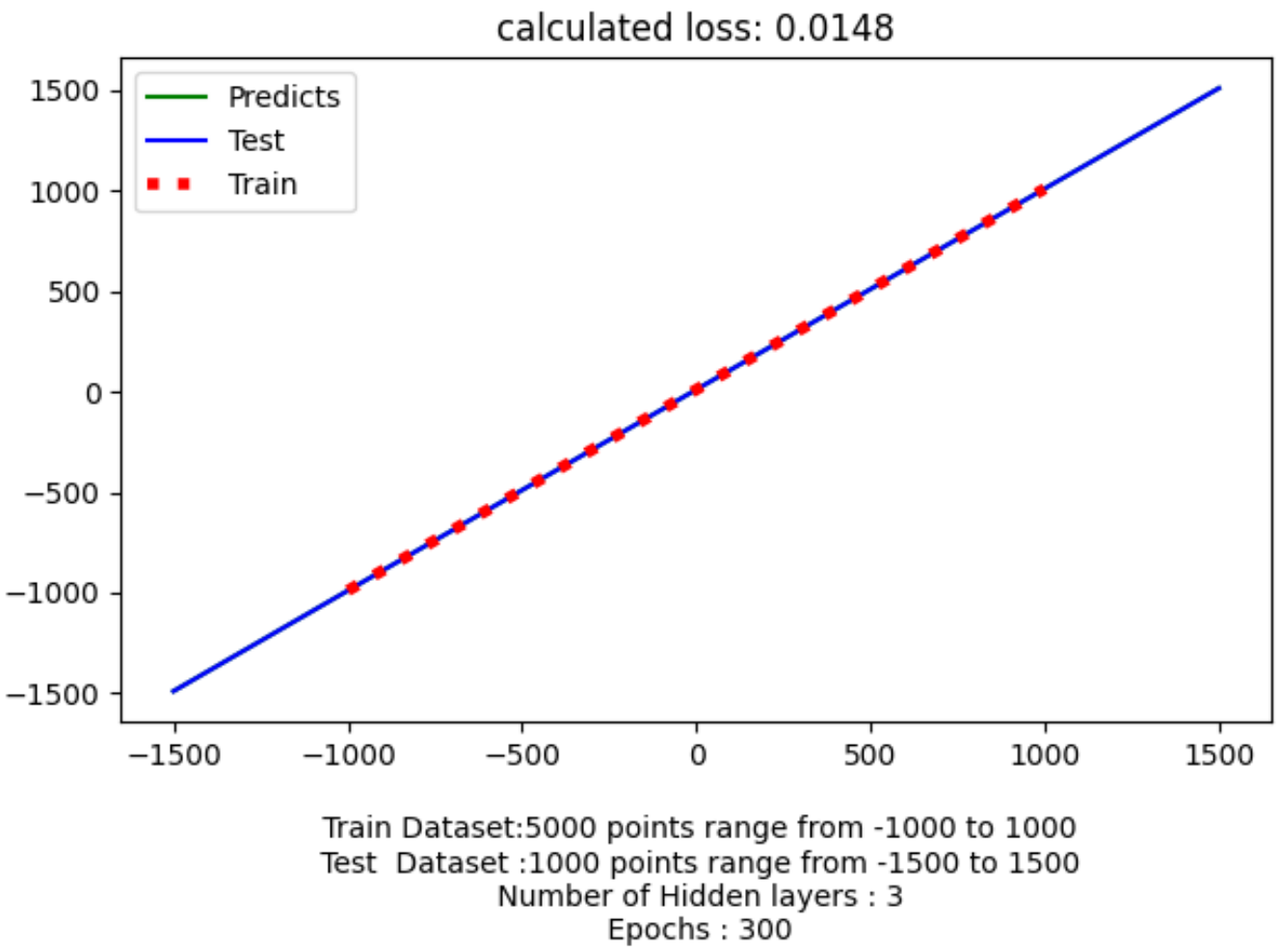




💡 as we see changing the number of neurons is something experimental and different strategies need to be checked for
optimal solution

Here is the most optimal neural network i found for this function

```
# three hidden layers with 30 , 20 , 10 neurons
# other configurations are mentioned in picture itself
Run((30,20,10),300,(5000,-1000,1000),(1000,-1500,1500))
```



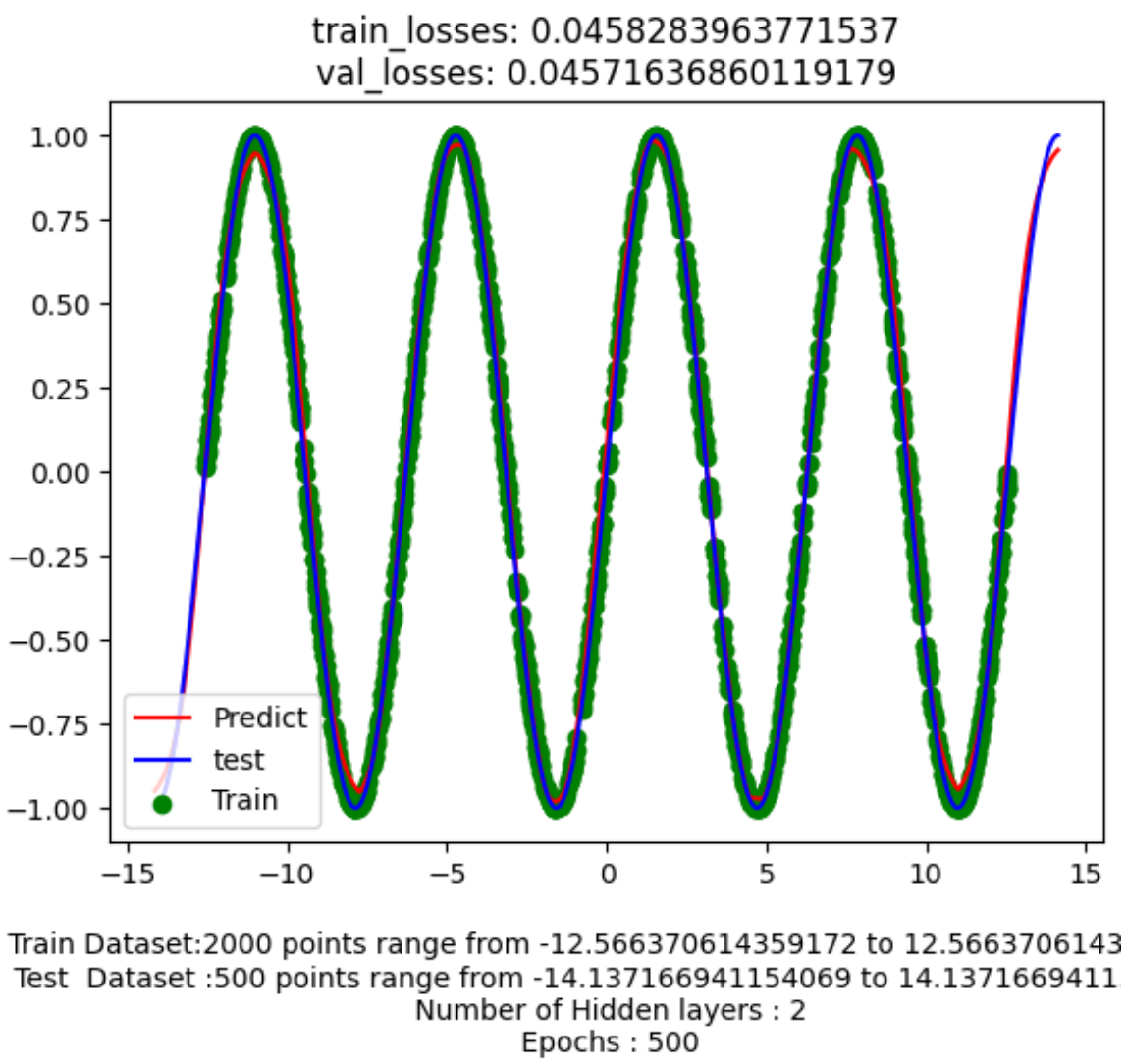
2. Function sin(t) :

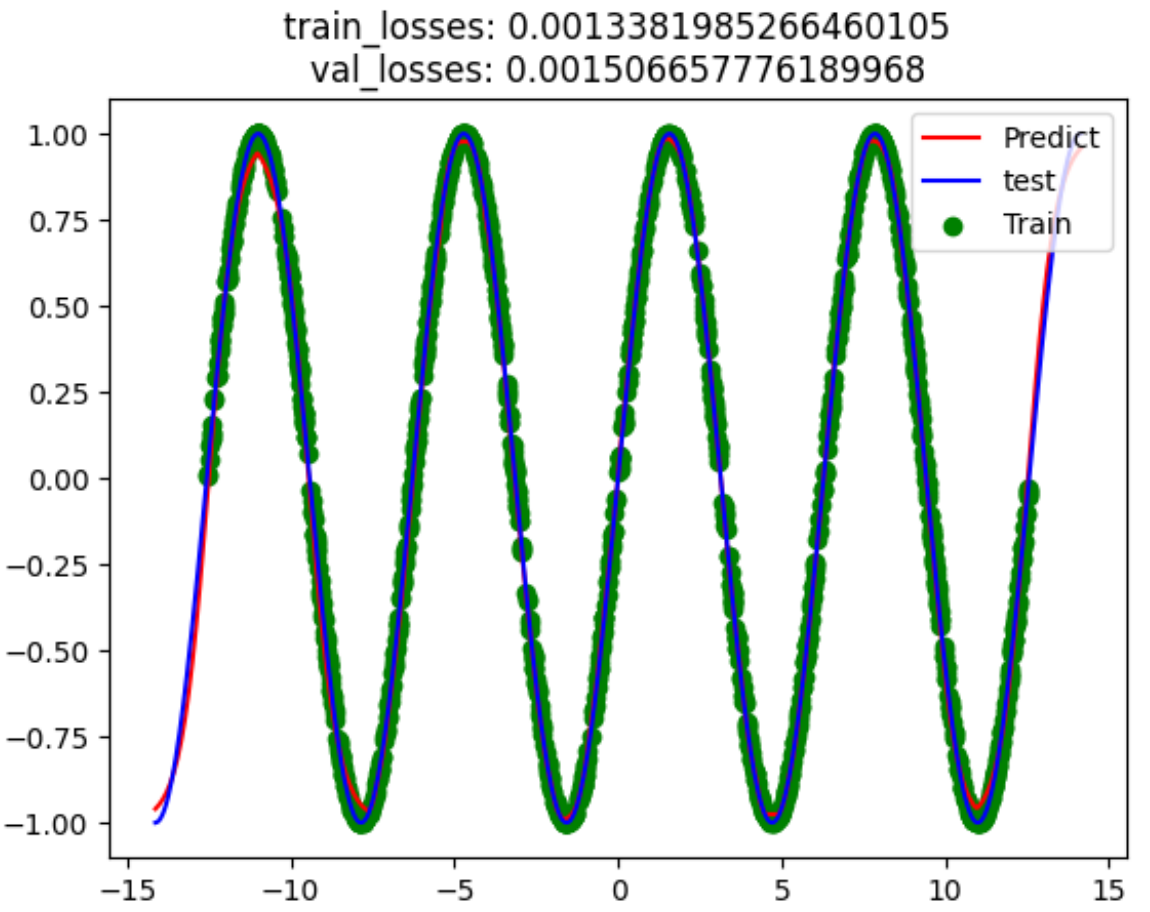
> here is the model and cross-validation implementation

```
def Run(hidden_layer, num_of_iteration, train_info, test_info):
    """
    Run training and evaluation process.

    Args:
    hidden_layer (list): List containing number of neurons in hidden layers.
    num_of_iteration (int): Number of training iterations.
    train_info (list): List containing information about training data.
    test_info (list): List containing information about test data.

    Returns:
    None
    """
    x_input, y_input = correct_input_output(
        train_info[0], train_info[1], train_info[2])
x_test, y_test = generate_test(
    test_info[0], test_info[1], test_info[2])
model = keras.Sequential()
for h in hidden_layer:
    model.add(layers.Dense(units=h, input_dim=1, activation='relu'))
model.add(layers.Dense(units=1, activation='tanh'))
model.compile(optimizer='adam', loss='mean_squared_error')
kfold = KFold(n_splits=4, shuffle=True, random_state=0)

# Perform k-fold cross-validation
for train_index, test_index in kfold.split(x_input):
    X_train, X_val = x_input[train_index], x_input[test_index]
    y_train, y_val = y_input[train_index], y_input[test_index]
    history = model.fit(X_train, y_train, epochs=num_of_iteration,batch_size=32, verbo
    predictions = model.predict(x_test)
    train_losses = np.mean(history.history['loss'])
    val_losses = np.mean(history.history['val_loss'])
```

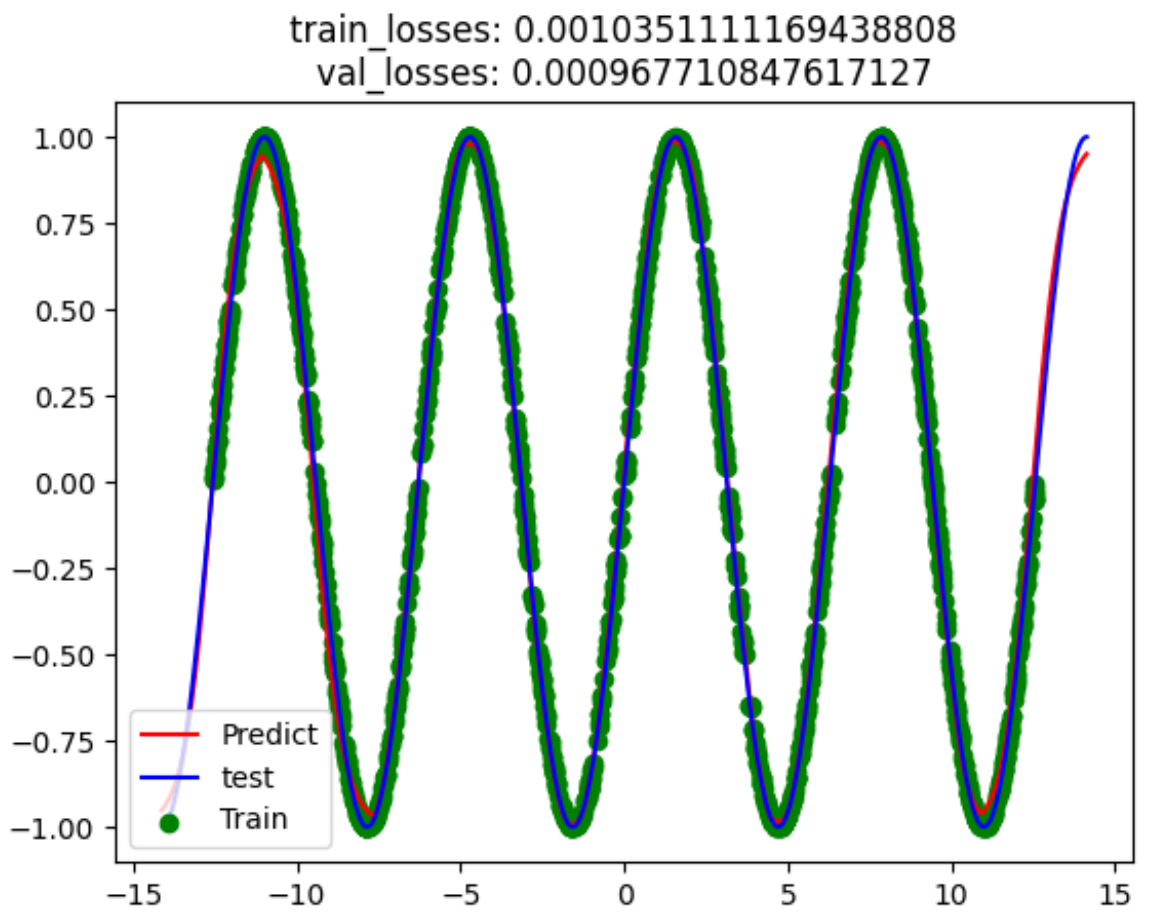
```
        fig, ax = plt.subplots()
        plt.title('train_losses: ' +
                  str(train_losses) +
                  '\nval_losses: ' +
                  str(val_losses))
        plt.plot(x_test, predictions, color='r', label='Predict')
        plt.plot(x_test, y_test, color='b', label='test')
        plt.scatter(X_train, y_train, label='Train', color='g')
        ax.set_xlabel('\n Train Dataset:'+str(train_info[0]) +
                      ' points range from ' + str(train_info[1]) +
                      ' to '+str(train_info[2]) +
                      '\n Test  Dataset :'+str(test_info[0]) +
                      ' points range from ' + str(test_info[1]) +
                      ' to '+str(test_info[2]) +
                      '\n Number of Hidden layers : '+
                      str(len(hidden_layer)) +
                      '\n Epochs : '+
                      str(num_of_iteration))

        plt.legend()
        plt.show()
```

▼ here is the result of the execution

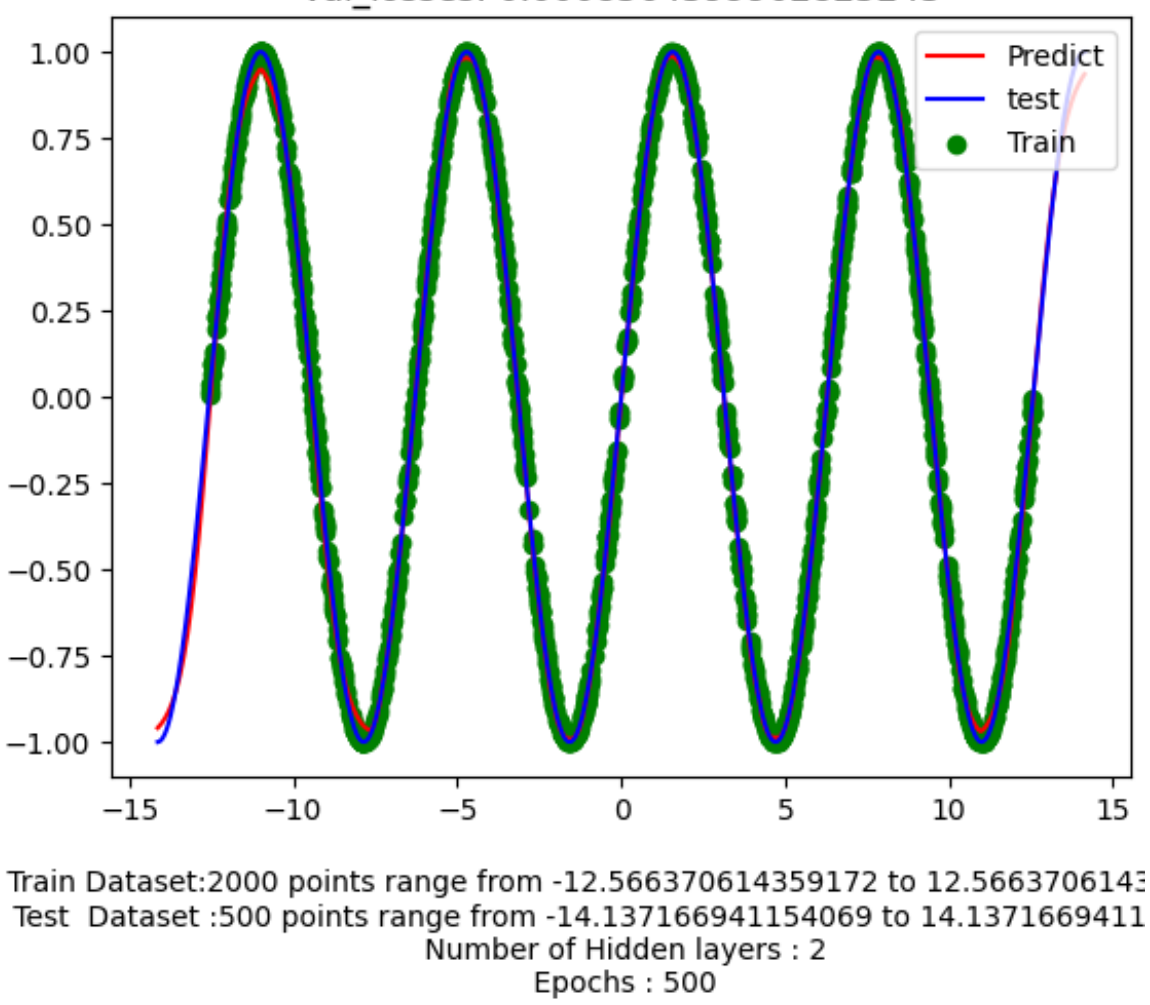




Train Dataset:2000 points range from -12.566370614359172 to 12.5663706143
Test Dataset :500 points range from -14.137166941154069 to 14.1371669411
Number of Hidden layers : 2
Epochs : 500



Train Dataset:2000 points range from -12.566370614359172 to 12.566370614
Test Dataset :500 points range from -14.137166941154069 to 14.1371669411
Number of Hidden layers : 2
Epochs : 500



> as we see in one execution the training loss and the validation loss continuously decreases

💡 as sin is more complex than x+10,we have to use more complex neural network

3. Function (x**3 - 5*x**2 - 9)

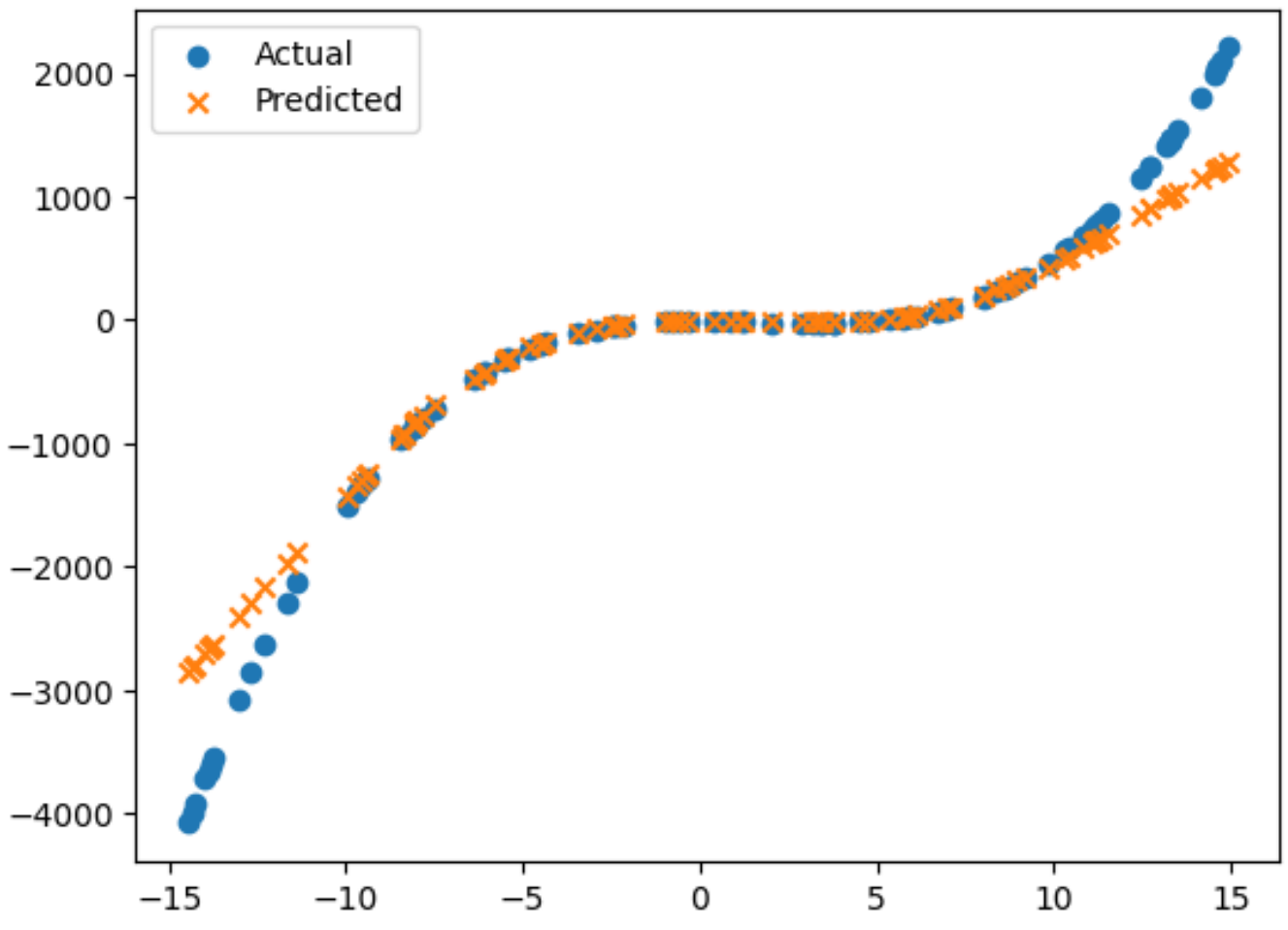
> this function is slightly complex so we need to increase number of neurons and also add some dropout layers to our neural
> network to prevent overfitting.

> here is the model that i designed

```
# Define the neural network model
model = Sequential()  # Create a sequential model
model.add(Dense(128, activation='relu' , input_dim=1))  # Add a fully connected layer with
model.add(Dropout(0.5))  # Add dropout regularization with a dropout rate of 0.5
model.add(Dense(256, activation='relu'))  # Add another fully connected layer with 256 neu
model.add(Dropout(0.5))  # Add dropout regularization with a dropout rate of 0.5
model.add(Dense(128, activation='relu'))  # Add another fully connected layer with 128 neu
model.add(Dense(units=1, activation='linear'))  # Add a fully connected output layer with

# Compile the model with a lower learning rate
opt = Adam(learning_rate=0.001)  # Initialize Adam optimizer with learning rate 0.001
model.compile(optimizer=opt, loss='mean_squared_error')  # Compile the model with Adam opt

# Train the model for more epochs
model.fit(X_train, y_train, epochs=400, batch_size=32, validation_split=0.2)  # Train the
```

```
# Generate test data (out of training range)
X_test = np.random.uniform(-15, 15, 100)  # Generate 100 random points between -15 and 15
y_test = target_function(X_test)  # Calculate target function values for the test points

# Make predictions
predictions = model.predict(X_test)  # Use the trained model to make predictions on the te

# Evaluate the model
mse = model.evaluate(X_test, y_test, verbose=0)  # Calculate mean squared error on the tes
print(f'Mean Squared Error: {mse:.4f}')  # Print the mean squared error

# Plot the actual vs predicted values
plt.scatter(X_test, y_test, label='Actual')  # Plot actual values
plt.scatter(X_test, predictions, label='Predicted', marker='x')  # Plot predicted values
plt.legend()  # Add legend
plt.show()  # Show the plot
```

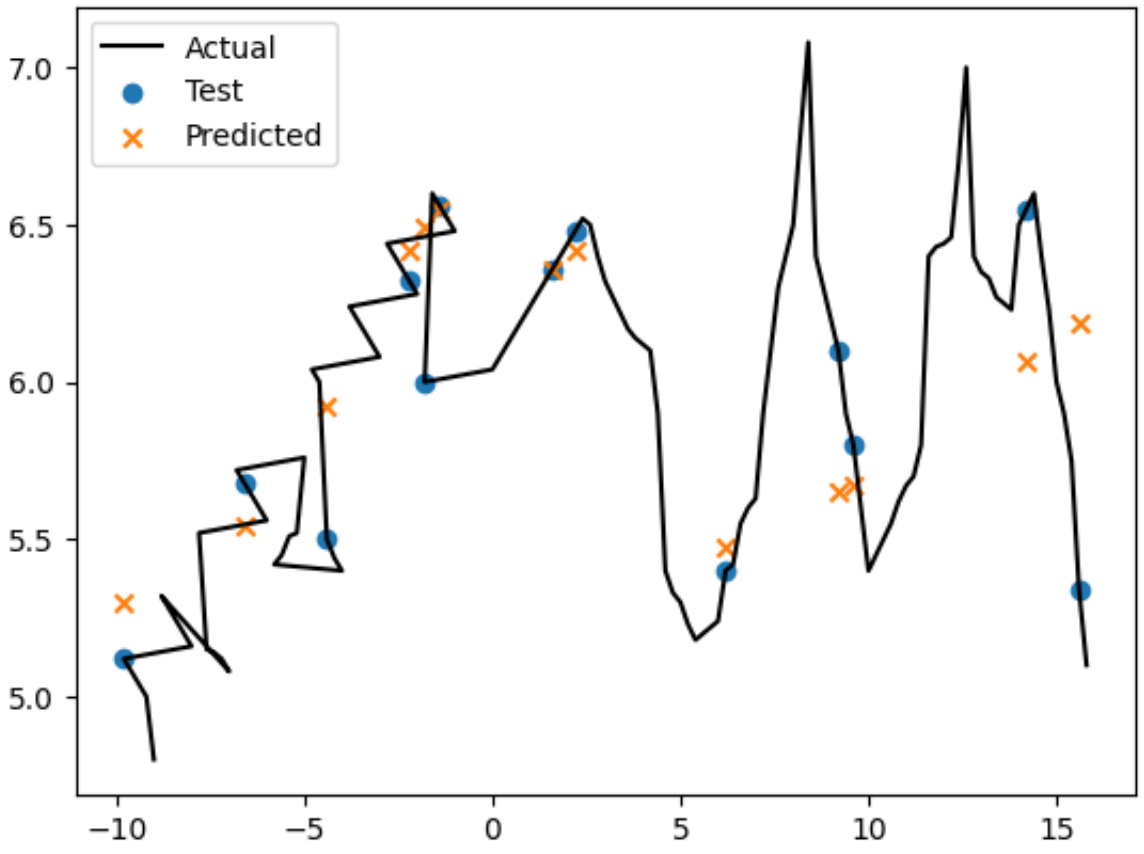
> here is the result of prediction



> 💡 the training range is from -10 to 10 and it includes 1000 point in there. As we see the predictions are perfectly matched during
> training range. Moreover, model works well for points that are out of training range.

Part two :

> plotted result shows everything

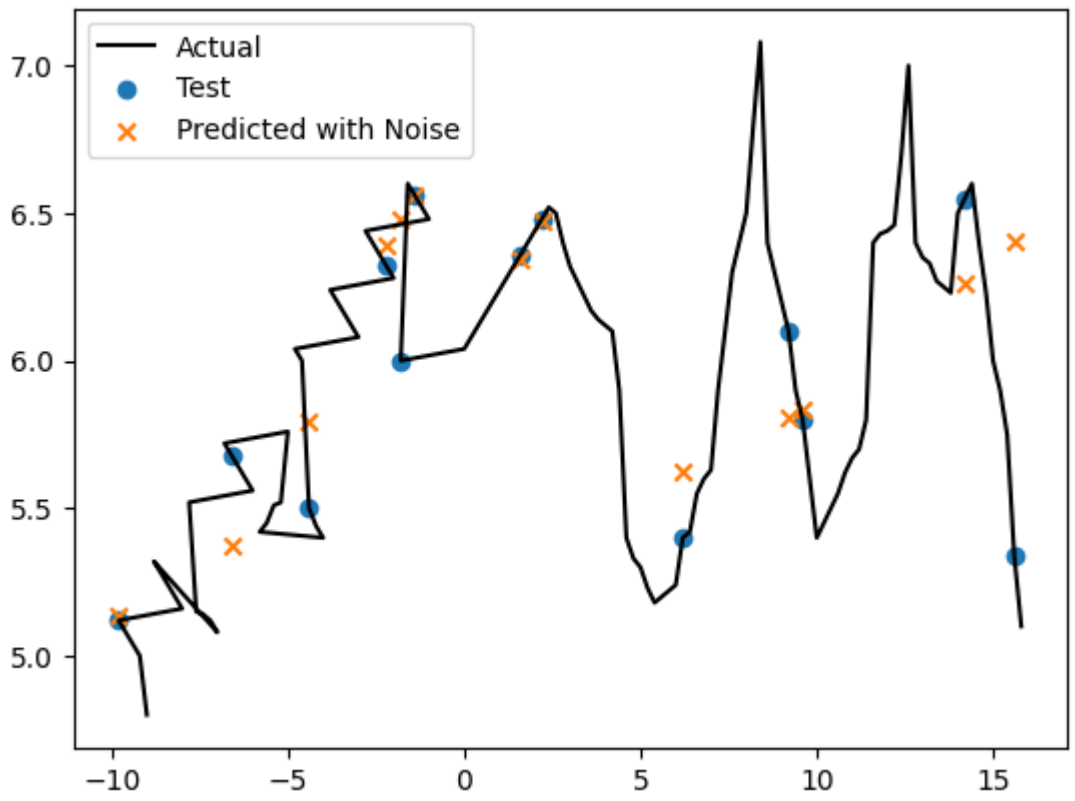


- there are 125 words in total
- 0.1 of total points are used for testing purpose
- obviously it has predicted most of numbers well except the ones that cause burst in function
- here is the preview of written code

```
# Define input and output data
X = np.array([-9.0, -9.2, -9.4, -9.6, -9.8, -8.0, -8.2, -8.4, -8.6, -8.8, -7.0, -7.2,
-7.4, -7.6, -7.8, -6.0, -6.2, -6.4, -6.6, -6.8, -5.0, -5.2, -5.4, -5.6, -5.8, -4.0,
-4.2, -4.4, -4.6, -4.8, -3.0, -3.2, -3.4, -3.6, -3.8, -2.0, -2.2, -2.4, -2.6, -2.8,
-1.0, -1.2, -1.4, -1.6, -1.8, 0.0, 0.2, 0.4, 0.6, 0.8, 1.0, 1.2, 1.4, 1.6, 1.8, 2.0,
2.2, 2.4, 2.6, 2.8, 3.0, 3.2, 3.4, 3.6, 3.8, 4.0, 4.2, 4.4, 4.6, 4.8, 5.0, 5.2, 5.4,
5.6, 5.8, 6.0, 6.2, 6.4, 6.6, 6.8, 7.0, 7.2, 7.4, 7.6, 7.8, 8.0, 8.2, 8.4, 8.6, 8.8,
9.0, 9.2, 9.4, 9.6, 9.8, 10.0, 10.2, 10.4, 10.6, 10.8, 11.0, 11.2, 11.4, 11.6, 11.8,
12.0, 12.2, 12.4, 12.6, 12.8, 13.0, 13.2, 13.4, 13.6, 13.8, 14.0, 14.2, 14.4, 14.6,
14.8, 15.0, 15.2, 15.4, 15.6, 15.8])
y = np.array([4.8, 5.0, 5.04, 5.08, 5.12, 5.16, 5.2, 5.24, 5.28, 5.32, 5.08, 5.12,
 5.14, 5.15, 5.52, 5.56, 5.6, 5.64, 5.68, 5.72, 5.76, 5.52, 5.51, 5.45, 5.42, 5.40,
 5.44, 5.50, 6, 6.04, 6.08, 6.12, 6.16, 6.2, 6.24, 6.28, 6.32, 6.36, 6.4, 6.44, 6.48,
 6.52, 6.56, 6.6, 6.0, 6.04, 6.08, 6.12, 6.16, 6.2, 6.24, 6.28, 6.32, 6.36, 6.4, 6.44,
  6.48, 6.52, 6.50, 6.40, 6.32, 6.27, 6.22, 6.17, 6.14, 6.12, 6.10, 5.9, 5.4, 5.33,
  5.30, 5.23, 5.18, 5.20, 5.22, 5.24, 5.40, 5.42, 5.55, 5.60, 5.63, 5.9, 6.1, 6.3,
  6.4, 6.5, 6.8, 7.08, 6.4, 6.3, 6.2, 6.1, 5.9, 5.8, 5.6, 5.4, 5.45, 5.50, 5.55,
  5.62, 5.67, 5.70, 5.80, 6.4, 6.43, 6.44, 6.46, 6.7, 7.0, 6.4, 6.35, 6.33, 6.27,
  6.25, 6.23, 6.5, 6.55, 6.60, 6.4, 6.23, 6.0, 5.9, 5.75, 5.34, 5.1])


# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.1, random_state=42)

# Define the neural network model
model = Sequential()  # Create a sequential model
model.add(Dense(128, activation='relu', input_dim=1))  # Add a fully connected layer with 128
model.add(Dense(256, activation='relu'))  # Add another fully connected layer with 256 neuron
model.add(Dense(128, activation='relu'))  # Add another fully connected layer with 128 neuron
model.add(Dense(units=1, activation='linear'))  # Add a fully connected output layer with lin
```

```
# Compile the model with Adam optimizer and mean squared error loss
opt = Adam(learning_rate=0.001)  # Initialize Adam optimizer with learning rate 0.001
model.compile(optimizer=opt, loss='mean_squared_error')

# Train the model for more epochs
model.fit(X_train, y_train, epochs=400, batch_size=32, validation_split=0.2)  # Train the mod
```

> After adding image noise, the diagram looks like this



- The well-trained model was able to remove the noise well and give us an approximation similar to the initial approximation
- here is the preview of written code

```
# Add noise to training data
noise = np.random.normal(0, 0.2, X_train.shape[0])  # Generate Gaussian noise with mean 0 and
X_train_noisy = X_train + noise  # Add noise to training data

# Define the neural network model
model_with_noise = Sequential()  # Create a new sequential model
model_with_noise.add(Dense(128, activation='relu', input_dim=1))  # Add a fully connected lay
model_with_noise.add(Dense(256, activation='relu'))  # Add another fully connected layer with
model_with_noise.add(Dense(128, activation='relu'))  # Add another fully connected layer with
model_with_noise.add(Dense(units=1, activation='linear'))  # Add a fully connected output lay

# Define the legacy optimizer
legacy_opt = legacy.Adam(learning_rate=0.001)

# Compile the model with the legacy optimizer
model_with_noise.compile(optimizer=legacy_opt, loss='mean_squared_error')

# Train the model with noisy training data
model_with_noise.fit(X_train_noisy, y_train, epochs=400, batch_size=32, validation_split=0.2)

# Make predictions on test data
predictions_with_noise = model_with_noise.predict(X_test)  # Use the trained model to make pr

# Plot the actual vs predicted values
plt.plot(X, y , color='black' , label='Actual')  # Plot actual values
plt.scatter(X_test, y_test,label='Test')  # Plot test data points
```

```
plt.scatter(X_test, predictions_with_noise,label='Predicted with Noise', marker='x')  # Plot
plt.legend()  # Add legend
plt.show()  # Show the plot
```

> If the diagram drawn in the neural network has points
> The jump is sudden, what happens?

💡 1-If the input data to the neural network has fluctuations or instability, it may cause jumps in normal conditions.

2-Improper Networks: Improperly Unusable Networks Large fluctuations in the network, such as weights and biases, can lead to dramatic jumps in neutral properties.

3-Large rating: Using too large a rate can cause trips to quickly deviate from the optimal path and turn into jumps.

These sudden jumps can appear as jump points in the cost function or the error function.

1. **Categories of images**

   Evaluation :

   - We have `10` classes, so if we pick a image and we randomly gues it class, we have `1/10` probability to be true.

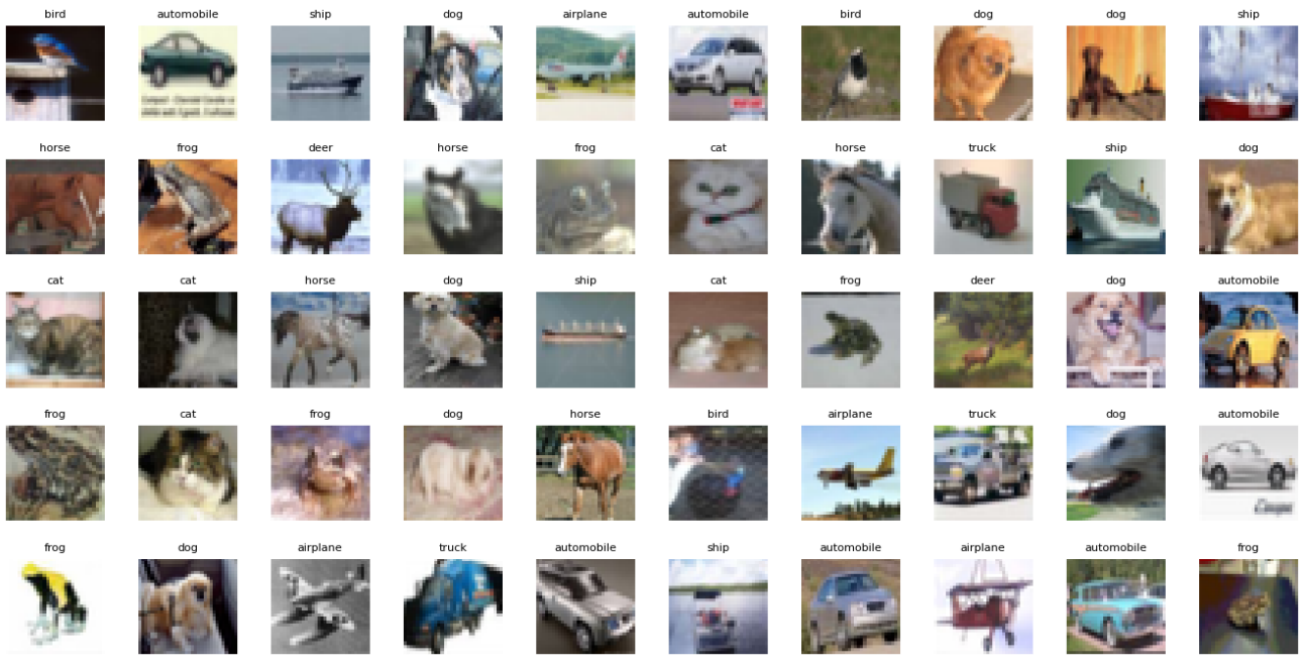
   Load the data and Data Visualization :

   - here is the preview of written code

```
(X_train, y_train), (X_test, y_test) = cifar10.load_data()  # Loading CIFAR-10 dataset and

print(f"X_train shape: {X_train.shape}")  # Printing the shape of the training data
print(f"y_train shape: {y_train.shape}")  # Printing the shape of the training labels
print(f"X_test shape: {X_test.shape}")  # Printing the shape of the testing data
print(f"y_test shape: {y_test.shape}")  # Printing the shape of the testing labels


# Define the labels of the dataset
labels = ['airplane', 'automobile', 'bird', 'cat', 'deer',
          'dog', 'frog', 'horse', 'ship', 'truck']

# Let's view more images in a grid format
# Define the dimensions of the plot grid
W_grid = 10
L_grid = 10

# fig, axes = plt.subplots(L_grid, W_grid)
# subplot return the figure object and axes object
# we can use the axes object to plot specific figures at various locations

fig, axes = plt.subplots(L_grid, W_grid, figsize = (17,17))

axes = axes.ravel() # flaten the 15 x 15 matrix into 225 array

n_train = len(X_train) # get the length of the train dataset

# Select a random number from 0 to n_train
for i in np.arange(0, W_grid * L_grid): # create evenly spaces variables

    # Select a random number
```

```
    index = np.random.randint(0, n_train)
    # read and display an image with the selected index
    axes[i].imshow(X_train[index,1:])
    label_index = int(y_train[index])
    axes[i].set_title(labels[label_index], fontsize = 8)
    axes[i].axis('off')

plt.subplots_adjust(hspace=0.4)
```

and result :

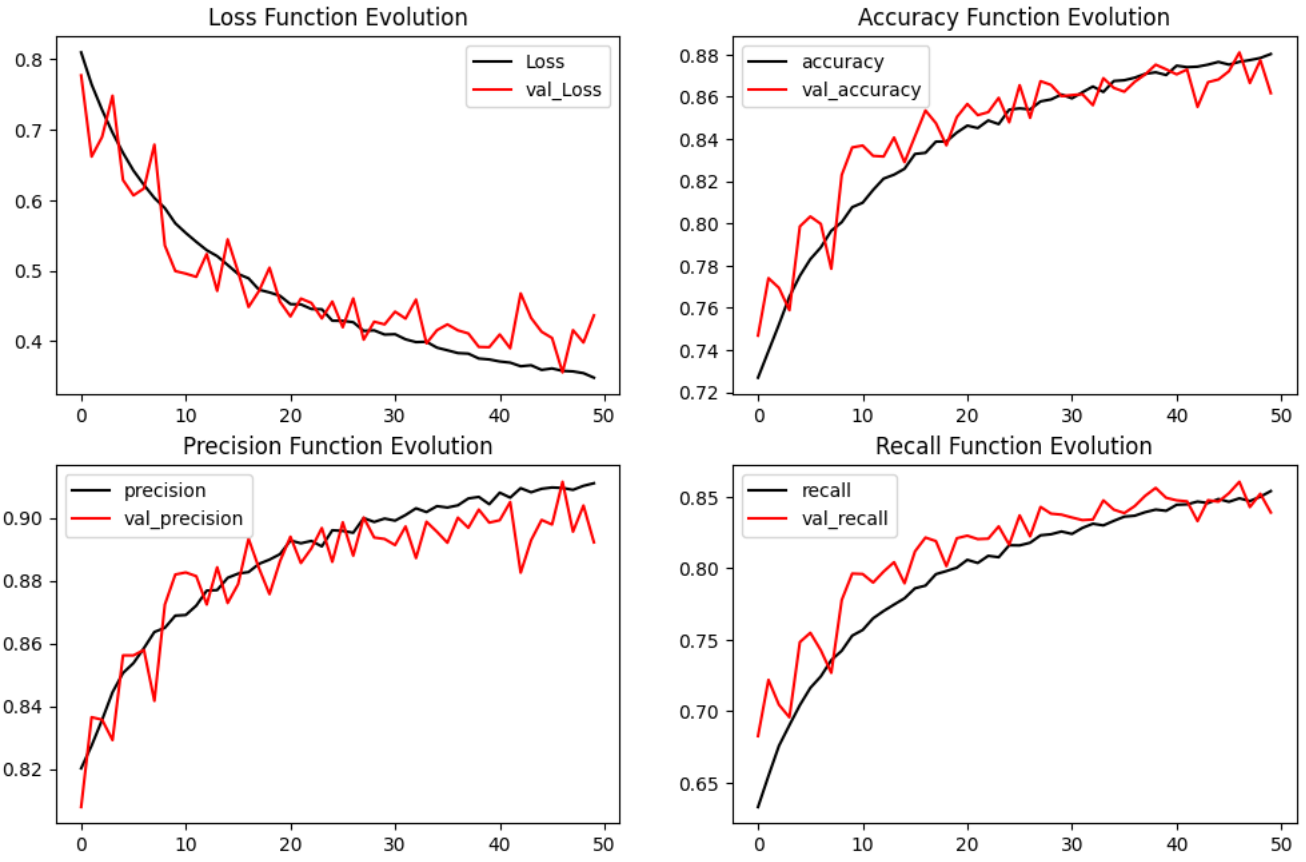


Data Augmentations :

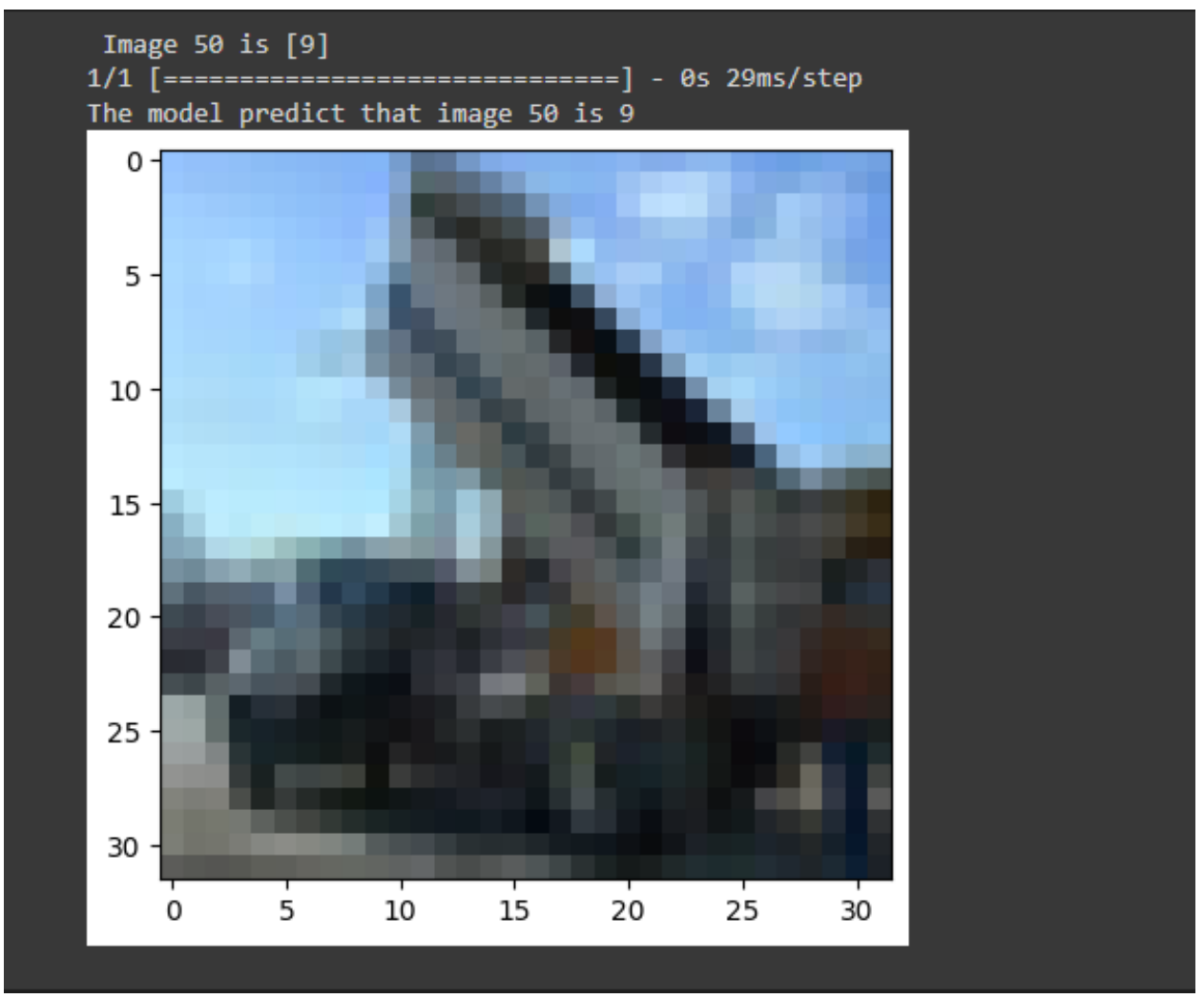
- In this part, we want to increase the accuracy by adding more data
- here is the preview of written code

```
batch_size = 32  # Setting the batch size for training

# Creating an ImageDataGenerator object with specified data augmentation parameters
data_generator = ImageDataGenerator(width_shift_range=0.1, height_shift_range=0.1, hori

# Creating a generator for training data using the defined ImageDataGenerator
train_generator = data_generator.flow(X_train, y_cat_train, batch_size)

# Calculating the number of steps per epoch for training
steps_per_epoch = X_train.shape[0] // batch_size

# Fitting the model to the training data using the generator
r = model.fit(train_generator,
              epochs=50,  # Setting the number of epochs
              steps_per_epoch=steps_per_epoch,  # Setting the steps per epoch
              validation_data=(X_test, y_cat_test)  # Providing validation data
             )
```

Model Evaluation :



Test on one image :



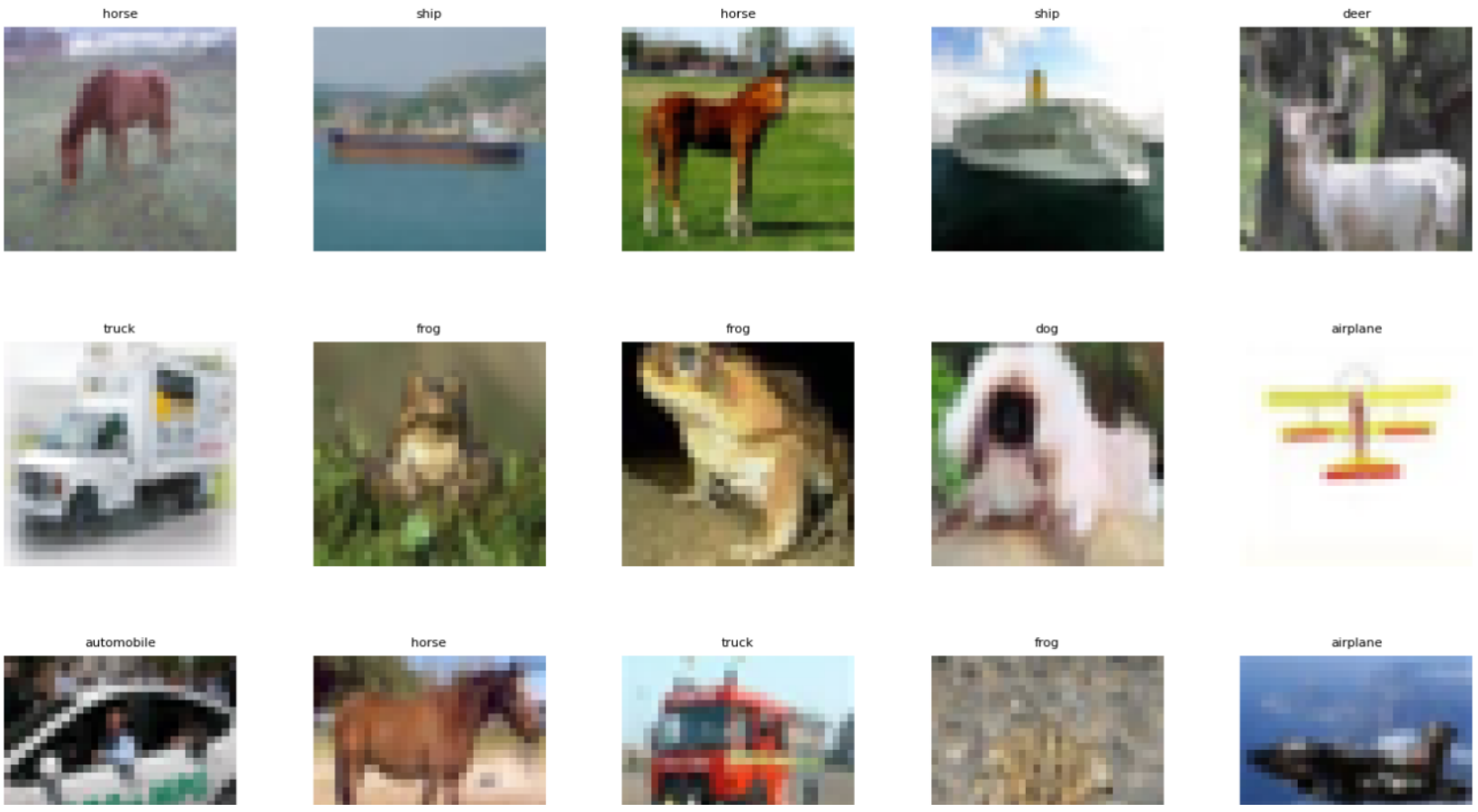
Trained model :

- here is the preview of written code

```
# Define the labels of the dataset
labels = ['airplane', 'automobile', 'bird', 'cat', 'deer',
          'dog', 'frog', 'horse', 'ship', 'truck']

# Let's view more images in a grid format
# Define the dimensions of the plot grid
W_grid = 5
L_grid = 5

# fig, axes = plt.subplots(L_grid, W_grid)
# subplot return the figure object and axes object
```

```
# we can use the axes object to plot specific figures at various locations

fig, axes = plt.subplots(L_grid, W_grid, figsize = (17,17))

axes = axes.ravel() # flaten the 15 x 15 matrix into 225 array

n_test = len(X_test) # get the length of the train dataset

# Select a random number from 0 to n_train
for i in np.arange(0, W_grid * L_grid): # create evenly spaces variables

    # Select a random number
    index = np.random.randint(0, n_test)
    # read and display an image with the selected index
    axes[i].imshow(X_test[index,1:])
    label_index = int(y_pred[index])
    axes[i].set_title(labels[label_index], fontsize = 8)
    axes[i].axis('off')

plt.subplots_adjust(hspace=0.4)
```

and result :



1. **simple face recognition using MLP**
    - First, we upload the zip file using the following commands and extract it

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
!unzip /content/drive/MyDrive/Dataset.zip
```

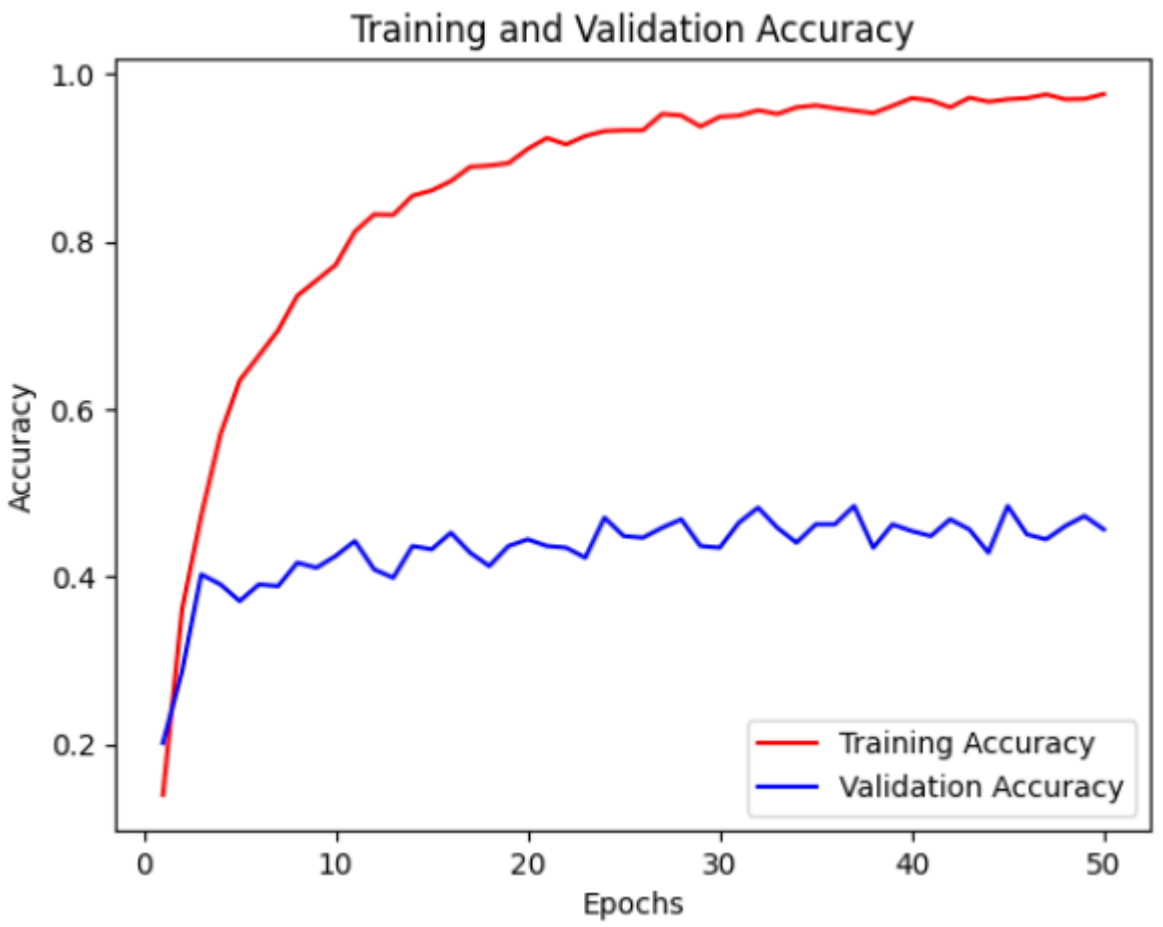
```
Streaming output truncated to the last 5000 lines.
  inflating: Dataset/Faces/Faces/Alia Bhatt_22.jpg
  inflating: Dataset/Faces/Faces/Alia Bhatt_23.jpg
  inflating: Dataset/Faces/Faces/Alia Bhatt_24.jpg
  inflating: Dataset/Faces/Faces/Alia Bhatt_25.jpg
  inflating: Dataset/Faces/Faces/Alia Bhatt_26.jpg
  inflating: Dataset/Faces/Faces/Alia Bhatt_27.jpg
  inflating: Dataset/Faces/Faces/Alia Bhatt_28.jpg
  inflating: Dataset/Faces/Faces/Alia Bhatt_29.jpg
  inflating: Dataset/Faces/Faces/Alia Bhatt_3.jpg
  inflating: Dataset/Faces/Faces/Alia Bhatt_30.jpg
```

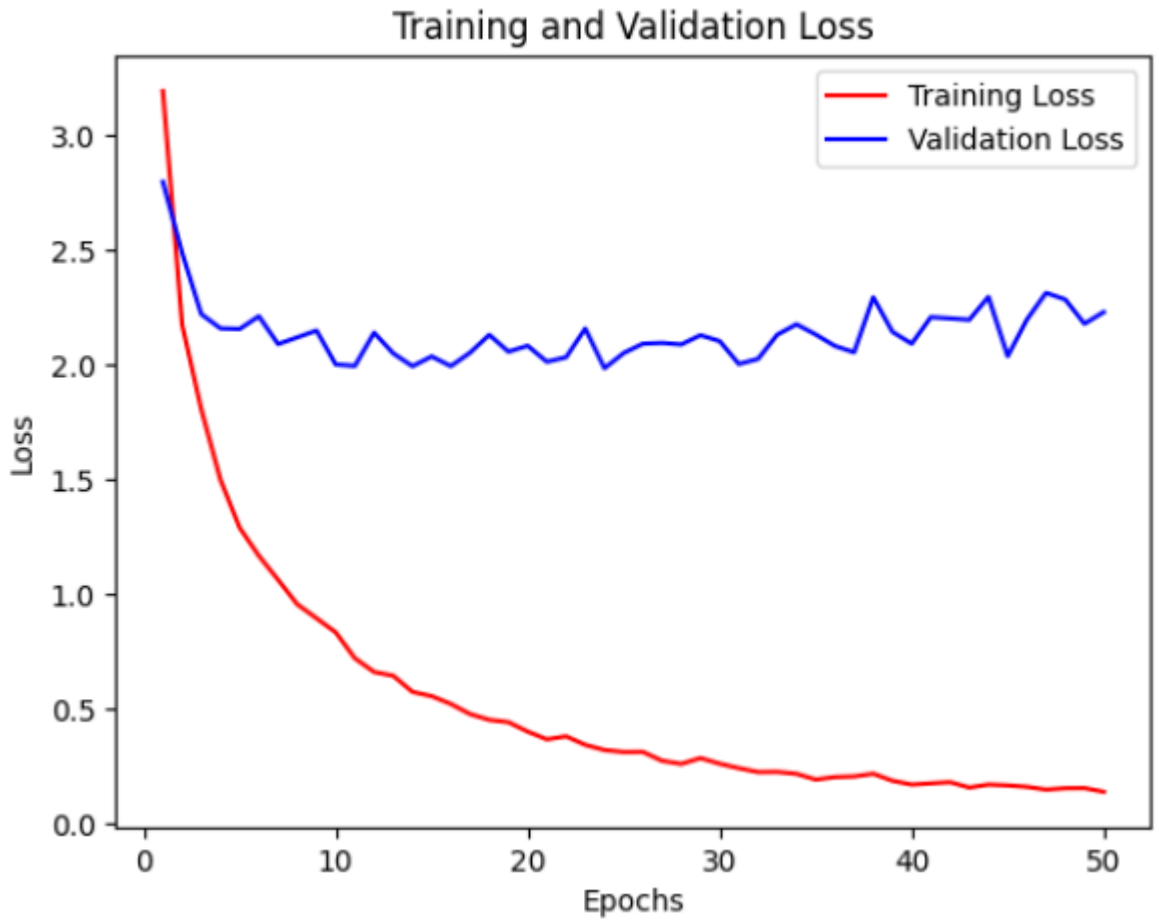
- Then we set the directory address and start to continue the project
- here is the preview of written code

```
import os
DIR = './Dataset/Original Images/Original Images'


generator = ImageDataGenerator(
    height_shift_range=0.1,
    rescale=1./255,
    shear_range=0.2,
    zoom_range=0.2,
    width_shift_range=0.1,
    horizontal_flip=True,
    validation_split=0.2
)

# train_ds, val_ds = keras.utils.image_dataset_from_directory(
#     DIR,
#     validation_split=0.2,
#     subset='both',
#     seed=1337,
#     image_size=(16, 16),
#     batch_size=32,
#     shuffle=True,
#     label_mode='categorical'
# )

file_count = 0
total_image_count = 0


for root, dirs, files in os.walk(DIR):
    for file in files:
        file_count += 1
        file_path = os.path.join(root, file)
        with open(file_path, 'r', encoding='latin-1') as f:
            image_count = sum(1 for _ in f)
            total_image_count += image_count
```

```
            print(f'File : {file_path}, Number of photos : {image_count}')

print(f'Number of All Files : {file_count}')
print(f'Number of All photos : {total_image_count}')

train_ds = generator.flow_from_directory(
    DIR,
    target_size=(128, 128),
    batch_size=32,
    shuffle=True,
    subset="training"
)

val_ds = generator.flow_from_directory(
    DIR,
    target_size=(128, 128),
    batch_size=32,
    shuffle=True,
    subset="validation"
)
```

- Then, after running the model, we get the accuracies





and result :

```
train_loss, train_accuracy = model.evaluate(train_ds)
print(f"Training Accuracy:" , train_accuracy)

validation_loss, validation_accuracy = model.evaluate(val_ds)
print(f"Validation Accuracy:" , train_accuracy)
```

```
65/65 [==============================] - 37s 575ms/step - loss: 0.1242 - accuracy: 0.9801
Training Accuracy: 0.9800971150398254
16/16 [==============================] - 11s 659ms/step - loss: 2.1312 - accuracy: 0.4602
Validation Accuracy: 0.9800971150398254
```

The end😌