EE6770_F24_Homework4

October 8, 2024

#

EE 6770 Fall 2024: Homework 4

1. **Write comments** in the code to explain your thoughts.
2. **Important: Execute the codes and show the results**.
3. **Do your own work.**

0.0.1 Submission:

- **Submit this notebook file and the pdf version** - remember to add your name in the filename.
- Deadline: 11:59 pm, 10/9 (Wednesday)

0.1 Assignment Objectives:

This assignment builds on the work from Homework 3, where you will use Keras to construct and experiment with a neural network as part of the Machine Learning Workflow. The dataset has been modified to include noisy samples, making the classification task more challenging. The dataset is split into three segments::

- **Train Data: 75% of the dataset**
- **Validation Data: 20% of the Train Data**
- **Test Data: 25% of the dataset**

Your task is to improve the provided baseline neural network model and achieve **at least 90% accuracy on the Test Data while preventing overfitting**. You will experiment with various network configurations, including architecture, regularization techniques, and hyperparameter tuning. The key objective is to optimize the model's performance while maintaining a balance between training and validation accuracy.

0.1.1 Steps to accomplish the tasks:

1. **Baseline Model:**
   - This Jupyter Notebook file contains a baseline model with the training, validation, and evaluation procedures already set up.
   - This is your starting point. **The accuracy on the Test Data is ~83%**.
2. **Modify the architecture and training configurations** to improve performance and prevent overfitting. This is the core of this exericse. You will experiement on combinations of different model architectures and hyperparameters such as:.
   - Network architecture (e.g., number of layers, number of neurons).

- Regularization techniques (e.g., L1, L2).
- Optimizers (e.g., Adam, SGD)
- Learning rates, Epochs
- Batch size
- Activation function

3. **Track and Document Your Trials**: For each of your trial, document network configurtions and performance metrics:
   - Model architecture (layers/neurons).
   - Regularization and optimizer used.
   - Training loss/accuracy, validation loss/accuracy, and test accuracy.
   - A reflection on whether the model overfitted or improved performance.
   - Present these results in a table format to summarize all your experiments
   - You will present `at least 5 trials` to gain full credit

0.1.2 Grading Breakdown:

1. **Achieving $\geq$ 90% Test Accuracy (65 points)**: Points will be deducted based on how far the model's performance is from the 90% accuracy target.
2. **Experimentation and Documentation (30 points)**: Test at least five configurations with clear documentation. This should include modifications to architecture, regularization methods, hyperparameters, and optimizations.
3. **Analysis of Results (5 points)**: Provide clear, well-reasoned explanations for the changes made and analyze how they affected performance. Include observations on underfitting or overfitting where relevant.

0.1.3 Import Tools

[59]:
```
import numpy as np
import matplotlib.pyplot as plt
import math

%matplotlib inline
```

0.1.4 Import Keras and TensorFlow

[60]:
```
import tensorflow as tf
from tensorflow import keras

from keras.layers import Dense
from keras.models import Sequential
```

0.2 Section 1: Generate the Spiral Dataset

0.2.1 Note: Do NOT change the noise level in the function.

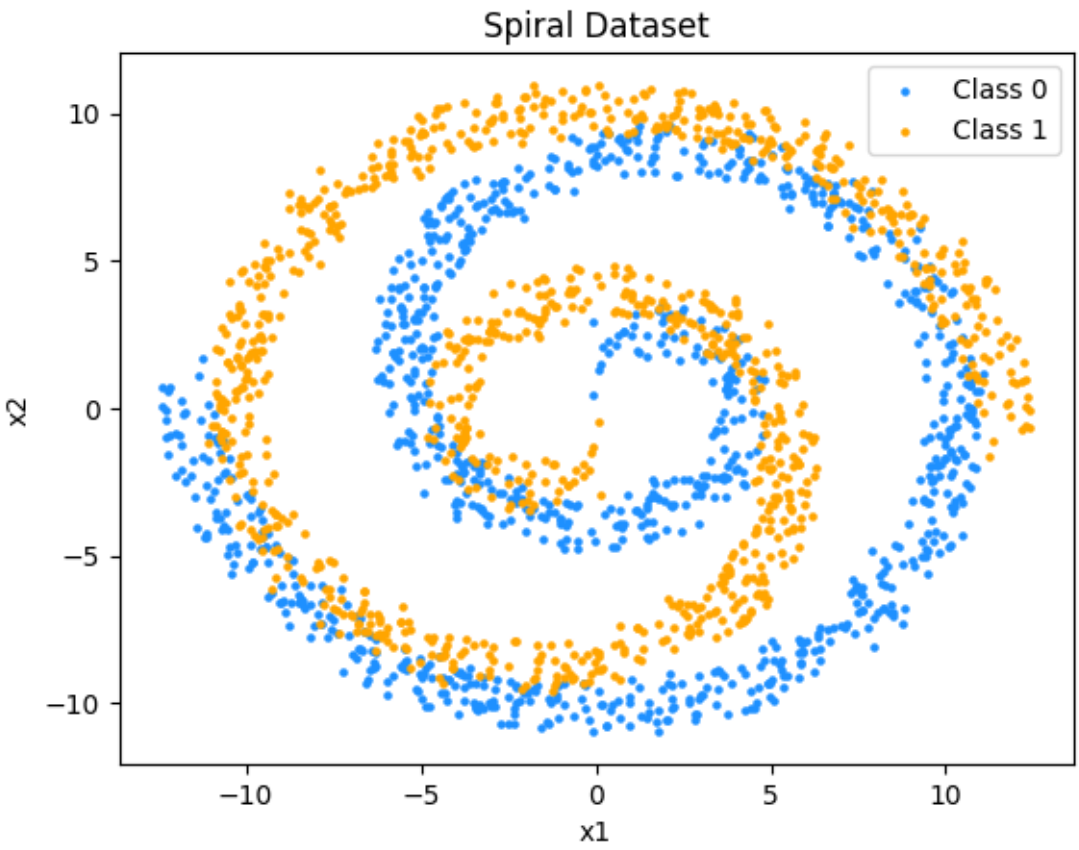
[61]:
```
def generate_spiral_data(n_points, noise=1.7):

    n = np.sqrt(np.random.rand(n_points, 1)) * 720 * (2*np.pi) / 360
    d1x = -np.cos(n) * n + np.random.rand(n_points, 1) * noise
    d1y = np.sin(n) * n + np.random.rand(n_points, 1) * noise

    return np.vstack((np.hstack((d1x, d1y)), np.hstack((-d1x, -d1y)))), np.
 ↪hstack((np.zeros(n_points), np.ones(n_points)))
```

0.2.2 Generate 1000 pairs of data points.

[62]:
```
# Notation:
# X = input data,  Y = labelled target (0 or 1)

X, y = generate_spiral_data(1000)
```

0.2.3 Plot the spiral dataset.

[63]:
```
# Plot the dataset with custom colors: Dodger Blue for class 0, Orange for␣
 ↪class 1

plt.scatter(X[y == 0, 0], X[y == 0, 1], color='dodgerblue', label='Class 0',␣
 ↪s=5)
plt.scatter(X[y == 1, 0], X[y == 1, 1], color='orange', label='Class 1', s=5)

plt.title('Spiral Dataset')
plt.legend()
plt.xlabel('x1')
plt.ylabel('x2')
plt.show()
```



0.2.4 Normalize the two input features with respect to the mean and standard deviation.

0.2.5 This is called z-score normalization, which helps to accelerate learning.

[64]:
```
def zscore_normalize(X):

    mu = np.mean(X, axis=0)          # find the mean of each column/feature

    sigma  = np.std(X, axis=0)       # find the standard deviation of each column/
↪feature

    X = (X - mu) / sigma             # Normalize each column wrt its mean and std

    return X, mu, sigma
```

0.2.6 Write codes to normalize the spiral dataset.

[65]:
```
## WRITE YOUR CODE.

X, mu, std = zscore_normalize(X)
```

0.3 Section 2: Train-Test Data Split

Before building a logistic regression model, let's start by splitting the data into a training set and test set.

We will accomplish this by using the `train_test_split` module from `Scikit-Learn`

0.3.1 Load Scikit Learn ML packages

[66]:
```
from sklearn.model_selection import train_test_split
```

[67]:
```
X.shape, y.shape
```

[67]:
```
((2000, 2), (2000,))
```

[68]:
```
# configure the test_size to split the dataset into Train and Test Data

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25,␣
 ↪random_state=101)
```

[69]:
```
# Verify the shapes of Train and Test Data

X_train.shape, X_test.shape, y_train.shape, y_test.shape
```

[69]:
```
((1500, 2), (500, 2), (1500,), (500,))
```

0.4 Section 3: Set Up Neural Network Architecture

0.4.1 Neural Network Parameters

Here is where you specify the NN architecture of your choice, which shall include: * Number of hidden layers * Number of neurons in each of the hidden layer * Number of neurons in the output layer (please set it as 1)

[70]:
```
n = X.shape[1]   # number of input features
m = X.shape[0]   # number of samples
```

[71]:
```
## This is the Baseline Model. You create your own model here.

n1 = 10           # number of neurons in the hidden layer 1
n2 = 10           # number of neurons in the hidden layer 2
n3 = 1           # number of neurons in the output layer
```

0.4.2 Section 3.1. Define the NN Model

[72]:
```
model = Sequential()

model.add(Dense(n1, input_dim=n, activation="tanh"))
model.add(Dense(n2, activation="tanh"))
model.add(Dense(n3, activation="sigmoid"))
```

Top-level summary of the model

[73]:
```
model.summary()
```

```
Model: "sequential_2"
```

Layer (type)	Output Shape	Param #
dense_6 (Dense)	(None, 10)	30
dense_7 (Dense)	(None, 10)	110
dense_8 (Dense)	(None, 1)	11

```
Total params: 151 (604.00 B)

Trainable params: 151 (604.00 B)

Non-trainable params: 0 (0.00 B)
```

Check out the weight initialization

[74]:
```
W = model.get_weights()

W
```

[74]:
```
[array([[-0.49460286, -0.65305614,  0.5228146 , -0.34540555,  0.5587571 ,
          0.23396033,  0.1243605 ,  0.13256484,  0.22136182,  0.44363075],
        [ 0.6725653 , -0.14868277, -0.60520077,  0.05517513,  0.6109064 ,
          0.5431333 , -0.6131289 ,  0.08461773, -0.18024153, -0.66811234]],
       dtype=float32),
 array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.], dtype=float32),
 array([[ 0.1776231 , -0.43583113, -0.21564877, -0.36069632,  0.31518573,
          0.11359876,  0.09711379, -0.11429048,  0.05779594,  0.13570285],
        [-0.4891618 , -0.11905977, -0.4308867 , -0.40795648,  0.20576632,
          0.36784935, -0.27093223,  0.19546062,  0.30919808, -0.13969532],
        [ 0.17702353, -0.03038096,  0.33528507, -0.33533943, -0.00557178,
```

```
        -0.33382812,  0.3876707 , -0.5148232 ,  0.12300104,  0.28540754],
       [ 0.15880531,  0.16040838,  0.4168883 ,  0.2312687 , -0.4127041 ,
        -0.42619967, -0.4928403 ,  0.1735251 ,  0.41966677,  0.20124185],
       [-0.27756238,  0.07634741, -0.35099617, -0.15150759,  0.5414934 ,
         0.49600625, -0.28028083, -0.4203918 ,  0.02656472,  0.31635362],
       [ 0.29864973, -0.21566796,  0.4962548 , -0.19593465, -0.09648138,
         0.37529534,  0.04909664, -0.43356818,  0.4106416 , -0.4114967 ],
       [-0.2605893 , -0.16209716, -0.484268  , -0.53001326,  0.20820045,
        -0.24337348,  0.16533208,  0.3729722 ,  0.23423392, -0.17352718],
       [ 0.06547552,  0.44336742, -0.132498  , -0.50997216, -0.00473028,
        -0.31978375, -0.48475993, -0.21082944,  0.2519669 ,  0.02173549],
       [-0.24278858, -0.09293804,  0.1309393 ,  0.488904  ,  0.44098407,
        -0.4306826 ,  0.1057536 ,  0.4197355 ,  0.26132375, -0.00954634],
       [-0.4804524 ,  0.46453118, -0.02122515,  0.54662204, -0.24345994,
         0.5326203 ,  0.42073095, -0.27513516, -0.35141888, -0.04358935]],
      dtype=float32),
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.], dtype=float32),
array([[ 0.37109917],
       [ 0.31948012],
       [ 0.29366904],
       [ 0.51628965],
       [ 0.68670684],
       [ 0.45109326],
       [ 0.4166711 ],
       [-0.4561218 ],
       [-0.18435216],
       [-0.29187492]], dtype=float32),
array([0.], dtype=float32)]
```

0.4.3 Section 3.2. Compile the Network Model

You need to configure the `compile` API with at least three parameters:

- **loss**: the loss function of choice. For binary classification, the default = `binary_crossentropy`
- **optimizer**: the variation of Gradient Descent algorithm. We choose `SGD = Stochastic GD` here
- **metrics**: the metric to evaluate the training performance

[75]:
```
sgd = keras.optimizers.SGD(learning_rate=0.1)

model.compile(loss="binary_crossentropy", optimizer=sgd, metrics=["accuracy"])
```

0.5 Section 4: Train and Fit the Model

0.5.1 Set up hyperparameters

- **Learning Rate**: It was already set in the previous step when the optimzer is chosen.

- **Number of Iterations (epochs)**: Vary the number of iterations (epochs) for training. Begin with a moderate number (e.g., 10000) and adjust accordingly.
- **Batch Size**: Default value = 32. Play with a few different numbers to see the behavior of the training process.
- **Steps Per Epoch**: Alternatively, you can set up the number of steps in each epoch and Keras would figure out the batch size automatically
- **Callbacks**: are powerful features executed at specific stages of the training process (such as at the end of an epoch, batch, or training). They are used to **monitor the training**, perform actions (like **saving the best model**), or **stop training early** based on certain conditions. We will use the `MakeCheckPoint` option to save the best model with the best validation loss or validation accuracy after training.

For more information on `model.fit`, please refer to the Keras API Documentation.

[76]:
```
from keras.callbacks import EarlyStopping, ModelCheckpoint

# Create the Early Stopping callback. For a choppy SGD optimizer, this may not␣
 ↪work as expected.
early_stopping = EarlyStopping(monitor='val_loss', patience=500,␣
 ↪restore_best_weights=True, verbose=0)

# Create the ModelCheckpoint callback. The best model is saved in H5 format.
checkpoint = ModelCheckpoint('best_model.keras', monitor='val_accuracy',␣
 ↪save_best_only=True, verbose = 0)
```

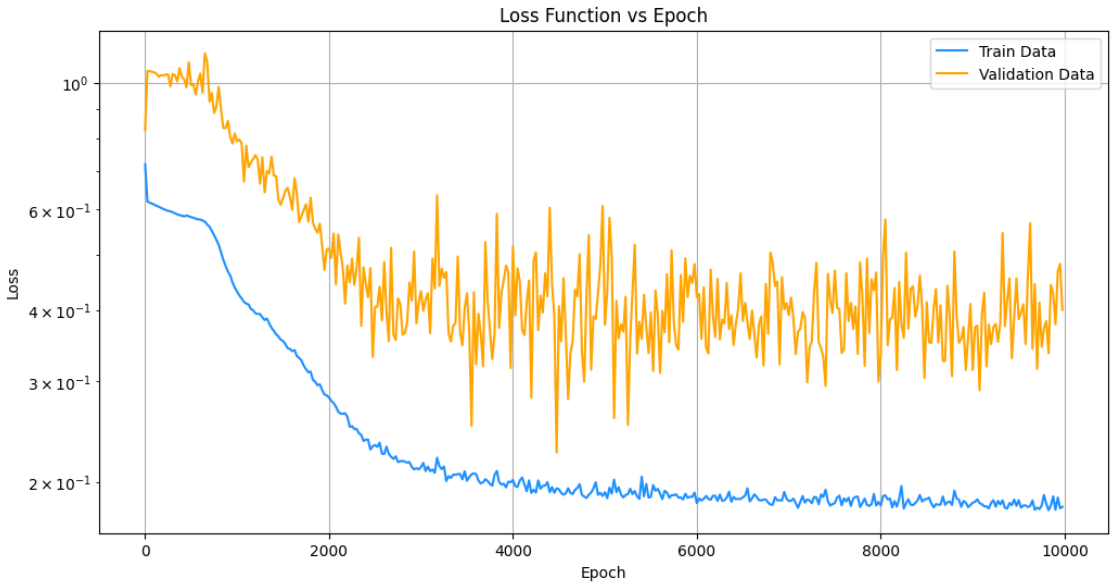
[77]:
```
history = model.fit(X, y, epochs=10000, steps_per_epoch=None, batch_size = 256,␣
 ↪validation_split=0.25, verbose = 0,
                callbacks=[checkpoint])
```

0.5.2 Display the history on `Loss` and `Accuracy` for both the Train Data and Validation Data

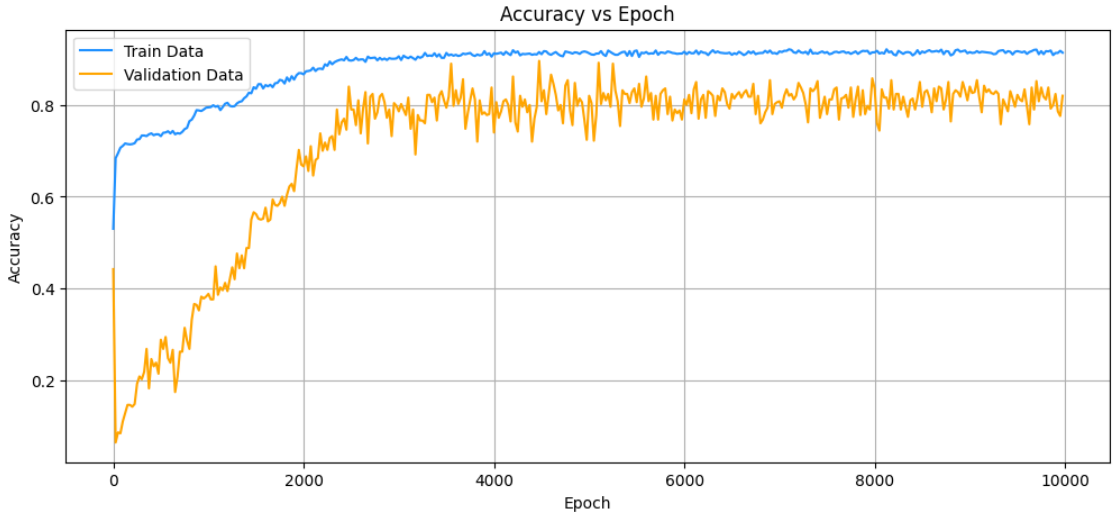
[78]:
```
# Retrieve training results

train_loss = history.history["loss"]
train_acc  = history.history["accuracy"]
valid_loss = history.history["val_loss"]
valid_acc  = history.history["val_accuracy"]
```

[79]:
```
plt.figure(figsize=(12,6))

# Adjust the value of 'step' to plot the results every 'step' epoch.
# This allows you to have cleaner plot when the number of epochs are large

step = 25
x_range = range(0, len(train_loss), step)
plt.plot(x_range, train_loss[ : :step], color='DodgerBlue', label='Train Data')
plt.plot(x_range, valid_loss[ : :step], color='orange', label='Validation Data')
```

```
plt.title('Loss Function vs Epoch')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.yscale('log')
plt.legend()
plt.grid()
plt.show()
```



[80]:
```
plt.figure(figsize=(12,5))

# Adjust the value of 'step' to plot the results every 'step' epoch.
# This allows you to have cleaner plot when the number of epochs are large

step = 25
x_range = range(0, len(train_acc), step)
plt.plot(x_range, train_acc[ : :step], color='DodgerBlue', label='Train Data')
plt.plot(x_range, valid_acc[ : :step], color='orange', label='Validation Data')

plt.title('Accuracy vs Epoch')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.grid()
plt.show()
```



0.5.3 Print the following outputs after model training:

-
-

[81]:
```
print (f"The final loss for the Train Data = {train_loss[-1] :.4f}")

print (f"The final loss for the Validation Data = {valid_loss[-1] :.4f}")
```

```
The final loss for the Train Data = 0.1805
The final loss for the Validation Data = 0.4230
```

[82]:
```
print (f"The accuracy for the Train Data = {train_acc[-1]*100 :.2f}%")

print (f"The accuracy for the Validation Data = {valid_acc[-1]*100 :.2f}%")
```

```
The accuracy for the Train Data = 91.53%
The accuracy for the Validation Data = 81.20%
```

0.6 Section 5: Model Evaluation

0.6.1 Section 5.1. Performance Analysis

We use the `evaluate` method to obtain the `loss` and `accuracy` performance of the `Test Data` using the best training model.

0.6.2 Load the model with the best `val_loss`

[83]:
```
from keras.models import load_model

# Load the best model
best_model = load_model('best_model.keras')
```

0.6.3 Use the loaded model to evaluate or make predictions

[84]:
```
loss, accuracy = best_model.evaluate(X_test, y_test, verbose=1)

print(f"Test Data Loss: {loss: .4f}")
print(f"Test Data Accuracy: {accuracy*100: .2f}%")
```

```
16/16             0s 955us/step -
accuracy: 0.9253 - loss: 0.1872
16/16             0s 955us/step -
accuracy: 0.9253 - loss: 0.1872
Test Data Loss:  0.1979
Test Data Accuracy:  91.60%
```

0.6.4 Section 5.2. Visualize the Decision Boundary

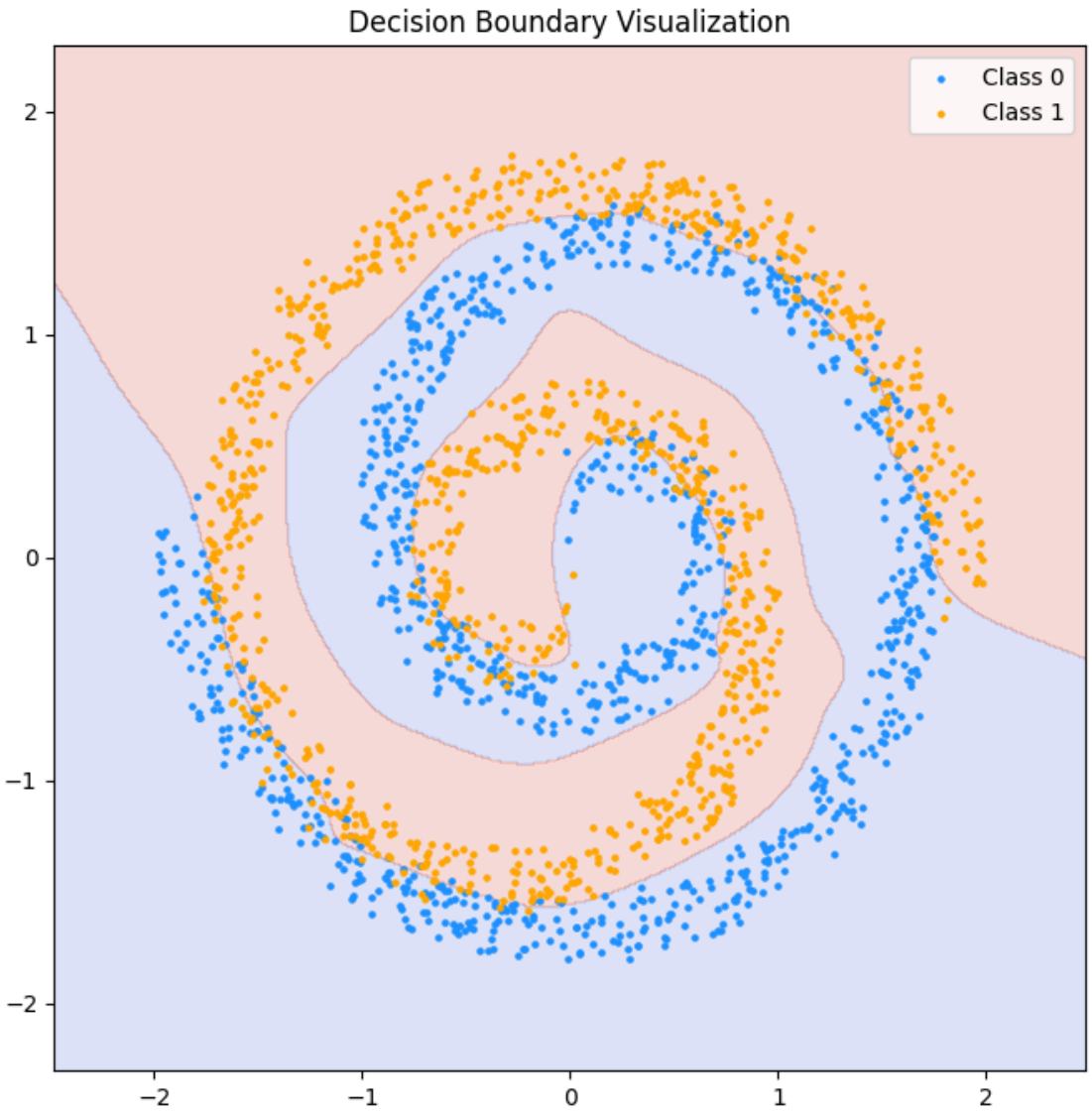
We will use the `predict()` method to revise the codes for plotting the decision bounday in Homework 3.

[85]:
```
import numpy as np
import matplotlib.pyplot as plt

# Generate a meshgrid of points
def create_meshgrid(X, h=0.01):
    x_min, x_max = X[:, 0].min() - .5, X[:, 0].max() + .5
    y_min, y_max = X[:, 1].min() - .5, X[:, 1].max() + .5
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
    return xx, yy

# Predict function that computes output for each point in the meshgrid
def predict_on_grid(xx, yy, model):
```

```
    grid = np.c_[xx.ravel(), yy.ravel()]  # Combine the grid points into a␣
 ↪shape for input

    # Use the trained model to predict the class for each point in the grid

    predictions = model.predict(grid)
    class_predictions = (predictions >= 0.5).astype(int)     # Convert sigmoid␣
 ↪output to binary class (0 or 1)

    return class_predictions.reshape(xx.shape)

# Plot the decision boundary
def plot_decision_boundary(X, y, model):

    xx, yy = create_meshgrid(X)
    Z = predict_on_grid(xx, yy, model)

    plt.figure(figsize=(8, 8))
    # Plot the contour map for decision boundary
    plt.contourf(xx, yy, Z, alpha=0.2, cmap='coolwarm')

    # Plot the original data points
    plt.scatter(X[y == 0, 0], X[y == 0, 1], c='dodgerblue', label='Class 0',␣
 ↪s=5)
    plt.scatter(X[y == 1, 0], X[y == 1, 1], c='orange', label='Class 1', s=5)
    plt.legend()
    plt.title('Decision Boundary Visualization')
    plt.show()
```

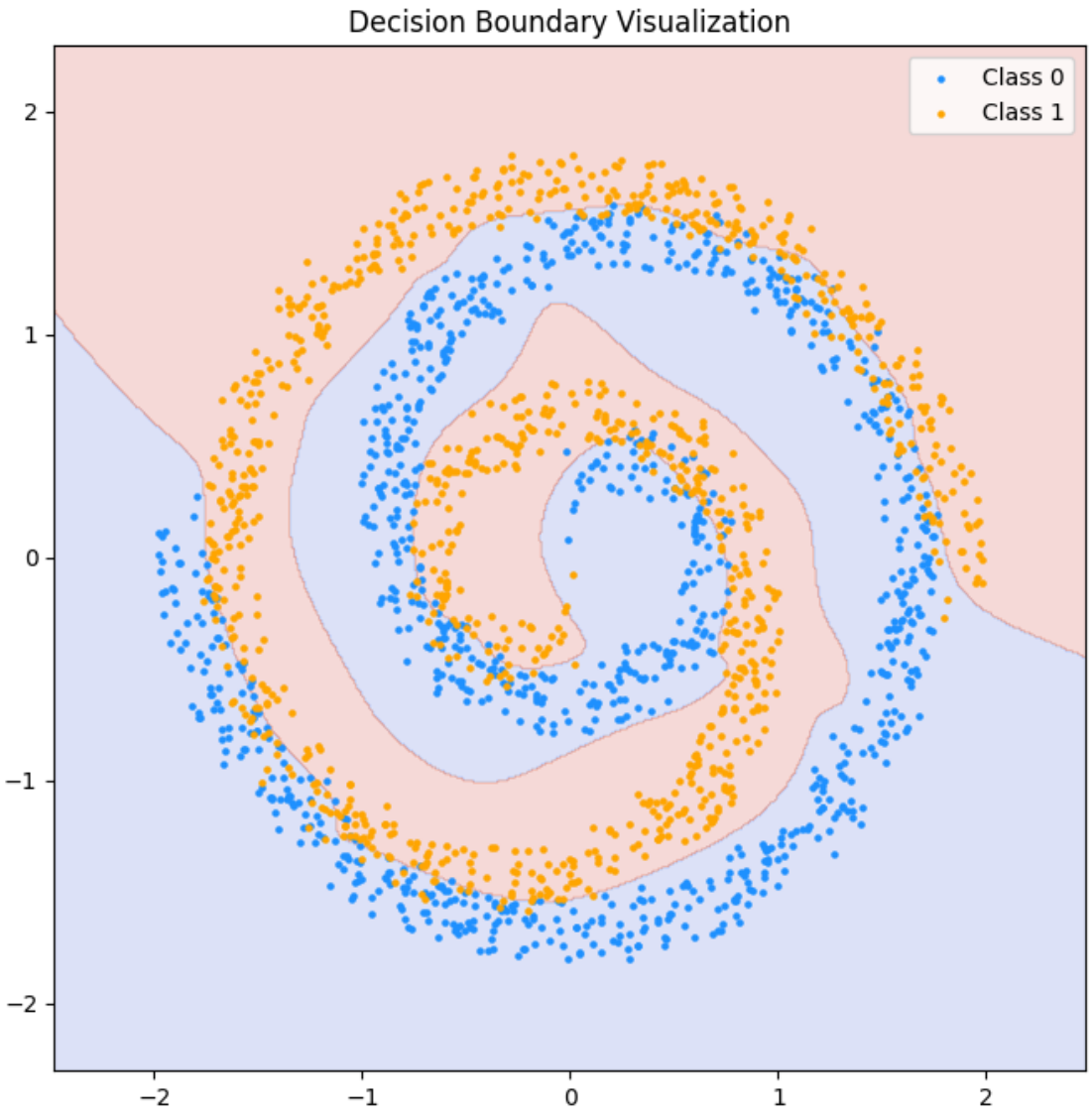
[86]:
```
# Example usage after training your neural network

plot_decision_boundary(X, y, best_model)
```

```
7160/7160             4s 573us/step
```



[87]:
```
# For comparison: Use the model at the end of training

plot_decision_boundary(X, y, model)
```

```
7160/7160              4s 569us/step
```


Decision Boundary Visualization
Class 0
Class 1
2
1
0
−1
−2
−2
−1
0
1
2

[ ]: