Autoencoders for Anomaly Detection

1. Provide brief details about the nature of your dataset. What is it about? What type of data are we encountering? How many entries and variables does the dataset comprise?

The dataset has been taken from Yahoo Webscope program. It consists of time-series data with labeled anomalies. The data represents real production traffic of some Yahoo properties. The dataset comprises of three columns, namely:

- timestamp: in the dataset, it is replaced by integers incremented by 1, which represents 1 hour worth of data.;
- value: value recorded at the corresponding timestamp.; and
- is_anomaly: Boolean indicating if the current value at the given timestamp is an anomaly or not.

There are a total of 67 files in the folder. I have combined all the files and loaded them into the dataframe. Therefore, the total number of entries in the dataframe is 94866.

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

	timestamp	value	is_anomaly
count	94866.000000	9.486600e+04	94866.000000
mean	713.461883	1.003524e+05	0.017593
std	414.700963	7.212159e+05	0.131468
min	1.000000	0.000000e+00	0.000000
25%	354.000000	4.000000e+00	0.000000
50%	708.000000	6.200000e+01	0.000000
75%	1072.000000	2.127000e+03	0.000000
max	1461.000000	7.845760e+06	1.000000

```
[ ] df.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 94866 entries, 0 to 94865
Data columns (total 3 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   timestamp   94866 non-null  int64
 1   value       94866 non-null  float64
 2   is_anomaly  94866 non-null  int64
dtypes: float64(1), int64(2)
memory usage: 2.2 MB
```

Why did I select this dataset?

- The dataset represents real production traffic data. It is highly relevant for all the companies to detect anomalies in their organization so that they take appropriate action whenever they encounter such scenarios.

2. Describe the details of your autoencoder models, including the layers, activation functions, and any specific configurations employed.

Details for autoencoder Model 1:

The first model consists of simple autoencoder architecture with dense layers (fully connected).

In the Encoder part:

- The input layers is a fully connected linear layer having dimensions (1, 64), followed by a ReLU activation function. Next, there are 2 hidden layers, fully connected linear layer, of dimensions (64, 32) and (32, 16) respectively with ReLU activation function used in both the layers.

In the Decoder part:

- In the decoder part, there are again 2 hidden layers of dimensions (16, 32) and (32, 64) with ReLU activation functions.
- The reconstruction output layer is fully connected linear layer with dimensions (64, 2) and activation function ReLU.

Loss Function and Optimizer: I have used Mean Squared Error (MSE) loss function with Adam optimizer.

Details for autoencoder Model 2:

The second model is an autoencoder architecture using LSTM.

Encoder part:

- The LSTM encoding layer accepts the input dimension of 2 and hidden dimensions of 64, with batch first parameter argument True.

Decoder part:

- The LSTM decoding layer has the input dimension of 62 and the output dimensions of 2, with batch first parameter argument True.

Activation Function: I have used the default activation function used in the LSTM layer internally, which is tanh() function.

Loss Function and Optimizer: I have used MSE loss function and Adam optimizer.

Details for autoencoder Model 3:

The third model is an extension of the first model, where I have changed some hyperparameters.

Encoder part:

- There are a total of 4 fully connected liner layers, having dimensions (2, 128), (128, 64), (64, 32), and (32, 16).

Decoder part:

- There are a total of 4 fully connected liner layers, having dimensions (16, 32), (32, 64), (64, 128), and (128, 2) to reconstruct the input data.

Activation Function: I have utilized tanh() activation functions in part the encoder and decoder parts.

Loss Function and Optimizer: I have used MSE loss function and Adam optimizer.

Details for autoencoder Model 4:

The fourth model is an extension of the third model, where I have changed some hyperparameters.

Encoder part:

- There are a total of 4 fully connected liner layers, having dimensions (2, 128), (128, 64), (64, 32), and (32, 16).

Decoder part:

- There are a total of 4 fully connected liner layers, having dimensions (16, 32), (32, 64), (64, 128), and (128, 2) to reconstruct the input data.

Activation Function: I have utilized LeakyRelu() activation functions in part the encoder and decoder parts.

Loss Function and Optimizer: I have used MSE loss function and Adam optimizer.

Dropout Layers: I have added dropout layers between fully connected linear layers, with the probability value of 0.2.

3. Discuss the results and provide relevant graphs:

a. Report training accuracy, training loss, validation accuracy, validation loss, testing accuracy, and testing loss.

Model 1:

Training Accuracy, Training Loss, Validation Accuracy, Validation Loss, Testing Accuracy, and Testing Loss of model 1:

```
[17] print(f"Training Accuracy: {training_r2_score*100:.4f}, Training Loss: {training_loss:.4f}")
     print(f"Validation Accuracy: {validation_r2_score*100:.4f}, Validation Loss: {validation_loss:.4f}")
     print(f"Testing Accuracy: {test_r2_sum*100:.4f}, Testing Loss: {test_loss:.4f}")

     Training Accuracy: 96.0662, Training Loss: 0.0392
     Validation Accuracy: 96.6134, Validation Loss: 0.0343
     Testing Accuracy: 96.6081, Testing Loss: 0.0340
```

Model 2:

Training Accuracy, Training Loss, Validation Accuracy, Validation Loss, Testing Accuracy, and Testing Loss of model 2:

```
[25] print(f"Training Accuracy: {training_r2_score*100:.4f}, Training Loss: {training_loss:.4f}")
     print(f"Validation Accuracy: {validation_r2_score*100:.4f}, Validation Loss: {validation_loss:.4f}")
     print(f"Testing Accuracy: {test_r2_sum*100:.4f}, Testing Loss: {test_loss:.4f}")

     Training Accuracy: 42.6913, Training Loss: 0.5674
     Validation Accuracy: 43.1275, Validation Loss: 0.5863
     Testing Accuracy: 43.2256, Testing Loss: 0.5720
```

Model 3:

Training Accuracy, Training Loss, Validation Accuracy, Validation Loss, Testing Accuracy, and Testing Loss of model 3:

```
[32] print(f"Training Accuracy: {training_r2_score*100:.4f}, Training Loss: {training_loss:.4f}")
     print(f"Validation Accuracy: {validation_r2_score*100:.4f}, Validation Loss: {validation_loss:.4f}")
     print(f"Testing Accuracy: {test_r2_sum*100:.4f}, Testing Loss: {test_loss:.4f}")

     Training Accuracy: 89.8215, Training Loss: 0.1006
     Validation Accuracy: 90.9222, Validation Loss: 0.0938
     Testing Accuracy: 91.0683, Testing Loss: 0.0901
```

Model 4:

Training Accuracy, Training Loss, Validation Accuracy, Validation Loss, Testing Accuracy, and Testing Loss of model 4:

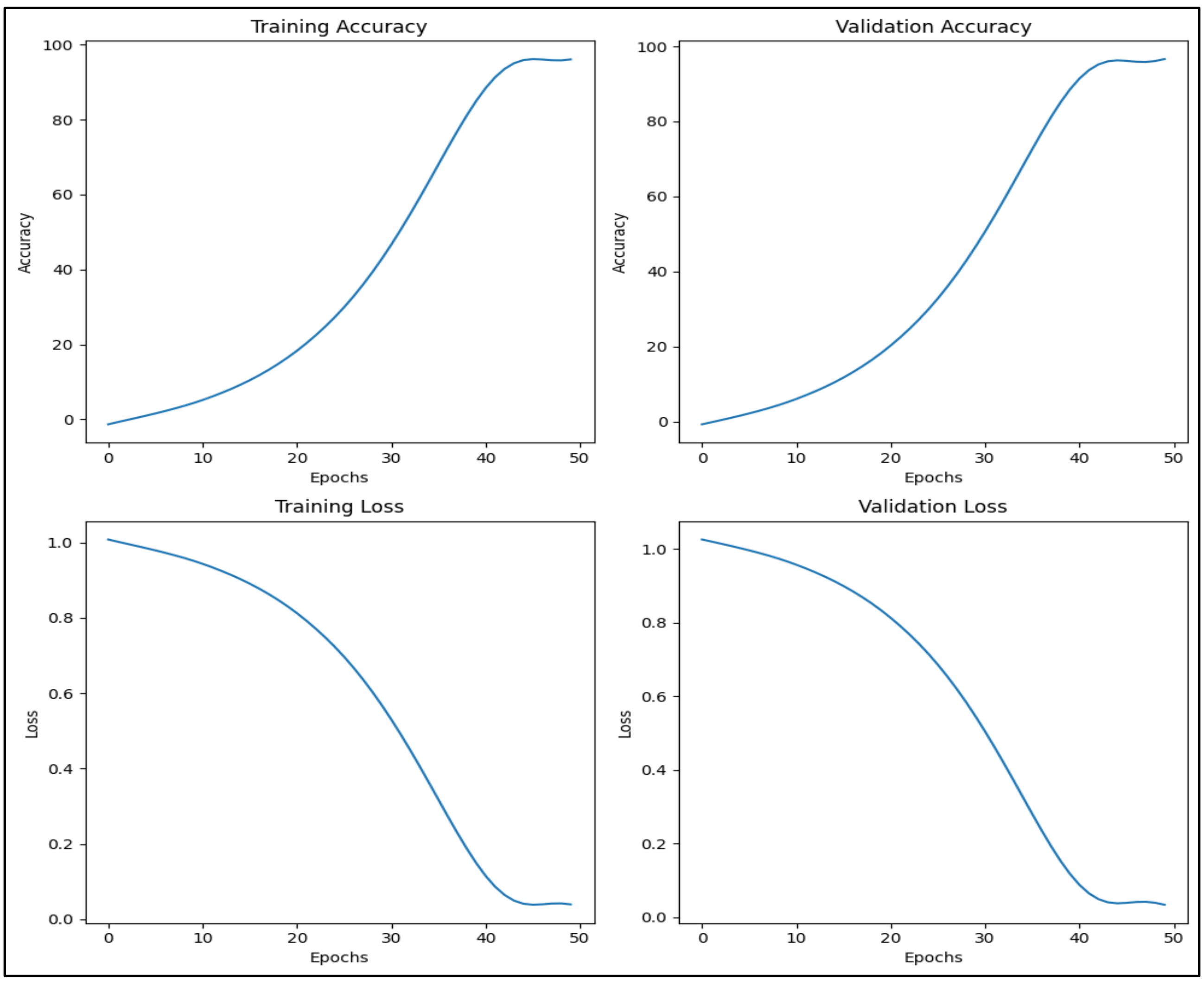
```
[41] print(f"Training Accuracy: {training_r2_score*100:.4f}, Training Loss: {training_loss:.4f}")
     print(f"Validation Accuracy: {validation_r2_score*100:.4f}, Validation Loss: {validation_loss:.4f}")
     print(f"Testing Accuracy: {test_r2_sum*100:.4f}, Testing Loss: {test_loss:.4f}")

     Training Accuracy: 89.3521, Training Loss: 0.1045
     Validation Accuracy: 95.2788, Validation Loss: 0.0475
     Testing Accuracy: 95.3889, Testing Loss: 0.0471
```

b. Plot the training and validation accuracy over time (epochs).

c. Plot the training and validation loss over time (epochs).

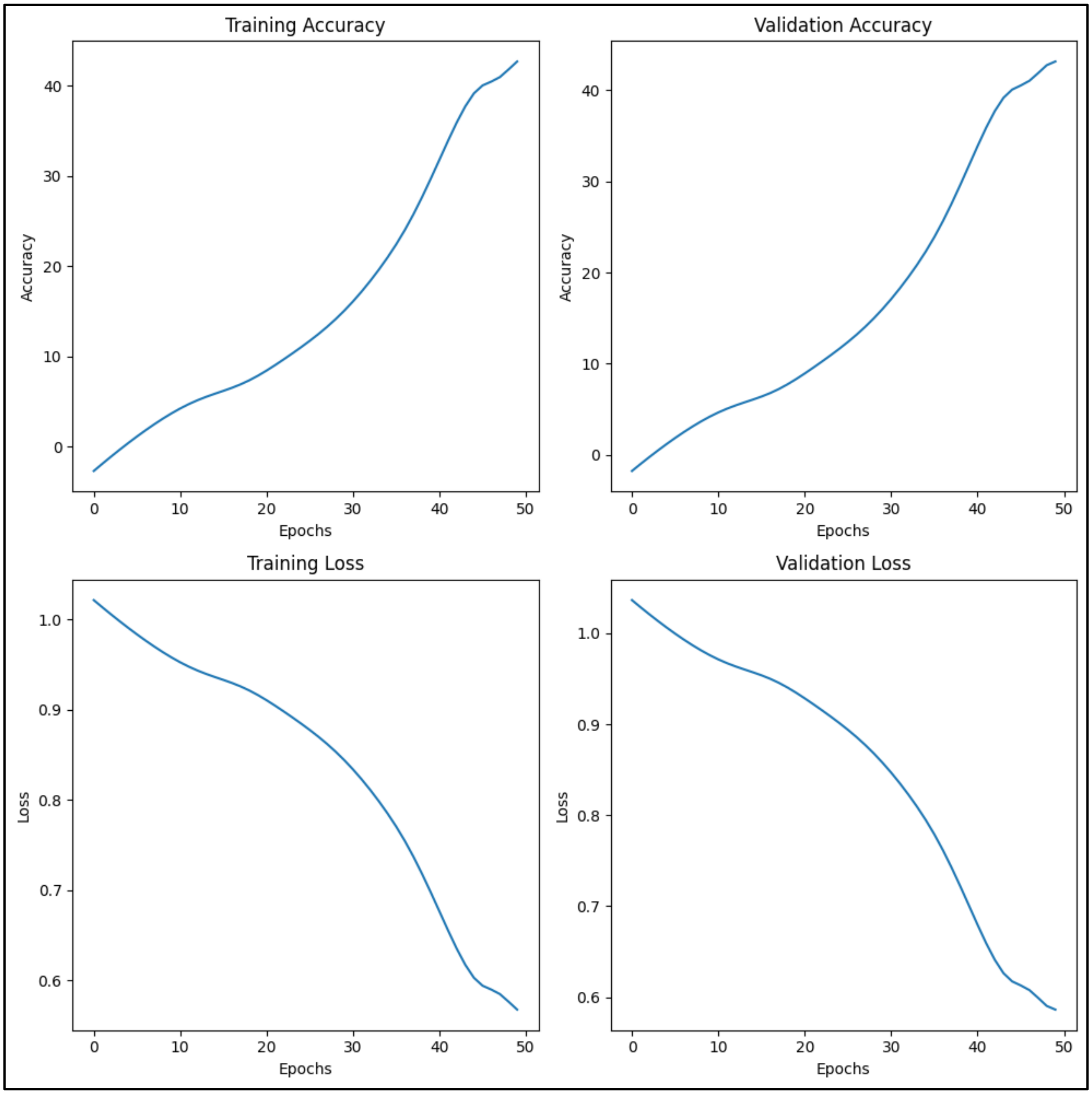
Model 1:

Plots for both part b and c:



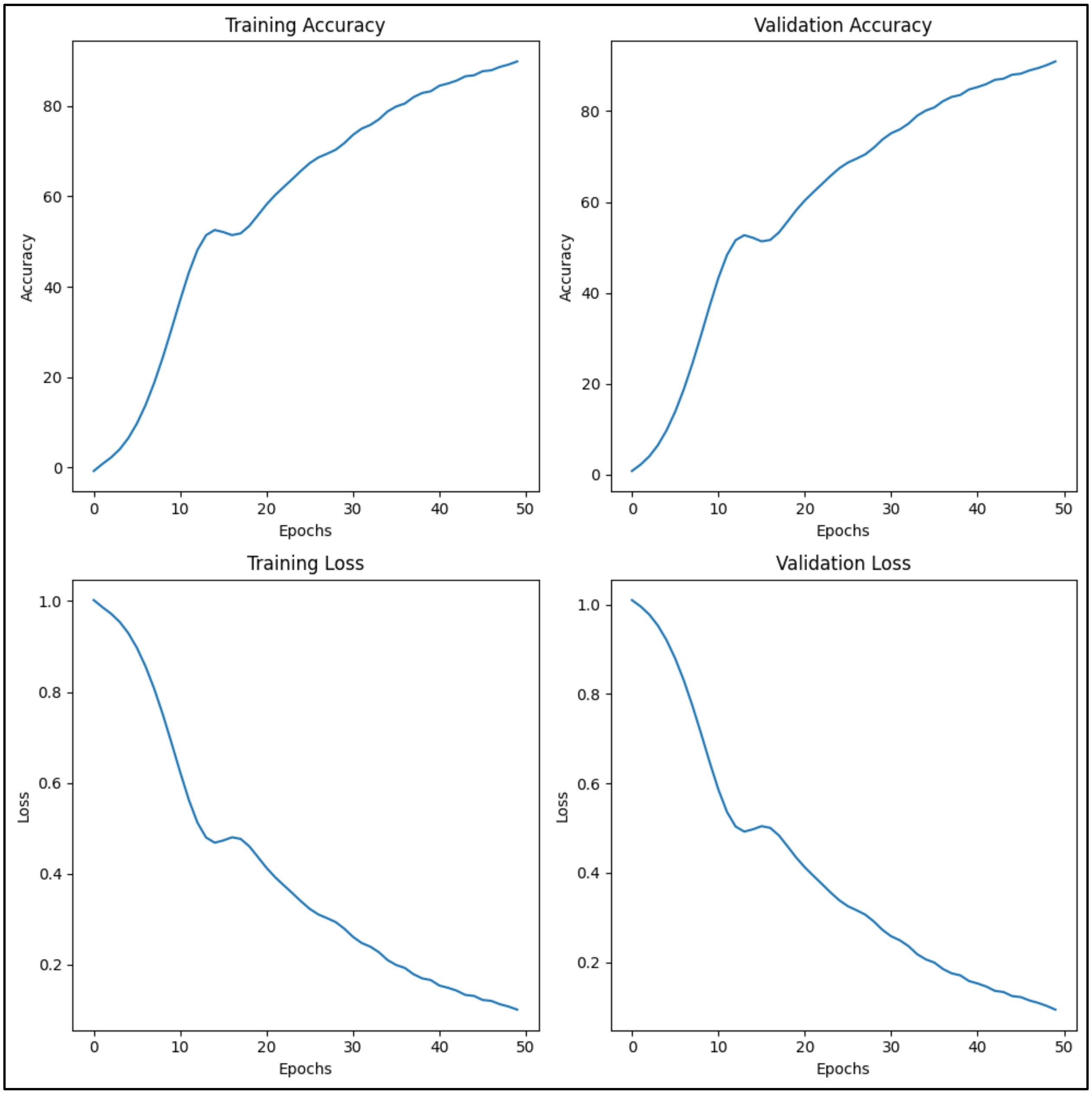
Model 2:

Plots for both part b and c:



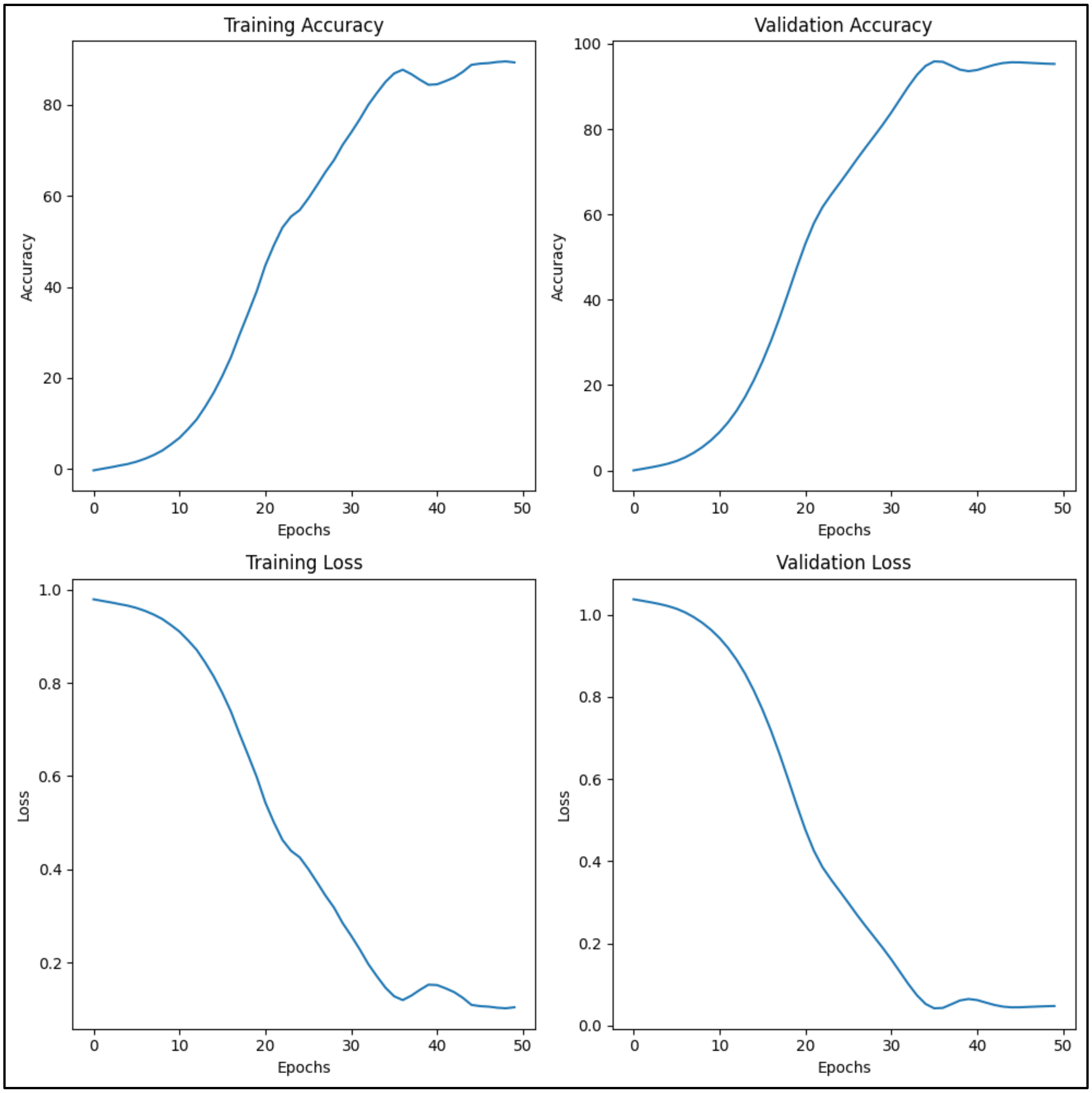
Model 3:

Plots for both part b and c:



Model 4:

Plots for both part b and c:



d. Generate a confusion matrix using the model's predictions on the test set.

Confusion Matrix for Model 1:

```
Confusion Matrix:
 [[17713   310]
 [  935    16]]
```

Confusion Matrix for Model 2:

```
Confusion Matrix:
 [[17782   280]
 [  866    46]]
```

Confusion Matrix for Model 3:

```
Confusion Matrix:
 [[17746   306]
 [  902    20]]
```

Confusion Matrix for Model 4:

```
Confusion Matrix:
 [[17651   339]
 [  979     5]]
```

e. Report any other evaluation metrics used to analyze the model's performance on the test set.

Model 1:

```
threshold = find_threshold(AE_model, X_train_tensor)
print(f"Threshold: {threshold}")
```

```
Threshold: 0.05142716318368912
```

```
[19] preds = get_predictions(AE_model, X_test_tensor, threshold)

# Evaluation
accuracy_test = accuracy_score(preds, Y_test_tensor) * 100
precision_test = precision_score(preds, Y_test_tensor, average='micro')
recall_test = recall_score(preds, Y_test_tensor, average='macro')
f1_test = f1_score(preds, Y_test_tensor, average='macro')
confusion_matrix_test = confusion_matrix(preds, Y_test_tensor)

print(f'Accuracy: {accuracy_test:.3f}')
print(f'Precision: {precision_test:.3f}')
print(f'Recall: {recall_test:.3f}')
print(f'F1 score: {f1_test:.3f}')
print('\nConfusion Matrix: \n', confusion_matrix_test)
```

```
Accuracy: 93.438
Precision: 0.934
Recall: 0.500
F1 score: 0.496

Confusion Matrix:
 [[17713   310]
 [  935    16]]
```

Model 2:

```
threshold = find_threshold(AE_model_2, X_train_tensor)
print(f"Threshold: {threshold}")

preds = get_predictions(AE_model_2, X_test_tensor, threshold)

# Evaluation
accuracy_test = accuracy_score(preds, Y_test_tensor) * 100
precision_test = precision_score(preds, Y_test_tensor, average='micro')
recall_test = recall_score(preds, Y_test_tensor, average='macro')
f1_test = f1_score(preds, Y_test_tensor, average='macro')
confusion_matrix_test = confusion_matrix(preds, Y_test_tensor)

print(f'\nAccuracy: {accuracy_test:.3f}')
print(f'Precision: {precision_test:.3f}')
print(f'Recall: {recall_test:.3f}')
print(f'F1 score: {f1_test:.3f}')
print('\nConfusion Matrix: \n', confusion_matrix_test)
```

```
Threshold: 0.5140899062156676

Accuracy: 93.960
Precision: 0.940
Recall: 0.517
F1 score: 0.522

Confusion Matrix:
 [[17782   280]
 [  866    46]]
```

Model 3:

```
threshold = find_threshold(AE_model_3, X_train_tensor)
print(f"Threshold: {threshold}")

preds = get_predictions(AE_model_3, X_test_tensor, threshold)

# Evaluation
accuracy_test = accuracy_score(preds, Y_test_tensor) * 100
precision_test = precision_score(preds, Y_test_tensor, average='micro')
recall_test = recall_score(preds, Y_test_tensor, average='macro')
f1_test = f1_score(preds, Y_test_tensor, average='macro')
confusion_matrix_test = confusion_matrix(preds, Y_test_tensor)

print(f'Accuracy: {accuracy_test:.3f}')
print(f'Precision: {precision_test:.3f}')
print(f'Recall: {recall_test:.3f}')
print(f'F1 score: {f1_test:.3f}')
print('\nConfusion Matrix: \n', confusion_matrix_test)
```

```
Threshold: 0.08289149105548856
Accuracy: 93.633
Precision: 0.936
Recall: 0.502
F1 score: 0.500

Confusion Matrix:
 [[17746   306]
 [  902    20]]
```

Model 4:

```
[42] threshold = find_threshold(AE_model_4, X_train_tensor)
     print(f"Threshold: {threshold}")

     preds = predictions(AE_model_4, X_test_tensor, threshold)

     # Evaluation
     accuracy_test = accuracy_score(preds, Y_test_tensor) * 100
     precision_test = precision_score(preds, Y_test_tensor, average='micro')
     recall_test = recall_score(preds, Y_test_tensor, average='macro')
     f1_test = f1_score(preds, Y_test_tensor, average='macro')
     confusion_matrix_test = confusion_matrix(preds, Y_test_tensor)

     print(f'Accuracy: {accuracy_test:.3f}')
     print(f'Precision: {precision_test:.3f}')
     print(f'Recall: {recall_test:.3f}')
     print(f'F1 score: {f1_test:.3f}')
     print('\nConfusion Matrix: \n', confusion_matrix_test)
```

```
Threshold: 0.1550538212060928
Accuracy: 93.054
Precision: 0.931
Recall: 0.493
F1 score: 0.486

Confusion Matrix:
 [[17651   339]
 [  979     5]]
```

Observations:

- Model 2 with LSTM architecture achieves the highest accuracy (93.93%), followed by Model 3 (93.63%), Model 1 (93.438%) and then Model 4 (93.05%).
- Model 2 has the highest precision and recall, which indicates that it can correctly identify and capture anomalies. It also shows the highest number of true positives and lowest number of false negatives.

4. Discuss the strengths and limitations of using autoencoders for anomaly detection.

Ans:

Strengths:

- Autoencoders can learn the complex relation within the data without requiring the labeled anomalies data during the training stage. Thus, it is well suited to be deployed in the monitor the real-time data in organizations, such as to monitor their CPU utilization, user activities, streaming etc.
- Autoencoders can also be used to perform dimensionality reduction.
- Autoencoders can be used for different types of data and problem domain.

Limitations:

- Autoencoders may struggle to generalize on unseen data if the data is significantly different from the data the model was trained on.
- Tuning the hyperparameters of the autoencoder models is a challenge to achieve optimal performance.