DSC 275/475: Time Series Analysis and Forecasting (Fall 2022)

Project 3.2 – LSTM-based Auto-encoders

Total points: 50

Submitted by: Aradhya Mathur and Lakshmi Nikhil Goduguluri

INSTRUCTIONS:

• You are welcome to work on this project individually or in teams (up to 2 members in each team max).
• If you plan to use PyTorch, a good resource is to review and modify the example code provided for the problem. We plan to review this example code in class as well.
• As outlined in the beginning of the code, you need to have the "arff2pandas" package to access the data files. For the submission, please make sure to hand in the following:
• A document (PDF, Word etc) that captures your responses to the questions below separately from the code to facilitate grading.
• Your code files and output
• Both team members on team should please submit the work to Blackboard.

Overview

In this project, you will work with LSTM-based autoencoders to classify human heart beats for heart disease diagnosis. The dataset contains 5,000 Time Series examples with 140 timesteps. Each time-series is an ECG or EKG signal that corresponds to a single heartbeat from a single patient with congestive heart failure. An electrocardiogram (ECG or EKG) is a test that checks how your heart is functioning by measuring the electrical activity of the heart. With each heart beat, an electrical impulse (or wave) travels through your heart. This wave causes the muscle to squeeze and pump blood from the heart. There are 5 types of hearbeats (classes) that can be classified: i) Normal (N); ii) R-on-T Premature Ventricular Contraction (R-on-T PVC); iii) Premature Ventricular Contraction (PVC); iv) Supra-ventricular Premature or Ectopic Beat (SP or EB); v) Unclassified Beat (UB). The shape of the time-series and the position of the impulses allows doctors to diagnose these different conditions. For the purposes of this project, we are interested in 2 classes: Normal and Abnormal (which includes class 2-5 above merged).
This is an example of an anomaly detection problem where class imbalance exists, i.e. number of each of the individual positive (abnormal) instances are smaller than the normal case. The autoencoder approach is suited well for such applications of anomaly detection. In anomaly detection, we learn the pattern of a normal process. Anything that does not follow this pattern is classified as an anomaly. For a binary classification of rare events, we can use a similar approach using autoencoders.
A sample code example (in Python) implementation of auto-encoder "AutoEncoders_anomaly_detection_ecg_SAMPLE.py" is provided. Review and run the code and answer the following questions:

2. In the above example, the embedding dimension (i.e. output length of encoder and input length of decoder) was set constant at 8.

In [1]:

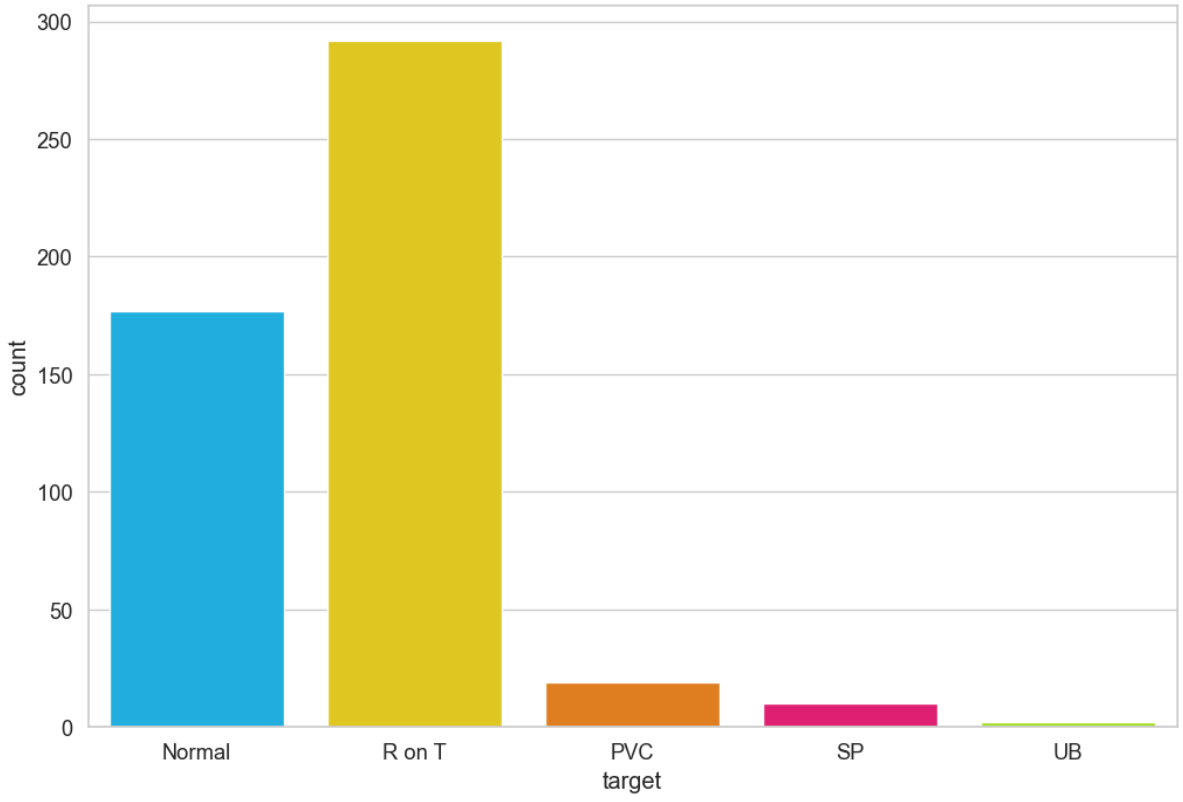
```
"""
# Time Series Anomaly Detection using LSTM Autoencoders with PyTorch in Python

# Uncomment the "pip" commands as necessary to install the packages

# Needed to access the data files
# !pip install -qq arff2pandas

#
# !pip install -q -U watermark

# !pip install -qq -U pandas

# Commented out IPython magic to ensure Python compatibility.
# %reload_ext watermark
# %watermark -v -p numpy,pandas,torch,arff2pandas

# Commented out IPython magic to ensure Python compatibility.
"""
import torch
import warnings
warnings.filterwarnings('ignore')
import copy
import numpy as np
import pandas as pd
import seaborn as sns
from pylab import rcParams
import matplotlib.pyplot as plt
from matplotlib import rc
from sklearn.model_selection import train_test_split

from torch import nn, optim

import torch.nn.functional as F
#from arff2pandas import a2p
from scipy.io import arff

# %matplotlib inline
# %config InlineBackend.figure_format='retina'

sns.set(style='whitegrid', palette='muted', font_scale=1.2)

HAPPY_COLORS_PALETTE = ["#01BEFE", "#FFDD00", "#FF7D00", "#FF006D", "#ADFF02", "#8

sns.set_palette(sns.color_palette(HAPPY_COLORS_PALETTE))

rcParams['figure.figsize'] = 12, 8

RANDOM_SEED = 42
np.random.seed(RANDOM_SEED)
torch.manual_seed(RANDOM_SEED)

"""
In this tutorial, you'll learn how to detect anomalies in Time Series data using ar
You're going to use real-world ECG data from a single patient with heart disease to

# Data
The [dataset](http://timeseriesclassification.com/description.php?Dataset=ECG5000)

> An electrocardiogram (ECG or EKG) is a test that checks how your heart is functio

We have 5 types of hearbeats (classes):
```

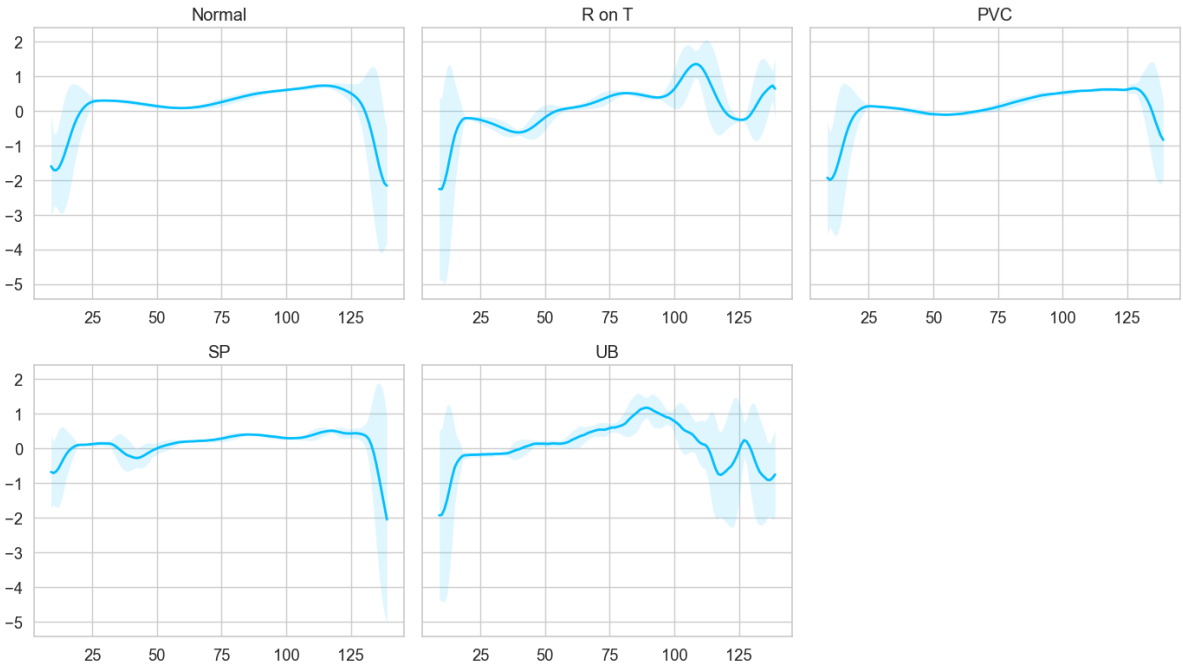
```
- Normal (N)
- R-on-T Premature Ventricular Contraction (R-on-T PVC)
- Premature Ventricular Contraction (PVC)
- Supra-ventricular Premature or Ectopic Beat (SP or EB)
- Unclassified Beat (UB).

> Assuming a healthy heart and a typical rate of 70 to 75 beats per minute, each ca
Frequency: 60–100 per minute (Humans)
Duration: 0.6–1 second (Humans) [Source](https://en.wikipedia.org/wiki/Cardiac_cycl
"""

#Load the arff files into Pandas data frames / Change Path as needed
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

train = arff.loadarff('ECG5000_TRAIN.arff')
test = arff.loadarff('ECG5000_TEST.arff')


#We'll combine the training and test data into a single data frame. This will give

df = pd.DataFrame(train[0])
df = df.sample(frac=1.0)
df['target'] = df['target'].apply(lambda x: str(x.decode()))

df.head()

"""We have 5,000 examples. Each row represents a single heartbeat record. Let's nar

CLASS_NORMAL = 1

class_names = ['Normal','R on T','PVC','SP','UB']

"""Next, we'll rename the last column to `target`, so its easier to reference it:"'

new_columns = list(df.columns)
new_columns[-1] = 'target'
df.columns = new_columns

"""## Exploratory Data Analysis

Let's check how many examples for each heartbeat class do we have:
"""

df.target.value_counts()

"""Let's plot the results:"""

ax = sns.countplot(df.target)
ax.set_xticklabels(class_names);

"""The normal class, has by far, the most examples.

Let's have a look at an averaged (smoothed out with one standard deviation on top a
"""


classes = df.target.unique()

fig, axs = plt.subplots(
  nrows=len(classes) // 3 + 1,
  ncols=3,
  sharey=True,
  figsize=(14, 8)
```

```
)

def plot_time_series_class(data, class_name, ax, n_steps=10):
    time_series_df = pd.DataFrame(data)

    smooth_path = time_series_df.rolling(n_steps).mean()
    path_deviation = 2 * time_series_df.rolling(n_steps).std()

    under_line = (smooth_path - path_deviation)[0]
    over_line = (smooth_path + path_deviation)[0]

    ax.plot(smooth_path, linewidth=2)
    ax.fill_between(
        path_deviation.index,
        under_line,
        over_line,
        alpha=.125
      )
    ax.set_title(class_name)

def create_dataset(df):

    sequences = df.astype(np.float32).to_numpy().tolist()

    dataset = [torch.tensor(s).unsqueeze(1).float() for s in sequences]

    n_seq, seq_len, n_features = torch.stack(dataset).shape

    return dataset, seq_len, n_features
def train_model(model, train_dataset, val_dataset, n_epochs):
    optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
    criterion = nn.L1Loss(reduction='sum').to(device)
    history = dict(train=[], val=[])

    best_model_wts = copy.deepcopy(model.state_dict())
    best_loss = 10000.0

    for epoch in range(1, n_epochs + 1):
        model = model.train()

        train_losses = []
        for seq_true in train_dataset:
            optimizer.zero_grad()

            seq_true = seq_true.to(device)
            seq_pred = model(seq_true)

            loss = criterion(seq_pred, seq_true)

            loss.backward()
            optimizer.step()

            train_losses.append(loss.item())

        val_losses = []
        model = model.eval()
        with torch.no_grad():
            for seq_true in val_dataset:

                seq_true = seq_true.to(device)
                seq_pred = model(seq_true)

                loss = criterion(seq_pred, seq_true)
                val_losses.append(loss.item())
```

```
        train_loss = np.mean(train_losses)
        val_loss = np.mean(val_losses)

        history['train'].append(train_loss)
        history['val'].append(val_loss)

        if val_loss < best_loss:
            best_loss = val_loss
            best_model_wts = copy.deepcopy(model.state_dict())

        print(f'Epoch {epoch}: train loss {train_loss} val loss {val_loss}')

    model.load_state_dict(best_model_wts)
    return model.eval(), history


def predict(model, dataset):
    predictions, losses = [], []
    criterion = nn.L1Loss(reduction='sum').to(device)
    with torch.no_grad():
        model = model.eval()
        for seq_true in dataset:
            seq_true = seq_true.to(device)
            seq_pred = model(seq_true)

            loss = criterion(seq_pred, seq_true)

            predictions.append(seq_pred.cpu().numpy().flatten())
            losses.append(loss.item())
    return predictions, losses

def plot_prediction(data, model, title, ax):
    predictions, pred_losses = predict(model, [data])

    ax.plot(data, label='true')
    ax.plot(predictions[0], label='reconstructed')
    ax.set_title(f'{title} (loss: {np.around(pred_losses[0], 2)})')
    ax.legend()

for i, cls in enumerate(classes):
    ax = axs.flat[i]
    data = df[df.target == cls] \
        .drop(labels='target', axis=1) \
        .mean(axis=0) \
        .to_numpy()
    plot_time_series_class(data, class_names[i], ax)

fig.delaxes(axs.flat[-1])
fig.tight_layout();
plt.show()

## LSTM Autoencoder

### Data Preprocessing

#Let's get all normal heartbeats and drop the target (class) column:

normal_df = df[df.target == str(CLASS_NORMAL)].drop(labels='target', axis=1)
normal_df.shape

#Merge all other classes and mark them as anomalies:"""

anomaly_df = df[df.target != str(CLASS_NORMAL)].drop(labels='target', axis=1)
```

```
anomaly_df.shape

#Split the normal examples into train, validation and test sets:"""

train_df, val_df = train_test_split(
  normal_df,
  test_size=0.15,
  random_state=RANDOM_SEED
)

val_df, test_df = train_test_split(
  val_df,
  test_size=0.33,
  random_state=RANDOM_SEED
)

#Convert our examples into tensors, so we can use them to train our Autoencoder.



#Each Time Series will be converted to a 2D Tensor in the shape *sequence length* x

#Create Train, Val and Test datasets:

train_dataset, seq_len, n_features = create_dataset(train_df)
val_dataset, _, _ = create_dataset(val_df)
test_normal_dataset, _, _ = create_dataset(test_df)
test_anomaly_dataset, _, _ = create_dataset(anomaly_df)
```





In [2]:

```
class Encoder(nn.Module):

    def __init__(self, seq_len, n_features, embedding_dim=64):
        super(Encoder, self).__init__()

        self.seq_len, self.n_features = seq_len, n_features
        #self.embedding_dim, self.hidden_dim = embedding_dim, 2 * embedding_dim
        self.embedding_dim, self.hidden_dim = embedding_dim, embedding_dim

        self.rnn1 = nn.LSTM(
          input_size=n_features,
          hidden_size=self.hidden_dim,
          num_layers=1,
          batch_first=True
        )

    def forward(self, x):
        x = x.reshape((1, self.seq_len, self.n_features))

        #x, (_, _) = self.rnn1(x)
        x, (hidden_n, _) = self.rnn1(x)
        #x, (hidden_n, _) = self.rnn2(x)

        return hidden_n.reshape((self.n_features, self.embedding_dim))

"""The *Encoder* uses LSTM layers to compress the Time Series data input.
Next, we'll decode the compressed representation using a *Decoder*:
"""

class Decoder(nn.Module):

    def __init__(self, seq_len, input_dim=64, n_features=1):
        super(Decoder, self).__init__()

        self.seq_len, self.input_dim = seq_len, input_dim
        # self.hidden_dim, self.n_features = 2 * input_dim, n_features
        self.hidden_dim, self.n_features = input_dim, n_features

        self.rnn1 = nn.LSTM(
          input_size=input_dim,
          hidden_size=input_dim,
          num_layers=1,
```

```
          batch_first=True
        )

        # self.rnn2 = nn.LSTM(
        #   input_size=input_dim,
        #   hidden_size=self.hidden_dim,
        #   num_layers=1,
        #   batch_first=True
        # )

        self.output_layer = nn.Linear(self.hidden_dim, n_features)

    def forward(self, x):
        x = x.repeat(self.seq_len, self.n_features)
        x = x.reshape((self.n_features, self.seq_len, self.input_dim))

        x, (hidden_n, cell_n) = self.rnn1(x)
        #x, (hidden_n, cell_n) = self.rnn2(x)
        x = x.reshape((self.seq_len, self.hidden_dim))

        return self.output_layer(x)

#Our Decoder contains LSTM layer and an output layer that gives the final reconstru

#Time to wrap everything into an easy to use module:


class RecurrentAutoencoder(nn.Module):

    def __init__(self, seq_len, n_features, embedding_dim=64):
        super(RecurrentAutoencoder, self).__init__()

        self.encoder = Encoder(seq_len, n_features, embedding_dim).to(device)
        self.decoder = Decoder(seq_len, embedding_dim, n_features).to(device)

    def forward(self, x):
        x = self.encoder(x)
        x = self.decoder(x)

        return x
```

In [3]:

```
def question2(embed_dim, c=False):
    # LSTM Autoencoder
    #The general Autoencoder architecture consists of two components. An *Encoder*


    """Our Autoencoder passes the input through the Encoder and Decoder. Let's crea

    model = RecurrentAutoencoder(seq_len, n_features, embed_dim)
    model = model.to(device)


    """At each epoch, the training process feeds our model with all training exampl

    Note that we're minimizing the [L1Loss](https://pytorch.org/docs/stable/nn.html

    We'll get the version of the model with the smallest validation error. Let's do
    """

    model, history = train_model(
      model,
      train_dataset,
      val_dataset,
```

```
  n_epochs=25
)

ax = plt.figure().gca()

ax.plot(history['train'])
ax.plot(history['val'])
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['train', 'test'])
plt.title('Loss over training epochs')
plt.show();

## Saving the model

#Let's store the model for later use:


MODEL_PATH = 'model.pth'

torch.save(model, MODEL_PATH)

"""## Choosing a threshold
"""

if c:


    _, losses = predict(model, train_dataset)

    sns.distplot(losses, bins=50, kde=True);

    THRESHOLD = 45

    """## Evaluation

    Using the threshold, we can turn the problem into a simple binary classifi

    - If the reconstruction loss for an example is below the threshold, we'll c
    - Alternatively, if the loss is higher than the threshold, we'll classify i

    ### Normal hearbeats

    Let's check how well our model does on normal heartbeats. We'll use the nor
    """

    predictions, pred_losses = predict(model, test_normal_dataset)
    sns.distplot(pred_losses, bins=50, kde=True);

    """We'll count the correct predictions:"""

    correct = sum(l <= THRESHOLD for l in pred_losses)
    print(f'Correct normal predictions: {correct}/{len(test_normal_dataset)}')

    normal_proportion = correct/len(test_normal_dataset)

    """### Anomalies

    We'll do the same with the anomaly examples, but their number is much highe
    """

    anomaly_dataset = test_anomaly_dataset[:len(test_normal_dataset)]
    #anomaly_dataset = test_anomaly_dataset
    """Now we can take the predictions of our model for the subset of anomalies
```

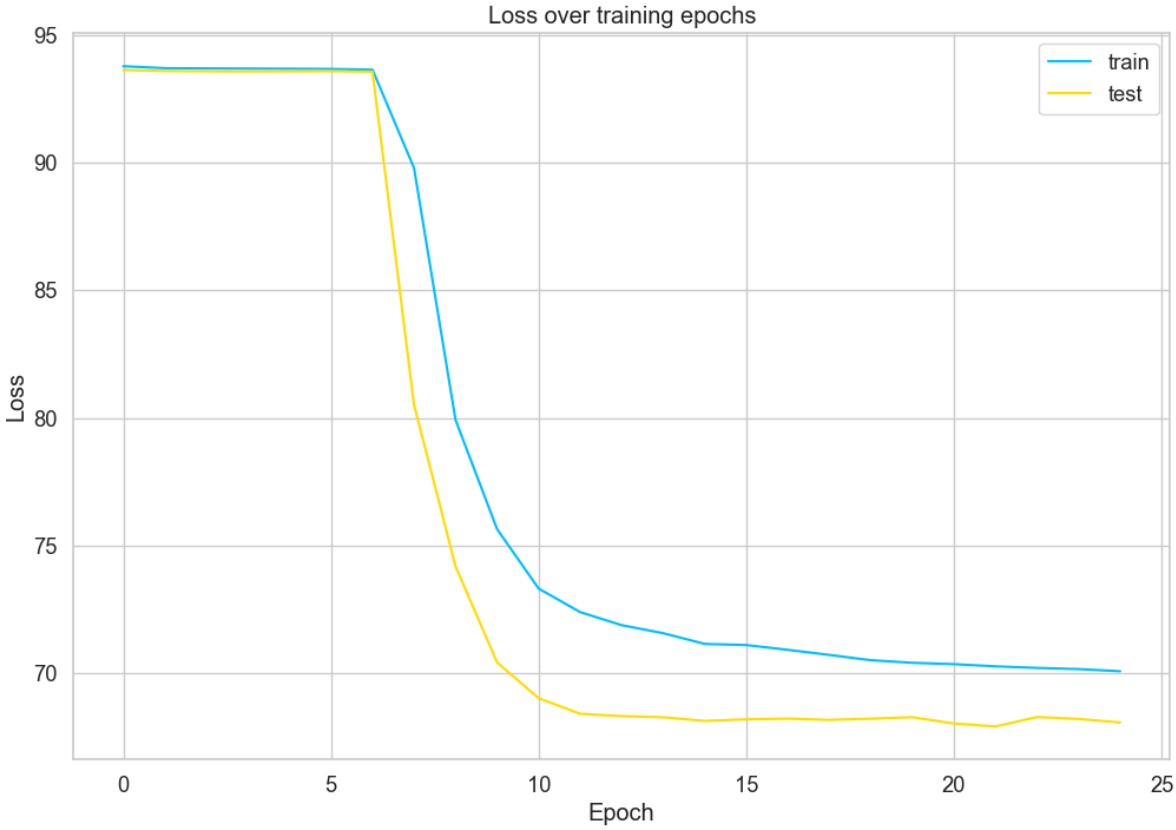
```
    predictions, pred_losses = predict(model, anomaly_dataset)
    sns.distplot(pred_losses, bins=50, kde=True);

    """Finally, we can count the number of examples above the threshold (consid

    correct = sum(l > THRESHOLD for l in pred_losses)
    print(f'Correct anomaly predictions: {correct}/{len(anomaly_dataset)}')

    anomaly_proportion = correct/len(anomaly_dataset)

    #### Looking at Examples

    #We can overlay the real and reconstructed Time Series values to see how cl


    fig, axs = plt.subplots(
      nrows=2,
      ncols=6,
      sharey=True,
      sharex=True,
      figsize=(22, 8)
    )

    for i, data in enumerate(test_normal_dataset[:6]):
        plot_prediction(data, model, title='Normal', ax=axs[0, i])

    for i, data in enumerate(test_anomaly_dataset[:6]):
        plot_prediction(data, model, title='Anomaly', ax=axs[1, i])

    fig.tight_layout();

    return (normal_proportion, anomaly_proportion)
```

a) Embedding dimension length is typically an important hyperparameter that can affect the performance of the technique. Vary the embedding dimension from 2 to 8 in increments of 2 and report the training and validation loss after 25 epochs.

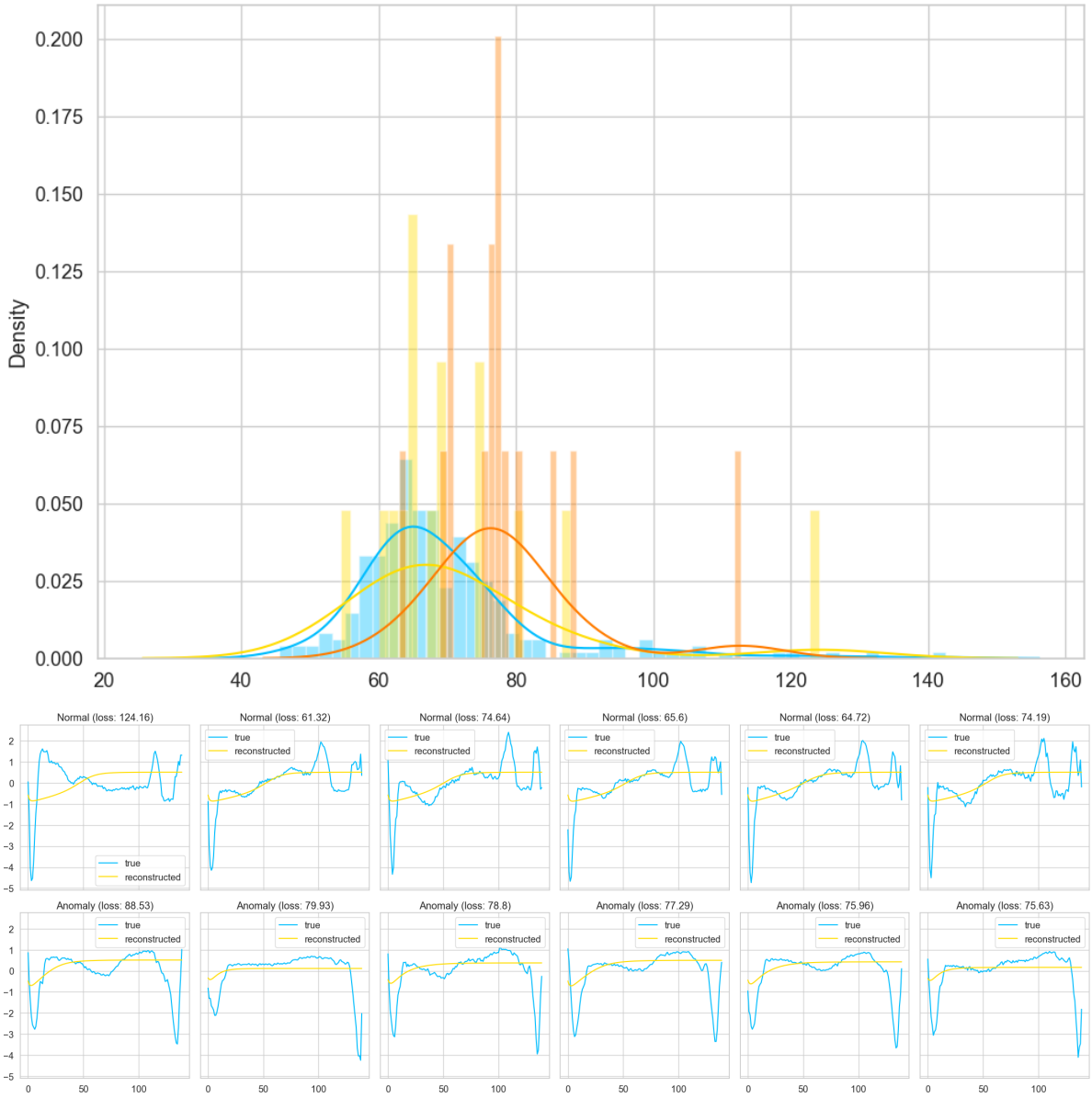
In [4]:
```
proportions = []
```

In [5]:
```
normal, anomaly = question2(2, c=True)
proportions.append([2, normal, anomaly])
```

```
Epoch 1: train loss 93.78667533013129 val loss 93.6387060757341
Epoch 2: train loss 93.7099097467238 val loss 93.59780436548694
Epoch 3: train loss 93.69962569205991 val loss 93.586262406974
Epoch 4: train loss 93.69364098579653 val loss 93.58293704328865
Epoch 5: train loss 93.68742349070888 val loss 93.5848775403253
Epoch 6: train loss 93.67841397562334 val loss 93.58957514269599
Epoch 7: train loss 93.64916423059279 val loss 93.57025093867861
Epoch 8: train loss 89.7936069426998 val loss 80.51026048331425
Epoch 9: train loss 79.91381009932488 val loss 74.17454936586577
Epoch 10: train loss 75.63544637926164 val loss 70.40972755695212
Epoch 11: train loss 73.3068858884996 val loss 69.0105051501044
Epoch 12: train loss 72.38400765388242 val loss 68.39844552401838
Epoch 13: train loss 71.87154297674856 val loss 68.30669192610115
Epoch 14: train loss 71.55706228748444 val loss 68.26220597891972
Epoch 15: train loss 71.1366602989935 val loss 68.11746544673525
Epoch 16: train loss 71.0925189756578 val loss 68.18136241518218
Epoch 17: train loss 70.90508411776635 val loss 68.2085602858971
Epoch 18: train loss 70.70530213079145 val loss 68.15883557549839
Epoch 19: train loss 70.4976746651434 val loss 68.20741100968986
Epoch 20: train loss 70.3968994232916 val loss 68.26303916141904
Epoch 21: train loss 70.3427391359883 val loss 68.01595503708413
Epoch 22: train loss 70.25764409957394 val loss 67.9005909623771
Epoch 23: train loss 70.19567797260899 val loss 68.27037390347185
Epoch 24: train loss 70.14956118983608 val loss 68.19655175044619
Epoch 25: train loss 70.06598649486419 val loss 68.05556606424265
```

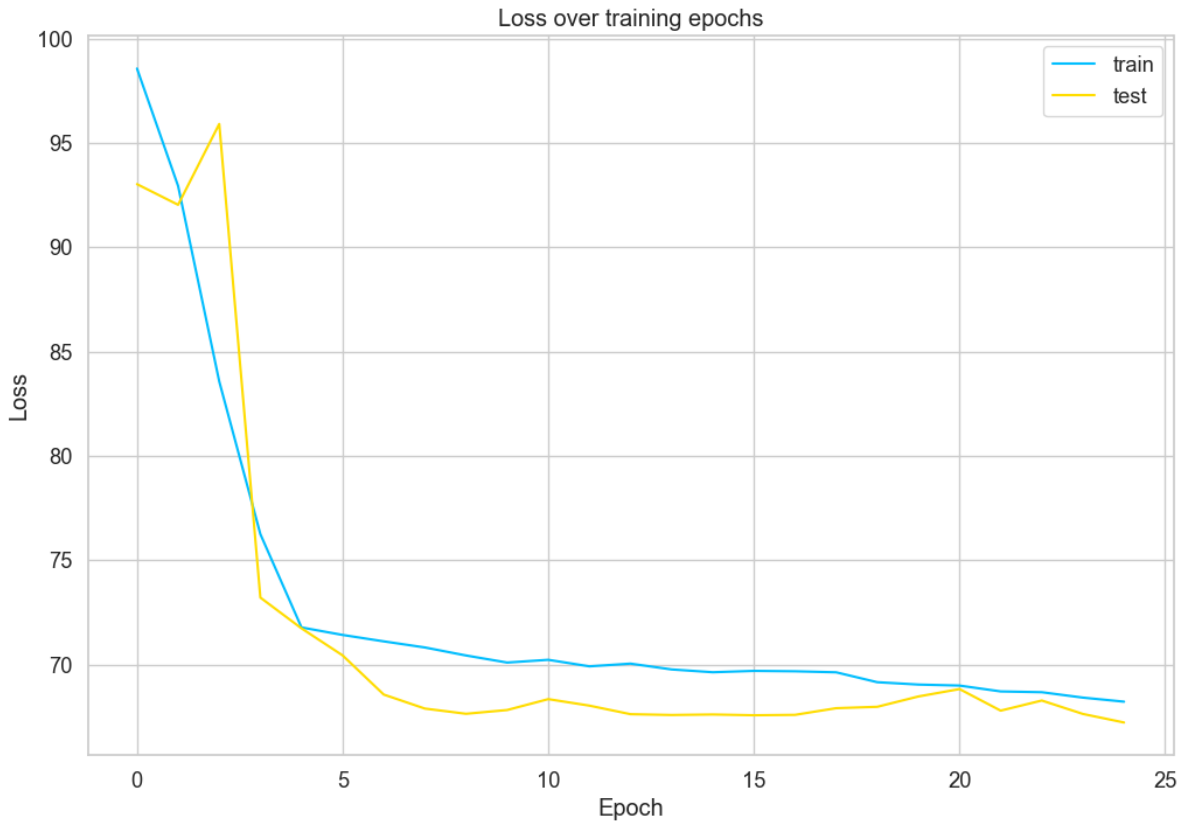



```
Correct normal predictions: 0/15
Correct anomaly predictions: 15/15
```

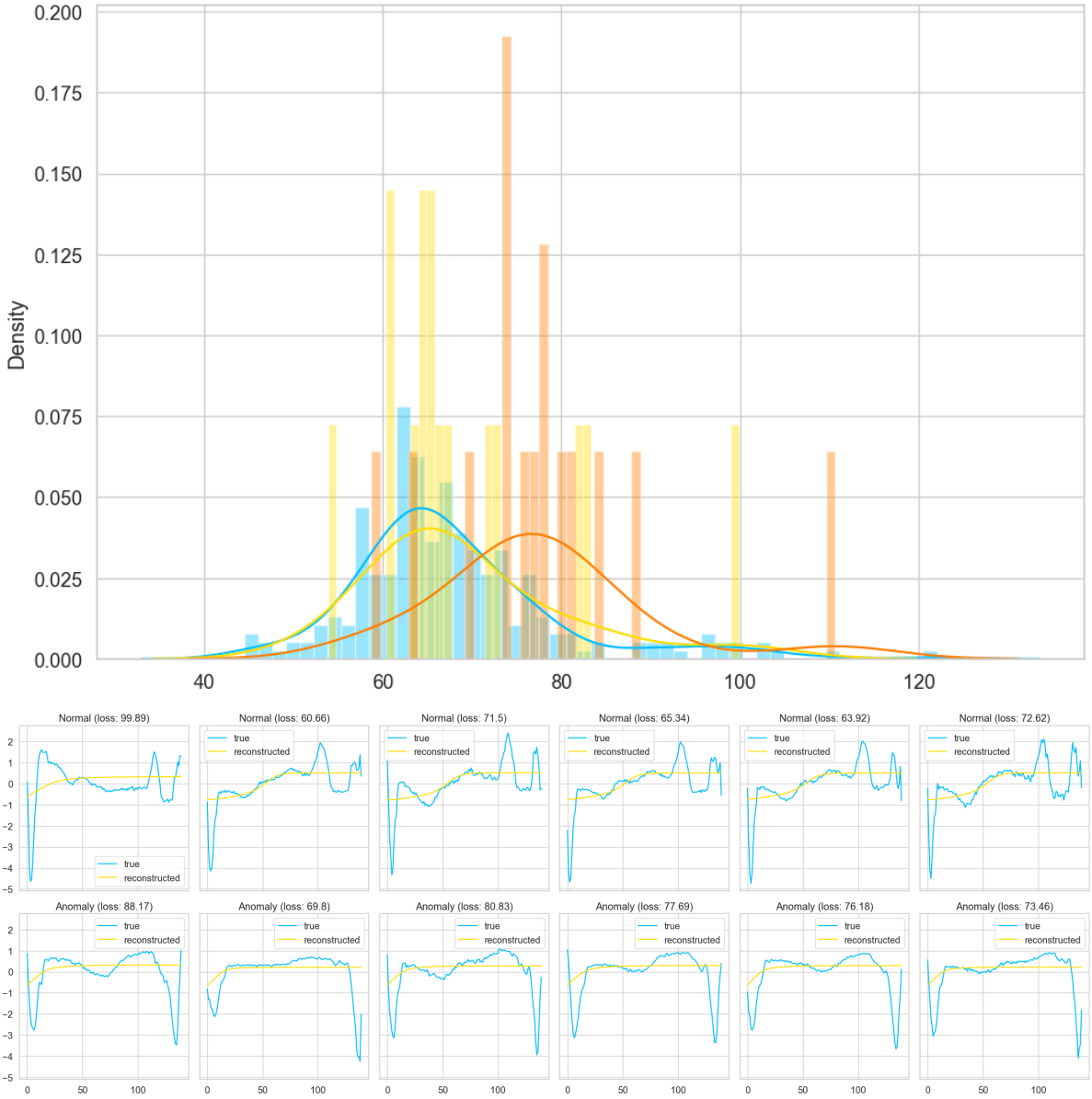


In [6]:
```
normal, anomaly = question2(4, c=True)
proportions.append([4, normal, anomaly])
```

```
Epoch 1: train loss 98.56293198370165 val loss 93.02433224382072
Epoch 2: train loss 92.93235098931098 val loss 92.03415206383015
Epoch 3: train loss 83.57163669217017 val loss 95.90755778345569
Epoch 4: train loss 76.25615609076715 val loss 73.21674215382544
Epoch 5: train loss 71.78726839250133 val loss 71.74287309317754
Epoch 6: train loss 71.42748080530474 val loss 70.44716091813713
Epoch 7: train loss 71.11970544630482 val loss 68.57073501060749
Epoch 8: train loss 70.82772664100894 val loss 67.89994667316306
Epoch 9: train loss 70.44395177595077 val loss 67.6505496584136
Epoch 10: train loss 70.10494818225983 val loss 67.83078173933357
Epoch 11: train loss 70.23590207869007 val loss 68.35265547653725
Epoch 12: train loss 69.92356946391445 val loss 68.04106008595434
Epoch 13: train loss 70.05351791074199 val loss 67.63103024712925
Epoch 14: train loss 69.77457392600274 val loss 67.59328552772259
Epoch 15: train loss 69.64098470441756 val loss 67.61674933597959
Epoch 16: train loss 69.70548666677168 val loss 67.58102548533472
Epoch 17: train loss 69.68636322021484 val loss 67.59925066191575
Epoch 18: train loss 69.63769641999275 val loss 67.91875536688443
Epoch 19: train loss 69.163504985071 val loss 67.98499429636988
Epoch 20: train loss 69.05072272208429 val loss 68.48028498682483
Epoch 21: train loss 69.00244002188406 val loss 68.83496304216057
Epoch 22: train loss 68.72153106812507 val loss 67.80120731222219
Epoch 23: train loss 68.68442266218123 val loss 68.28575831446155
Epoch 24: train loss 68.42456777634159 val loss 67.64249854252256
Epoch 25: train loss 68.23039791660923 val loss 67.23227454876077
```

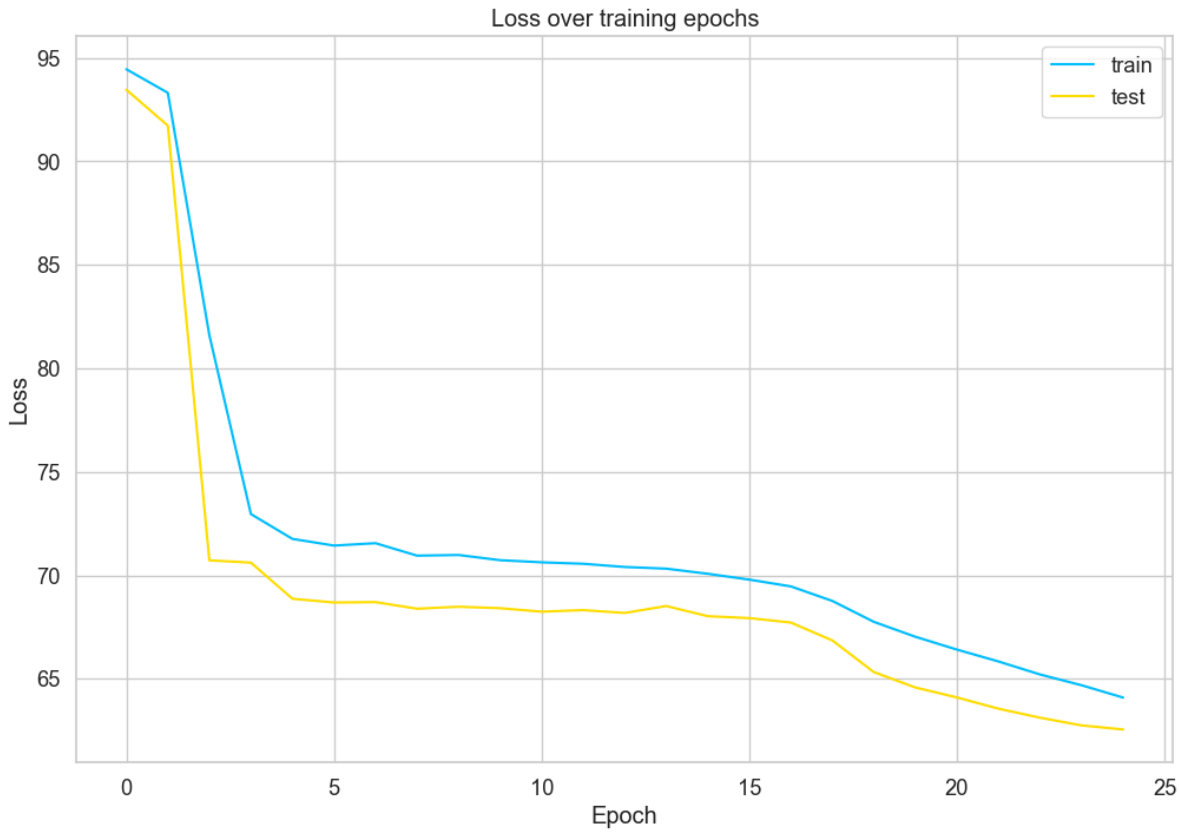



```
Correct normal predictions: 0/15
Correct anomaly predictions: 15/15
```

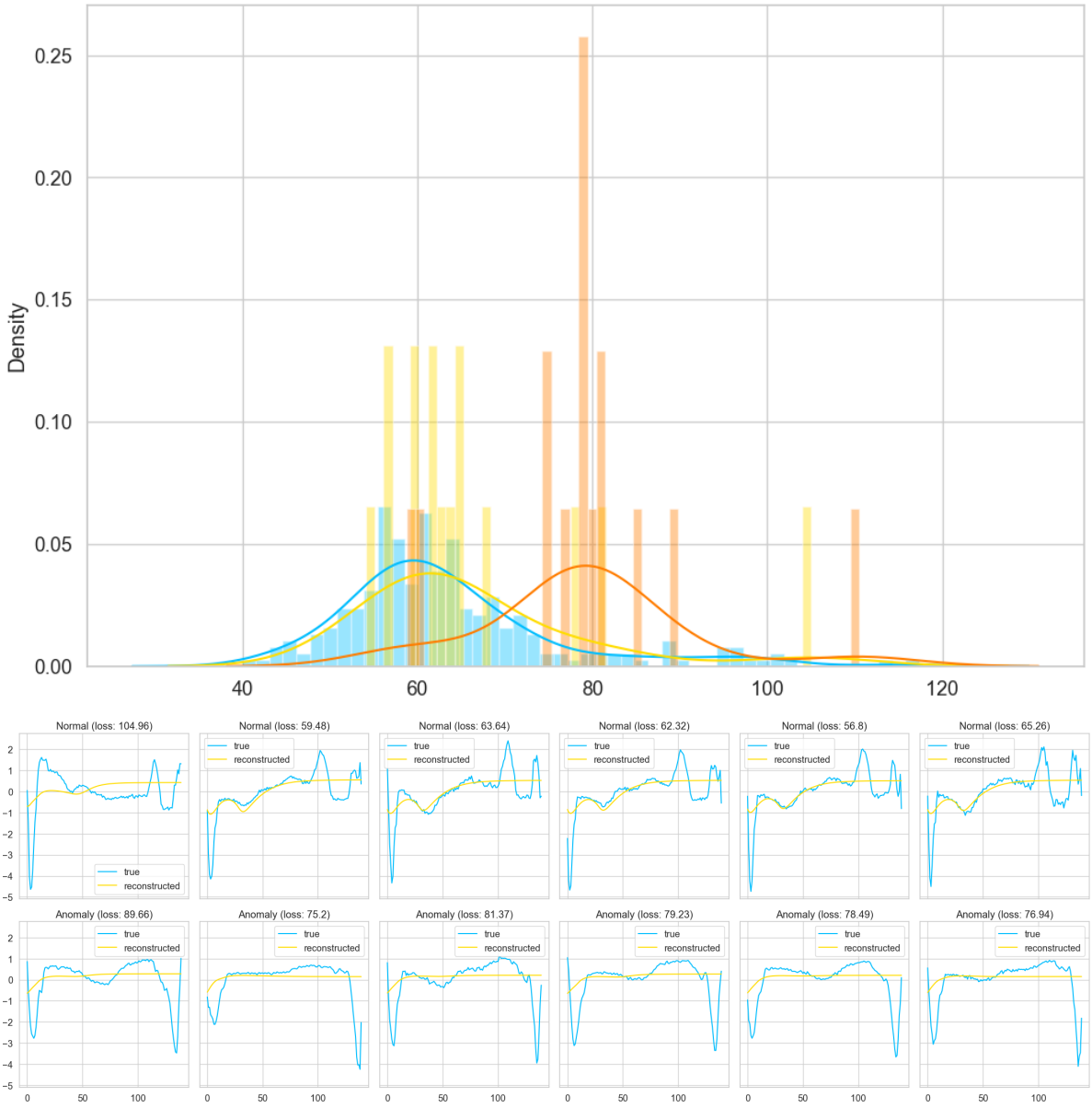


In [7]:
```
normal, anomaly = question2(6, c=True)
proportions.append([6, normal, anomaly])
```

```
Epoch 1: train loss 94.45761089940225 val loss 93.4689036402209
Epoch 2: train loss 93.31762218475342 val loss 91.73227744266904
Epoch 3: train loss 81.59975065723542 val loss 70.72512212292901
Epoch 4: train loss 72.96273043847853 val loss 70.61607808080213
Epoch 5: train loss 71.7624418350958 val loss 68.87083356133823
Epoch 6: train loss 71.4403236604506 val loss 68.68908323090652
Epoch 7: train loss 71.55468351610246 val loss 68.70831969688679
Epoch 8: train loss 70.95356007545224 val loss 68.39016789403455
Epoch 9: train loss 70.98504849403135 val loss 68.48396787972285
Epoch 10: train loss 70.7336403323758 val loss 68.41417680937668
Epoch 11: train loss 70.63093230032152 val loss 68.24523465386753
Epoch 12: train loss 70.56036304658458 val loss 68.3210958283523
Epoch 13: train loss 70.40621014564267 val loss 68.1841750309385
Epoch 14: train loss 70.32429109081146 val loss 68.5190947960163
Epoch 15: train loss 70.0773570153021 val loss 68.02801092739763
Epoch 16: train loss 69.79474458386821 val loss 67.93495743850181
Epoch 17: train loss 69.46783422654674 val loss 67.72052935896248
Epoch 18: train loss 68.76406278917867 val loss 66.85545691128435
Epoch 19: train loss 67.75323404804352 val loss 65.32424098047717
Epoch 20: train loss 67.03241909703901 val loss 64.58208347189016
Epoch 21: train loss 66.41337393176171 val loss 64.10021512261753
Epoch 22: train loss 65.83950774900374 val loss 63.55824463942955
Epoch 23: train loss 65.21161067101264 val loss 63.12285127310917
Epoch 24: train loss 64.69086205574774 val loss 62.75009812979862
Epoch 25: train loss 64.09545456978583 val loss 62.55447374541184
```

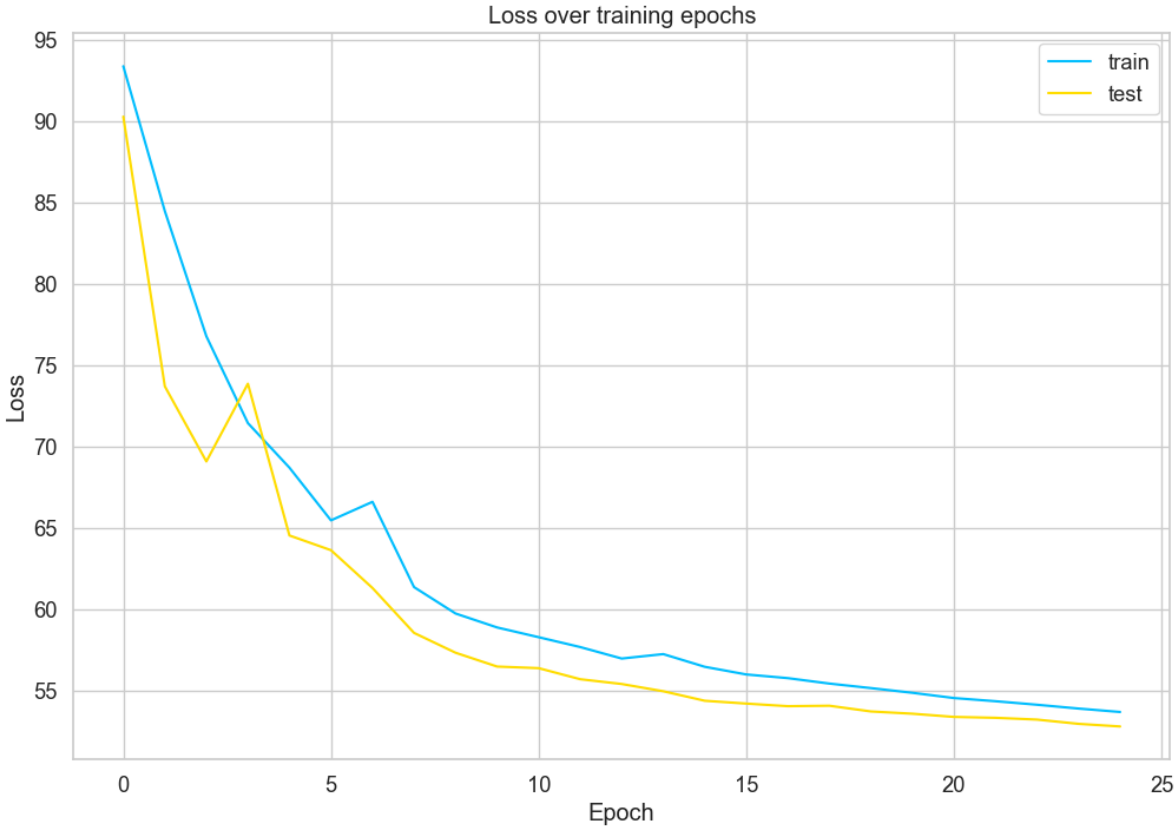



```
Correct normal predictions: 0/15
Correct anomaly predictions: 15/15
```

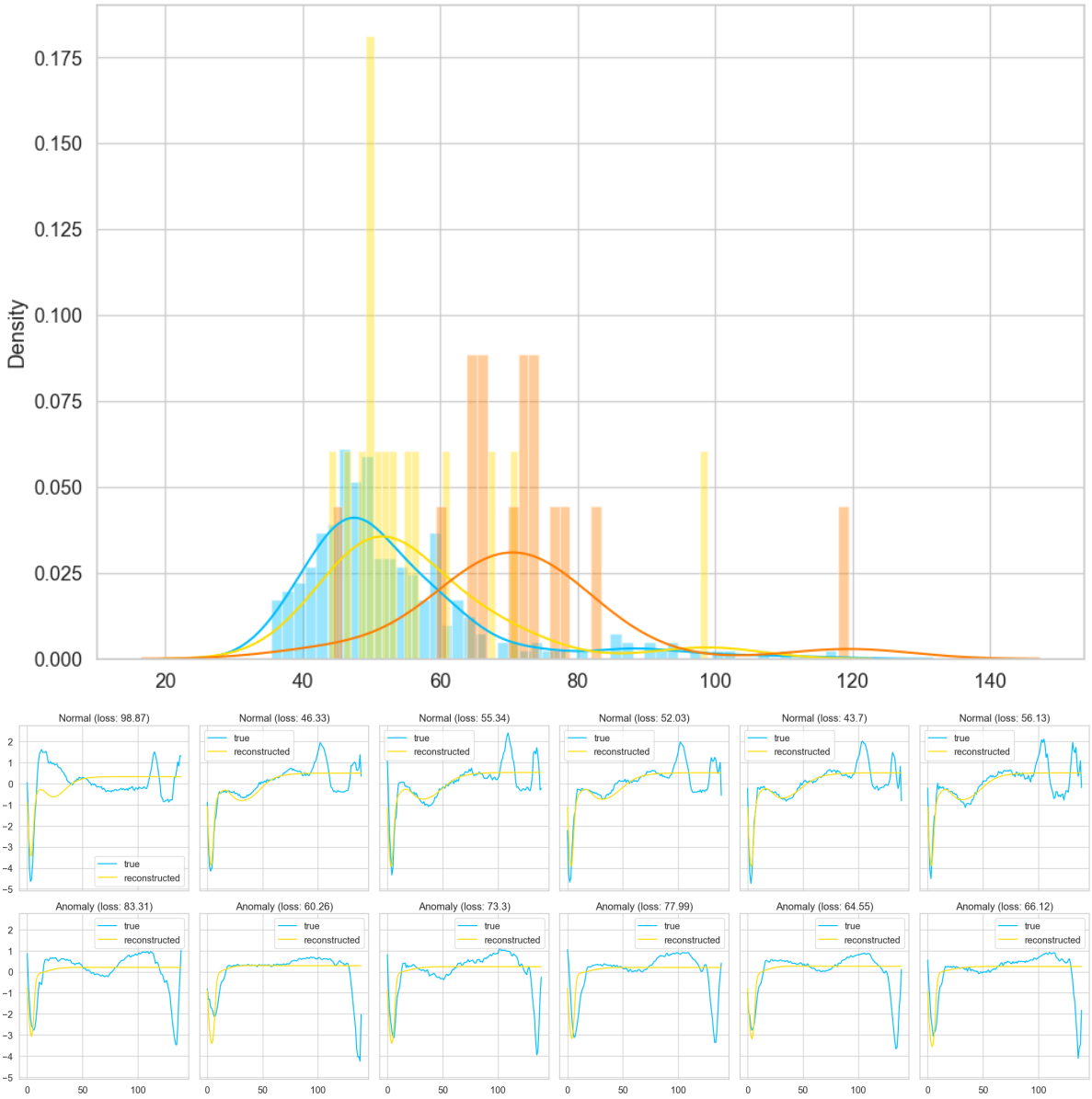


In [8]:
```
normal, anomaly = question2(8, c=True)
proportions.append([8, normal, anomaly])
```

```
Epoch 1: train loss 93.37557989551175 val loss 90.27811537117793
Epoch 2: train loss 84.47186048569218 val loss 73.7037774447737
Epoch 3: train loss 76.77927644791141 val loss 69.08760649582436
Epoch 4: train loss 71.44441541548699 val loss 73.86386555638806
Epoch 5: train loss 68.70537027235954 val loss 64.53788099617793
Epoch 6: train loss 65.46723502682102 val loss 63.63479561641299
Epoch 7: train loss 66.60672355467274 val loss 61.309836815143456
Epoch 8: train loss 61.366446064364524 val loss 58.55190737494107
Epoch 9: train loss 59.74282612339143 val loss 57.332778404498924
Epoch 10: train loss 58.8864051295865 val loss 56.48347157445447
Epoch 11: train loss 58.28807733904931 val loss 56.385160248855065
Epoch 12: train loss 57.68253135681152 val loss 55.706000492490574
Epoch 13: train loss 56.97598481947376 val loss 55.41137642695986
Epoch 14: train loss 57.249237937311975 val loss 54.96666441292599
Epoch 15: train loss 56.468590121115405 val loss 54.381541285021555
Epoch 16: train loss 55.99747074803999 val loss 54.2077794568292
Epoch 17: train loss 55.77182145272532 val loss 54.04855964923727
Epoch 18: train loss 55.43763279145764 val loss 54.07570569268589
Epoch 19: train loss 55.16026609174667 val loss 53.724202780888
Epoch 20: train loss 54.870335809646114 val loss 53.58535621906149
Epoch 21: train loss 54.548709623275265 val loss 53.388390508191335
Epoch 22: train loss 54.35306321420977 val loss 53.33461024843413
Epoch 23: train loss 54.13687233771047 val loss 53.22474210015659
Epoch 24: train loss 53.902127019820675 val loss 52.962251860519935
Epoch 25: train loss 53.69145231862222 val loss 52.80211494708883
```




```
Correct normal predictions: 1/15
Correct anomaly predictions: 14/15
```



b) Briefly explain the trend you see in the training and validation loss

Answer) It can be evidently observed that training and validation loss show decreasing trend with respect to increase in embedding dimension from 2 to 8. Although there is not much of a difference for dimension 2 and dimension 4 but still there is a decrease.

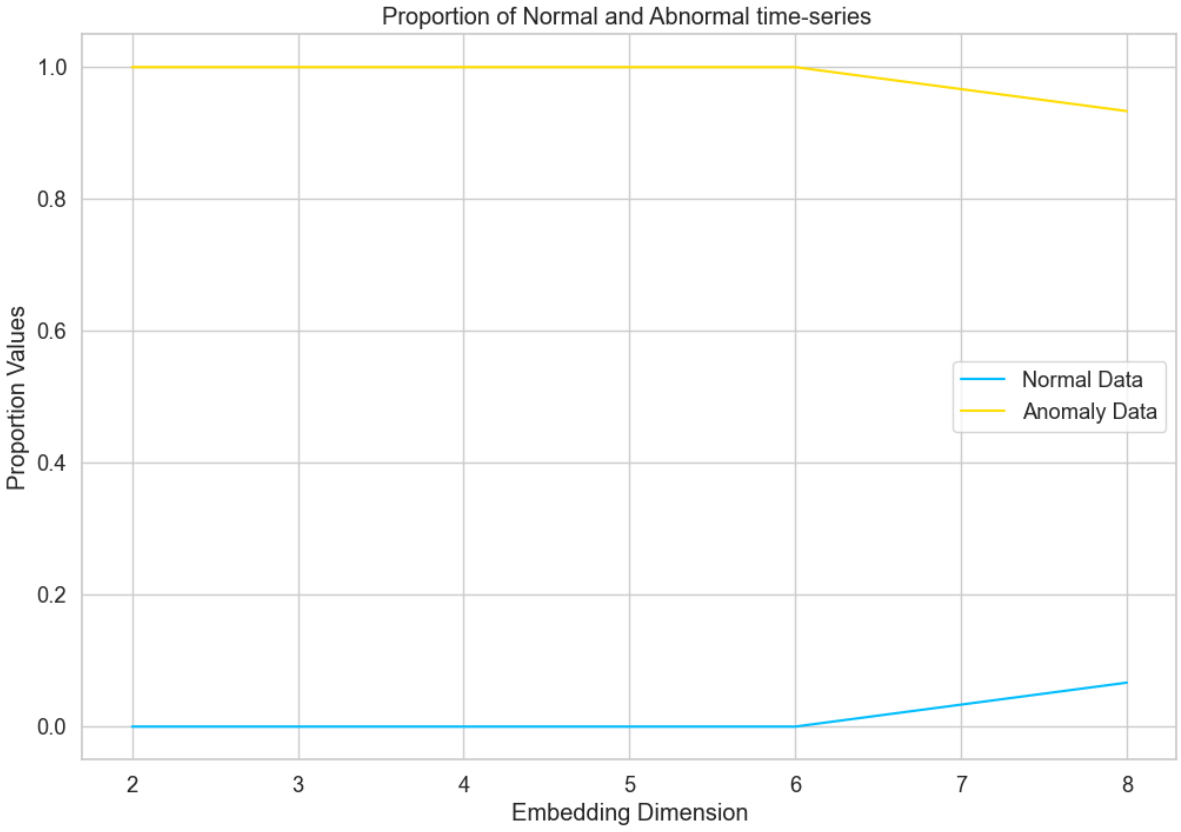
c) Compute the proportion of normal and abnormal time-series correctly classified (i.e. Recall) for the same test set in Q.1 above for each of the embedding dimension values from (a). You can set the threshold to 45.

In [9]:
```
proportions_df = pd.DataFrame(proportions, columns=['Embedding Dimesnion', 'Propor
proportions_df
```

Out[9]:

	Embedding Dimesnion	Proportion of Normal	Proportion of Anomaly
0	2	0.000000	1.000000
1	4	0.000000	1.000000
2	6	0.000000	1.000000
3	8	0.066667	0.933333

In [12]:
```
plt.plot(proportions_df['Embedding Dimesnion'], proportions_df['Proportion of Norma
plt.plot(proportions_df['Embedding Dimesnion'], proportions_df['Proportion of Anoma
plt.title("Proportion of Normal and Abnormal time-series ")
plt.xlabel('Embedding Dimension')
plt.ylabel('Proportion Values')
plt.legend()
plt.show()
```



d) Briefly explain the trend you see in the Recall in part (c) above

Answer) It is evident from the above graph that for constant threshold of 45, proportion of normal and anomaly remains constant 0 and 1 respectively till embedding dimension of 6. For embedding dimension 8, proportion of Normal increases to 0.06667 and proportion of Anomaly decreases to 0.93333.
Ideally, normal value should increase and anomaly value should decrease with respect to increase in embedding layer.
In my opinion, threshold 45 is little less to compare for embedding layer 2,4 and 6. It was clearly observed that normal increases and anomaly decreases with increase in embedding layer from 2 to 8 for threshold 65 (additional file submitted).

In [ ]:
```
```