DSC 275/475: Time Series Analysis and Forecasting (Fall 2023) Project 3.2 – LSTM-based Auto-encoders, Total points: 50

In [7]:
```
import torch

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 scipy.io import arff
import warnings
warnings.filterwarnings("ignore")

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

HAPPY_COLORS_PALETTE = ["#01BEFE", "#FFDD00", "#FF7D00", "#FF006D", "#ADFF02", "#8F00FF"]

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)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

1. A critical hyper-parameter when using auto-encoders is the threshold applied to the reconstructed time-series to classify between normal and abnormal. The default threshold in the code is set to 45. Run the code for 50 epochs.

In [8]:
```
#train
data1 = arff.loadarff(r'/Users/tangsirui/Downloads/ECG5000_TRAIN.arff')
train = pd.DataFrame(data1[0])
train["target"] = train['target'].str.decode("utf-8")
train.head()

#test
data1 = arff.loadarff(r'/Users/tangsirui/Downloads/ECG5000_TEST.arff')
test = pd.DataFrame(data1[0])
test["target"] = test['target'].str.decode("utf-8")
test.head()

#df = train.append(test)
df = pd.concat([train,test])

df = df.sample(frac=1.0)
df.shape

df.head()
```

Out[8]:

	att1	att2	att3	att4	att5	att6	att7	att8	att9	att10	...	att132	att133	att1
1001	1.469756	-1.048520	-3.394356	-4.254399	-4.162834	-3.822570	-3.003609	-1.799773	-1.500033	-1.025095	...	0.945178	1.275588	1.6172
2086	-1.998602	-3.770552	-4.267091	-4.256133	-3.515288	-2.554540	-1.699639	-1.566366	-1.038815	-0.425483	...	1.008577	1.024698	1.0511
2153	-1.187772	-3.365038	-3.695653	-4.094781	-3.992549	-3.425381	-2.057643	-1.277729	-1.307397	-0.623098	...	1.085007	1.467196	1.4138
555	0.604969	-1.671363	-3.236131	-3.966465	-4.067820	-3.551897	-2.582864	-1.804755	-1.688151	-1.025897	...	0.545222	0.649363	0.9868
205	-1.197203	-3.270123	-3.778723	-3.977574	-3.405060	-2.392634	-1.726322	-1.572748	-0.920075	-0.388731	...	0.828168	0.914338	1.0630

5 rows × 141 columns

In [9]:
```
"""We have 5,000 examples. Each row represents a single heartbeat record. Let's name the possible classes:

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()

#ax = sns.countplot(df.target);
"""
ax = sns.countplot(x="target", data=df,order = df['target'].value_counts().index);
ax.set_xticklabels(class_names);

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

classes = df.target.unique()

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

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();
'''
## 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:"""

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

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
)
```

a) For the normal and abnormal test set defined in the code as “test_normal_dataset” and “anomaly_dataset”, vary the threshold value from 15 to 75 (both included) in increments of 10 and report (as a graph or a table) the proportion of normal and abnormal time-series that were correctly classified, i.e. recall. (10 points)

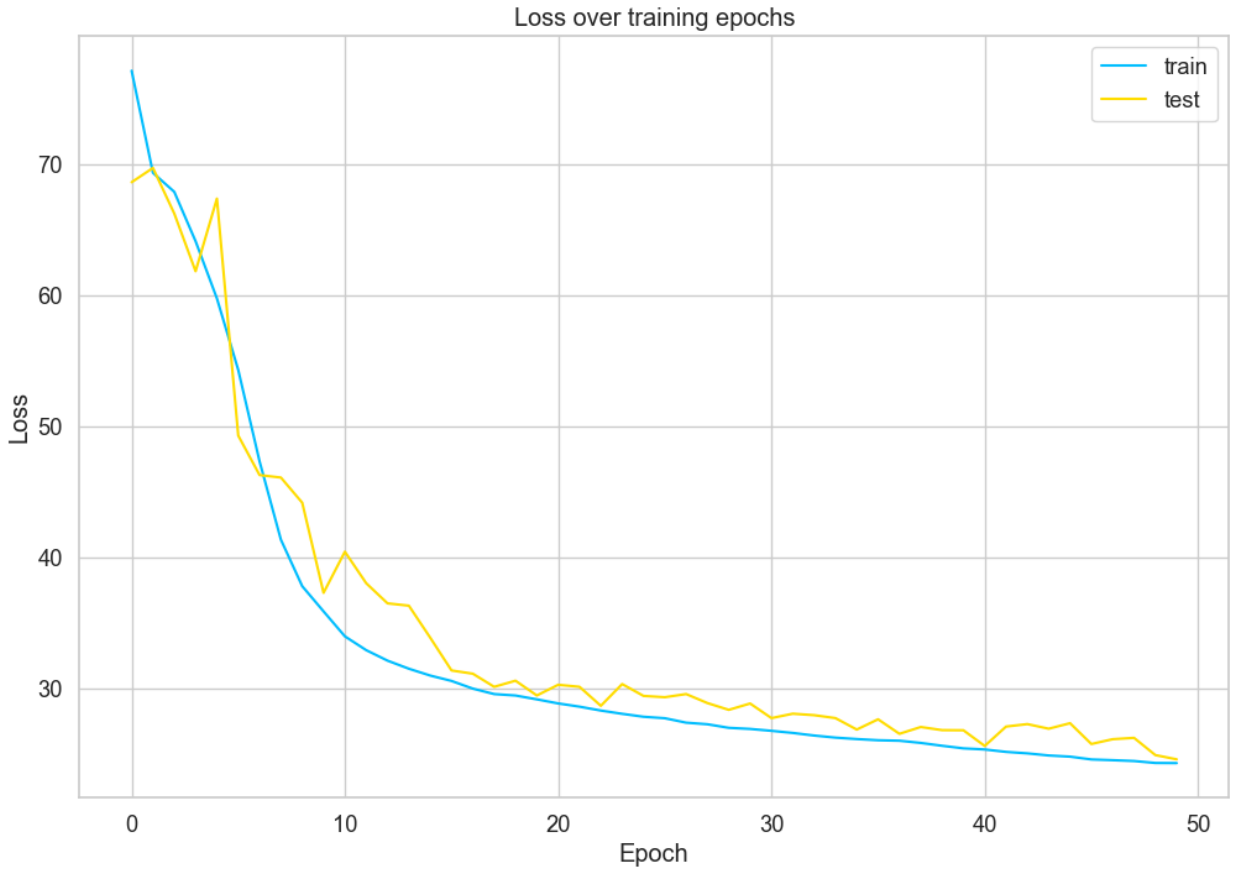
In [10]:
```
reate_dataset(df):

equences = df.astype(np.float32).to_numpy().tolist()
ataset = [torch.tensor(s).unsqueeze(1).float() for s in sequences]

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

eturn dataset, seq_len, n_features

 Time Series will be converted to a 2D Tensor in the shape *sequence length* x *number of features* (140x1

te Train, Val and Test datasets:
_dataset, seq_len, n_features = create_dataset(train_df)
ataset, _, _ = create_dataset(val_df)
normal_dataset, _, _ = create_dataset(test_df)
anomaly_dataset, _, _ = create_dataset(anomaly_df)
```

In [11]:
```
# LSTM Autoencoder
#The general Autoencoder architecture consists of two components.

#An *Encoder* that compresses the input
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))

# *Decoder* that tries to reconstruct it.
"""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 reconstruction.

#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

"""Our Autoencoder passes the input through the Encoder and Decoder. Let's create an instance of it:"""

#model = RecurrentAutoencoder(seq_len, n_features, 128)
```

```
#model = RecurrentAutoencoder(seq_len, n_features, 8)
model = RecurrentAutoencoder(seq_len, n_features, 8)
model = model.to(device)
```

In [12]:
```
## Training

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

model, history = train_model(
  model,
  train_dataset,
  val_dataset,
  n_epochs=50
)

print("Finish training with 50 Epoches on the train dataset")
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)
```

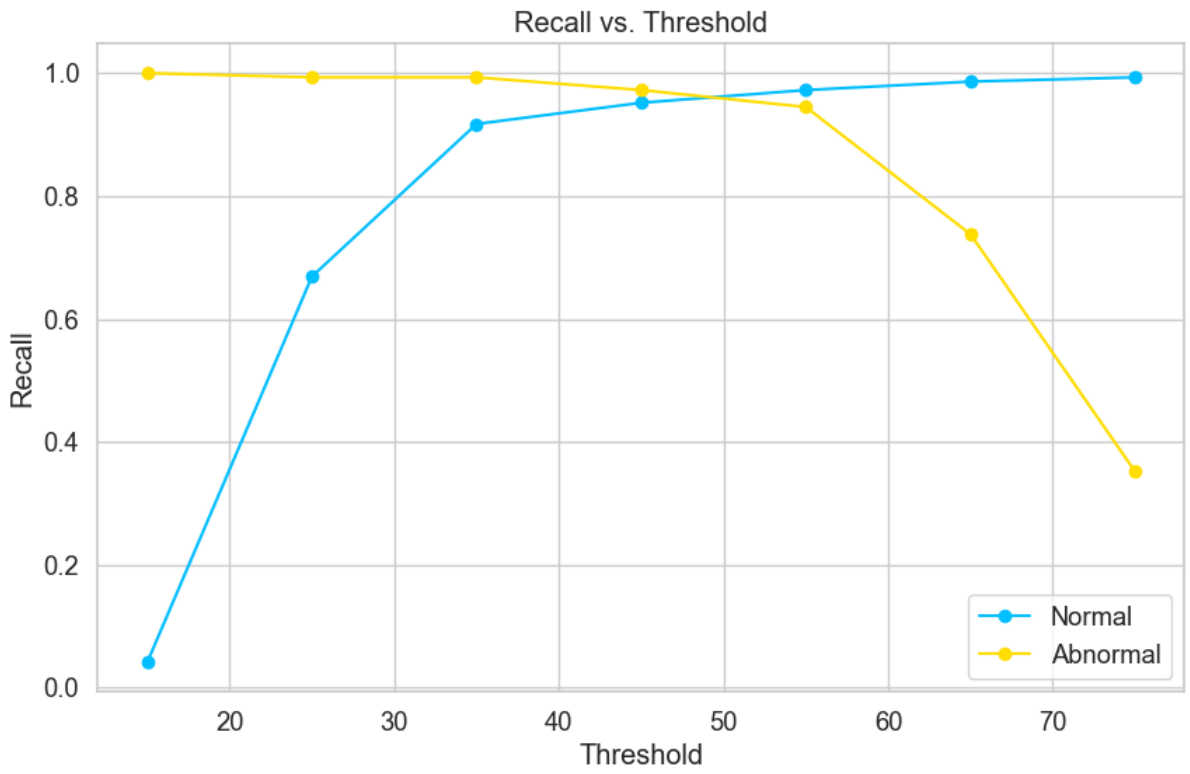
```
Finish training with 50 Epoches on the train dataset
```


Loss over training epochs
train
test
Loss
70
60
50
40
30
0
10
20
30
40
50
Epoch

In [13]:
```
# Choosing a threshold

 our model at hand, we can have a look at the reconstruction error on the training set.
s start by writing a helper function to get predictions from our model:


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

ur function goes through each example in the dataset and records the predictions and losses. Let's get the

osses = predict(model, train_dataset)

s.distplot(losses, bins=50, kde=True);
ll_normal = []
ll_anomaly = []

i in range(15, 76, 10):
# #15 to 75
THRESHOLD = i
print("Threshold: ", i)
## Evaluation
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)
normal = correct/len(test_normal_dataset)
recall_normal.append(normal)
print(f'Correct normal predictions: {correct}/{len(test_normal_dataset)}, {normal}')

### Anomalies
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 (considered as anomalies):"""
correct = sum(l > THRESHOLD for l in pred_losses)
abnormal = correct/len(anomaly_dataset)
recall_anomaly.append(abnormal)
print(f'Correct anomaly predictions: {correct}/{len(anomaly_dataset)}, {abnormal}')
```

```
Threshold:  15
Correct normal predictions: 6/145, 0.041379310344827586
Correct anomaly predictions: 145/145, 1.0
Threshold:  25
Correct normal predictions: 97/145, 0.6689655172413793
Correct anomaly predictions: 144/145, 0.993103448275862
Threshold:  35
Correct normal predictions: 133/145, 0.9172413793103448
Correct anomaly predictions: 144/145, 0.993103448275862
Threshold:  45
Correct normal predictions: 138/145, 0.9517241379310345
Correct anomaly predictions: 141/145, 0.9724137931034482
Threshold:  55
Correct normal predictions: 141/145, 0.9724137931034482
Correct anomaly predictions: 137/145, 0.9448275862068966
Threshold:  65
Correct normal predictions: 143/145, 0.9862068965517241
Correct anomaly predictions: 107/145, 0.7379310344827587
Threshold:  75
Correct normal predictions: 144/145, 0.993103448275862
Correct anomaly predictions: 51/145, 0.35172413793103446
```

In [14]:
```
# def evaluate_threshold(model, test_dataset, anomaly_dataset, threshold):
#     # Evaluate the model on the normal test set
#     predictions_normal, losses_normal = predict(model, test_dataset)
#     correct_normal = sum(l <= threshold for l in losses_normal)
#     recall_normal = correct_normal / len(test_dataset)

#     # Evaluate the model on the anomaly test set
#     predictions_anomaly, losses_anomaly = predict(model, anomaly_dataset)
#     correct_anomaly = sum(l > threshold for l in losses_anomaly)
#     recall_anomaly = correct_anomaly / len(anomaly_dataset)

#     return recall_normal, recall_anomaly

# # Range of threshold values
threshold_values = list(range(15, 76, 10))

# recall_normal_list = []
# recall_anomaly_list = []

# # Iterate over threshold values
# for threshold in threshold_values:
#     print(f"Evaluating threshold: {threshold}")

#     # Evaluate the model for the current threshold
#     recall_normal, recall_anomaly = evaluate_threshold(model, test_normal_dataset, anomaly_dataset, thre
#     print(f"Accuracy of normal and abnormal test set: {recall_normal, recall_anomaly}")
#     # Append the results to the lists
#     recall_normal_list.append(recall_normal)
#     recall_anomaly_list.append(recall_anomaly)

# Plotting the results
plt.figure(figsize=(10, 6))

# Plot normal recall
plt.plot(threshold_values, recall_normal, label='Normal', linestyle='-', marker='o')

# Plot anomaly recall
plt.plot(threshold_values, recall_anomaly, label='Abnormal', linestyle='-', marker='o')

plt.xlabel('Threshold')
plt.ylabel('Recall')
plt.title('Recall vs. Threshold')
plt.legend()
plt.grid(True)
plt.show()
```



b) Briefly explain the trend you see in the recall values as you increase the threshold. (5 points)

According to the graph, we can see that the model tends to be more conservative in classifying instances as abnormal. This can lead to a decrease in the number of true positives and an increase in false negatives, resulting in a decrease in recall for the abnormal class. In contrast, the model becomes more lenient in classifying instances as normal. This can lead to an increase in true negatives and a decrease in false positives, resulting in an increase in recall for the normal class.

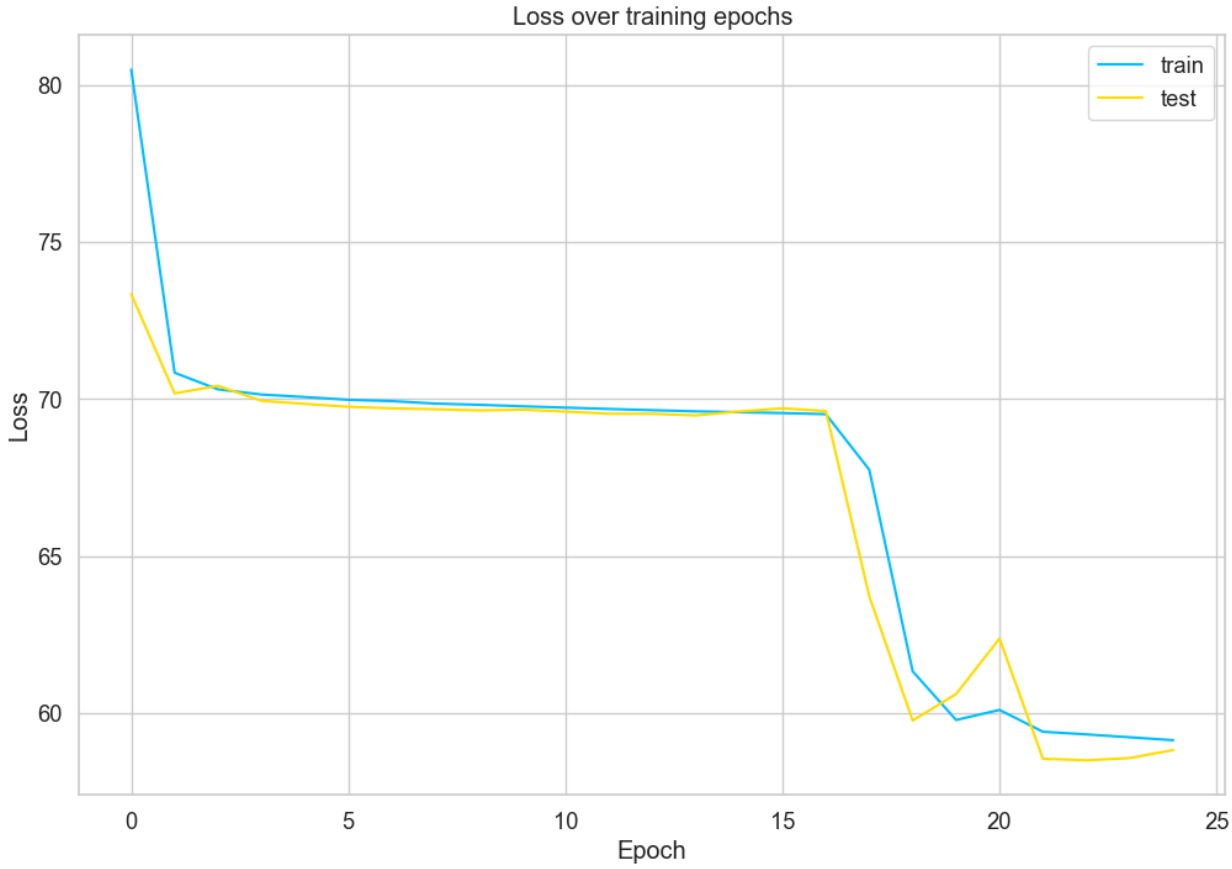
2. In the above example, the embedding dimension (i.e. output length of encoder and input length of decoder) was set constant at 8.

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. (15 points)

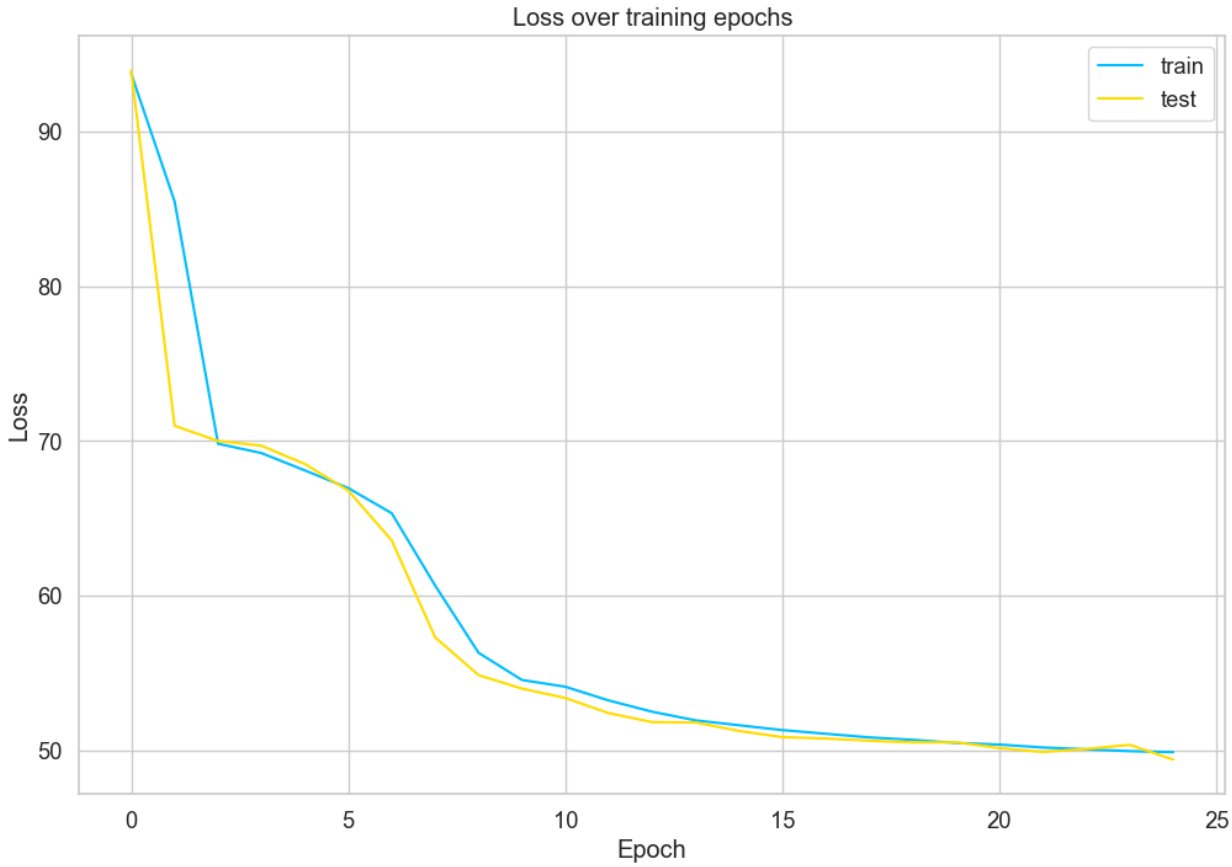
In [23]:
```
model_list = []

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

final_train_losses = []
final_val_losses = []
embedding_dims = list(range(2, 9, 2))

for i in embedding_dims:

    print('Dimension: ', i)
    model = RecurrentAutoencoder(seq_len, n_features, i)
    model = model.to(device)

    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();

    MODEL_PATH = 'model.pth'
    torch.save(model, MODEL_PATH)
    model_list.append(model)

    # Store the final train and validation loss
    final_train_losses.append(history['train'][-1])
```

```
    final_val_losses.append(history['val'][-1])

# Plotting the final train and validation loss for each embedding dimension
plt.figure(figsize=(10, 6))
plt.plot(embedding_dims, final_train_losses, label='Train Loss', marker='*', color='blue')
plt.plot(embedding_dims, final_val_losses, label='Validation Loss', marker='*', color='orange')

# Labeling the axes
plt.title('Train and Validation Loss versus Embedding Dimension')
plt.xlabel('Embedding Dimension')
plt.ylabel('Loss')
plt.legend()
plt.grid(True)
plt.show()
```

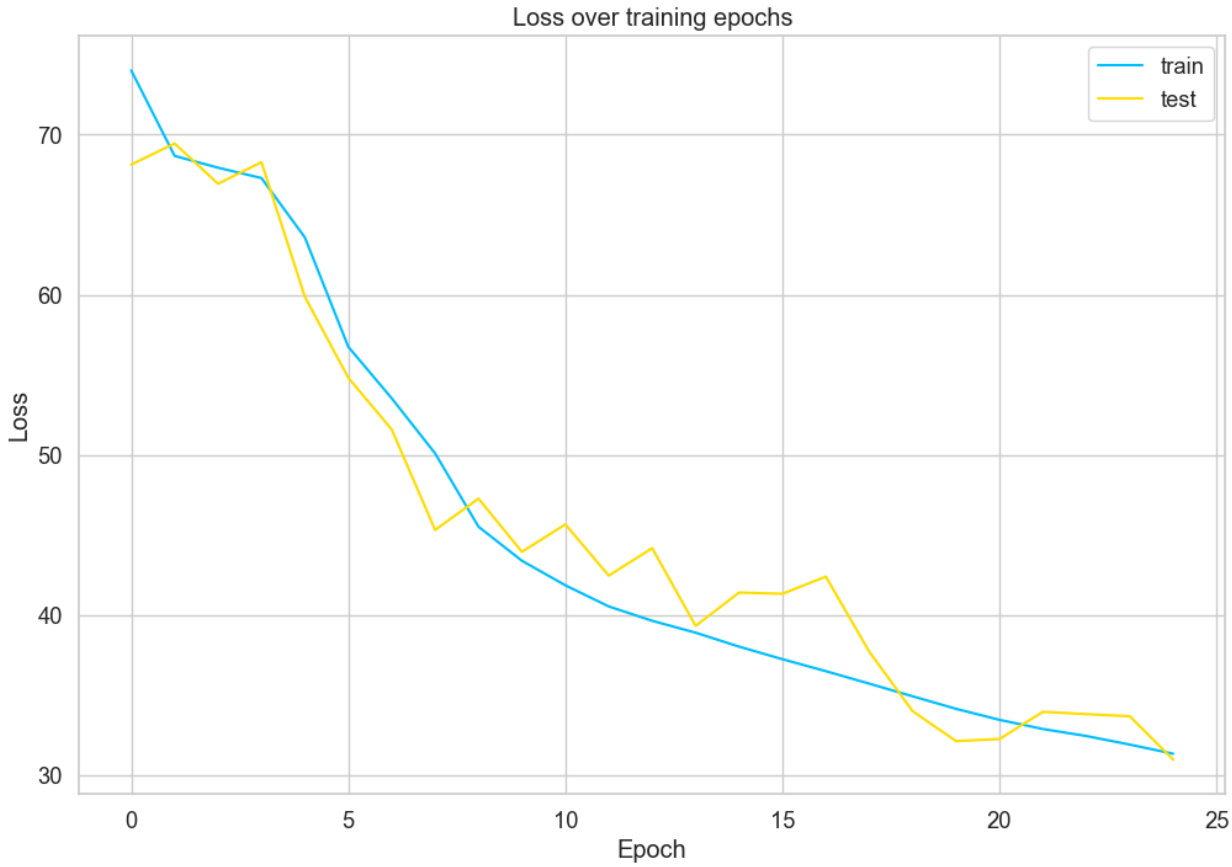
```
Dimension:  2
Epoch 1: train loss 80.48806168272917 val loss 73.34458409071782
Epoch 2: train loss 70.84128534270121 val loss 70.18353061871316
Epoch 3: train loss 70.31151395072767 val loss 70.42497433086304
Epoch 4: train loss 70.14828646658698 val loss 69.9486737951077
Epoch 5: train loss 70.06821205480499 val loss 69.85036420333914
Epoch 6: train loss 69.98122176755008 val loss 69.75853295700541
Epoch 7: train loss 69.93869578689397 val loss 69.70750519684151
Epoch 8: train loss 69.8602324719873 val loss 69.68079055942366
Epoch 9: train loss 69.82224447118904 val loss 69.64214016390335
Epoch 10: train loss 69.7739075534241 val loss 69.66741186929644
Epoch 11: train loss 69.73118553792023 val loss 69.6055789075207
Epoch 12: train loss 69.68886958856056 val loss 69.53623364809837
Epoch 13: train loss 69.6500543458286 val loss 69.53452040636499
Epoch 14: train loss 69.61359624005479 val loss 69.48303397116807
Epoch 15: train loss 69.58560095482996 val loss 69.60829146974322
Epoch 16: train loss 69.55785093880237 val loss 69.70930997828981
Epoch 17: train loss 69.52277523411324 val loss 69.6197628893543
Epoch 18: train loss 67.75538487955237 val loss 63.730141792687945
Epoch 19: train loss 61.34084995981668 val loss 59.77360658352693
Epoch 20: train loss 59.79280853732754 val loss 60.6182904552681
Epoch 21: train loss 60.11270896853193 val loss 62.38283346130579
Epoch 22: train loss 59.41822442389553 val loss 58.55877817043265
Epoch 23: train loss 59.33464093806041 val loss 58.50794301993204
Epoch 24: train loss 59.242106970064775 val loss 58.581583263931016
Epoch 25: train loss 59.14912433309067 val loss 58.83589195798282
```



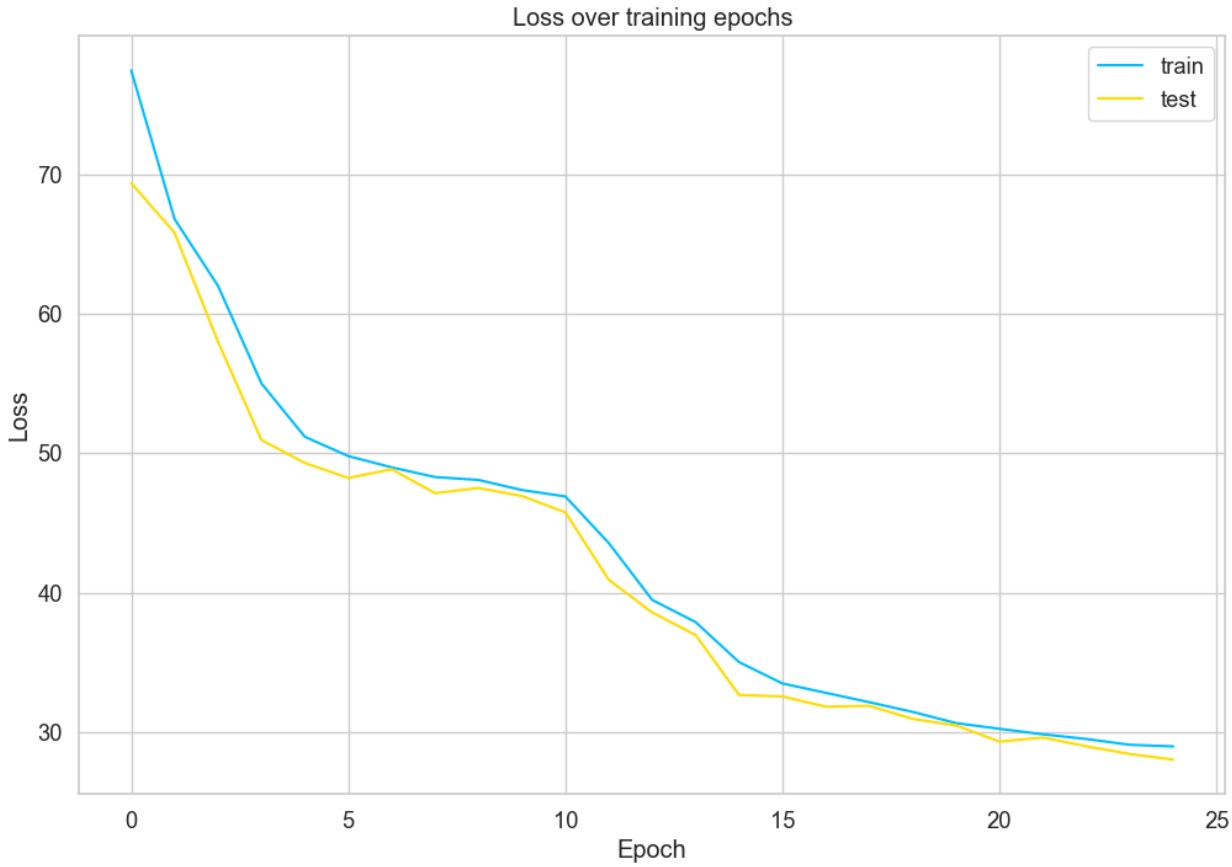
```
Dimension:  4
Epoch 1: train loss 93.72888563679669 val loss 93.93757626790642
Epoch 2: train loss 85.47360529471001 val loss 70.99061213418486
Epoch 3: train loss 69.83265621611208 val loss 70.0101520681544
Epoch 4: train loss 69.22840487577415 val loss 69.70544692680697
Epoch 5: train loss 68.11022252087054 val loss 68.52531524241581
Epoch 6: train loss 66.96925457789503 val loss 66.80255292300072
Epoch 7: train loss 65.34666000667573 val loss 63.59580260657613
Epoch 8: train loss 60.67985441799079 val loss 57.33436200399041
Epoch 9: train loss 56.336138772560865 val loss 54.89379700540276
Epoch 10: train loss 54.57922221671569 val loss 54.022282043821576
Epoch 11: train loss 54.14280606200261 val loss 53.41563833451515
Epoch 12: train loss 53.254938224014474 val loss 52.435505421088415
Epoch 13: train loss 52.53054422027206 val loss 51.859538420068525
Epoch 14: train loss 51.9711464724681 val loss 51.83328215823646
Epoch 15: train loss 51.65614167781954 val loss 51.28775744389348
Epoch 16: train loss 51.34229285560371 val loss 50.88831096297645
Epoch 17: train loss 51.106892201747684 val loss 50.800743793057904
Epoch 18: train loss 50.87187282234755 val loss 50.656853848349925
Epoch 19: train loss 50.71403445528669 val loss 50.54922952749623
Epoch 20: train loss 50.51154550646928 val loss 50.55498371840337
Epoch 21: train loss 50.405868834711185 val loss 50.18415705579947
Epoch 22: train loss 50.219976548944835 val loss 49.93436598533656
Epoch 23: train loss 50.10268510344143 val loss 50.12468944718813
Epoch 24: train loss 49.984367964105324 val loss 50.3904111360934
Epoch 25: train loss 49.91755699179241 val loss 49.43688750022914
```

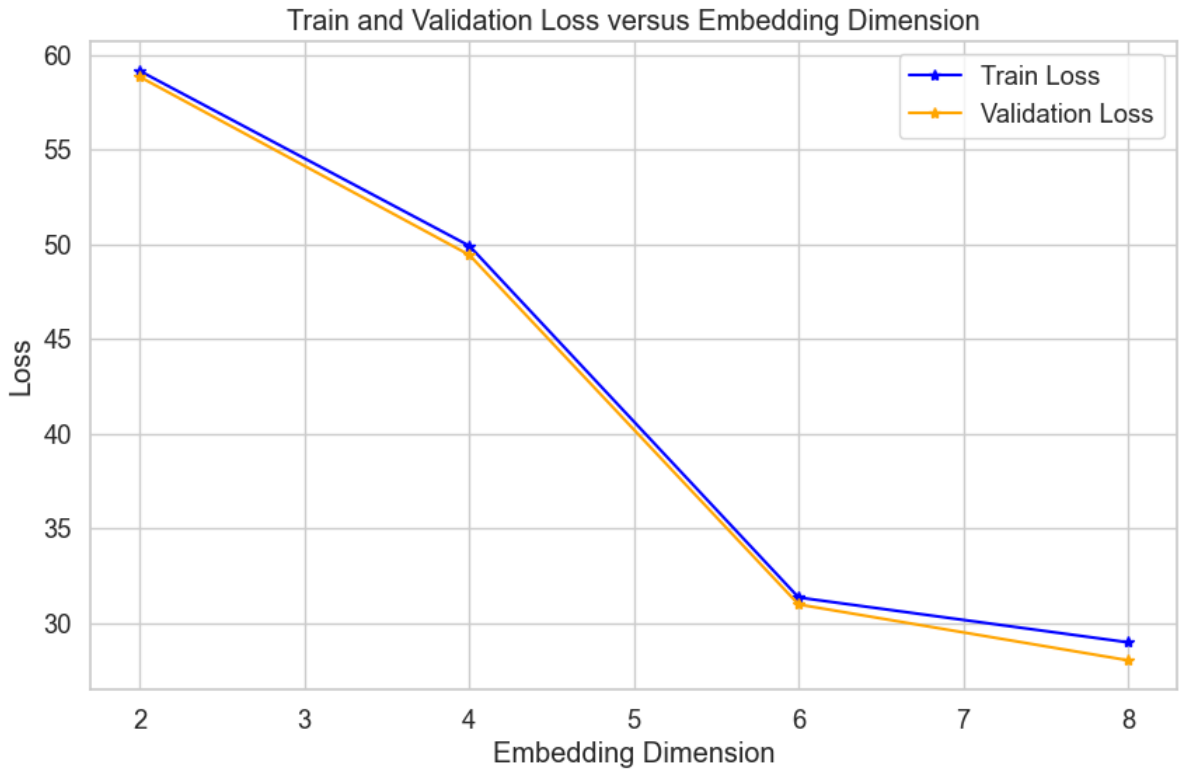


```
Dimension:  6
Epoch 1: train loss 74.00963207369416 val loss 68.1268830803881
Epoch 2: train loss 68.67036604198992 val loss 69.44512820976179
Epoch 3: train loss 67.93264932259586 val loss 66.93109163407986
Epoch 4: train loss 67.28610650938589 val loss 68.27715141374503
Epoch 5: train loss 63.57910777579772 val loss 59.8731815871932
Epoch 6: train loss 56.75061891425862 val loss 54.819888521786844
Epoch 7: train loss 53.53527907218149 val loss 51.58976771644358
Epoch 8: train loss 50.094888375776996 val loss 45.31444285265822
Epoch 9: train loss 45.516371715073625 val loss 47.27050166081243
Epoch 10: train loss 43.388690643279794 val loss 43.946849002773035
Epoch 11: train loss 41.85793856890247 val loss 45.663594763433565
Epoch 12: train loss 40.537808523981475 val loss 42.46275445462901
Epoch 13: train loss 39.6424096786702 val loss 44.176078217021434
Epoch 14: train loss 38.89935107156184 val loss 39.324935984692885
Epoch 15: train loss 38.029639125302744 val loss 41.40203906244792
Epoch 16: train loss 37.24347641563185 val loss 41.32908220909561
Epoch 17: train loss 36.496238699270705 val loss 42.39918214062375
Epoch 18: train loss 35.71817270299688 val loss 37.71382527790786
Epoch 19: train loss 34.926611946840914 val loss 33.99430749196648
Epoch 20: train loss 34.14459934411439 val loss 32.12968131706576
Epoch 21: train loss 33.45675312266721 val loss 32.2594453401533
Epoch 22: train loss 32.8852912137125 val loss 33.95901569327397
Epoch 23: train loss 32.447807181318744 val loss 33.814186014819875
Epoch 24: train loss 31.91141751830206 val loss 33.6810411642029
Epoch 25: train loss 31.342119381822727 val loss 30.973987520758204
```



```
Dimension:  8
Epoch 1: train loss 77.47917250787381 val loss 69.39843561218053
Epoch 2: train loss 66.80659279598434 val loss 65.8230228619364
Epoch 3: train loss 62.015703929330684 val loss 58.00090279432694
Epoch 4: train loss 55.01643993680586 val loss 50.95640074433727
Epoch 5: train loss 51.187186535209094 val loss 49.30882269693316
Epoch 6: train loss 49.80315115566169 val loss 48.231431603024845
Epoch 7: train loss 48.994934486410685 val loss 48.857563624203
Epoch 8: train loss 48.2987650925476 val loss 47.14318938792362
Epoch 9: train loss 48.09240965437668 val loss 47.51172398381673
Epoch 10: train loss 47.36665696878676 val loss 46.938771856522806
Epoch 11: train loss 46.90965001614811 val loss 45.767253211740744
Epoch 12: train loss 43.56128109966925 val loss 40.92747929804154
Epoch 13: train loss 39.48453522266663 val loss 38.57967376058012
Epoch 14: train loss 37.892829067041106 val loss 36.964159369875546
Epoch 15: train loss 35.021828130578285 val loss 32.662560199307904
Epoch 16: train loss 33.48787307047354 val loss 32.55873153803699
Epoch 17: train loss 32.81623338932282 val loss 31.8198632835935
Epoch 18: train loss 32.14272832197799 val loss 31.883112041616602
Epoch 19: train loss 31.447821254645657 val loss 30.946204218034453
Epoch 20: train loss 30.643132501530676 val loss 30.471285165780234
Epoch 21: train loss 30.23452000616059 val loss 29.31863470044966
Epoch 22: train loss 29.83935671379277 val loss 29.603839359999515
Epoch 23: train loss 29.5043109730817 val loss 28.980425212977284
Epoch 24: train loss 29.08684157517974 val loss 28.42815722377634
Epoch 25: train loss 28.97410983416593 val loss 28.024176379519517
```





In [20]:
```
print(type(model_list[0]))
```

```
<class '__main__.RecurrentAutoencoder'>
```

b) Briefly explain the trend you see in the training and validation loss (5 points)

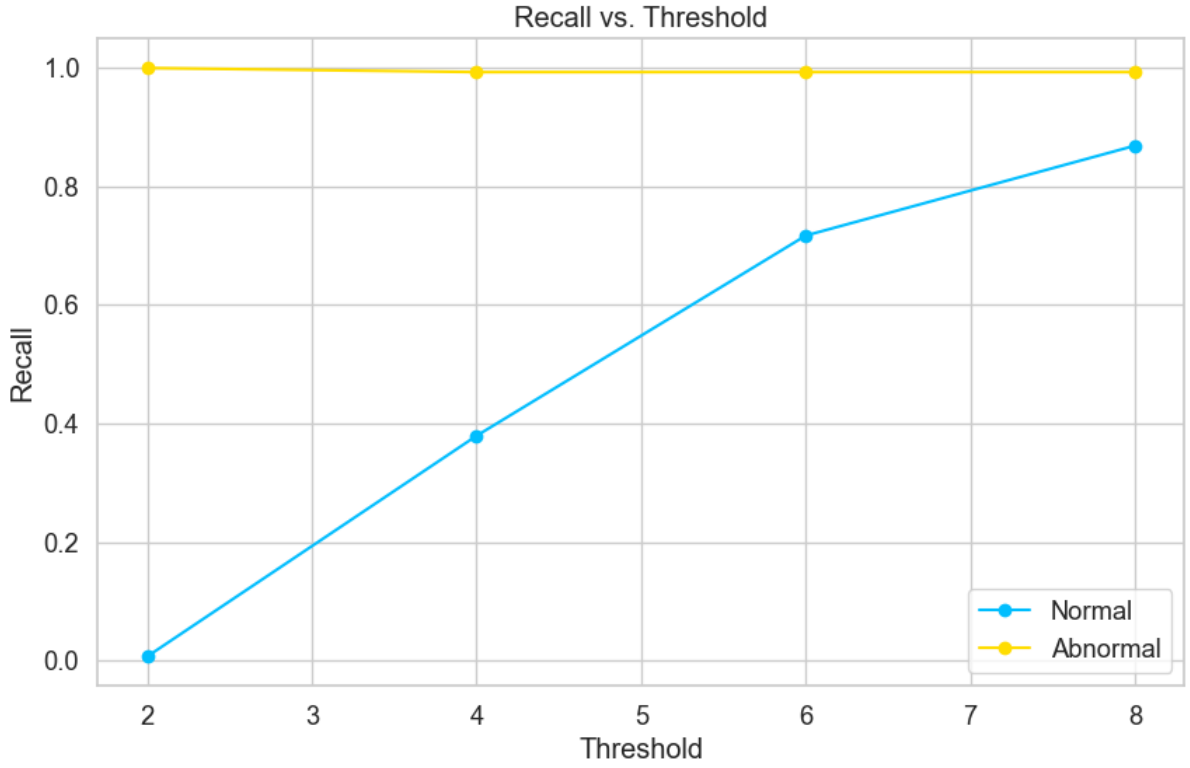
We can see that for both training and validation loss there are decreasing trend. As the embedding dimension increases, both training and validation losses decrease, indicating more effective learning and generalization. The most substantial improvements are observed as the dimension increases from 2 to 4 and then from 4 to 6, with a continued but less pronounced improvement from 6 to 8. This pattern suggests that higher-dimensional embeddings capture more complex patterns, enhancing model performance.

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. (10 points)

In [21]:
```
"## Choosing a threshold

th our model at hand, we can have a look at the reconstruction error on the training set. Let's start by wr
"

f 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


call_normal = []
call_anomaly = []
"Our function goes through each example in the dataset and records the predictions and losses. Let's get th
r i in range(len(model_list)):
  _, losses = predict(model_list[i], train_dataset)

  #sns.distplot(losses, bins=50, kde=True);
  #45
  THRESHOLD = 45

  """## Evaluation

  Using the threshold, we can turn the problem into a simple binary classification task:

  - If the reconstruction loss for an example is below the threshold, we'll classify it as a *normal* heart
  - Alternatively, if the loss is higher than the threshold, we'll classify it as an anomaly

  ### Normal hearbeats

  Let's check how well our model does on normal heartbeats. We'll use the normal heartbeats from the test s
  """

  predictions, pred_losses = predict(model_list[i], 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)
  normal = correct/len(test_normal_dataset)
  recall_normal.append(normal)
  print(f'Correct normal predictions: {correct}/{len(test_normal_dataset)}, {normal}')

  """### Anomalies

  We'll do the same with the anomaly examples, but their number is much higher. We'll get a subset that has
  """

  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_list[i], anomaly_dataset)
  #sns.distplot(pred_losses, bins=50, kde=True);

  """Finally, we can count the number of examples above the threshold (considered as anomalies):"""

  correct = sum(l > THRESHOLD for l in pred_losses)
  abnormal = correct/len(anomaly_dataset)
  recall_anomaly.append(abnormal)
  print(f'Correct anomaly predictions: {correct}/{len(anomaly_dataset)}, {abnormal}')
```

```
Correct normal predictions: 1/145, 0.006896551724137931
Correct anomaly predictions: 145/145, 1.0
Correct normal predictions: 55/145, 0.3793103448275862
Correct anomaly predictions: 144/145, 0.993103448275862
Correct normal predictions: 104/145, 0.7172413793103448
Correct anomaly predictions: 144/145, 0.993103448275862
Correct normal predictions: 126/145, 0.8689655172413793
Correct anomaly predictions: 144/145, 0.993103448275862
```

In [22]:
```
# # Range of threshold values
threshold_values = list(range(2, 9, 2))

# Plotting the results
plt.figure(figsize=(10, 6))

# Plot normal recall
plt.plot(threshold_values, recall_normal, label='Normal', linestyle='-', marker='o')

# Plot anomaly recall
plt.plot(threshold_values, recall_anomaly, label='Abnormal', linestyle='-', marker='o')

plt.xlabel('Threshold')
plt.ylabel('Recall')
plt.title('Recall vs. Threshold')
plt.legend()
plt.grid(True)
plt.show()
```



d) Briefly explain the trend you see in the Recall in part (c) above (5 points)

As the embedding dimension increases, there is a clear upward trend in the recall for normal time-series, indicating improved accuracy in correctly classifying normal cases. This suggests that higher dimensions may be capturing more relevant features for normal time-series classification. The recall for abnormal time-series remains high across all dimensions, although there is a slight decrease as the dimension increases from 2 to 6, followed by a return to the highest level at dimension 8. This indicates consistent performance in identifying abnormal cases across different dimensions.

In [ ]: