Homework 7

References

- Lectures 24-26 (inclusive).

Instructions

- Type your name and email in the "Student details" section below.
- Develop the code and generate the figures you need to solve the problems using this notebook.
- For the answers that require a mathematical proof or derivation you should type them using latex. If you have never written latex before and you find it exceedingly difficult, we will likely accept handwritten solutions.
- The total homework points are 100. Please note that the problems are not weighed equally.

In [91]:

```
import numpy as np
np.set_printoptions(precision=3)
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
sns.set(rc={"figure.dpi":100, "savefig.dpi":300})
sns.set_context("notebook")
sns.set_style("ticks")

import scipy
import scipy.stats as st
import urllib.request
import os

def download(
    url : str,
    local_filename : str = None
):
    """Download a file from a url.

    Arguments
    url            -- The url we want to download.
    local_filename -- The filemame to write on. If not
```

```
                      specified
    """
    if local_filename is None:
        local_filename = os.path.basename(url)
    urllib.request.urlretrieve(url, local_filename)
```

Student details

- **First Name:** Ben
- **Last Name:** McAteer
- **Email:** bmcateer@purdue.edu

Problem 1 - Using DNNs to Analyze Experimental Data

In this problem you have to use a deep neural network (DNN) to perform a regression task. The dataset we are going to use is the [Airfoil Self-Noise Data Set])https://archive.ics.uci.edu/ml/datasets/Airfoil+Self-Noise#) From this reference, the descreption of the dataset is as follows:

> The NASA data set comprises different size NACA 0012 airfoils at various wind tunnel speeds and angles of attack. The span of the airfoil and the observer position were the same in all of the experiments.
>
> Attribute Information: This problem has the following inputs:
>
> 1. Frequency, in Hertzs.
> 2. Angle of attack, in degrees.
> 3. Chord length, in meters.
> 4. Free-stream velocity, in meters per second.
> 5. Suction side displacement thickness, in meters.
>
> The only output is:
>
> 1. Scaled sound pressure level, in decibels.

You will have to do regression between the inputs and the output using a DNN. Before we start, let's download and load the data.

In [92]:
```
url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/00291/airfoil_self_noise.dat'
```

```
download(url)
```

The data are in simple text format. Here is how we can load them:

In [93]:
```
data = np.loadtxt('airfoil_self_noise.dat')
data
```

Out[93]:
```
array([[8.000e+02, 0.000e+00, 3.048e-01, 7.130e+01, 2.663e-03, 1.262e+02],
       [1.000e+03, 0.000e+00, 3.048e-01, 7.130e+01, 2.663e-03, 1.252e+02],
       [1.250e+03, 0.000e+00, 3.048e-01, 7.130e+01, 2.663e-03, 1.260e+02],
       ...,
       [4.000e+03, 1.560e+01, 1.016e-01, 3.960e+01, 5.285e-02, 1.066e+02],
       [5.000e+03, 1.560e+01, 1.016e-01, 3.960e+01, 5.285e-02, 1.062e+02],
       [6.300e+03, 1.560e+01, 1.016e-01, 3.960e+01, 5.285e-02, 1.042e+02]])
```

You may work directly with `data`, but, for your convenience, I am going to put them also in a nice Pandas DataFrame:

In [94]:
```
import pandas as pd
df = pd.DataFrame(data, columns=['Frequency', 'Angle_of_attack', 'Chord_length',
                                  'Velocity', 'Suction_thickness', 'Sound_pressure'])
```

Part A - Analyze the data visually

It is always a good idea to visualize the data before you start doing anything with them.

Part A.I - Do the histogtrams of all variables

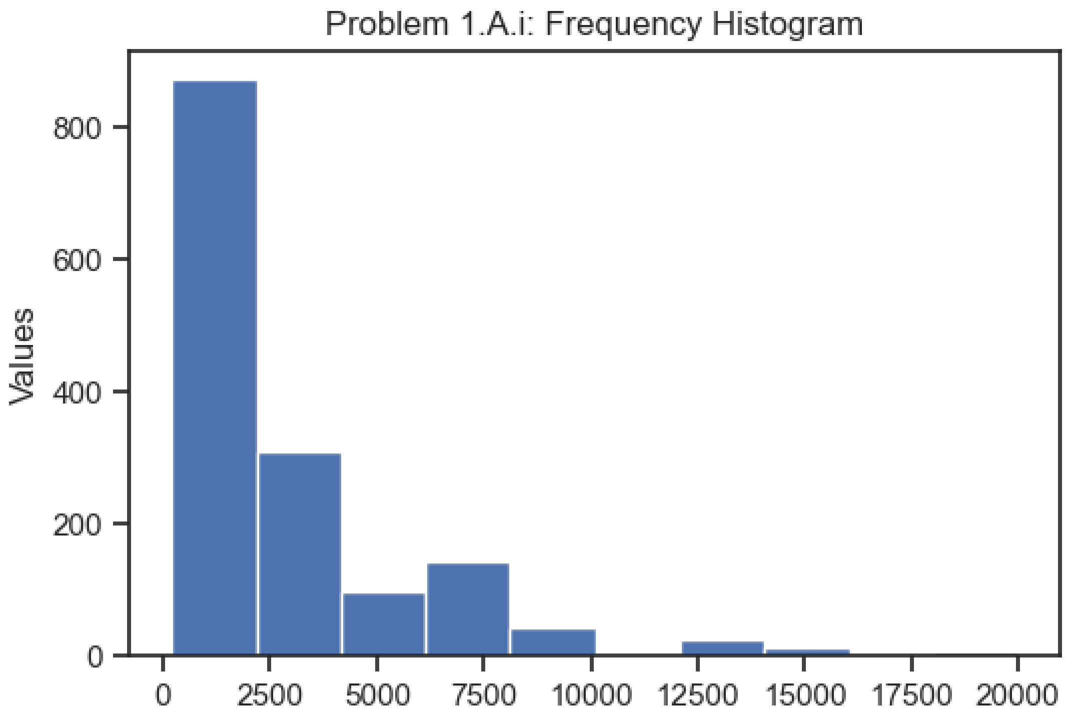
Use as many code segments you need below to plot the histogram of each variable (all inputs and the output in separate plots) Discuss whether or not you need to standarize the data before moving to regression.

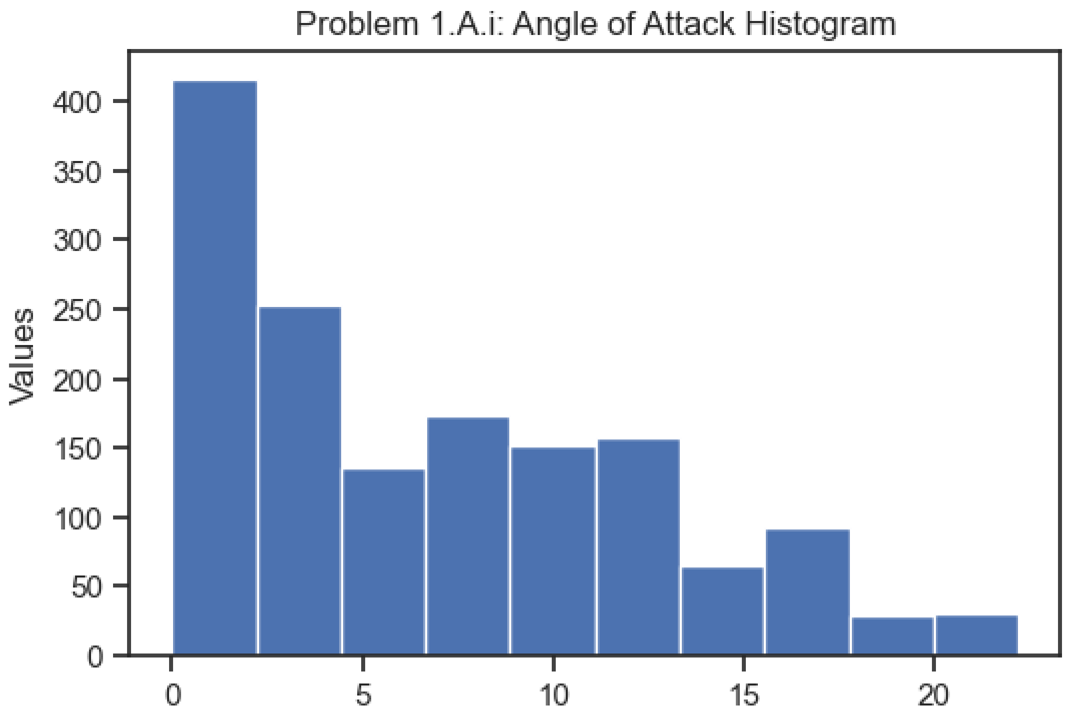
Answer: The data does not need to be standardized until regression but it will eventually need to be standardized so that each parameter has equal weight

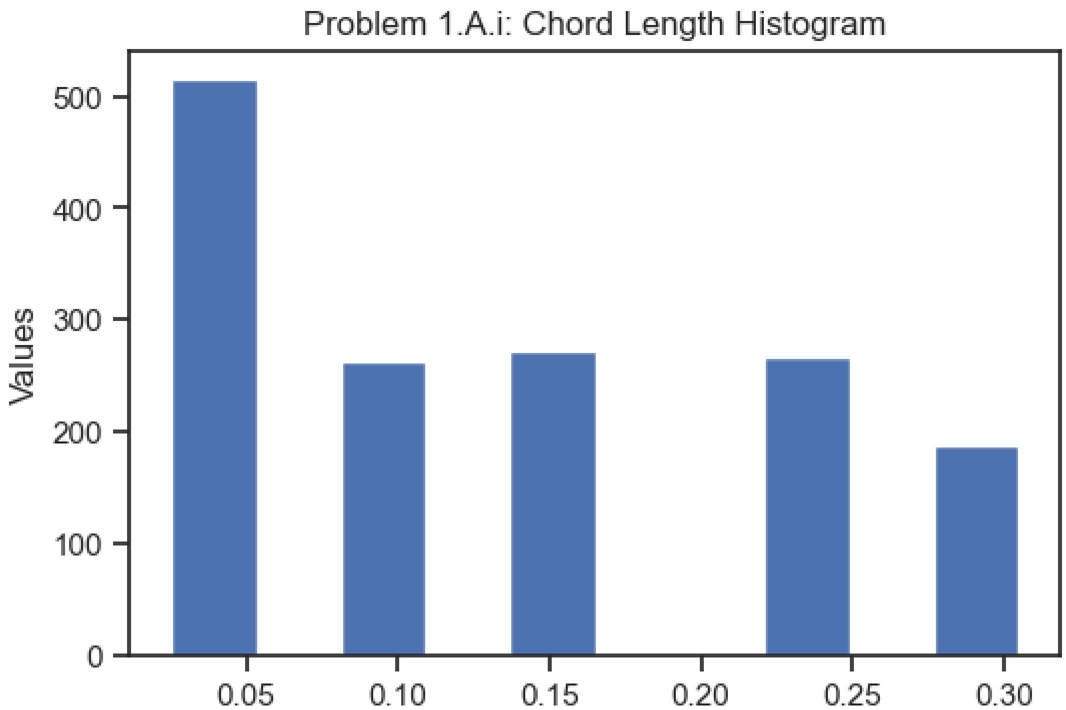
In [95]:
```
#part A

Frequency = df['Frequency'].values # Hz
AoA = df['Angle_of_attack'].values #degrees
ChordLen = df['Chord_length'].values # meters
Vel = df['Velocity'].values #m/s
SuckThicc = df['Suction_thickness'].values #meteres
```

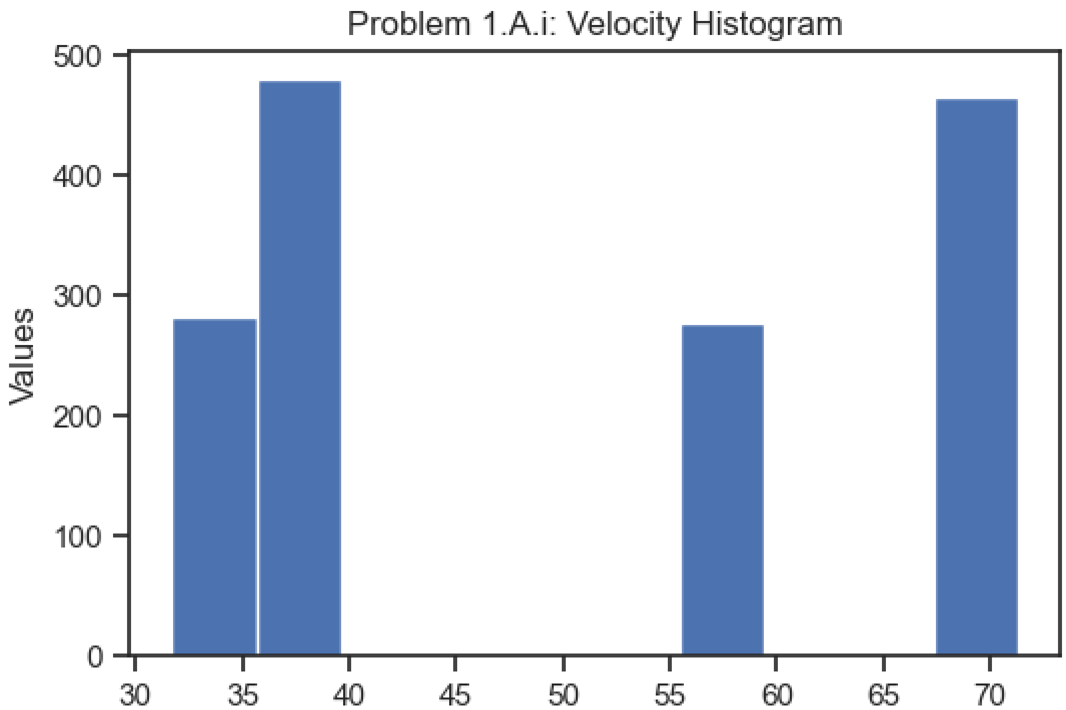
```python
SoundPres = df['Sound_pressure'].values #Only Output in dB


plt.figure()
plt.hist(Frequency)
plt.title('Problem 1.A.i: Frequency Histogram')
plt.ylabel('Values')

plt.figure()
plt.hist(AoA)
plt.title('Problem 1.A.i: Angle of Attack Histogram')
plt.ylabel('Values')

plt.figure()
plt.hist(ChordLen)
plt.title('Problem 1.A.i: Chord Length Histogram')
plt.ylabel('Values')

plt.figure()
plt.hist(Vel)
plt.title('Problem 1.A.i: Velocity Histogram')
plt.ylabel('Values')

plt.figure()
plt.hist(SuckThicc)
plt.title('Problem 1.A.i: Suction Thickness Histogram')
plt.ylabel('Values')
plt.figure()
plt.hist(SoundPres)
plt.title('Problem 1.A.i: Sound Pressure Histogram')
plt.ylabel('Values')
```

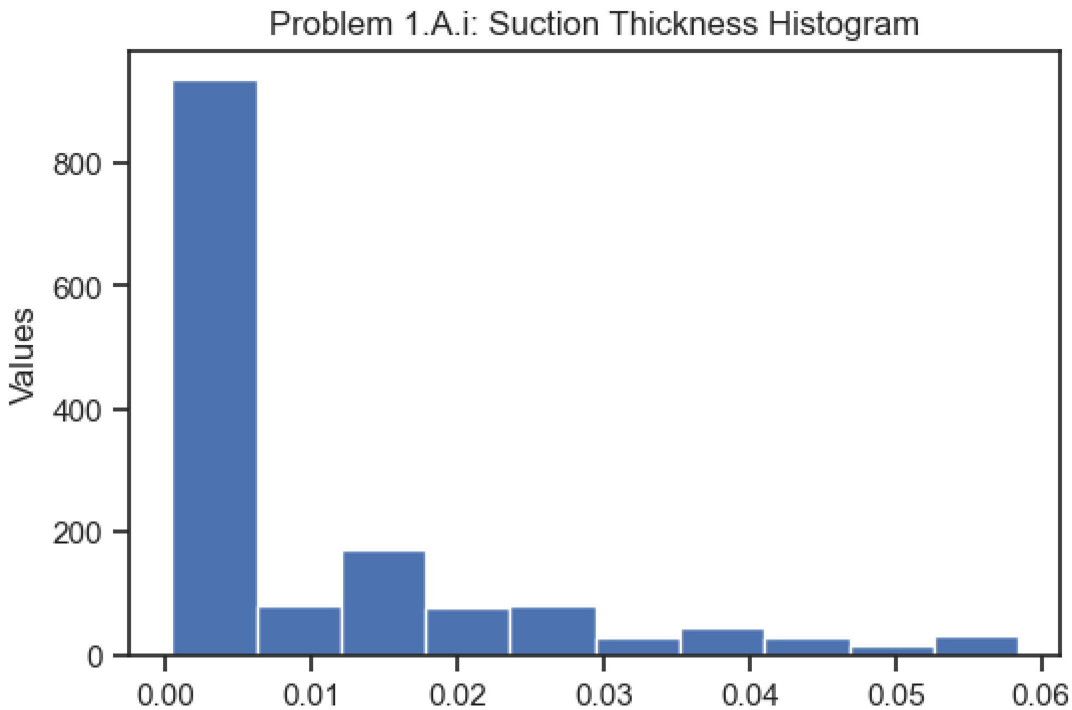
Out[95]:
```
Text(0, 0.5, 'Values')
```

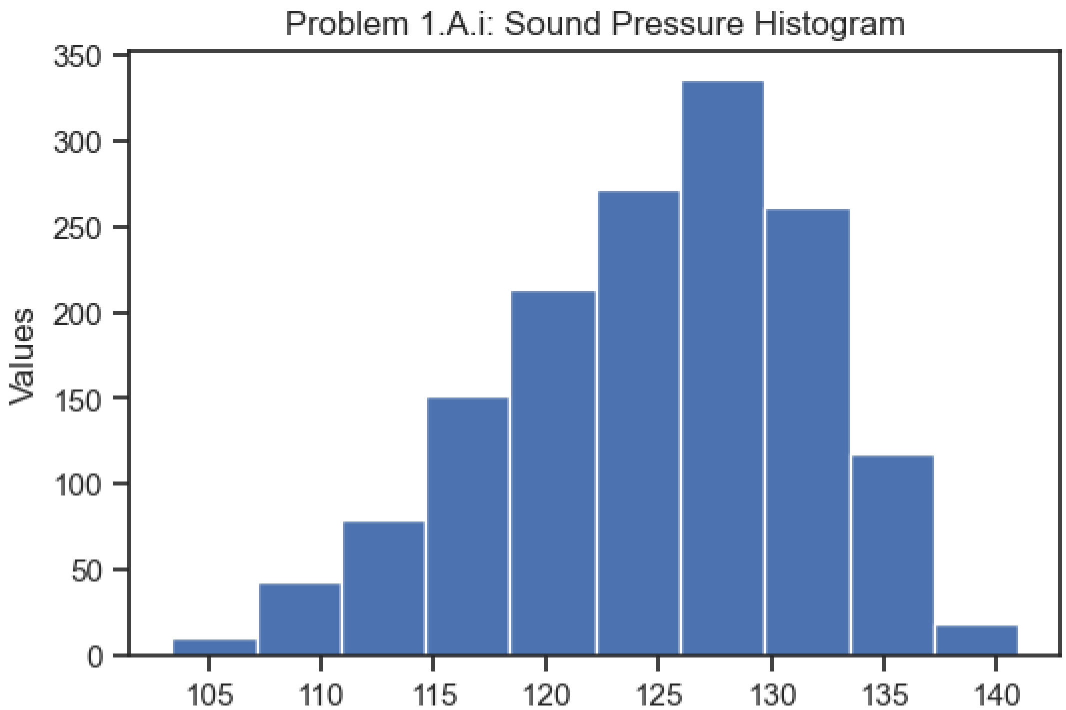

Problem 1.A.i: Frequency Histogram
Values
0
200
400
600
800
0
2500
5000
7500
10000
12500
15000
17500
20000


Problem 1.A.i: Angle of Attack Histogram
Values
400
350
300
250
200
150
100
50
0
0
5
10
15
20


Problem 1.A.i: Chord Length Histogram
Values
0
100
200
300
400
500
0.05
0.10
0.15
0.20
0.25
0.30


Problem 1.A.i: Velocity Histogram
Values
0
100
200
300
400
500
30
35
40
45
50
55
60
65
70


Problem 1.A.i: Suction Thickness Histogram
Values
800
600
400
200
0
0.00
0.01
0.02
0.03
0.04
0.05
0.06



Part A.II - Do the scatter plots between all input variables

Do the scatter plot between all input variables. This will give you an idea of the range of experimental conditions. Whatever model you build will only be valid inside the domain implicitly defined with your experimental conditions. Are there any holes in the dataset, i.e., places where you have no data?

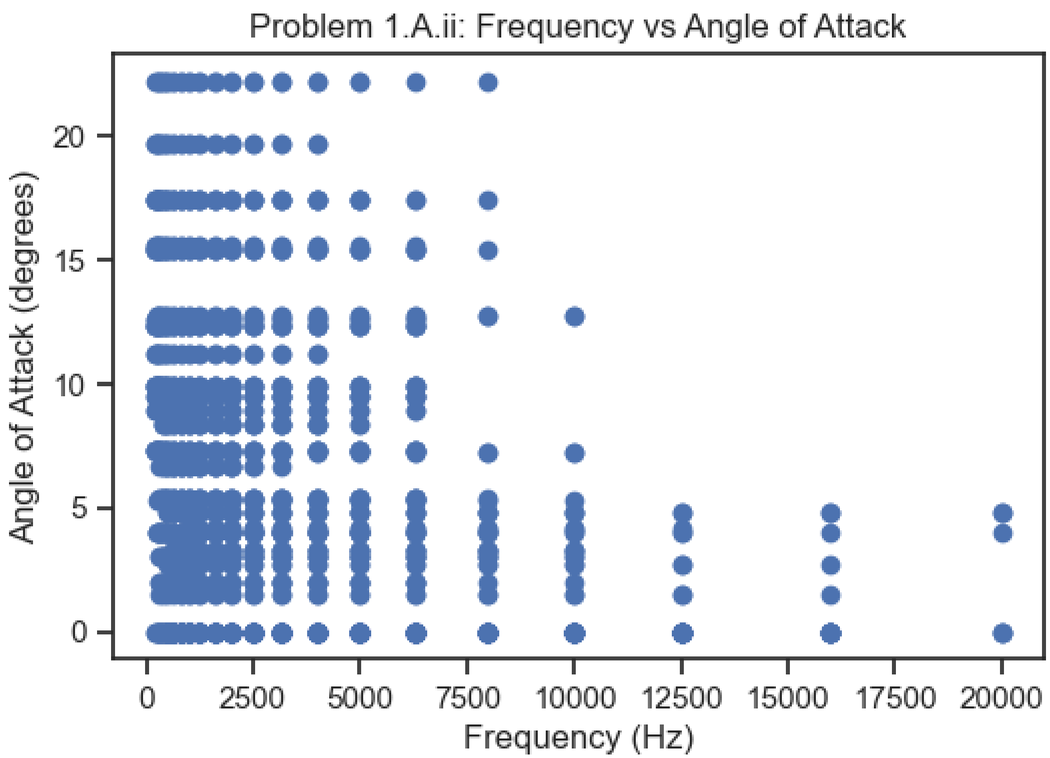
Answer: There are a few holes in the velocity input variable when compared to other inputs. This hole is relatively small though and unlikely to cause too large of a discrepency

In [22]:

```
##Part A.ii

plt.figure()
plt.scatter(Frequency,AoA)
plt.title('Problem 1.A.ii: Frequency vs Angle of Attack')
plt.xlabel('Frequency (Hz)')
plt.ylabel('Angle of Attack (degrees)')
plt.show()

plt.figure()
```

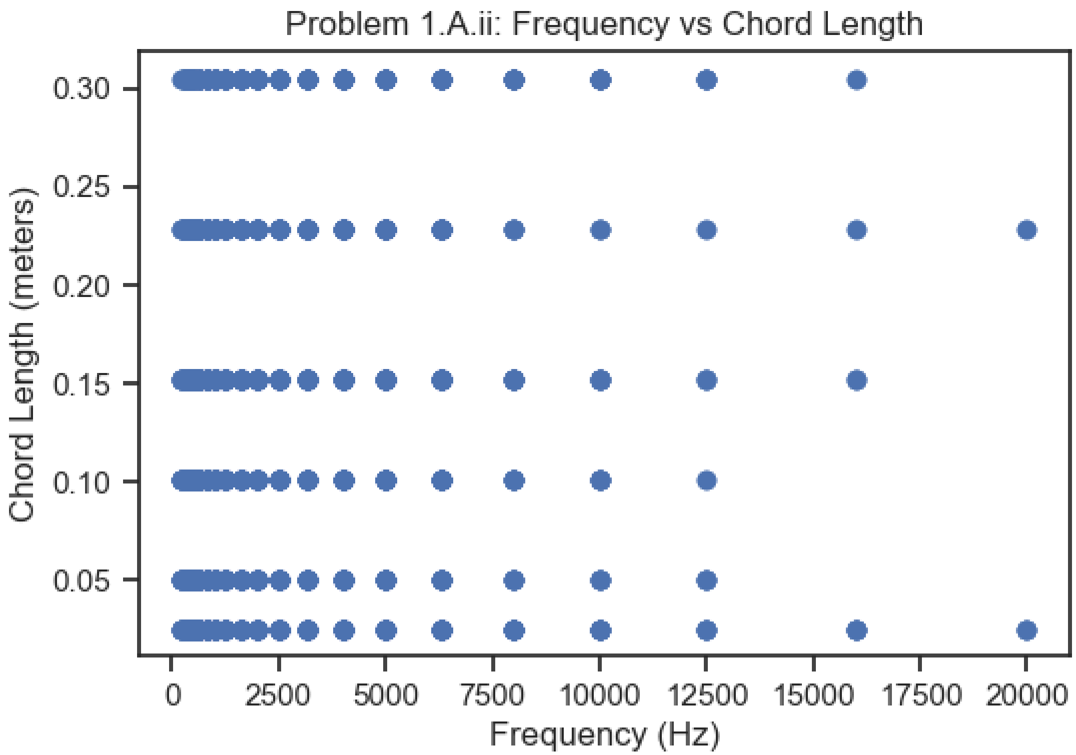
```
plt.scatter(Frequency,ChordLen)
plt.title('Problem 1.A.ii: Frequency vs Chord Length')
plt.xlabel('Frequency (Hz)')
plt.ylabel('Chord Length (meters)')

plt.figure()
plt.scatter(Frequency,Vel)
plt.title('Problem 1.A.ii: Frequency vs Velocity')
plt.xlabel('Frequency (Hz)')
plt.ylabel('Velocity (m/s)')

plt.figure()
plt.scatter(Frequency,SuckThicc)
plt.title('Problem 1.A.ii: Frequency vs Suction Displacement Thickness')
plt.xlabel('Frequency (Hz)')
plt.ylabel('Displacement (meters)')

plt.figure()
plt.scatter(AoA,ChordLen)
plt.title('Problem 1.A.ii: Angle of Attack vs Chord Length')
plt.ylabel('Chord Length (meters)')
plt.xlabel('Angle of Attack (degrees)')

plt.figure()
plt.scatter(AoA,Vel)
plt.title('Problem 1.A.ii: Angle of Attack vs Velocity')
plt.ylabel('Velocity (m/s)')
plt.xlabel('Angle of Attack (degrees)')

plt.figure()
plt.scatter(AoA,SuckThicc)
plt.title('Problem 1.A.ii: Angle of Attack vs Suction Displacement Thickness')
plt.ylabel('Suction Displacement Thickness (meters)')
plt.xlabel('Angle of Attack (degrees)')

plt.figure()
plt.scatter(ChordLen, Vel)
plt.title('Problem 1.A.ii: Chord Length vs Velocity')
plt.xlabel('Chord Length (m)')
plt.ylabel('Velocity (m/s)')

plt.figure()
plt.scatter(ChordLen,SuckThicc)
plt.title('Problem 1.A.ii: Chord Length vs Suction Displacement Thickness')
plt.xlabel('Chord Length (m)')
```

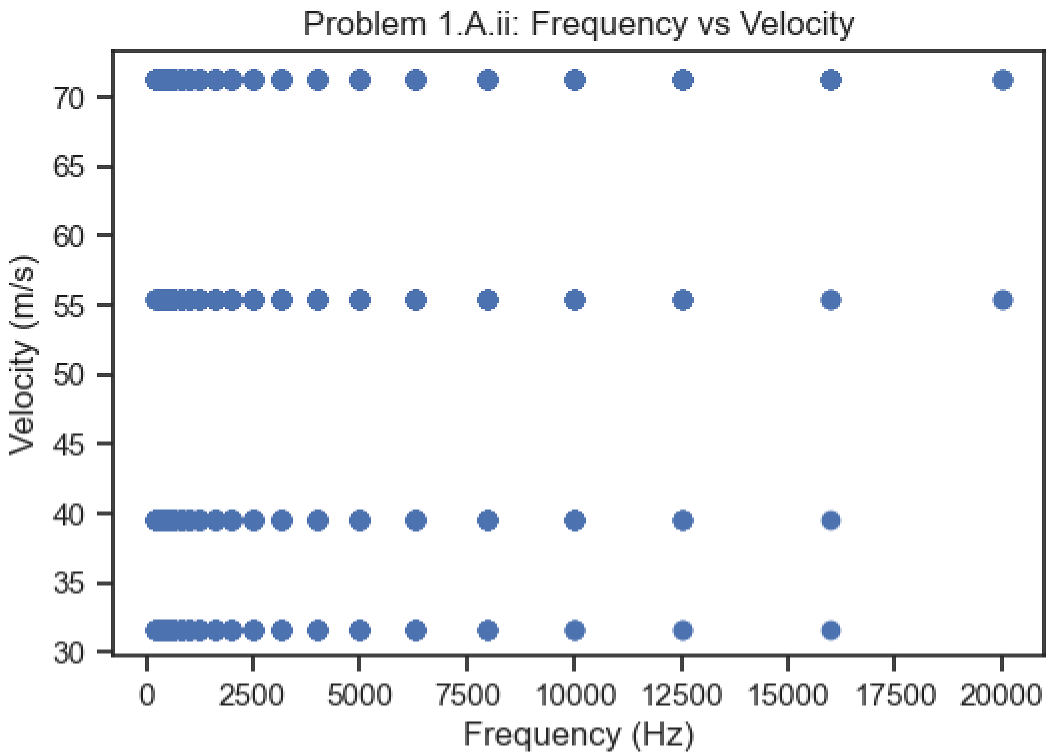
```
plt.ylabel('Suction Displacement Thickness (meters)')

plt.figure()
plt.scatter(Vel,SuckThicc)
plt.title('Problem 1.A.ii: Velocity vs Suction Displacement Thickness')
plt.xlabel('Velocity (m/s)')
plt.ylabel('Suction Displacement Thickness (meters)')
```

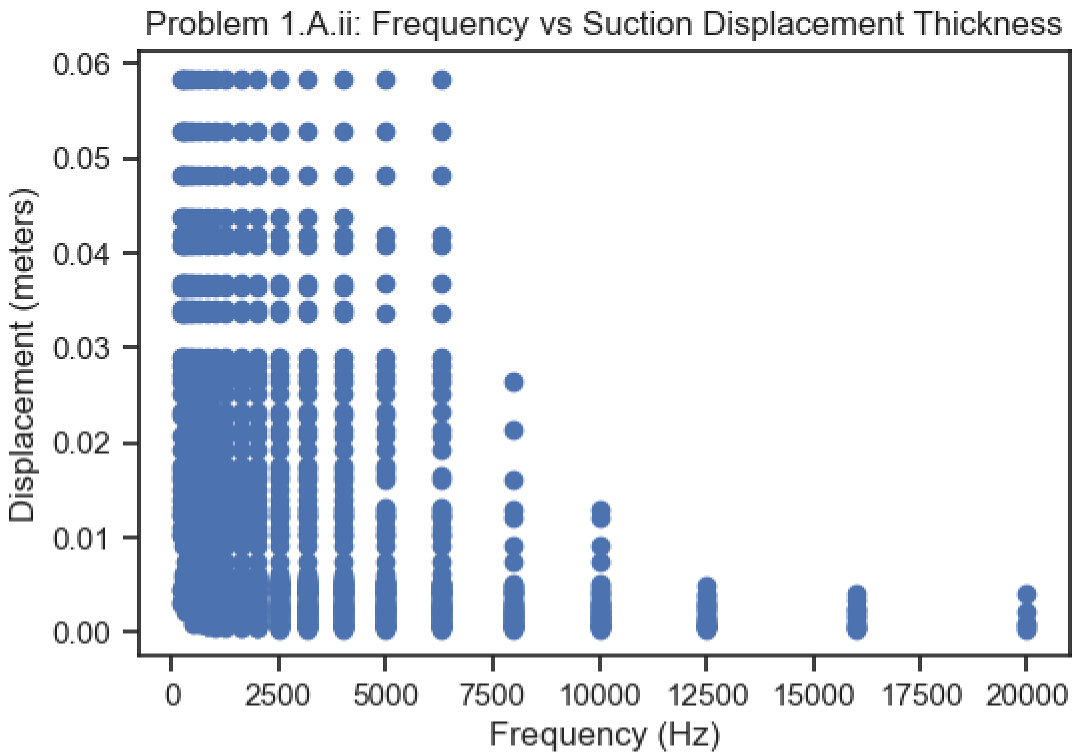


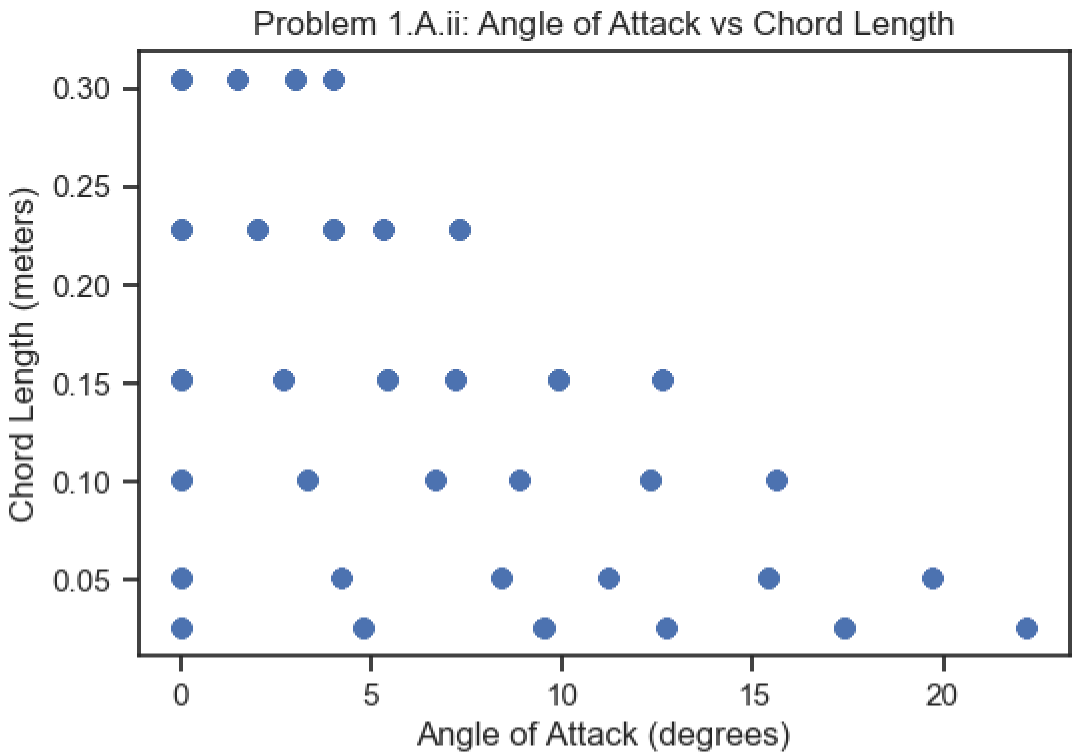
Out[22]:

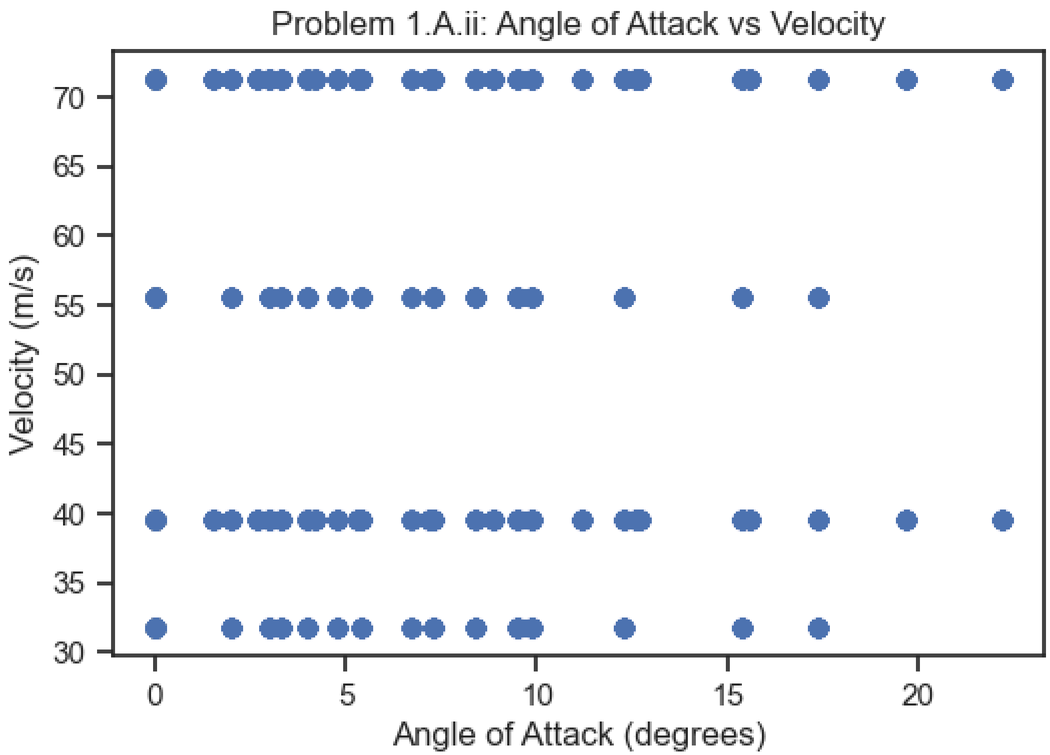
```
Text(0, 0.5, 'Suction Displacement Thickness (meters)')
```

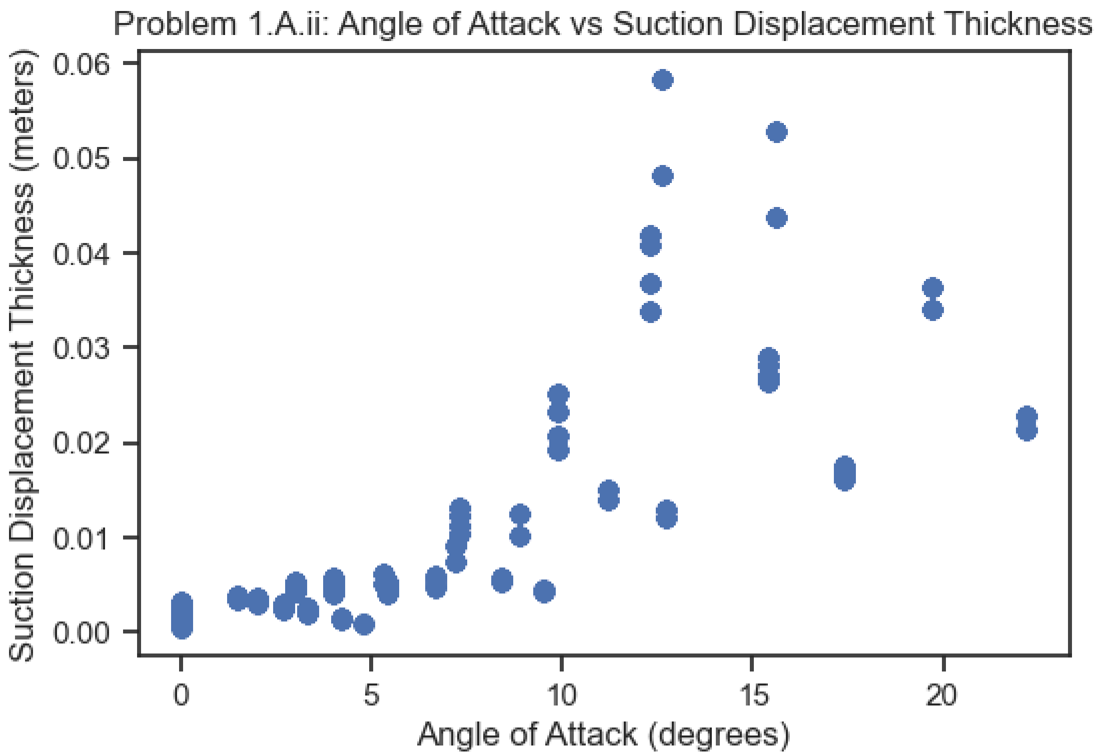

Problem 1.A.ii: Frequency vs Chord Length
Chord Length (meters)
0.30
0.25
0.20
0.15
0.10
0.05
0
2500
5000
7500
10000
12500
15000
17500
20000
Frequency (Hz)

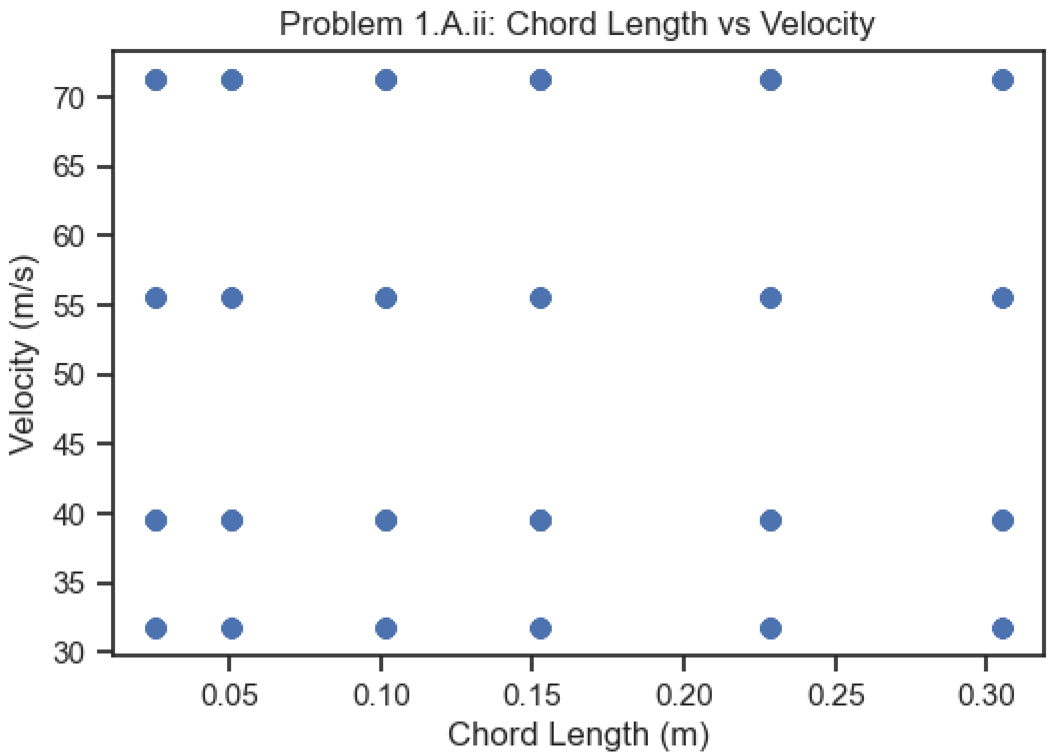

Problem 1.A.ii: Frequency vs Velocity
Velocity (m/s)
70
65
60
55
50
45
40
35
30
0
2500
5000
7500
10000
12500
15000
17500
20000
Frequency (Hz)

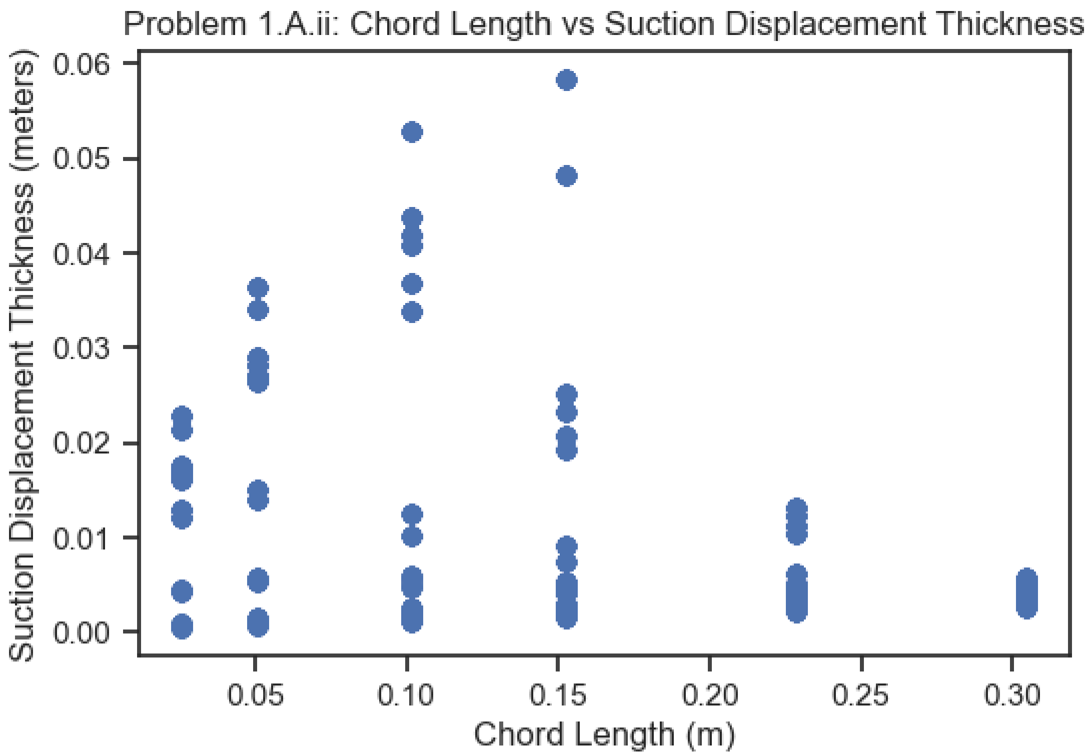

Problem 1.A.ii: Frequency vs Suction Displacement Thickness
Displacement (meters)
0.00
0.01
0.02
0.03
0.04
0.05
0.06
0
2500
5000
7500
10000
12500
15000
17500
20000
Frequency (Hz)

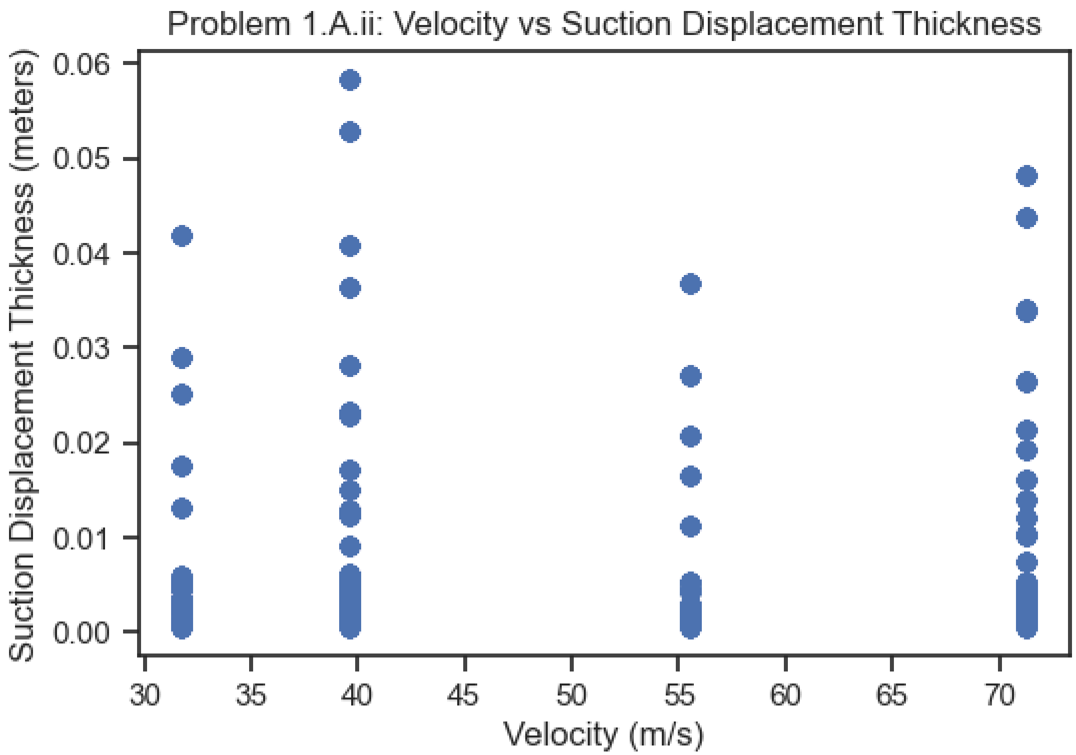

Problem 1.A.ii: Angle of Attack vs Chord Length
Chord Length (meters)
0.30
0.25
0.20
0.15
0.10
0.05
0
5
10
15
20
Angle of Attack (degrees)


Problem 1.A.ii: Angle of Attack vs Velocity
Velocity (m/s)
70
65
60
55
50
45
40
35
30
0
5
10
15
20
Angle of Attack (degrees)


Problem 1.A.ii: Angle of Attack vs Suction Displacement Thickness
Suction Displacement Thickness (meters)
0.06
0.05
0.04
0.03
0.02
0.01
0.00
0
5
10
15
20
Angle of Attack (degrees)


Problem 1.A.ii: Chord Length vs Velocity
Velocity (m/s)
70
65
60
55
50
45
40
35
30
0.05
0.10
0.15
0.20
0.25
0.30
Chord Length (m)


Problem 1.A.ii: Chord Length vs Suction Displacement Thickness
Suction Displacement Thickness (meters)
0.00
0.01
0.02
0.03
0.04
0.05
0.06
0.05
0.10
0.15
0.20
0.25
0.30
Chord Length (m)



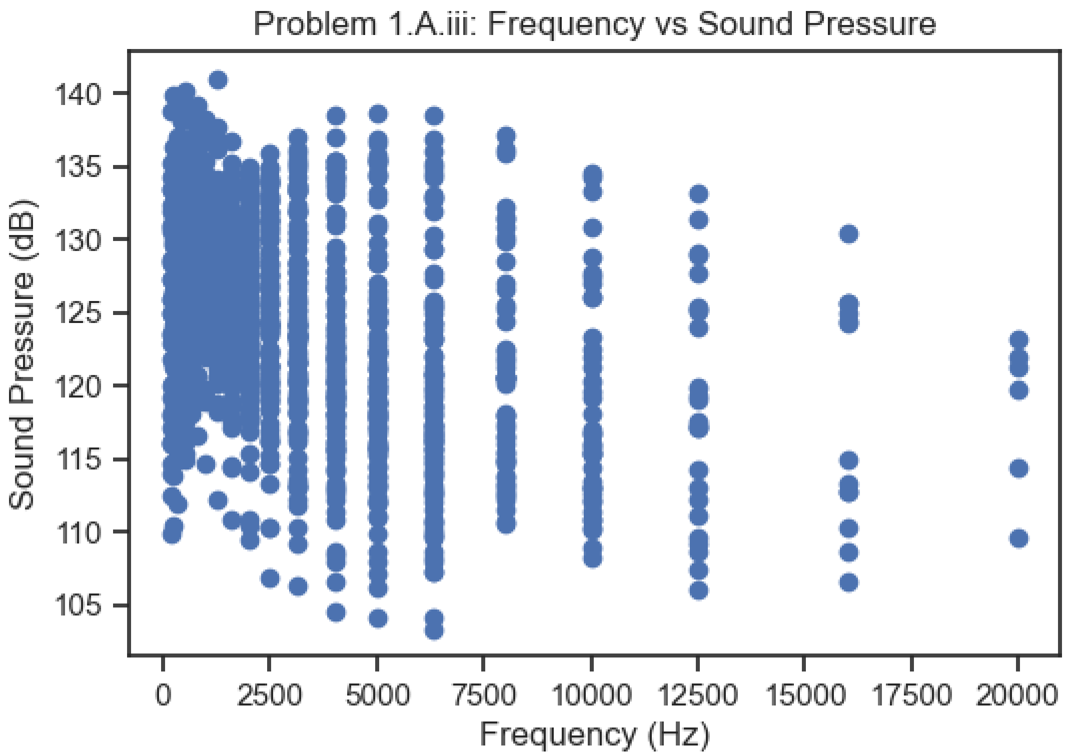
Part A.III - Do the scatter plots between each input and the output

Do the scatter plot between each input variable and the output. This will give you an idea of the functional relationship between the two. Do you observe any obvious patterns?

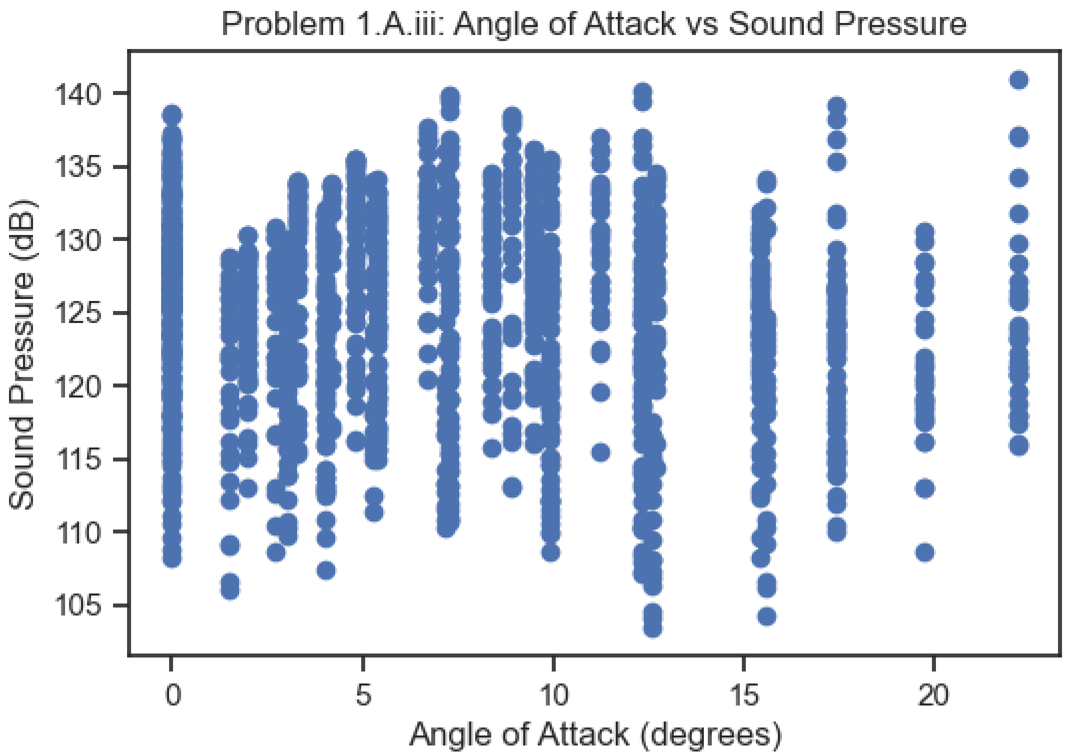
Answer: No obvious patterns are observed between the inputs and output

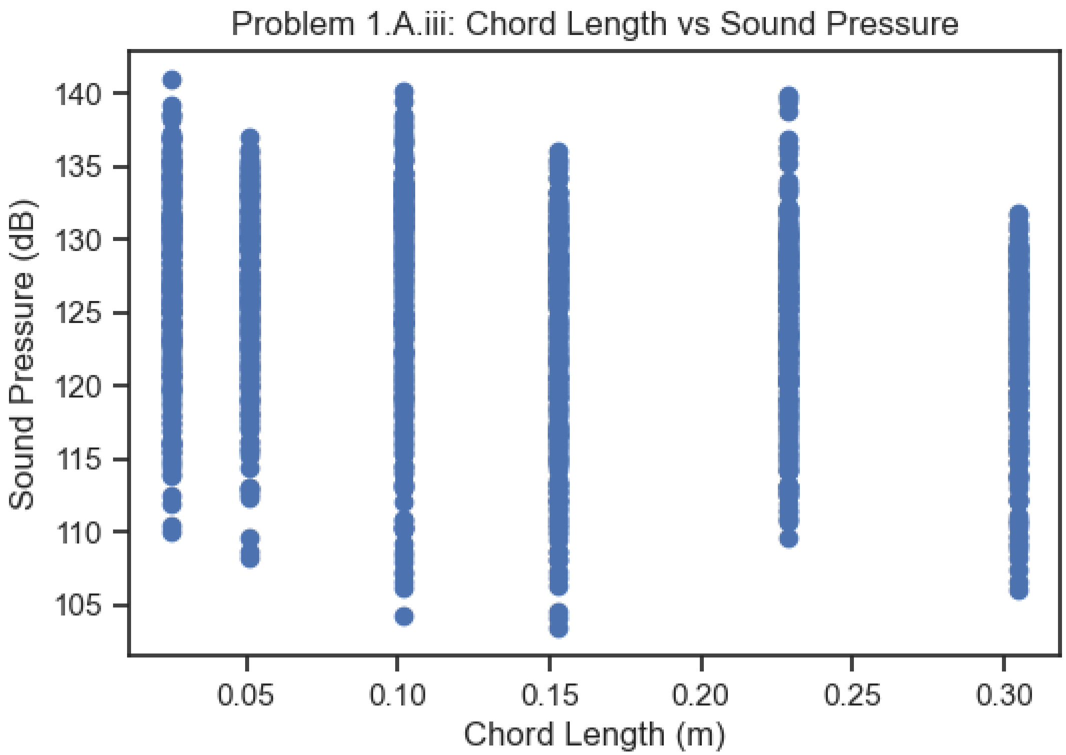
In [23]:
```
# part A.iii

plt.figure()
plt.scatter(Frequency,SoundPres)
plt.title('Problem 1.A.iii: Frequency vs Sound Pressure')
plt.xlabel('Frequency (Hz)')
plt.ylabel('Sound Pressure (dB)')
plt.show()


plt.figure()
```

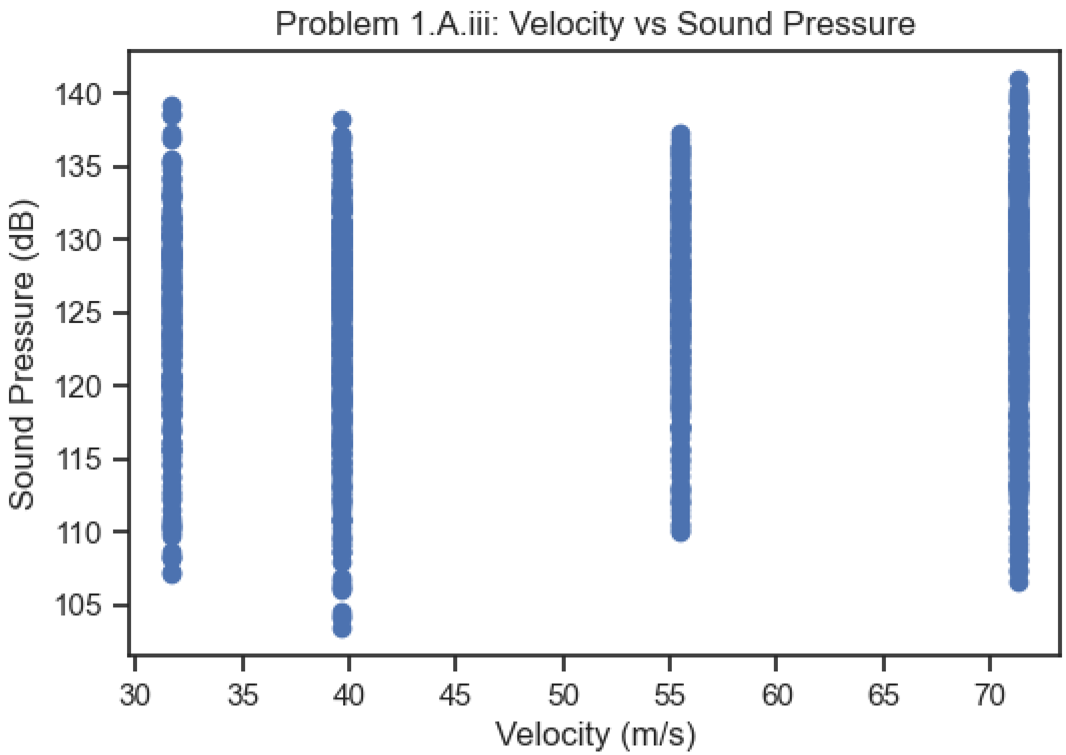
```
plt.scatter(AoA,SoundPres)
plt.title('Problem 1.A.iii: Angle of Attack vs Sound Pressure')
plt.xlabel('Angle of Attack (degrees)')
plt.ylabel('Sound Pressure (dB)')

plt.figure()
plt.scatter(ChordLen, SoundPres)
plt.title('Problem 1.A.iii: Chord Length vs Sound Pressure')
plt.xlabel('Chord Length (m)')
plt.ylabel('Sound Pressure (dB)')

plt.figure()
plt.scatter(Vel,SoundPres)
plt.title('Problem 1.A.iii: Velocity vs Sound Pressure')
plt.xlabel('Velocity (m/s)')
plt.ylabel('Sound Pressure (dB)')

plt.figure()
plt.scatter(SuckThicc,SoundPres)
plt.title('Problem 1.A.iii: Suction Displacement Thickness vs Sound Pressure')
plt.ylabel('Sound Pressure (dB)')
plt.xlabel('Suction Displacement Thickness (meters)')
```

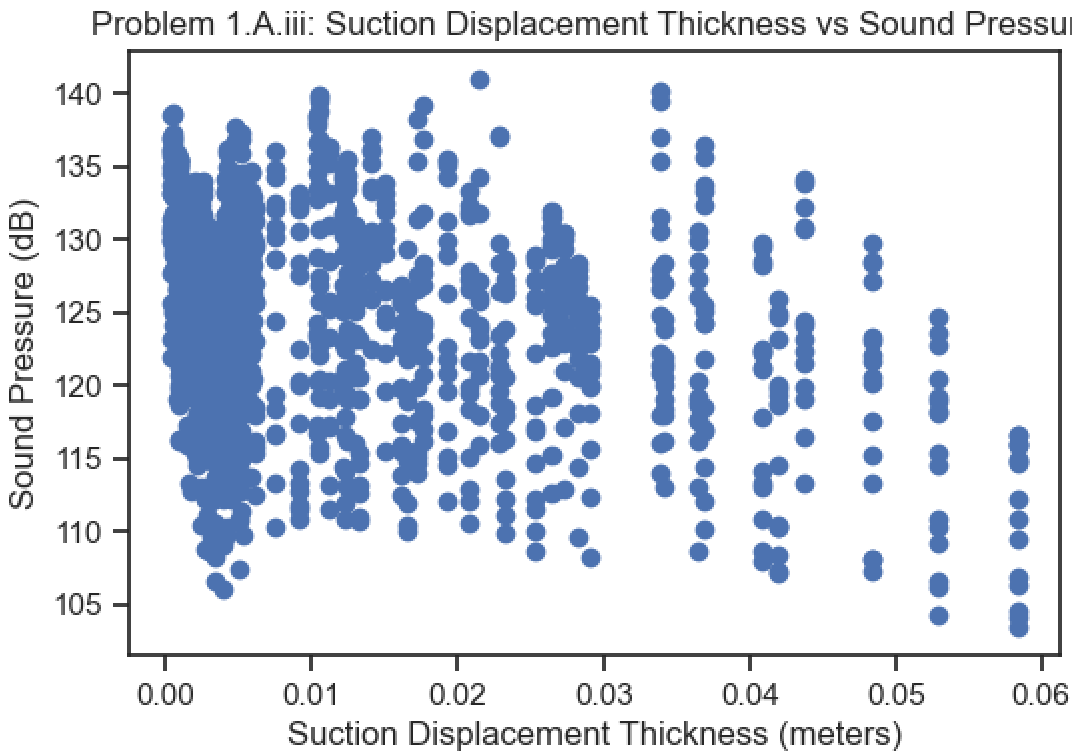


Out[23]:
```
Text(0.5, 0, 'Suction Displacement Thickness (meters)')
```


Problem 1.A.iii: Angle of Attack vs Sound Pressure
Sound Pressure (dB)
140
135
130
125
120
115
110
105
0
5
10
15
20
Angle of Attack (degrees)


Problem 1.A.iii: Chord Length vs Sound Pressure
Sound Pressure (dB)
140
135
130
125
120
115
110
105
0.05
0.10
0.15
0.20
0.25
0.30
Chord Length (m)


Problem 1.A.iii: Velocity vs Sound Pressure
Sound Pressure (dB)
140
135
130
125
120
115
110
105
30
35
40
45
50
55
60
65
70
Velocity (m/s)



Part B - Use DNN to do regression

Let start by separating inputs and outputs for you:

In [96]:
```
X = data[:, :-1]
y = data[:, -1][:, None]
```

Part B.I - Make the loss

Use standard torch functionality, to create a function that gives you the sum of square error followed by an L2 regularization term for the weights and biaseas of all netework parameters (remember that the L2 regularization is like putting a Gaussian prior on each parameter). Follow the instructions bellow and fill in the missing code.

Answer:

In [97]:
```
import torch
```

```python
import torch.nn as nn

# Use standard torch functionality to define a function
# mse_loss(y_obs, y_pred) which gives you the mean of the sum of the square
# of the difference between y_obs and y_pred
# Hint: This is already implemented in PyTorch. You can just reuse it.
mse_loss = nn.MSELoss()
```

In [98]:

```python
# Test your code here
y_obs_tmp = np.random.randn(100, 1)
y_pred_tmp = np.random.randn(100, 1)
print('Your mse_loss: {0:1.2f}'.format(mse_loss(torch.Tensor(y_obs_tmp),
                                               torch.Tensor(y_pred_tmp))))
print('What you should be getting: {0:1.2f}'.format(np.mean((y_obs_tmp - y_pred_tmp) ** 2)))
```

```
Your mse_loss: 1.87
What you should be getting: 1.87
```

In [99]:

```python
# Now, we will create a regularization term for the loss
# I'm just going to give you this one:
def l2_reg_loss(params):
    """
    This needs an iterable object of network parameters.
    You can get it by doing `net.parameters()`.

    Returns the sum of the squared norms of all parameters.
    """
    l2_reg = torch.tensor(0.)
    for p in params:
        l2_reg += torch.norm(p) ** 2
    return l2_reg
```

In [100…

```python
# Finally, let's add the two together to make a mean square error loss
# plus some weight (which we will call reg_weight) times the sum of the squared norms
# of all parameters.
# I give you the signature and you have to implement the rest of the code:
def loss_func(y_obs, y_pred, reg_weight, params):
    """
    Parameters:
    y_obs      -    The observed outputs
    y_pred     -    The predicted outputs
    reg_weight -    The regularization weight (a positive scalar)
```

```
    params     -    An iterable containing the parameters of the network

    Returns the sum of the MSE loss plus reg_weight times the sum of the squared norms of
    all parameters.
    """
    weightedMSEloss = mse_loss(y_obs, y_pred) + reg_weight * l2_reg_loss(params)

    return weightedMSEloss
```

In [101…

```
# You can try your final code here
# First, here is a dummy model
dummy_net = nn.Sequential(nn.Linear(10, 20),
                          nn.Sigmoid(),
                          nn.Linear(20, 1))
loss = loss_func(torch.Tensor(y_obs_tmp), torch.Tensor(y_pred_tmp),
                 0.0,
                 dummy_net.parameters())
print('The loss without regularization: {0:1.2f}'.format(loss.item()))
print('This should be the same as this: {0:1.2f}'.format(mse_loss(torch.Tensor(y_obs_tmp), torch.Tensor(y_pred_tmp))))
loss = loss_func(torch.Tensor(y_obs_tmp), torch.Tensor(y_pred_tmp),
                 0.01,
                 dummy_net.parameters())
print('The loss with regularization: {0:1.2f}'.format(loss.item()))
```

```
The loss without regularization: 1.87
This should be the same as this: 1.87
The loss with regularization: 1.95
```

Part B.III - Write flexible code to perform regression

When training neural networks you have to hand-pick many parameters: from the structure of the network to the activation functions to the regularization parameters to the details of the stochatic optimization. Instead of blindly going through trial and error, it is better to think about the parameters you want to investigate (vary) and write code that allows you to repeatly train networks with all different parameter variations. In what follows, I will guide you through writing code for training an arbitrary regression network having the flexibility to:

- standarize the inputs and output or not
- experiment with various levels of regularization
- change the learning rate of the stochatic optimization algorithm
- change the batch size of the optimization algorithm

- change the number of epochs (how many times the optimization algorithm does a complete sweep through all the data.

Answer:

In [102…

```
# We are going to start by creating a class that encapsulates a regression
# network so that we can turn on or off input/output standarization
# without too much fuss.
# The class will essentially represent a trained network model.
# It will "know" whether or not during training we standarized the data.
# I am not asking you to do anything here, so you may just run this code segment
# or read through if you want to know about the details.
from sklearn.preprocessing import StandardScaler

class TrainedModel(object):
    """
    A class that represents a trained network model.
    The main reason I created this class is to encapsulate the standarization
    process in a nice way.

    Parameters:

    net             -    A network.
    standarized     -    True if the network expects standarized features and outputs
                         standarized targets. False otherwise.
    feature_scaler -    A feature scalar - Ala scikit learn. Must have transform()
                         and inverse_transform() implemented.
    target_scaler  -    Similar to feature_scaler but for targets...
    """

    def __init__(self, net, standarized=False, feature_scaler=None, target_scaler=None):
        self.net = net
        self.standarized = standarized
        self.feature_scaler = feature_scaler
        self.target_scaler = target_scaler

    def __call__(self, X):
        """
        Evaluates the model at X.
        """
        # If not scaled, then the model is just net(X)
        if not self.standarized:
            return self.net(X)
        # Otherwise:
        # Scale X:
```

```
        X_scaled = self.feature_scaler.transform(X)
        # Evaluate the network output - which is also scaled:
        y_scaled = self.net(torch.Tensor(X_scaled))
        # Scale the output back:
        y = self.target_scaler.inverse_transform(y_scaled.detach().numpy())
        return y
```

In [103…

```
# Go through the code that follows and fill in the missing parts
from sklearn.model_selection import train_test_split
# We need this for a progress bar:
from tqdm import tqdm

def train_net(X, y, net, reg_weight, n_batch, epochs, lr, test_size=0.33,
              standarize=True):
    """
    A function that trains a regression neural network using stochatic gradient
    descent and returns the trained network. The loss function being minimized is
    `loss_func`.

    Arguments:

    X          -    The observed features
    y          -    The observed targets
    net        -    The network you want to fit
    n_batch    -    The batch size you want to use for stochastic optimization
    epochs     -    How many times do you want to pass over the training dataset.
    lr         -    The learning rate for the stochastic optimization algorithm.
    test_size  -    What percentage of the data should be used for testing (validation).
    standarize -    Whether or not you want to standarize the features and the targets.
    """
    # Split the data
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33)

    # Standarize the data
    if standarize:
        # Build the scalers
        feature_scaler = StandardScaler().fit(X)
        target_scaler = StandardScaler().fit(y)
        # Get scaled versions of the data
        X_train_scaled = feature_scaler.transform(X_train)
        y_train_scaled = target_scaler.transform(y_train)
        X_test_scaled = feature_scaler.transform(X_test)
        y_test_scaled = target_scaler.transform(y_test)
    else:
```

```
        feature_scaler = None
        target_scaler = None
        X_train_scaled = X_train
        y_train_scaled = y_train
        X_test_scaled = X_test
        y_test_scaled = y_test

    # Turn all the numpy arrays to torch tensors
    X_train_scaled = torch.Tensor(X_train_scaled)
    X_test_scaled = torch.Tensor(X_test_scaled)
    y_train_scaled = torch.Tensor(y_train_scaled)
    y_test_scaled = torch.Tensor(y_test_scaled)

    # This is pytorch magick to enable shuffling of the
    # training data every time we go through them
    train_dataset = torch.utils.data.TensorDataset(X_train_scaled, y_train_scaled)
    train_data_loader = torch.utils.data.DataLoader(train_dataset,
                                                    batch_size=n_batch,
                                                    shuffle=True)

    # Create an Adam optimizing object for the neural network `net`
    # with learning rate `lr`
    #raise NotImplementedError('Define the optimizer object! Delete me then!')
    optimizer = torch.optim.Adam(net.parameters(), lr = lr)

    # This is a place to keep track of the test loss
    test_loss = []

    # Iterate the optimizer.
    # Remember, each time we go through the entire dataset we complete an `epoch`
    # I have wrapped the range around tqdm to give you a nice progress bar
    # to look at
    for e in tqdm(range(epochs)):
        # This loop goes over all the shuffled training data
        # That's why the DataLoader class of PyTorch is convenient
        for X_batch, y_batch in train_data_loader:
            # Perform a single optimization step with loss function
            # loss_func(y_batch, y_pred, reg_weight, net.parameters())
            # Hint 1: You have defined loss_func() already
            # Hint 2: Consult the hands-on activities for an example
            optimizer.zero_grad()
            y_pred= net(X_batch) ## output
            loss = loss_func(y_batch, y_pred, reg_weight,net.parameters())
            loss.backward()
            optimizer.step()
```

```
        # Evaluate the test loss and append it on the list `test_loss`
        y_pred_test = net(X_test_scaled)
        ts_loss = mse_loss(y_test_scaled, y_pred_test)
        test_loss.append(ts_loss.item())

    # Make a TrainedModel
    trained_model = TrainedModel(net, standarized=standarize,
                                 feature_scaler=feature_scaler,
                                 target_scaler=target_scaler)

    # Make sure that we return properly scaled

    # Return everything we need to analyze the results
    return trained_model, test_loss, X_train, y_train, X_test, y_test
```

Use this to test your code:

In [32]:

```
# A simple one-layer network with 10 neurons
net = nn.Sequential(nn.Linear(5, 20),
                    nn.Sigmoid(),
                    nn.Linear(20, 1))
epochs = 1000
lr = 0.01
reg_weight = 0
n_batch = 100
model, test_loss, X_train, y_train, X_test, y_test = train_net(
    X,
    y,
    net,
    reg_weight,
    n_batch,
    epochs,
    lr
)
```

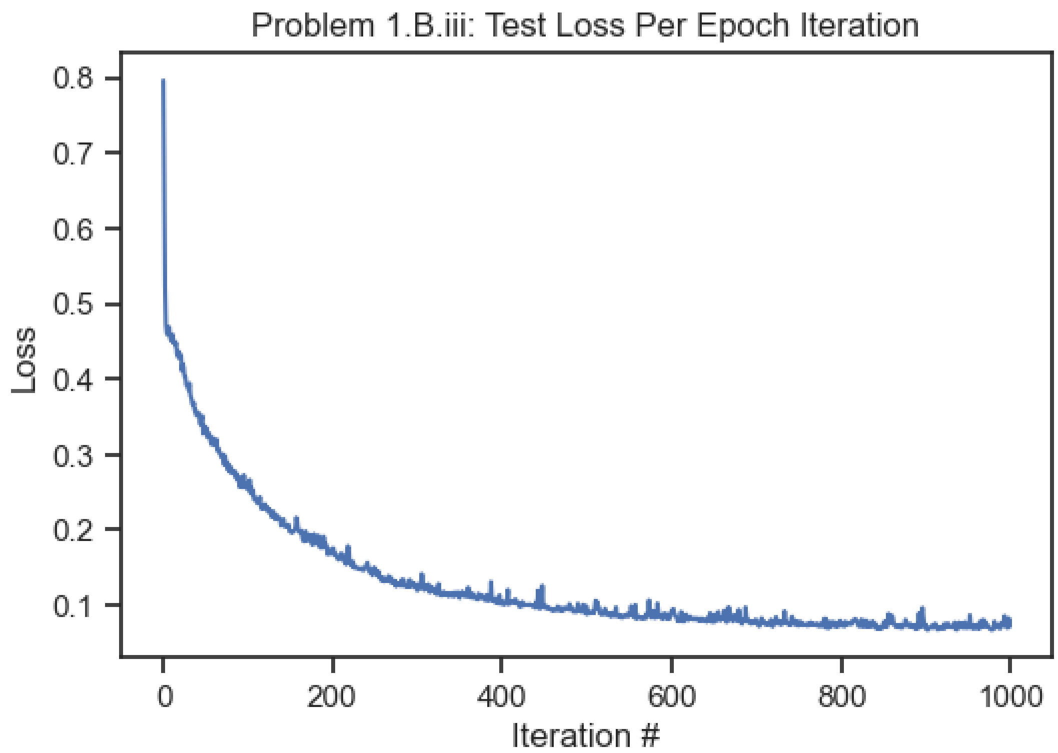
```
100%|██████████| 1000/1000 [00:25<00:00, 39.92it/s]
```

There are a few more things for you to do here. First, plot the evolution of the test loss as a function of the number of epochs:

In [33]:

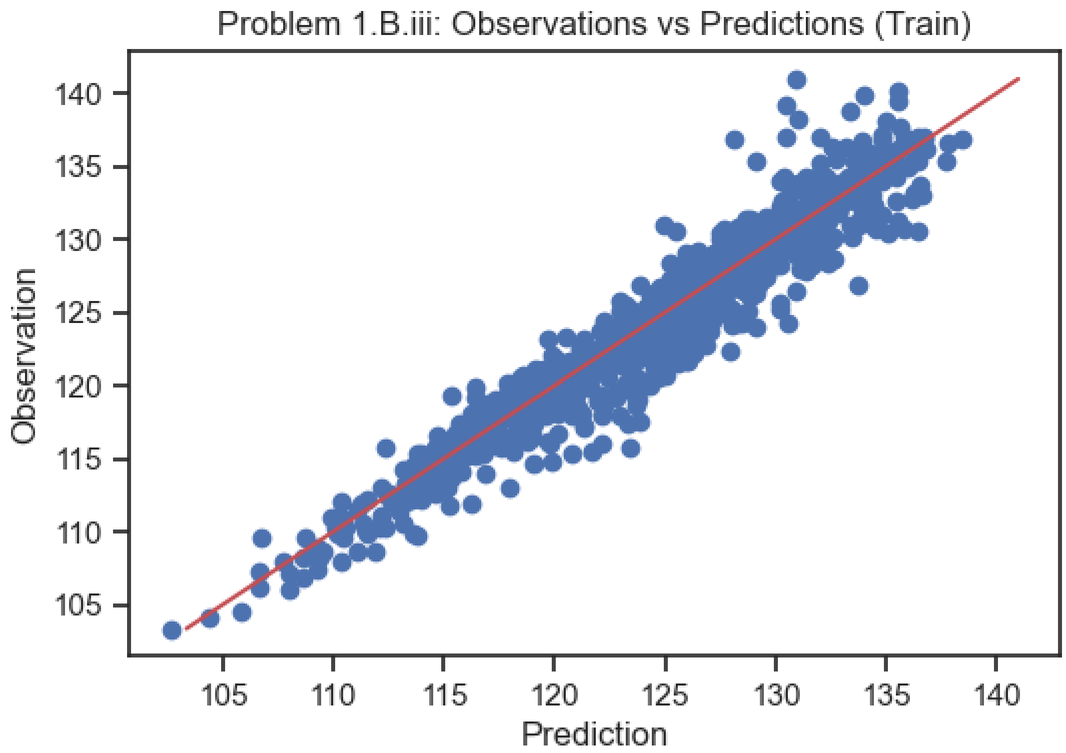
```
plt.figure()
plt.plot(test_loss)
plt.xlabel('Iteration #')
```

```
plt.ylabel('Loss')
plt.title('Problem 1.B.iii: Test Loss Per Epoch Iteration')
plt.show()
```



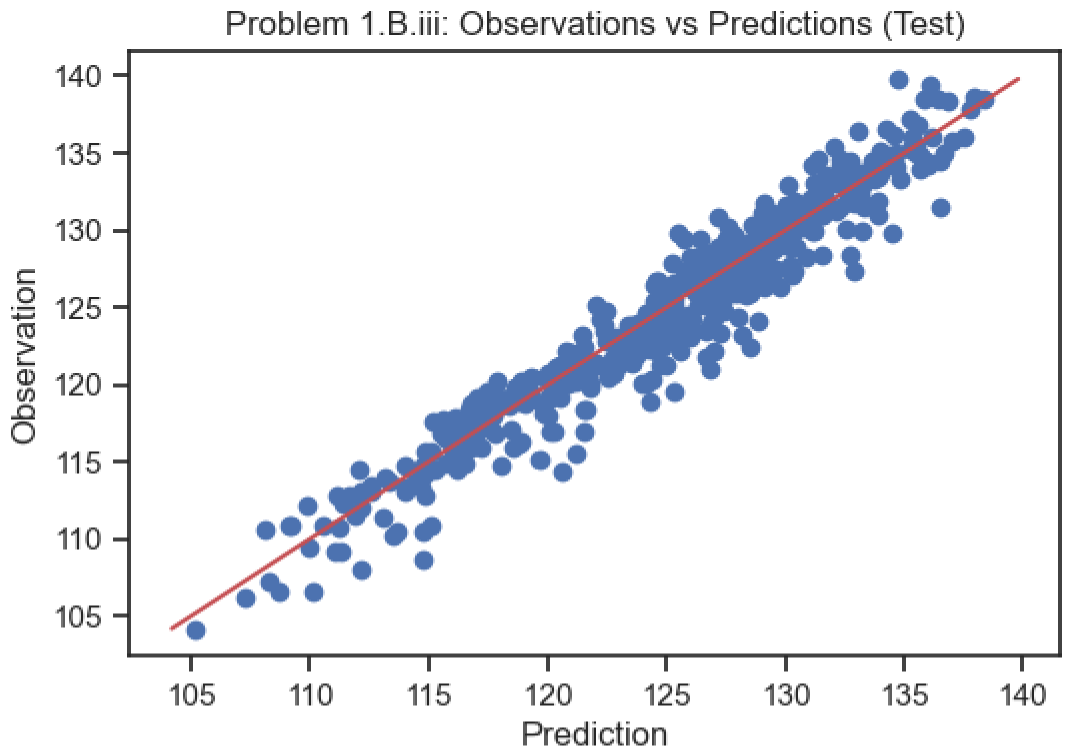
Now plot the observations vs predictions plot for the training data:

In [34]:
```
train_prediction = model(X_train)
plt.scatter(train_prediction, y_train)
bestline = torch.linspace(y_train.min(), y_train.max(),30)
plt.plot(bestline,bestline, 'r')
plt.xlabel('Prediction')
plt.ylabel('Observation')
plt.title('Problem 1.B.iii: Observations vs Predictions (Train)')
plt.show()
```



And do the observations vs predictions plot for the test data:

In [35]:
```
test_prediction = model(X_test)
plt.scatter(test_prediction, y_test)
bestline = torch.linspace(y_test.min(), y_test.max(),30)
plt.plot(bestline,bestline, 'r')
plt.xlabel('Prediction')
plt.ylabel('Observation')
plt.title('Problem 1.B.iii: Observations vs Predictions (Test)')
plt.show()
```



Part C.I - Investigate the effect of the batch size

For the given network, try batch sizes of 10, 25, 50 and 100 for 400 epochs. In the sample plot, show the evolution of the test loss function for each case. Which batch sizes lead to faster training times and why? Which one would you choose?

Answer: See Print statement below

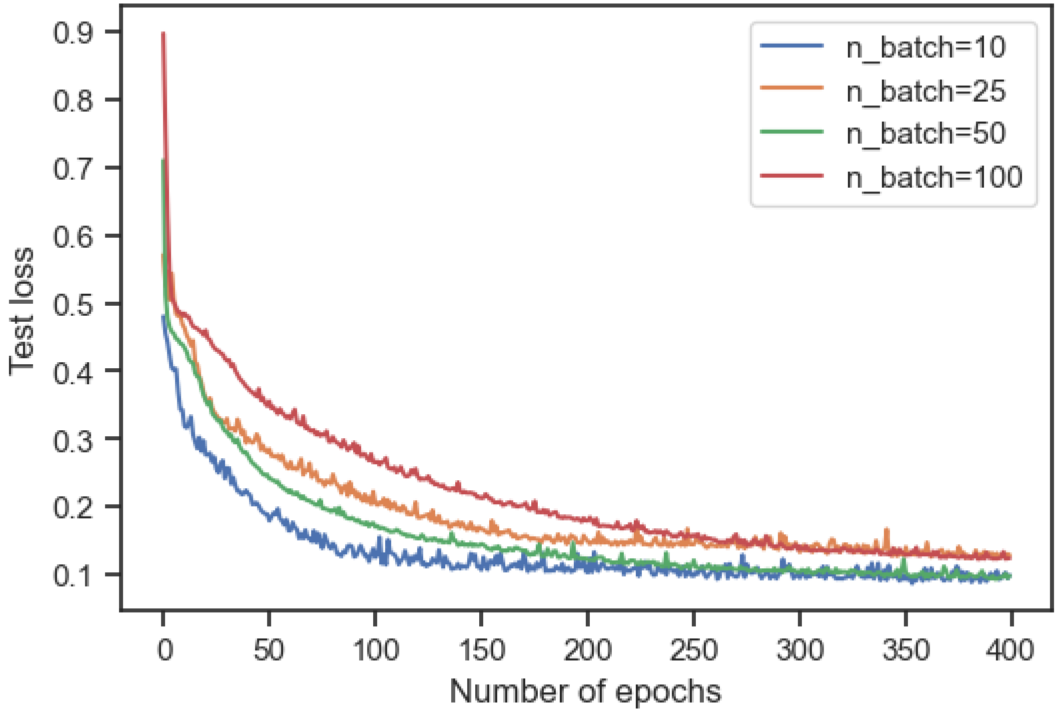
In [36]:
```
epochs = 400 # given
lr = lr # not sure what to pick
reg_weight = reg_weight # not sure what to pick
test_losses = []
models = []
batches = [10,25,50,100]# make me a list with the right batch sizes
for n_batch in batches:
    print('Training n_batch: {0:d}'.format(n_batch))
    net = nn.Sequential(nn.Linear(5, 20),
                        nn.Sigmoid(),
                        nn.Linear(20, 1))
```

```
    model, test_loss, X_train, y_train, X_test, y_test = train_net(
        X,
        y,
        net,
        reg_weight,
        n_batch,
        epochs,
        lr
    )
    test_losses.append(test_loss)
    models.append(model)

print('Problem 1.C.i: The smaller batch sizes are more accurate with less loss, but due to run time, a higher batch size
print('Larger batch sizes lead to faster training times due to the available memory space needed according to: (https://m
chosen_Batch = 50
```

```
Training n_batch: 10
100%|██████████| 400/400 [00:54<00:00,  7.34it/s]
Training n_batch: 25
100%|██████████| 400/400 [00:23<00:00, 17.25it/s]
Training n_batch: 50
100%|██████████| 400/400 [00:13<00:00, 30.72it/s]
Training n_batch: 100
100%|██████████| 400/400 [00:08<00:00, 48.17it/s]
Problem 1.C.i: The smaller batch sizes are more accurate with less loss, but due to run time, a higher batch size of 50 w
ill be used for the remaining segments!
Larger batch sizes lead to faster training times due to the available memory space needed according to: (https://medium.c
om/mini-distill/effect-of-batch-size-on-training-dynamics-21c14f7a716e)
```

In [37]:

```
fig, ax = plt.subplots(dpi=100)
for tl, n_batch in zip(test_losses, batches):
    ax.plot(tl, label='n_batch={0:d}'.format(n_batch))
ax.set_xlabel('Number of epochs')
ax.set_ylabel('Test loss')
plt.legend(loc='best');
```



Problem 1.C.i: The smaller batch sizes are more accurate with less loss, but due to run time, a higher batch size of 50 will be used for the remaining segments! Larger batch sizes lead to faster training times due to the available memory space needed according to: (https://medium.com/mini-distill/effect-of-batch-size-on-training-dynamics-21c14f7a716e)

Part C.II - Investigate the effect of the learning rate

Fix the batch size to best one you identified in Part C.I. For the given network, try learning rates of 1, 0.1, 0.01 and 0.001 for 400 epochs. In the sample plot, show the evolution of the test loss function for each case. Does the algorithm converge for all learning rates? Which learning rate would you choose?

Answer: See print statement below- they all converge but some are smoother than others

In [38]:

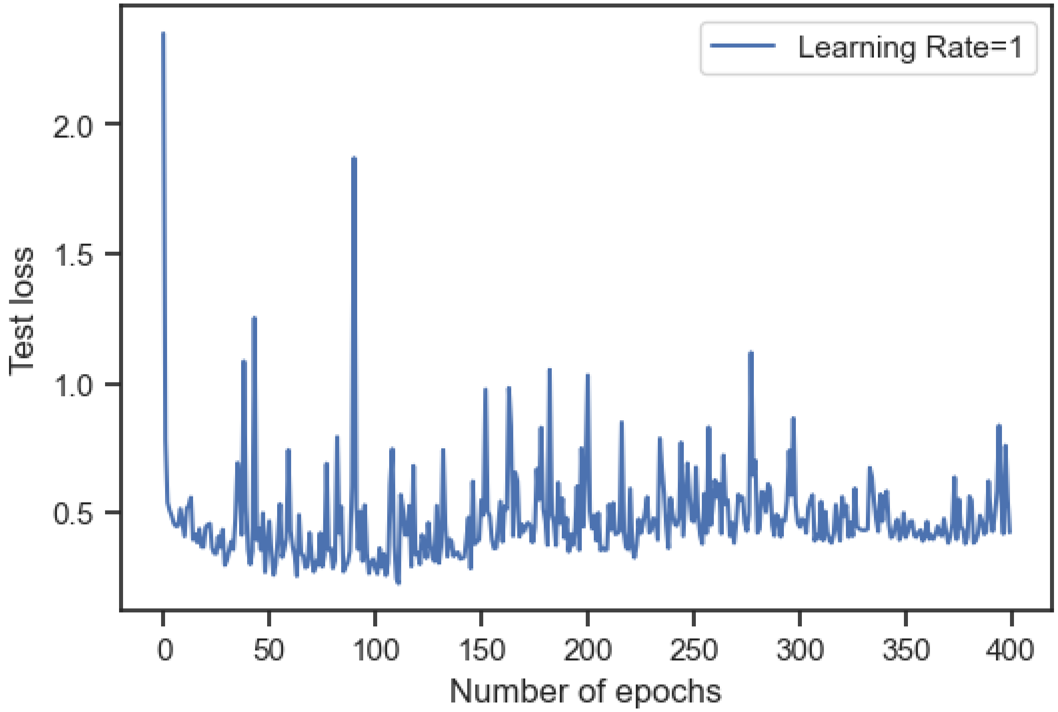
```
## Part 1C.ii
#yes I am being lazy and hard coding. Sorry



epochs = 400 # given
lr = 1
```

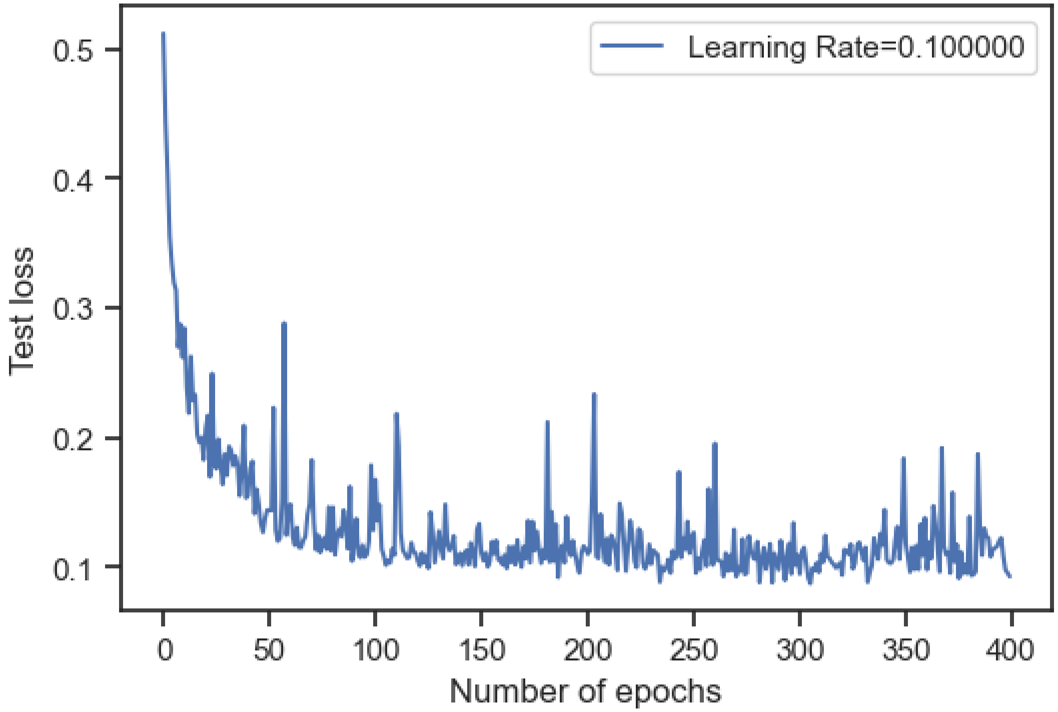
```
reg_weight = reg_weight # not sure what to pick
test_losses = []
models = []
batches = [chosen_Batch]# make me a list with the right batch sizes
for n_batch in batches:
    print('Training n_batch: {0:d}'.format(n_batch))
    net = nn.Sequential(nn.Linear(5, 20),
                        nn.Sigmoid(),
                        nn.Linear(20, 1))
    model, test_loss, X_train, y_train, X_test, y_test = train_net(
        X,
        y,
        net,
        reg_weight,
        n_batch,
        epochs,
        lr
    )
    test_losses.append(test_loss)
    models.append(model)

fig, ax = plt.subplots(dpi=100)
for tl, n_batch in zip(test_losses, batches):
    ax.plot(tl, label='Learning Rate={0:d}'.format(lr))
ax.set_xlabel('Number of epochs')
ax.set_ylabel('Test loss')
plt.legend(loc='best');

epochs = 400 # given
lr = .1
reg_weight = reg_weight # not sure what to pick
test_losses = []
models = []
batches = [chosen_Batch]# make me a list with the right batch sizes
for n_batch in batches:
    print('Training n_batch: {0:f}'.format(n_batch))
    net = nn.Sequential(nn.Linear(5, 20),
                        nn.Sigmoid(),
                        nn.Linear(20, 1))
    model, test_loss, X_train, y_train, X_test, y_test = train_net(
        X,
        y,
        net,
        reg_weight,
        n_batch,
```

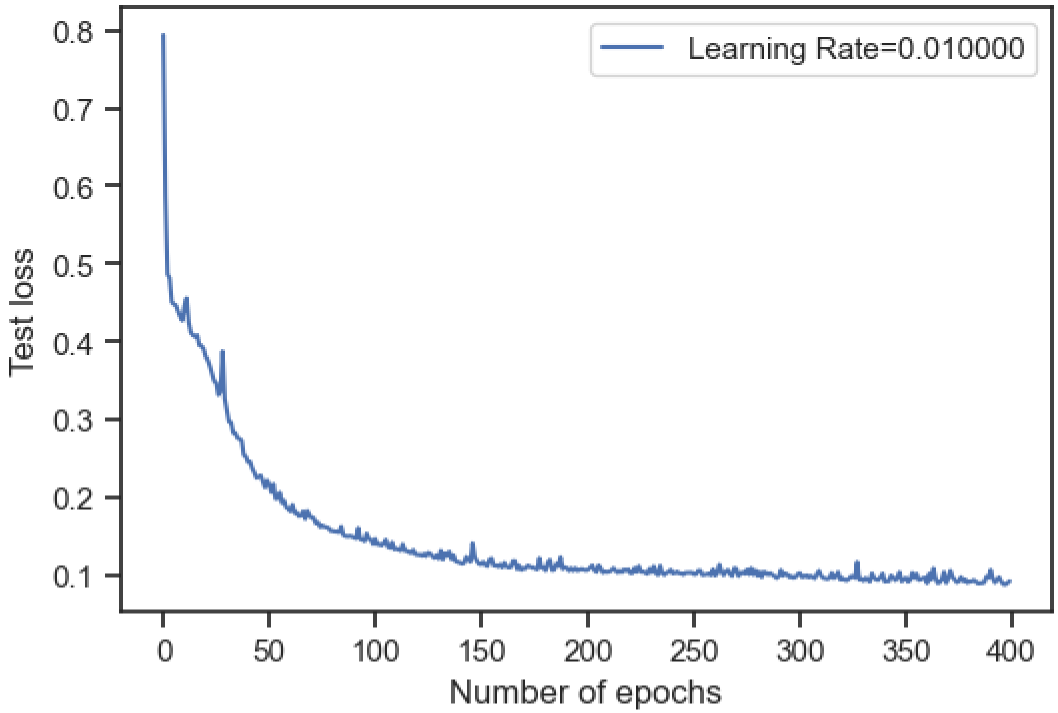
```
        epochs,
        lr
    )
    test_losses.append(test_loss)
    models.append(model)

fig, ax = plt.subplots(dpi=100)
for tl, n_batch in zip(test_losses, batches):
    ax.plot(tl, label='Learning Rate={0:f}'.format(lr))
ax.set_xlabel('Number of epochs')
ax.set_ylabel('Test loss')
plt.legend(loc='best');

epochs = 400 # given
lr = 0.01
reg_weight = reg_weight # not sure what to pick
test_losses = []
models = []
batches = [chosen_Batch]# make me a list with the right batch sizes
for n_batch in batches:
    print('Training n_batch: {0:f}'.format(n_batch))
    net = nn.Sequential(nn.Linear(5, 20),
                    nn.Sigmoid(),
                    nn.Linear(20, 1))
    model, test_loss, X_train, y_train, X_test, y_test = train_net(
        X,
        y,
        net,
        reg_weight,
        n_batch,
        epochs,
        lr
    )
    test_losses.append(test_loss)
    models.append(model)

fig, ax = plt.subplots(dpi=100)
for tl, n_batch in zip(test_losses, batches):
    ax.plot(tl, label='Learning Rate={0:f}'.format(lr))
ax.set_xlabel('Number of epochs')
ax.set_ylabel('Test loss')
plt.legend(loc='best');
epochs = 400 # given
lr = 0.001
reg_weight = reg_weight # not sure what to pick
```

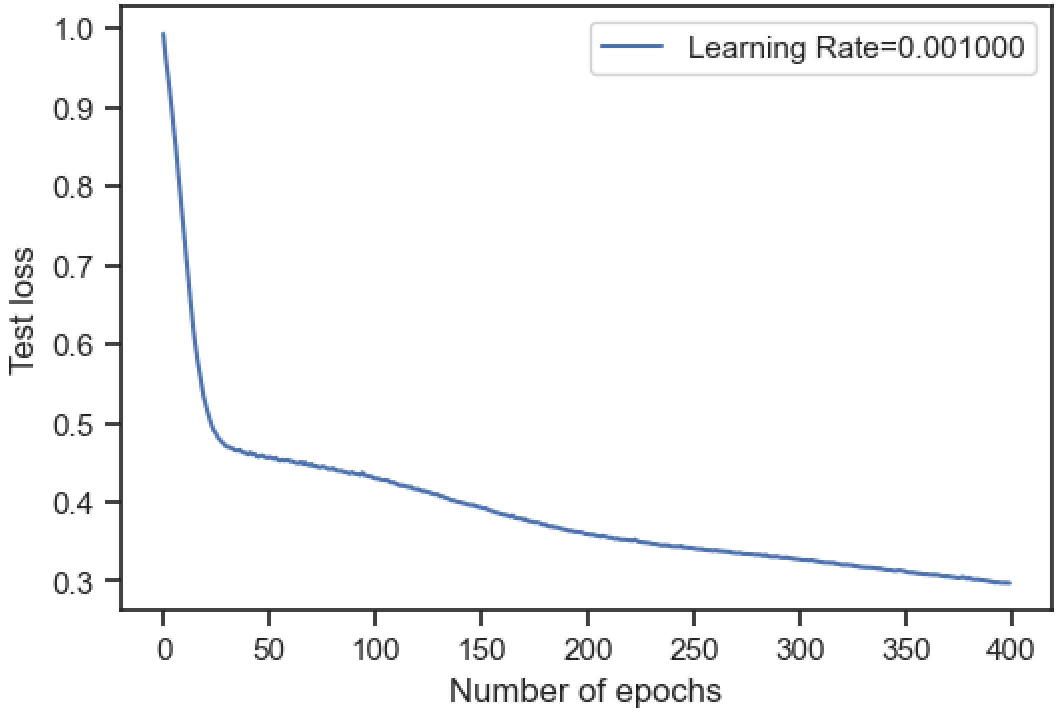
```
test_losses = []
models = []
batches = [chosen_Batch]# make me a list with the right batch sizes
for n_batch in batches:
    print('Training n_batch: {0:d}'.format(n_batch))
    net = nn.Sequential(nn.Linear(5, 20),
                    nn.Sigmoid(),
                    nn.Linear(20, 1))
    model, test_loss, X_train, y_train, X_test, y_test = train_net(
        X,
        y,
        net,
        reg_weight,
        n_batch,
        epochs,
        lr
    )
    test_losses.append(test_loss)
    models.append(model)

fig, ax = plt.subplots(dpi=100)
for tl, n_batch in zip(test_losses, batches):
    ax.plot(tl, label='Learning Rate={0:f}'.format(lr))
ax.set_xlabel('Number of epochs')
ax.set_ylabel('Test loss')
plt.legend(loc='best');

print('Problem 1.C.ii: The Larger Learning rates converge faster but are more noisy. Since all learning rates converge, I
chosen_learningrate = 0.001
```

```
Training n_batch: 50
100%|██████████| 400/400 [00:13<00:00, 30.74it/s]
Training n_batch: 50.000000
100%|██████████| 400/400 [00:12<00:00, 31.00it/s]
Training n_batch: 50.000000
100%|██████████| 400/400 [00:13<00:00, 30.53it/s]
Training n_batch: 50
100%|██████████| 400/400 [00:13<00:00, 30.20it/s]
Problem 1.C.ii: The Larger Learning rates converge faster but are more noisy. Since all learning rates converge, I am cho
osing the smallest to denoise
```


Learning Rate=1
Test loss
2.0
1.5
1.0
0.5
0
50
100
150
200
250
300
350
400
Number of epochs


Learning Rate=0.100000
Test loss
Number of epochs
0.5
0.4
0.3
0.2
0.1
0
50
100
150
200
250
300
350
400


Learning Rate=0.010000
Test loss
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
50
100
150
200
250
300
350
400
Number of epochs



Part C.III - Investigate the effect of the regularization weight

Fix the batch size to the value you selected in C.I and the learning rate to the value you selected in C.II. For the given network, try regularization weights of 0, 1e-16, 1e-12, 1e-6, and 1e-3 for 400 epochs. In the sample plot, show the evolution of the test loss function for each case. Which regularization weight seems to be the best and why?

Answer: See print statement below

In [39]:

```
## Part 1C.iii
#yes I am being lazy and hard coding. Sorry



epochs = 400 # given
lr = chosen_learningrate
reg_weight = 0 # not sure what to pick
test_losses = []
models = []
batches = [chosen_Batch]# make me a list with the right batch sizes
```

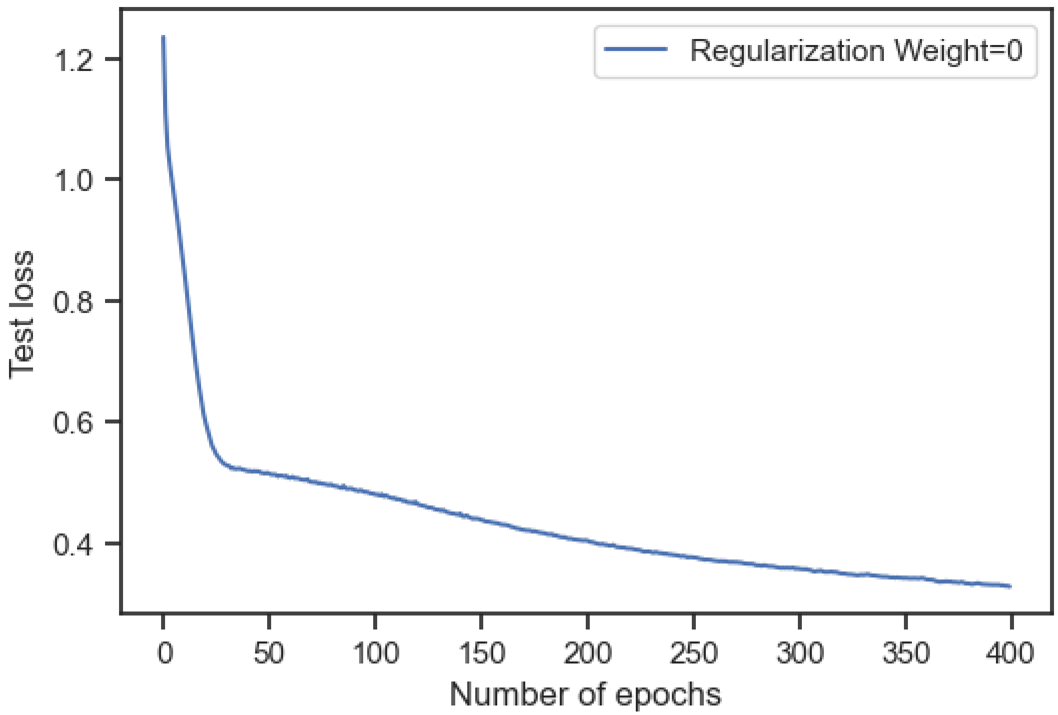
```
for n_batch in batches:
    print('Training n_batch: {0:f}'.format(n_batch))
    net = nn.Sequential(nn.Linear(5, 20),
                        nn.Sigmoid(),
                        nn.Linear(20, 1))
    model, test_loss, X_train, y_train, X_test, y_test = train_net(
        X,
        y,
        net,
        reg_weight,
        n_batch,
        epochs,
        lr
    )
    test_losses.append(test_loss)
    models.append(model)

fig, ax = plt.subplots(dpi=100)
for tl, n_batch in zip(test_losses, batches):
    ax.plot(tl, label='Regularization Weight=0')
ax.set_xlabel('Number of epochs')
ax.set_ylabel('Test loss')
plt.legend(loc='best');

epochs = 400 # given
lr = .1
reg_weight = 1E-16  # not sure what to pick
test_losses = []
models = []
batches = [chosen_Batch]# make me a list with the right batch sizes
for n_batch in batches:
    print('Training n_batch: {0:f}'.format(n_batch))
    net = nn.Sequential(nn.Linear(5, 20),
                        nn.Sigmoid(),
                        nn.Linear(20, 1))
    model, test_loss, X_train, y_train, X_test, y_test = train_net(
        X,
        y,
        net,
        reg_weight,
        n_batch,
        epochs,
        lr
    )
    test_losses.append(test_loss)
```

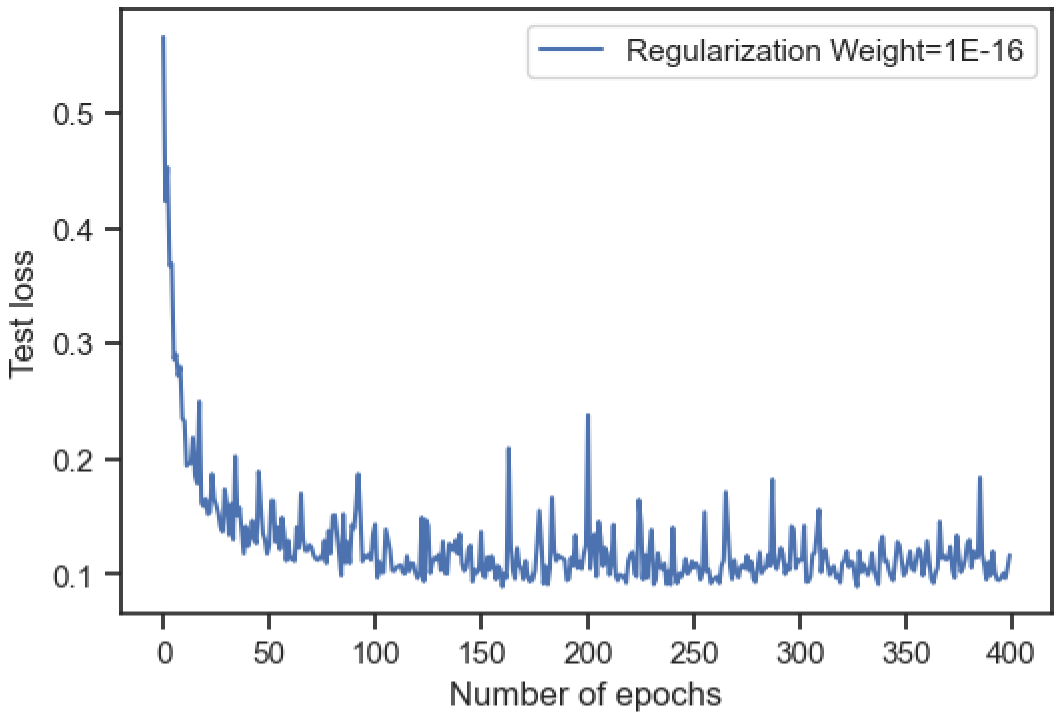
```
    models.append(model)

fig, ax = plt.subplots(dpi=100)
for tl, n_batch in zip(test_losses, batches):
    ax.plot(tl, label='Regularization Weight=1E-16')
ax.set_xlabel('Number of epochs')
ax.set_ylabel('Test loss')
plt.legend(loc='best');

epochs = 400 # given
lr = 0.01
reg_weight = 1E-12 # not sure what to pick
test_losses = []
models = []
batches = [chosen_Batch]# make me a list with the right batch sizes
for n_batch in batches:
    print('Training n_batch: {0:f}'.format(n_batch))
    net = nn.Sequential(nn.Linear(5, 20),
                        nn.Sigmoid(),
                        nn.Linear(20, 1))
    model, test_loss, X_train, y_train, X_test, y_test = train_net(
        X,
        y,
        net,
        reg_weight,
        n_batch,
        epochs,
        lr
    )
    test_losses.append(test_loss)
    models.append(model)

fig, ax = plt.subplots(dpi=100)
for tl, n_batch in zip(test_losses, batches):
    ax.plot(tl, label='Regularization Weight=1E-12')
ax.set_xlabel('Number of epochs')
ax.set_ylabel('Test loss')
plt.legend(loc='best');

epochs = 400 # given
lr = 0.001
reg_weight = 1E-6 # not sure what to pick
test_losses = []
models = []
batches = [chosen_Batch]# make me a list with the right batch sizes
```

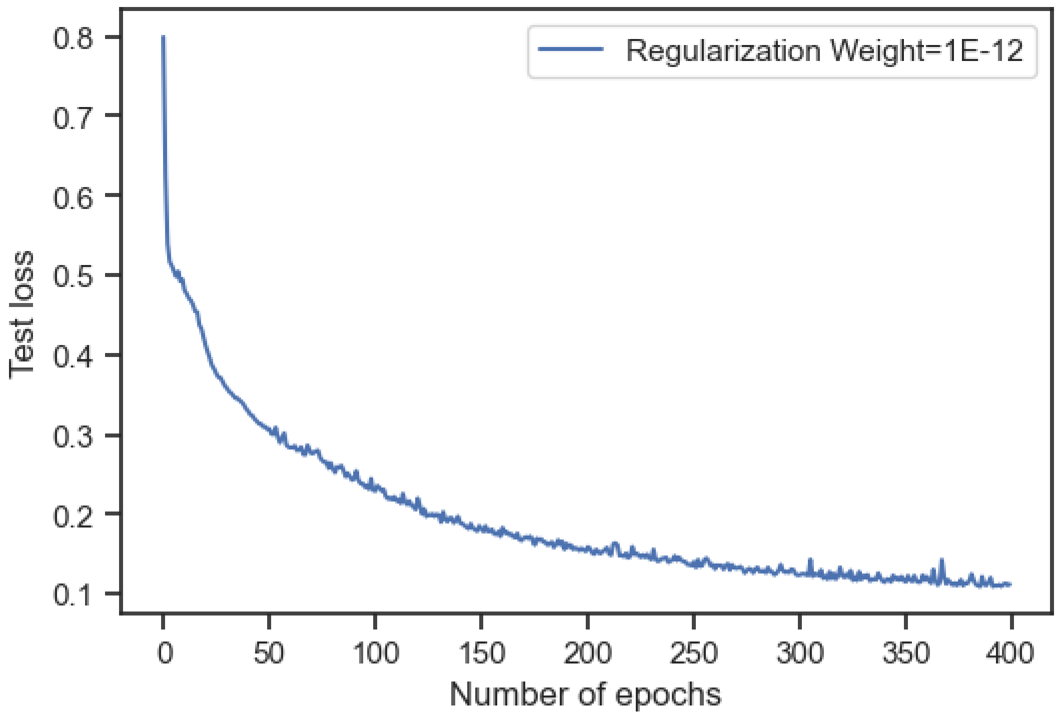
```
for n_batch in batches:
    print('Training n_batch: {0:d}'.format(n_batch))
    net = nn.Sequential(nn.Linear(5, 20),
                        nn.Sigmoid(),
                        nn.Linear(20, 1))
    model, test_loss, X_train, y_train, X_test, y_test = train_net(
        X,
        y,
        net,
        reg_weight,
        n_batch,
        epochs,
        lr
    )
    test_losses.append(test_loss)
    models.append(model)

fig, ax = plt.subplots(dpi=100)
for tl, n_batch in zip(test_losses, batches):
    ax.plot(tl, label='Regularization Weight=1E-6')
ax.set_xlabel('Number of epochs')
ax.set_ylabel('Test loss')
plt.legend(loc='best');

epochs = 400 # given
lr = 0.001
reg_weight = 1E-3 # not sure what to pick
test_losses = []
models = []
batches = [chosen_Batch]# make me a list with the right batch sizes
for n_batch in batches:
    print('Training n_batch: {0:d}'.format(n_batch))
    net = nn.Sequential(nn.Linear(5, 20),
                        nn.Sigmoid(),
                        nn.Linear(20, 1))
    model, test_loss, X_train, y_train, X_test, y_test = train_net(
        X,
        y,
        net,
        reg_weight,
        n_batch,
        epochs,
        lr
    )
    test_losses.append(test_loss)
```

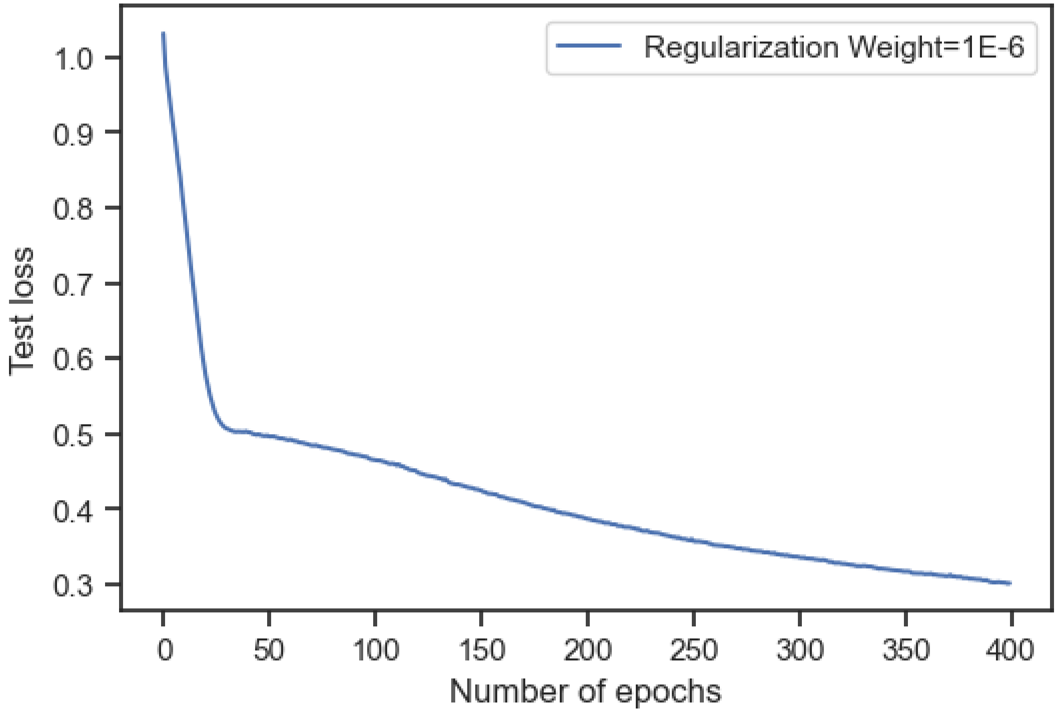
```
    models.append(model)

fig, ax = plt.subplots(dpi=100)
for tl, n_batch in zip(test_losses, batches):
    ax.plot(tl, label='Regularization Weight=1E-3')
ax.set_xlabel('Number of epochs')
ax.set_ylabel('Test loss')
plt.legend(loc='best');

chosen_reg_weight = 1E-12
print('Problem 1.C.iii: The best regularization weight appears to be 1E-12 as it has the fastest convergence with minimal
```

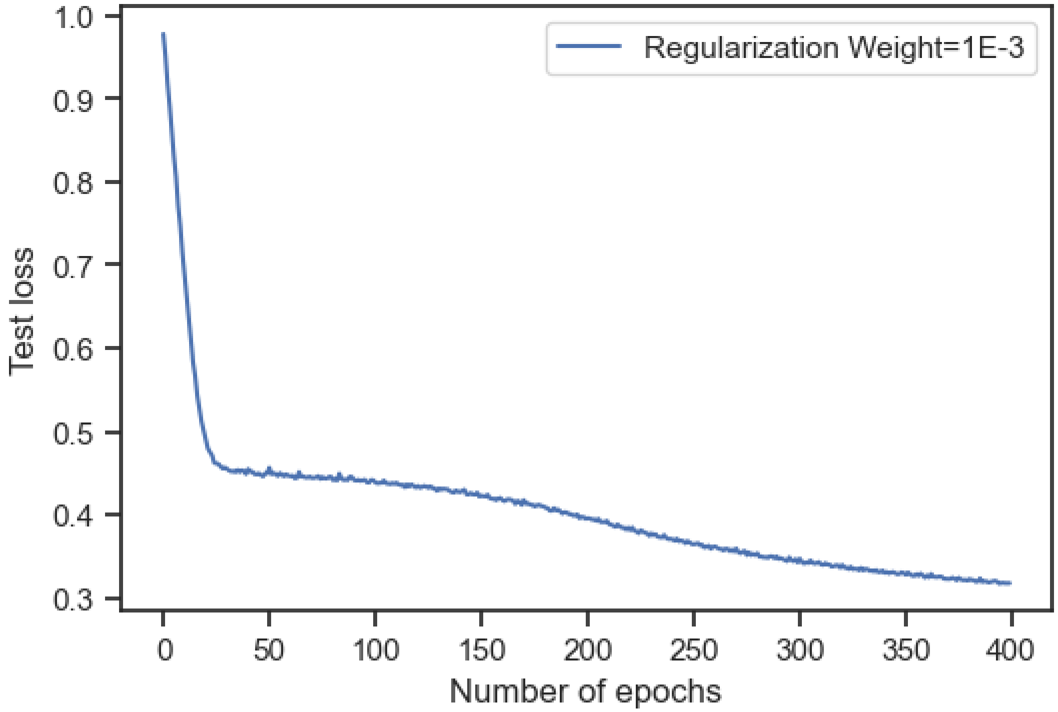
```
Training n_batch: 50.000000
100%|██████████| 400/400 [00:13<00:00, 28.72it/s]
Training n_batch: 50.000000
100%|██████████| 400/400 [00:20<00:00, 19.86it/s]
Training n_batch: 50.000000
100%|██████████| 400/400 [00:16<00:00, 24.02it/s]
Training n_batch: 50
100%|██████████| 400/400 [00:12<00:00, 31.77it/s]
Training n_batch: 50
100%|██████████| 400/400 [00:12<00:00, 31.22it/s]
Problem 1.C.iii: The best regularization weight appears to be 1E-12 as it has the fastest convergence with minimal noise
```


Regularization Weight=0
Test loss
1.2
1.0
0.8
0.6
0.4
0
50
100
150
200
250
300
350
400
Number of epochs


Regularization Weight=1E-16
Test loss
0.1
0.2
0.3
0.4
0.5
0
50
100
150
200
250
300
350
400
Number of epochs


Regularization Weight=1E-12
Test loss
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
50
100
150
200
250
300
350
400
Number of epochs


Regularization Weight=1E-6
Test loss
Number of epochs
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0
50
100
150
200
250
300
350
400



Part D.I - Train a bigger network

Now that you have developed some intuition about the parameters involved in training a network, train a larger one. In particular, use a 5-layer deep network with 100 neurons per layer. You can use the sigmoid activation function or you can change it to something else. Make sure you plot:

- the evolution of the test loss a a function of the epochs
- the observations vs predictions plot for the test data

Answer:

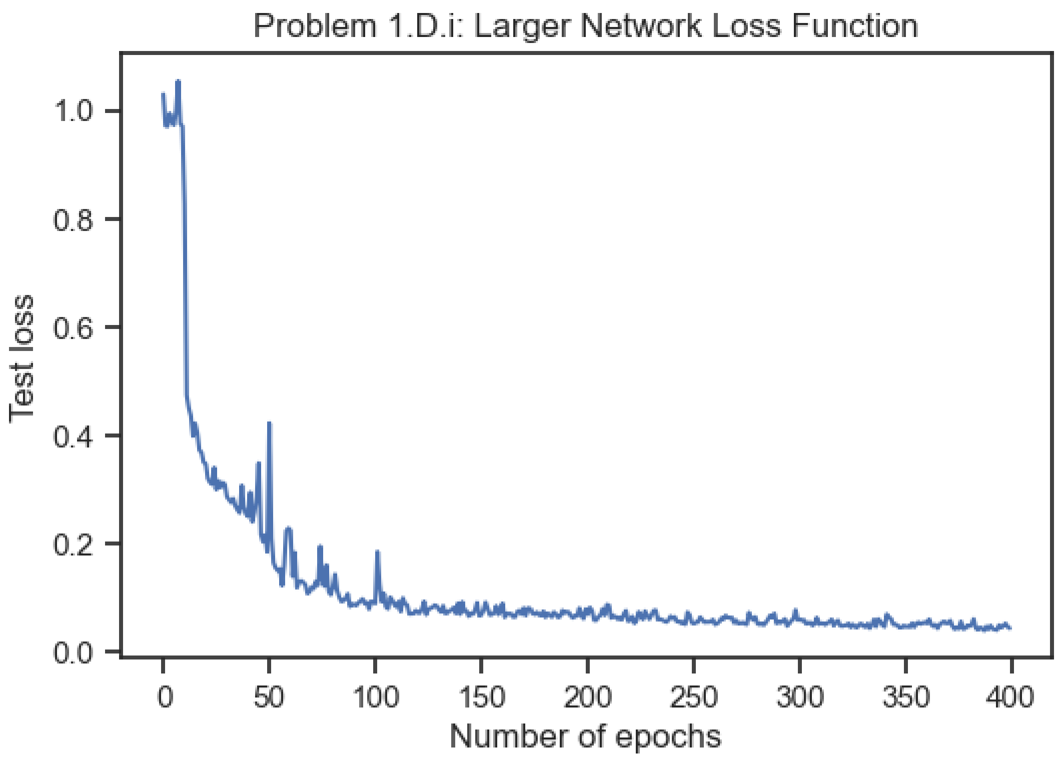
In [104…

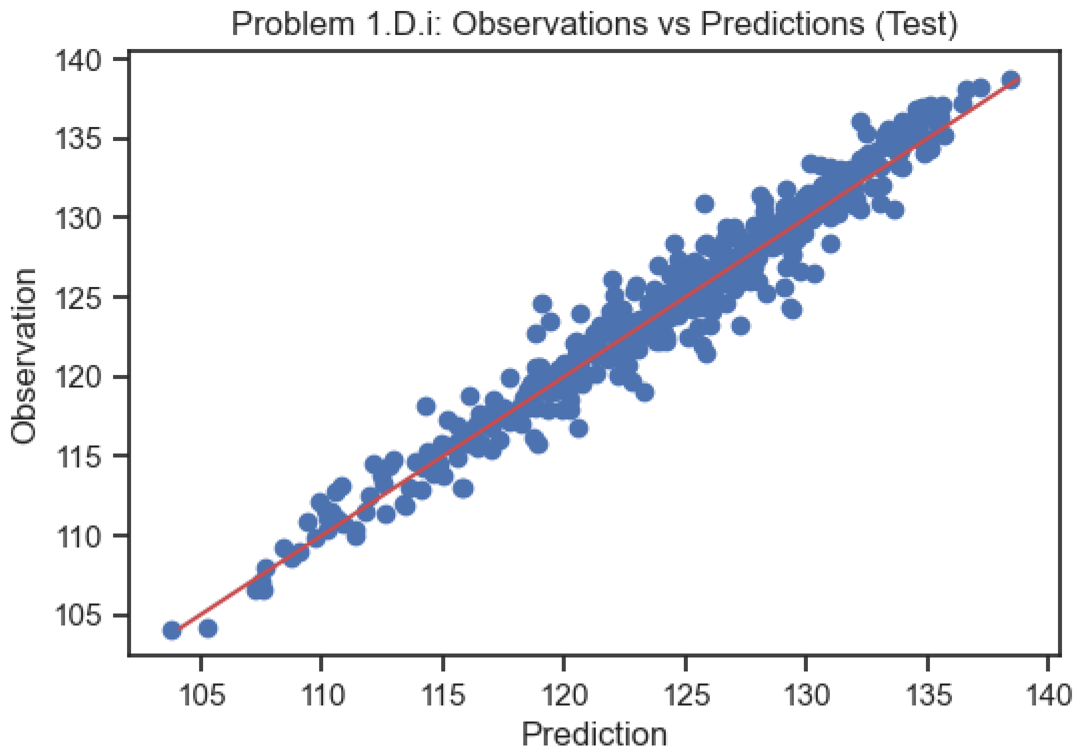
```
#part D.i


epochs2 = 400 # given
lr2 = 0.01
reg_weight2 = 0 #best at 0
test_losses2 = []
models2 = []
```

```
batches2 = [100]
for n_batch2 in batches2:
    print('Training n_batch: {0:d}'.format(n_batch2))
    net2 = nn.Sequential(nn.Linear(5, 100),
                    nn.Sigmoid(),
                    nn.Linear(100, 100),
                    nn.Sigmoid(),
                    nn.Linear(100, 100),
                    nn.Sigmoid(),
                    nn.Linear(100, 100),
                    nn.Sigmoid(),
                    nn.Linear(100, 1))
    model2, test_loss2, X_train2, y_train2, X_test2, y_test2 = train_net(
        X,
        y,
        net2,
        reg_weight2,
        n_batch2,
        epochs2,
        lr2
    )
    test_losses2.append(test_loss2)
    models2.append(model2)

fig, ax = plt.subplots(dpi=100)
for tl, n_batch2 in zip(test_losses2, batches2):
    ax.plot(tl)
ax.set_xlabel('Number of epochs')
plt.title('Problem 1.D.i: Larger Network Loss Function')
ax.set_ylabel('Test loss')
#plt.legend(loc='best');

print('Model parameters (lr, batch, and regularization weight) all are found with trial and error. Batch size remained hi

test_prediction2 = model2(X_test2)
plt.figure()
plt.scatter(test_prediction2, y_test2)
bestline = torch.linspace(y_test2.min(), y_test2.max(),30)
plt.plot(bestline,bestline, 'r')
plt.xlabel('Prediction')
plt.ylabel('Observation')
plt.title('Problem 1.D.i: Observations vs Predictions (Test)')
plt.show()
```

```
Training n_batch: 100
```

```
100%|██████████| 400/400 [00:28<00:00, 14.04it/s]
Model parameters (lr, batch, and regularization weight) all are found with trial and error. Batch size remained higher th
an ideal for the sake of computing time
```





Part D.II - Make a prediction

Visualize the scaled sound level as a function of the streem velocity for a fixed frequency of 2500 Hz, a chord lentgh of 0.1 m, a sucction side displacement thickness of 0.01 m, and an angle of attack of 0, 5, and 10 degrees.

Answer:

This is just a sanity check for your model. You will just have to run the following code segmenets for the best model you have found.

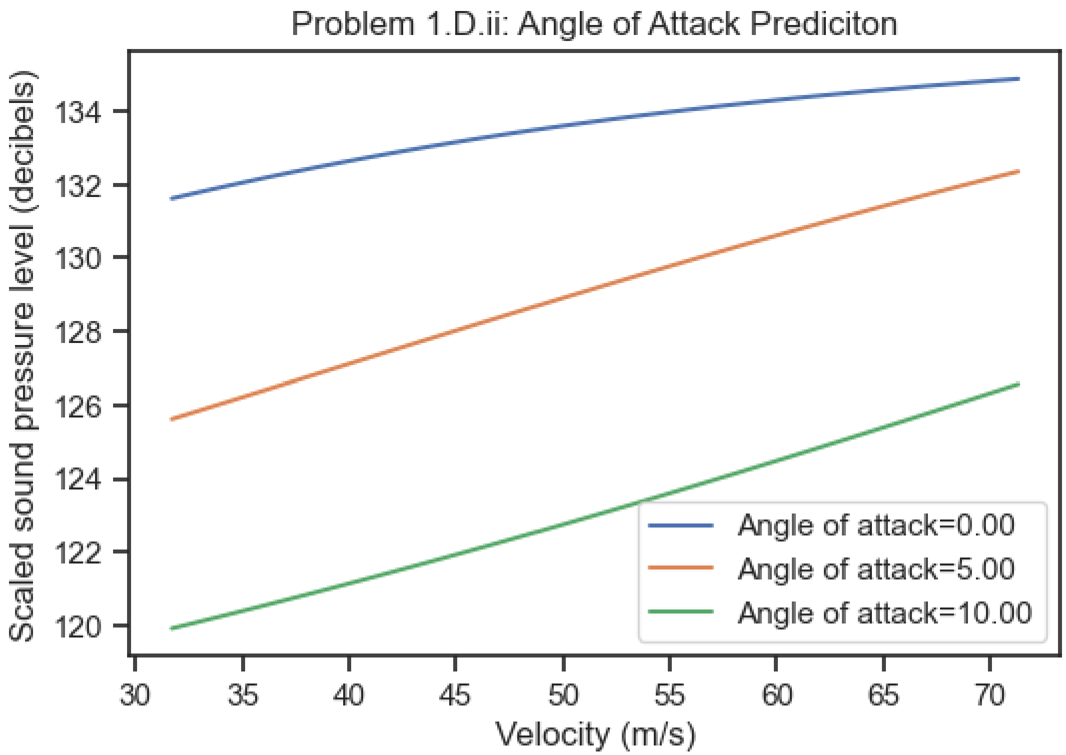
In [90]:

```
#part1.D.ii
best_model = model2# set this equal to your best model

def plot_sound_level_as_func_of_stream_vel(
    freq=2500,
    angle_of_attack=10,
    chord_length=0.1,
    suc_side_disp_thick=0.01,
    ax=None,
```

```
    label=None
):

    if ax is None:
        fig, ax = plt.subplots(dpi=100)

    # The velocities on which we want to evaluate the model
    vel = np.linspace(X[:, 3].min(), X[:, 3].max(), 100)[:, None]

    # Make the input for the model
    freqs = freq * np.ones(vel.shape)
    angles = angle_of_attack * np.ones(vel.shape)
    chords = chord_length * np.ones(vel.shape)
    sucs = suc_side_disp_thick * np.ones(vel.shape)

    # Put all these into a single array
    XX = np.hstack([freqs, angles, chords, vel, sucs])

    ax.plot(vel, best_model(XX), label=label)

    ax.set_xlabel('Velocity (m/s)')
    ax.set_ylabel('Scaled sound pressure level (decibels)')
```

In [42]:

```
fig, ax = plt.subplots(dpi=100)
for aofa in [0, 5, 10]:
    plot_sound_level_as_func_of_stream_vel(
        angle_of_attack=aofa,
        ax=ax,
        label='Angle of attack={0:1.2f}'.format(aofa)
)
plt.legend(loc='best');
plt.title('Problem 1.D.ii: Angle of Attack Prediciton');
```



Problem 2 - Classification with DNNs

This homework problem was kindly provided by Dr. Ali Lenjani. It is based on our joint work on this paper: Hierarchical convolutional neural networks information fusion for activity source detection in smart buildings. The data come from the Human Activity Benchmark published by Dr. Juan M. Caicedo.

So the problem is as follows. You want to put sensors on a building so that it can figure out what is going on insider it. This has applications in industrial facilities (e.g., detecting if there was an accident), public infrastructure, hospitals (e.g., did a patient fall off a bed), etc. Typically, the problem is addressed using cameras. Instead of cameras, we are going to investigate the ability of acceleration sensors to tell us what is going on.

Four acceleration sensors have been placed in different locations in the benchmark building to record the floor vibration signals of different objects falling from several heights. A total of seven cases cases were considered:

- **bag-high:** 450 g bag containing plastic pieces is dropped roughly from 2.10 m

- **bag-low:** 450 g bag containing plastic pieces is dropped roughly from 1.45 m
- **ball-high:** 560 g basketball is dropped roughly from 2.10 m
- **ball-low:** 560 g basketball is dropped roughly from 1.45 m
- **j-jump:** person 1.60 m tall, 55 kg jumps approximately 12 cm high
- **d-jump:** person 1.77 m tall, 80 kg jumps approximately 12 cm high
- **w-jump:** person 1.85 m tall, 85 kg jumps approximately 12 cm high

Each of these seven cases was repeated 115 times at 5 different locations of the building. The original data are here, but I have repackaged them for you in a more convenient format. Let's download them:

In [46]:
```
!curl -O 'https://dl.dropboxusercontent.com/s/n8dczk7t8bx0pxi/human_activity_data.npz'
```

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed

  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0curl: (6) Could not resolve host: 'https
```

Here is how to load the data:

In [47]:
```
data = np.load('human_activity_data.npz')
```

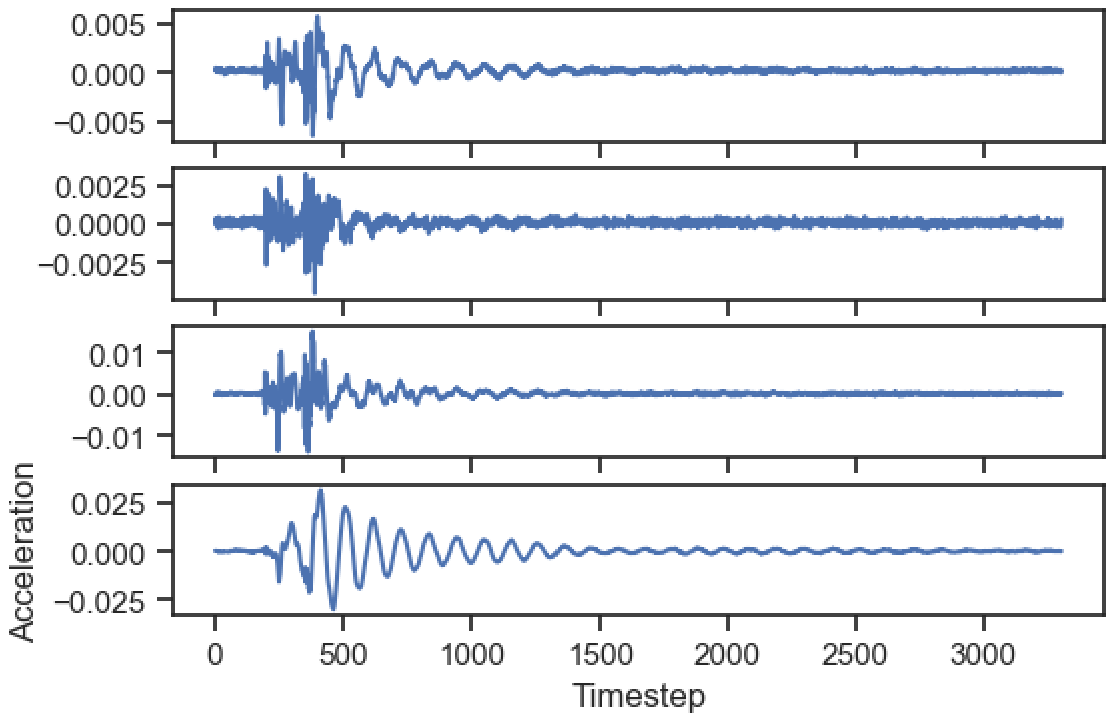
This is a Python dictionary that contains the following entries:

In [48]:
```
for key in data.keys():
    print(key, ':', data[key].shape)
```

```
features : (4025, 4, 3305)
labels_1 : (4025,)
labels_2 : (4025,)
loc_ids : (4025,)
```

Let's go over these one by one. First, the `features`. These are the accelertion sensor measurements. Here is how you visualize them:

In [50]:
```
fig, ax = plt.subplots(4, 1, dpi=100)
# Loop over sensors
for j in range(4):
        ax[j].plot(data['features'][0, j])
ax[-1].set_xlabel('Timestep')
ax[-1].set_ylabel('Acceleration');
```



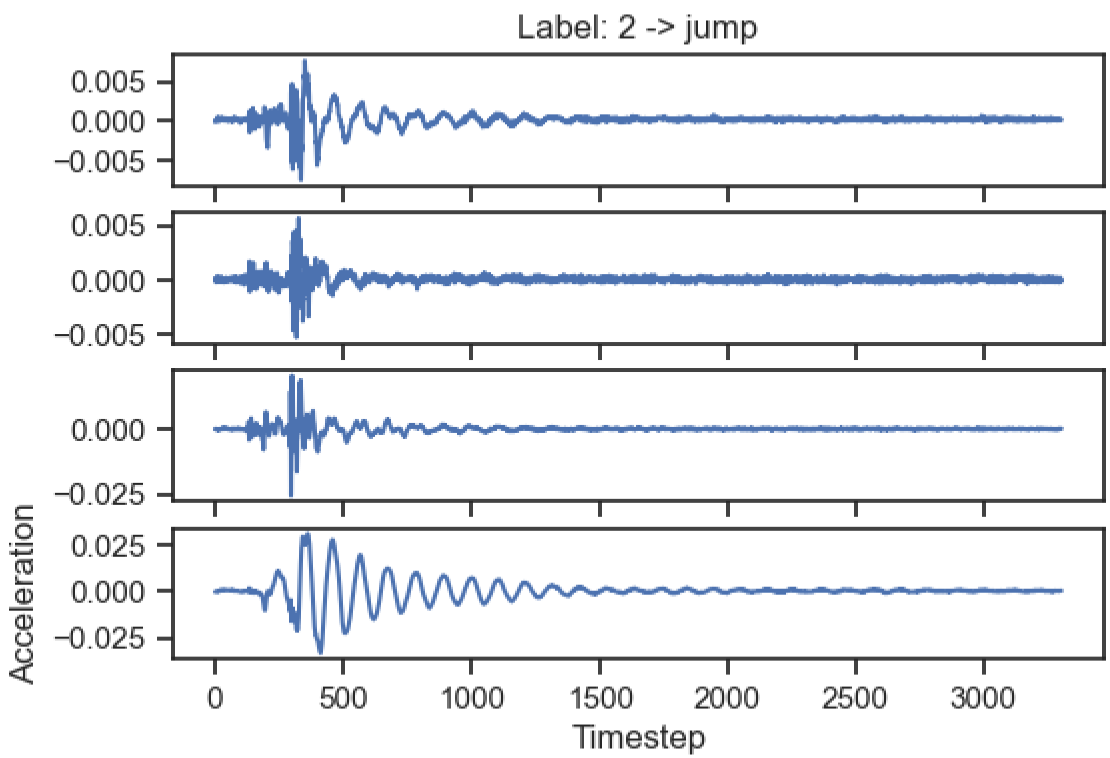
The second key, `labels_1` , is a bunch of integers ranging from 0 to 2 indicating whether the entry corresponds to a "bag," a "ball" or a "jump." For your reference, the correspondence is:

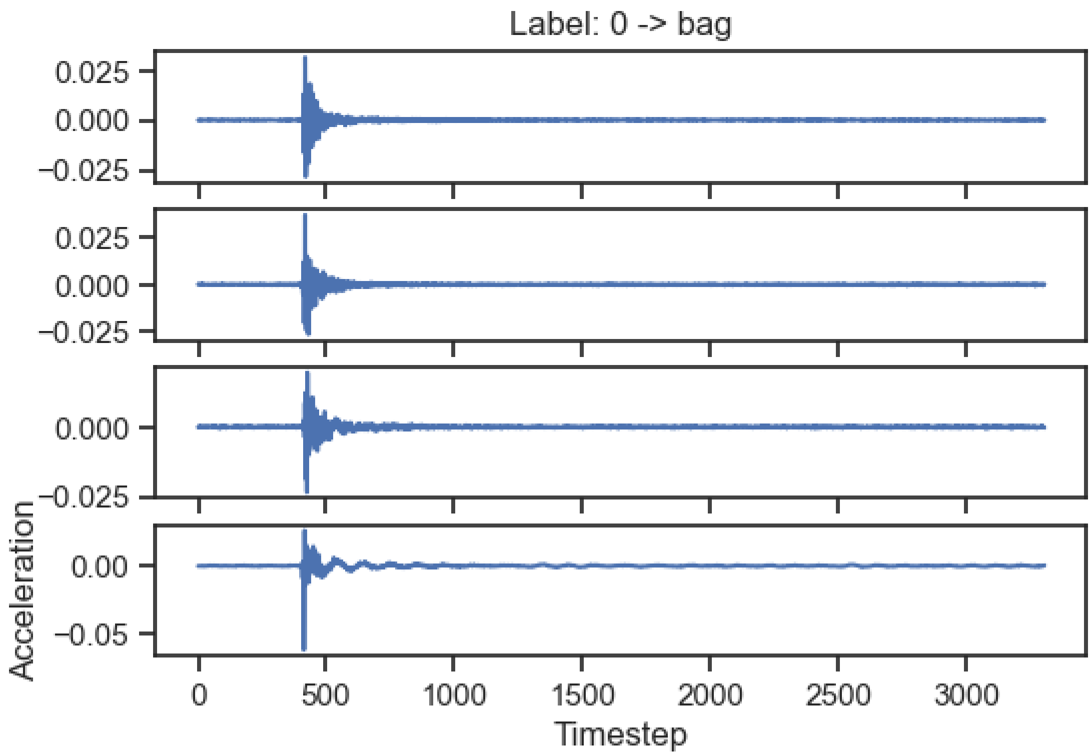
In [51]:
```
LABELS_1_TO_TEXT = {
    0: 'bag',
    1: 'ball',
    2: 'jump'
}
```

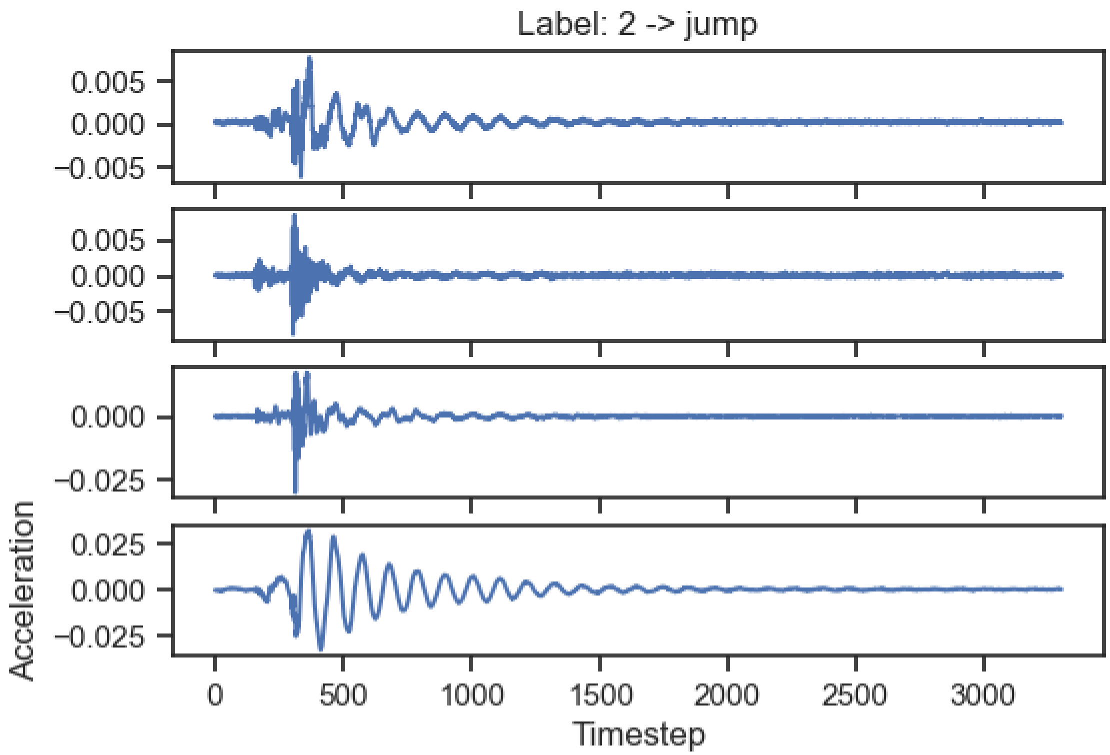
And here are a few examples:

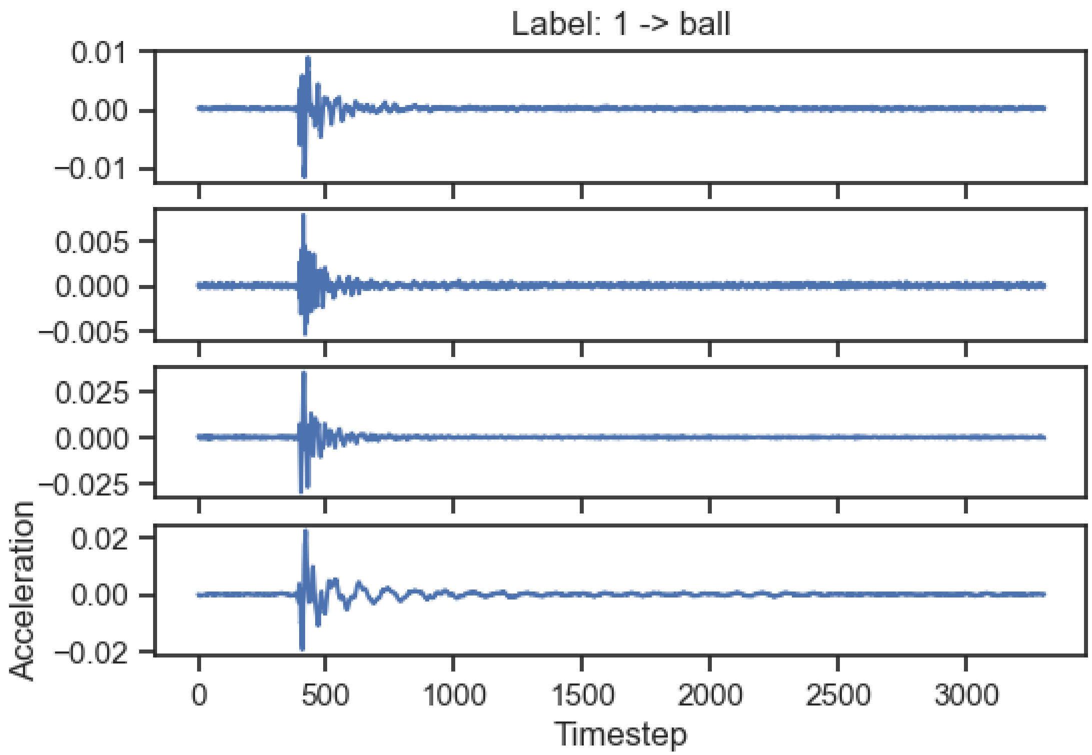
In [52]:
```
for _ in range(5):
    i = np.random.randint(0, data['features'].shape[0])
    fig, ax = plt.subplots(4, 1, dpi=100)
    for j in range(4):
        ax[j].plot(data['features'][i, j])
    ax[-1].set_xlabel('Timestep')
    ax[-1].set_ylabel('Acceleration')
```

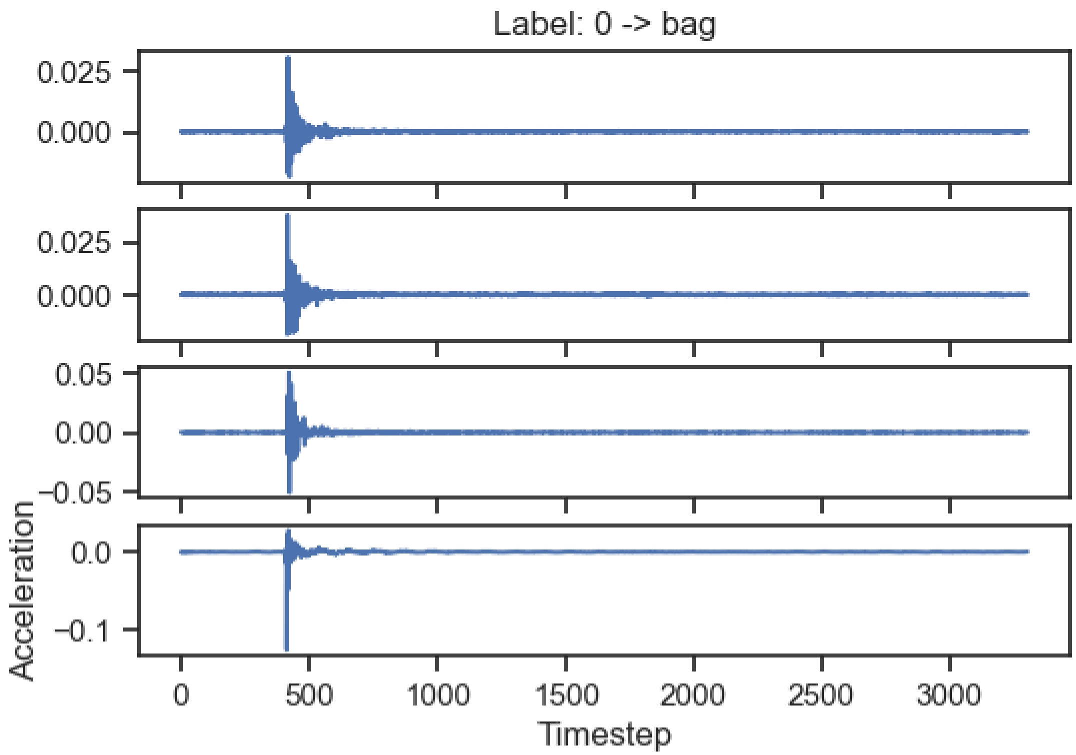
```
    ax[0].set_title('Label: {0:d} -> {1:s}'.format(data['labels_1'][i],
                                                   LABELS_1_TO_TEXT[data['labels_1'][i]]))
```




Label: 0 -> bag
0.025
0.000
−0.025
0.025
0.000
−0.025
0.000
−0.025
0.00
−0.05
Acceleration
0
500
1000
1500
2000
2500
3000
Timestep


Label: 2 -> jump
0.005
0.000
−0.005
0.005
0.000
−0.005
0.000
−0.025
0.025
0.000
−0.025
Acceleration
0
500
1000
1500
2000
2500
3000
Timestep


Label: 1 -> ball
0.01
0.00
−0.01
0.005
0.000
−0.005
0.025
0.000
−0.025
0.02
0.00
−0.02
Acceleration
0
500
1000
1500
2000
2500
3000
Timestep



The array `labels_2` includes integers from 0 to 6 indicating the detailed label of the experiment. The correspondence between integers and text labels is:

In [69]:

```
LABELS_2_TO_TEXT = {
    0: 'bag-high',
    1: 'bag-low',
    2: 'ball-high',
    3: 'ball-low',
    4: 'd-jump',
    5: 'j-jump',
    6: 'w-jump'
}
```

Finally, the field `loc_ids` takes values from 0 to 4 indicating five distinct locations in the building.

Before moving forward with the questions, let's extract the data in a more covenient form:

In [70]:

```
# The features
```

```
X = data['features']
# The labels_1
y1 = data['labels_1']
# The labels_2
y2 = data['labels_2']
# The locations
y3 = data['loc_ids']
```

Part A - Train a CNN to predict the the high-level type of observation (bag, ball, or jump)

Fill in the blanks in the code blocks below to train a classification neural network that is going to take you from the four acceleration sensor data to the high-level type of each observation. You can keep the structure of the network fixed, but you can experiment with the learning rate, the number of epochs, or anything else. Just keep in mind that for this particular dataset it is possible to hit an accuracy of almost 100%.

Answer:

The first thing that we need to do is pick a neural network structure. I suggest that we use 1D convolutional layers at the very beginning. These are the same as the 2D (image) convolutional layers, but in 1D. The reason I am proposing this is mainly that the convolutional layers are invariant to small translations of the acceleration signal (just like the labels are). Here is what I propose:

In [71]:

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):
    def __init__(self, num_labels=3):
        super(Net, self).__init__()
        # A convolutional layer:
        # 3 = input channels (sensors),
        # 6 = output channels (features),
        # 5 = kernel size
        self.conv1 = nn.Conv1d(4, 8, 10)
        # A 2 x 2 max pooling layer - we are going to use it two times
        self.pool = nn.MaxPool1d(5)
        # Another convolutional layer
        self.conv2 = nn.Conv1d(8, 16, 5)
        # Some linear layers
        self.fc1 = nn.Linear(16 * 131, 200)
        self.fc2 = nn.Linear(200, 50)
        self.fc3 = nn.Linear(50, num_labels)
```

```
    def forward(self, x):
        # This function implements your network output
        # Convolutional layer, followed by relu, followed by max pooling
        x = self.pool(F.relu(self.conv1(x)))
        # Same thing
        x = self.pool(F.relu(self.conv2(x)))
        # Flatting the output of the convolutional layers
        x = x.view(-1, 16 * 131)
        # Go throught the first dense linear layer followed by relu
        x = F.relu(self.fc1(x))
        # Through the second dense layer
        x = F.relu(self.fc2(x))
        # Finish up with a linear transformation
        x = self.fc3(x)
        return x
```

In [72]:

```
# You can make the network like this:
net = Net(3)
```

Now, you need to pick the right loss function for classification tasks:

In [74]:

```
cnn_loss_func = nn.CrossEntropyLoss()
```

Just like before, let's organize our training code in a convenient function that allows us to play with the parameters of training. Fill in the missing code.

In [75]:

```
import torch.optim as optim
def train_cnn(X, y, net, n_batch, epochs, lr, test_size=0.33):
    """
    A function that trains a regression neural network using stochatic gradient
    descent and returns the trained network. The loss function being minimized is
    `loss_func`.

    Parameters:

    X           -    The observed features
    y           -    The observed targets
    net         -    The network you want to fit
    n_batch     -    The batch size you want to use for stochastic optimization
    epochs      -    How many times do you want to pass over the training dataset.
```

```
    lr          -    The learning rate for the stochastic optimization algorithm.
    test_size   -    What percentage of the data should be used for testing (validation).
    """
    # Split the data
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33)

    # Turn all the numpy arrays to torch tensors
    X_train = torch.Tensor(X_train)
    X_test = torch.Tensor(X_test)
    y_train = torch.LongTensor(y_train)
    y_test = torch.LongTensor(y_test)

    # This is pytorch magick to enable shuffling of the
    # training data every time we go through them
    train_dataset = torch.utils.data.TensorDataset(X_train, y_train)
    train_data_loader = torch.utils.data.DataLoader(train_dataset,
                                                    batch_size=n_batch,
                                                    shuffle=True)

    # Create an Adam optimizing object for the neural network `net`
    # with learning rate `lr`

    optimizer = optim.Adam(net.parameters(),lr = lr)

    # This is a place to keep track of the test loss
    test_loss = []
    # This is a place to keep track of the accuracy on each epoch
    accuracy = []

    # Iterate the optimizer.
    # Remember, each time we go through the entire dataset we complete an `epoch`
    # I have wrapped the range around tqdm to give you a nice progress bar
    # to look at
    for e in range(epochs):
        # This loop goes over all the shuffled training data
        # That's why the DataLoader class of PyTorch is convenient
        for X_batch, y_batch in train_data_loader:
            # Perform a single optimization step with loss function
            # Hint 1: You have defined cnn_loss_func() already
            # Hint 2: Consult the hands-on activities for an example
            optimizer.zero_grad()
            y_pred= net(X_batch) ## output
            #reg_weight = 0
            loss = cnn_loss_func( y_pred, y_batch)#, reg_weight,net.parameters())
            loss.backward()
```

```
            optimizer.step()

        # Evaluate the test loss and append it on the list `test_loss`
        y_pred_test = net(X_test)
        ts_loss = cnn_loss_func(y_pred_test, y_test)
        test_loss.append(ts_loss.item())
        # Evaluate the accuracy
        _, predicted = torch.max(y_pred_test.data, 1)
        correct = (predicted == y_test).sum().item()
        accuracy.append(correct / y_test.shape[0])
        # Print something about the accuracy
        print('Epoch {0:d}: accuracy = {1:1.5f}%'.format(e+1, accuracy[-1]*100))
    trained_model = net

    # Return everything we need to analyze the results
    return trained_model, test_loss, accuracy, X_train, y_train, X_test, y_test
```

Now experiment with the epochs, the learning rate, and the batch size until this works.

In [76]:

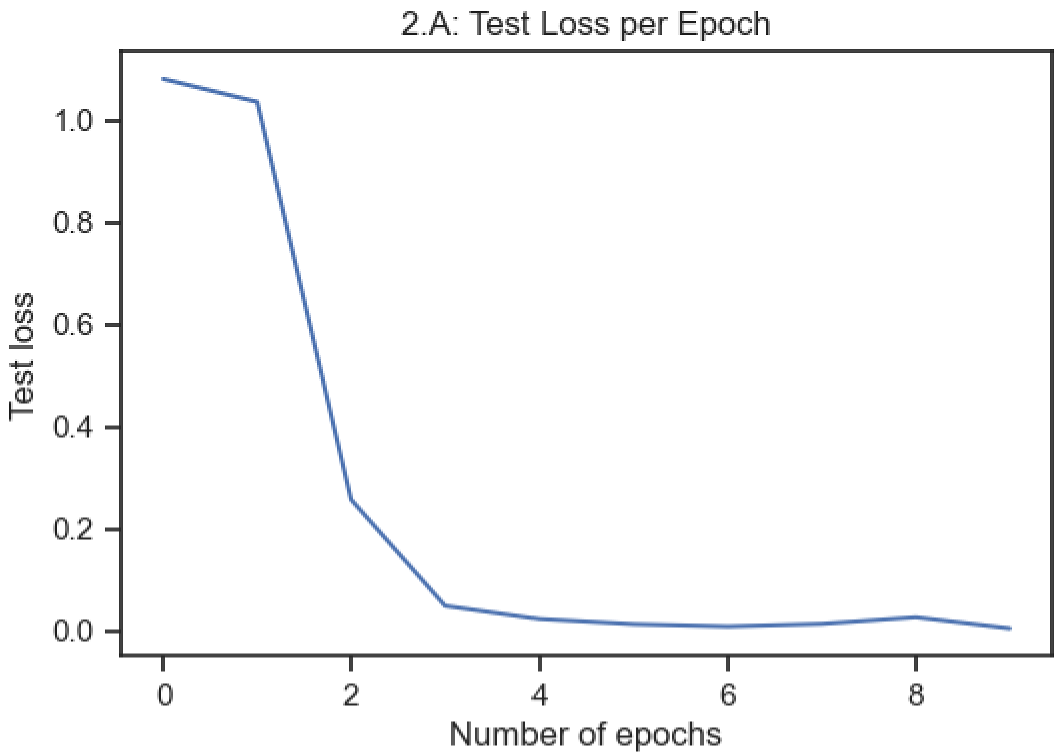
```
epochs = 10
lr = 0.01
n_batch = 100
trained_model, test_loss, accuracy, X_train, y_train, X_test, y_test = train_cnn(X, y1, net, n_batch, epochs, lr)
```

```
Epoch 1: accuracy = 42.66366%
Epoch 2: accuracy = 42.66366%
Epoch 3: accuracy = 90.66968%
Epoch 4: accuracy = 98.57035%
Epoch 5: accuracy = 99.24755%
Epoch 6: accuracy = 99.54853%
Epoch 7: accuracy = 99.84951%
Epoch 8: accuracy = 99.54853%
Epoch 9: accuracy = 99.17231%
Epoch 10: accuracy = 99.92476%
```

Plot the evolution of the test loss as a function of epochs.

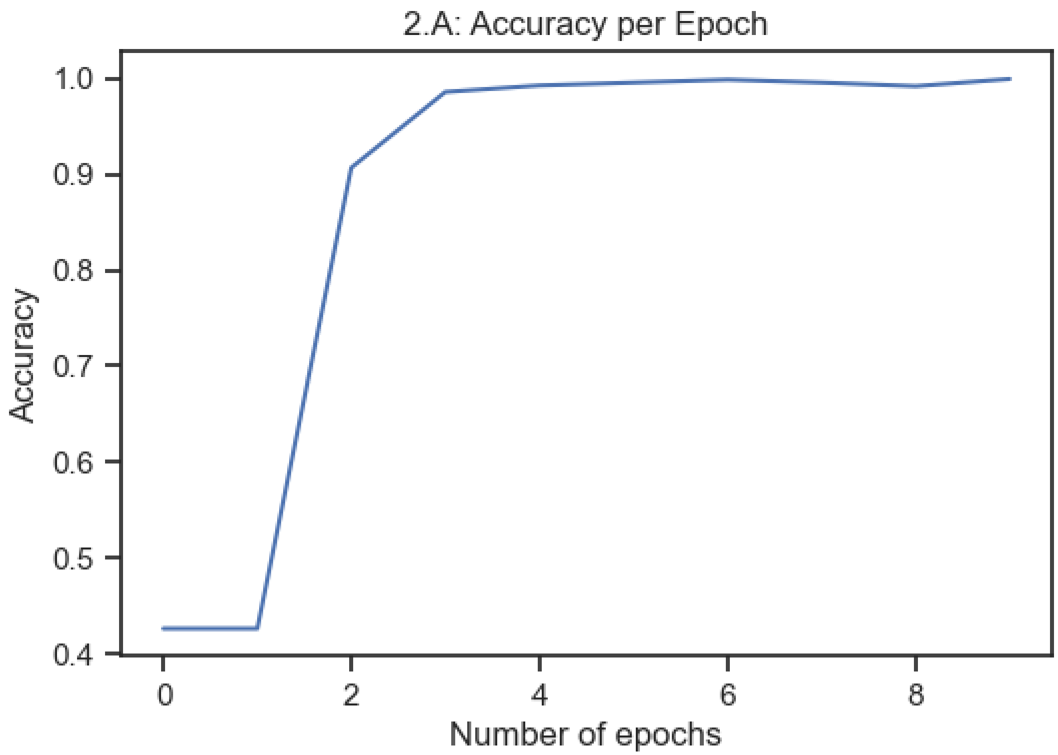
In [79]:

```
fig, ax = plt.subplots(dpi=100)
ax.plot(test_loss)
ax.set_xlabel('Number of epochs')
ax.set_ylabel('Test loss');
plt.title('2.A: Test Loss per Epoch');
plt.show()
```



Plot the evolution of the accuracy as a function of epochs.

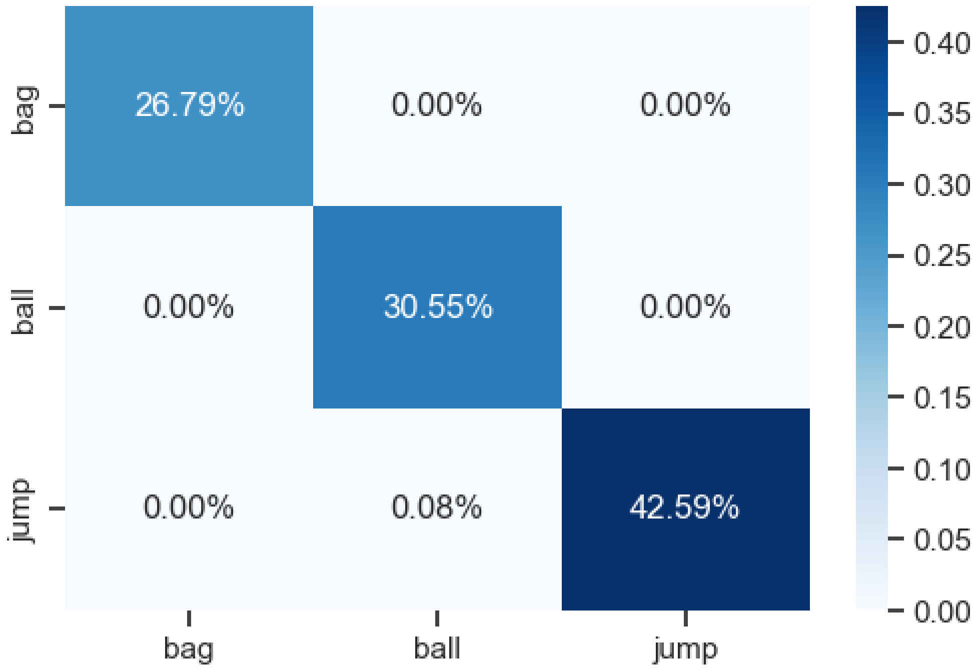
In [78]:
```
fig, ax = plt.subplots(dpi=100)
ax.plot(accuracy)
ax.set_xlabel('Number of epochs')
ax.set_ylabel('Accuracy');
plt.title('2.A: Accuracy per Epoch');
plt.show()
```



Plot the confusion matrix.

In [80]:
```
from sklearn.metrics import confusion_matrix
# Predict on the test data
y_pred_test = trained_model(X_test)
# Remember that the prediction is probabilistic
# We need to simply pick the label with the highest probability:
_, y_pred_labels = torch.max(y_pred_test, 1)
# Here is the confusion matrix:
cf_matrix = confusion_matrix(y_test, y_pred_labels)
```

In [81]:
```
sns.heatmap(cf_matrix/np.sum(cf_matrix), annot=True,
            fmt='.2%', cmap='Blues',
            xticklabels=LABELS_1_TO_TEXT.values(),
            yticklabels=LABELS_1_TO_TEXT.values());
```



Part B - Train a CNN to predict the the low-level type of observation (bag-high, bag-low, etc.)

Repeat what you did above for `y2` .

Answer:

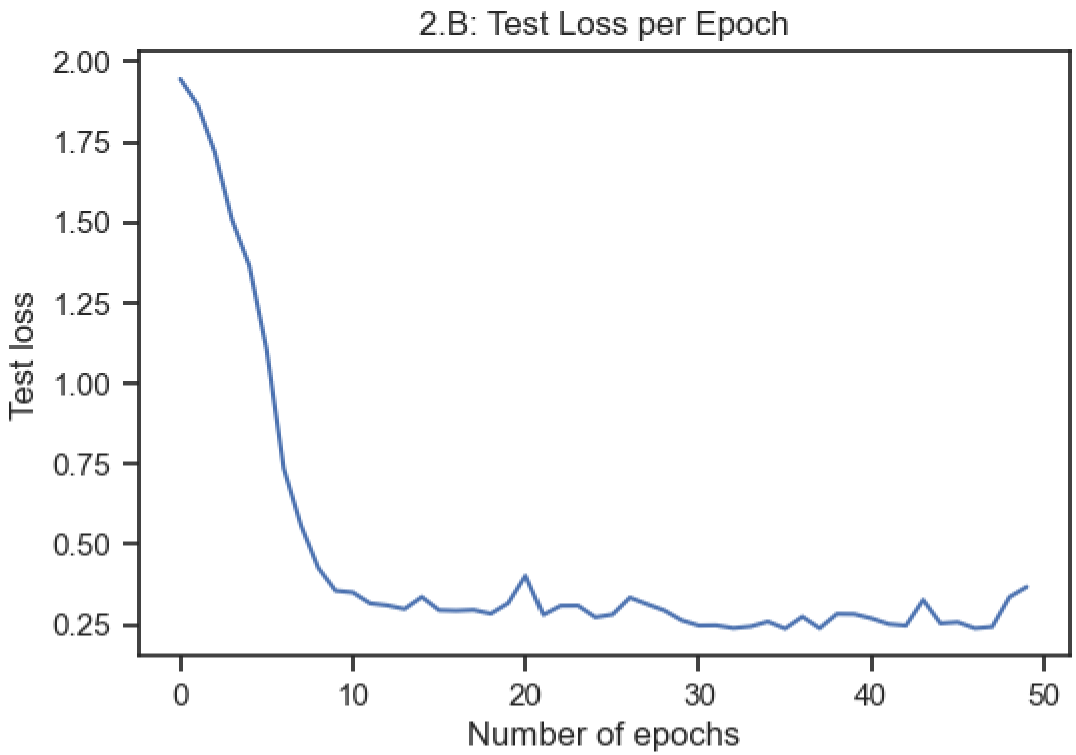
In [84]:

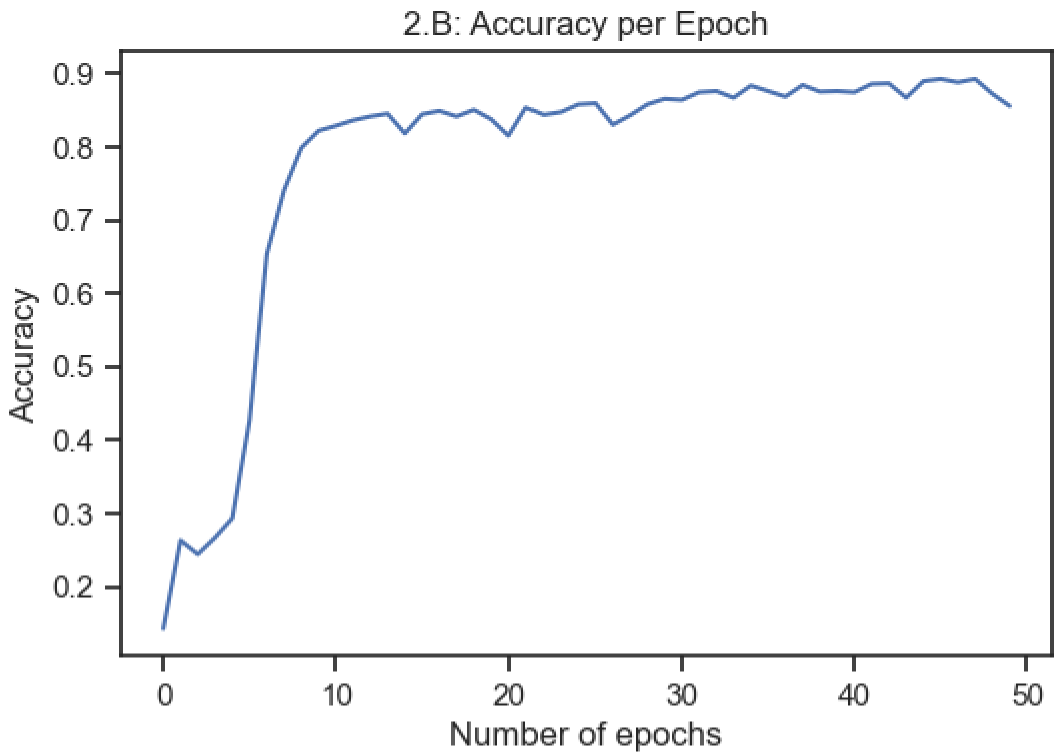
```
##part B

# You can make the network like this:
net = Net(7)

epochs = 50
lr = 0.01
n_batch = 100
trained_model, test_loss, accuracy, X_train, y_train, X_test, y_test = train_cnn(X, y2, net, n_batch, epochs, lr)

fig, ax = plt.subplots(dpi=100)
ax.plot(test_loss)
ax.set_xlabel('Number of epochs')
ax.set_ylabel('Test loss');
plt.title('2.B: Test Loss per Epoch');
```

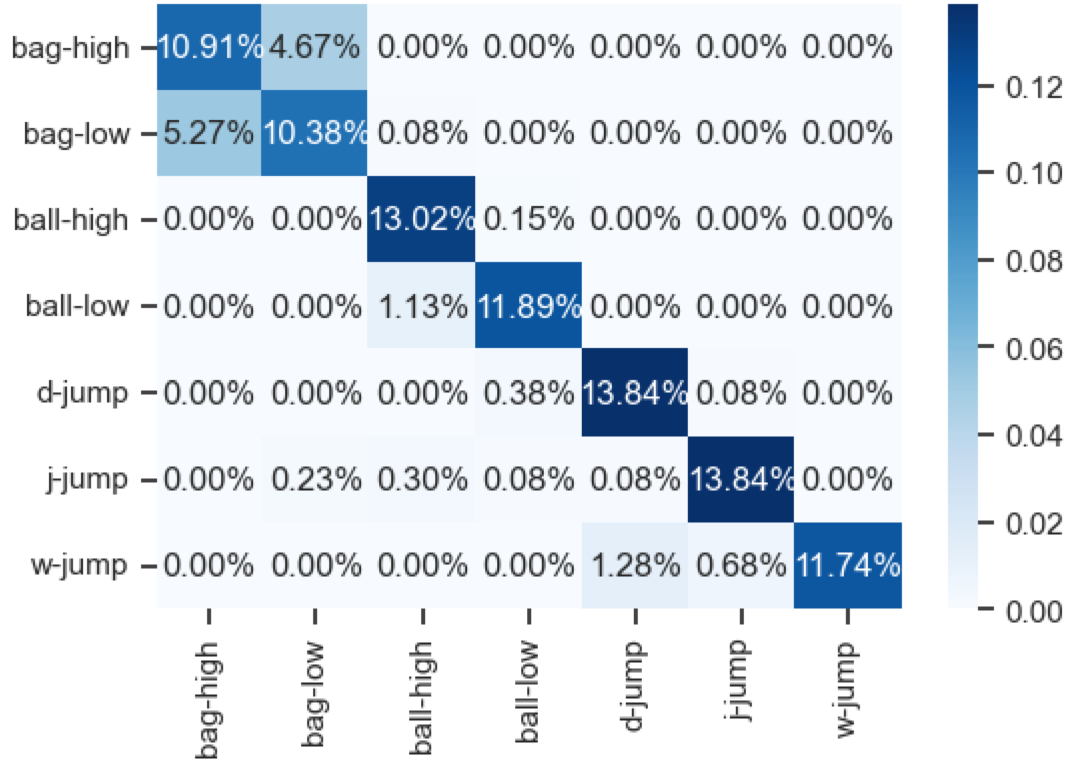
```
plt.show()

fig, ax = plt.subplots(dpi=100)
ax.plot(accuracy)
ax.set_xlabel('Number of epochs')
ax.set_ylabel('Accuracy');
plt.title('2.B: Accuracy per Epoch');
plt.show()

from sklearn.metrics import confusion_matrix
# Predict on the test data
y_pred_test = trained_model(X_test)
# Remember that the prediction is probabilistic
# We need to simply pick the label with the highest probability:
_, y_pred_labels = torch.max(y_pred_test, 1)
# Here is the confusion matrix:
cf_matrix = confusion_matrix(y_test, y_pred_labels)

sns.heatmap(cf_matrix/np.sum(cf_matrix), annot=True,
            fmt='.2%', cmap='Blues',
            xticklabels=LABELS_2_TO_TEXT.values(),
            yticklabels=LABELS_2_TO_TEXT.values());
```

```
Epoch 1: accuracy = 14.29646%
Epoch 2: accuracy = 26.33559%
Epoch 3: accuracy = 24.45448%
Epoch 4: accuracy = 26.71181%
Epoch 5: accuracy = 29.34537%
Epoch 6: accuracy = 42.73890%
Epoch 7: accuracy = 65.38751%
Epoch 8: accuracy = 74.11588%
Epoch 9: accuracy = 79.90971%
Epoch 10: accuracy = 82.24229%
Epoch 11: accuracy = 82.91949%
Epoch 12: accuracy = 83.67193%
Epoch 13: accuracy = 84.19865%
Epoch 14: accuracy = 84.57487%
Epoch 15: accuracy = 81.86606%
Epoch 16: accuracy = 84.49962%
Epoch 17: accuracy = 84.95109%
Epoch 18: accuracy = 84.19865%
Epoch 19: accuracy = 85.10158%
Epoch 20: accuracy = 83.82242%
Epoch 21: accuracy = 81.56509%
Epoch 22: accuracy = 85.40256%
```

```
Epoch 23: accuracy = 84.42438%
Epoch 24: accuracy = 84.80060%
Epoch 25: accuracy = 85.85403%
Epoch 26: accuracy = 86.00451%
Epoch 27: accuracy = 83.06998%
Epoch 28: accuracy = 84.34913%
Epoch 29: accuracy = 85.85403%
Epoch 30: accuracy = 86.60647%
Epoch 31: accuracy = 86.45598%
Epoch 32: accuracy = 87.50941%
Epoch 33: accuracy = 87.65989%
Epoch 34: accuracy = 86.75696%
Epoch 35: accuracy = 88.41234%
Epoch 36: accuracy = 87.65989%
Epoch 37: accuracy = 86.90745%
Epoch 38: accuracy = 88.48758%
Epoch 39: accuracy = 87.58465%
Epoch 40: accuracy = 87.65989%
Epoch 41: accuracy = 87.50941%
Epoch 42: accuracy = 88.63807%
Epoch 43: accuracy = 88.71332%
Epoch 44: accuracy = 86.75696%
Epoch 45: accuracy = 89.01430%
Epoch 46: accuracy = 89.31527%
Epoch 47: accuracy = 88.86381%
Epoch 48: accuracy = 89.31527%
Epoch 49: accuracy = 87.28367%
Epoch 50: accuracy = 85.62829%
```


2.B: Test Loss per Epoch
Test loss
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0
10
20
30
40
50
Number of epochs


2.B: Accuracy per Epoch
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
Accuracy
0
10
20
30
40
50
Number of epochs


bag-high
bag-low
ball-high
ball-low
d-jump
j-jump
w-jump
10.91% 4.67% 0.00% 0.00% 0.00% 0.00% 0.00%
5.27% 10.38% 0.08% 0.00% 0.00% 0.00% 0.00%
0.00% 0.00% 13.02% 0.15% 0.00% 0.00% 0.00%
0.00% 0.00% 1.13% 11.89% 0.00% 0.00% 0.00%
0.00% 0.00% 0.00% 0.38% 13.84% 0.08% 0.00%
0.00% 0.23% 0.30% 0.08% 0.08% 13.84% 0.00%
0.00% 0.00% 0.00% 0.00% 1.28% 0.68% 11.74%
0.12
0.10
0.08
0.06
0.04
0.02
0.00