ME539.Homework.7

August 2, 2022

1 Homework 7

1.1 References

- Lectures 24-26 (inclusive).

1.2 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.

[ ]:
```
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

!sudo apt install texlive texlive-latex-extra texlive-fonts-recommended dvipng
!sudo apt install cm-super

from matplotlib import rc
rc('text', usetex=True)

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 filename to write on. If not
                      specified
    """
    if local_filename is None:
        local_filename = os.path.basename(url)
    urllib.request.urlretrieve(url, local_filename)
```

```
Reading package lists… Done
Building dependency tree
Reading state information… Done
dvipng is already the newest version (1.15-1).
texlive is already the newest version (2017.20180305-1).
texlive-fonts-recommended is already the newest version (2017.20180305-1).
texlive-latex-extra is already the newest version (2017.20180305-2).
The following package was automatically installed and is no longer required:
  libnvidia-common-460
Use 'sudo apt autoremove' to remove it.
0 upgraded, 0 newly installed, 0 to remove and 49 not upgraded.
Reading package lists… Done
Building dependency tree
Reading state information… Done
cm-super is already the newest version (0.3.4-11).
The following package was automatically installed and is no longer required:
  libnvidia-common-460
Use 'sudo apt autoremove' to remove it.
0 upgraded, 0 newly installed, 0 to remove and 49 not upgraded.
```

[ ]:
```
def make_hist(
    df,
    names,
    units,
    idx,
    ax=None
):
    """Generate a histogram from a particular column of a pandas dataframe

    Arguments
    df           -- the pandas dataframe object to index from
    names        -- the column names of the dataframe
    units        -- the units that correspond to the data from each dataframe␣
 ↪column
```

```
    idx            -- the index of which column to use for the histogram
    ax             -- the axes matplotlib.pyplot object to plot on
    """
    if ax is None:
      fig, ax = plt.subplots(dpi=100)
    ax.hist(df[names[idx]], alpha=0.5)
    ax.set_xlabel(names[idx] + ' [' + units[idx] + ']')
    ax.set_ylabel('Statistical Frequency')
    ax.set_title(names[idx] + ' Histogram');
# reference(s): https://en.wikipedia.org/wiki/Frequency_(statistics)
```

[ ]:
```
def make_scatter(
    df,
    names,
    units,
    idx1,
    idx2,
    ax=None
):
    """Generate a scater plot of particular data from a pandas dataframe

    Arguments
    df             -- the pandas dataframe object to index from
    names          -- the column names of the dataframe
    units          -- the units that correspond to the data from each dataframe␣
 ↪column
    idx1           -- the index of which column to use for the x-axis of the␣
 ↪scatter plot
    idx2           -- the index of which column to use for the y-axis of the␣
 ↪scatter plot
    ax             -- the axes matplotlib.pyplot object to plot on
    """
    if ax is None:
      fig, ax = plt.subplots(dpi=100)
    ax.scatter(df[names[idx1]], df[names[idx2]], s=1)
    ax.set_xlabel(names[idx1] + " [" + units[idx1] + "]")
    ax.set_ylabel(names[idx2] + " [" + units[idx2] + "]")
```

1.3 Student details

- **First Name:** Jack
- **Last Name:** Girard
- **Email:** girard2@purdue.edu

Student Notes:

- This code was executed with Google Colab GPUs

- The code block numbering is missing because the PDF was generated in a separate Colab session from when the code was executed
- Results slightly vary from run-to-run (expected)

1.4 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 description 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: 6. 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.

[ ]:
```
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:

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

[ ]:
```
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:

[ ]:
```
import pandas as pd
df = pd.DataFrame(data, columns=['Frequency', 'Angle\_of\_attack',␣
 ↪'Chord\_length',
                                 'Velocity', 'Suction\_thickness',␣
 ↪'Sound\_pressure'])
df
```

```
[ ]:        Frequency  Angle\_of\_attack  Chord\_length  Velocity  \
     0         800.0                0.0        0.3048      71.3
     1        1000.0                0.0        0.3048      71.3
     2        1250.0                0.0        0.3048      71.3
     3        1600.0                0.0        0.3048      71.3
     4        2000.0                0.0        0.3048      71.3
     …          …                …             …         …
     1498     2500.0               15.6        0.1016      39.6
     1499     3150.0               15.6        0.1016      39.6
     1500     4000.0               15.6        0.1016      39.6
     1501     5000.0               15.6        0.1016      39.6
     1502     6300.0               15.6        0.1016      39.6

            Suction\_thickness  Sound\_pressure
     0               0.002663         126.201
     1               0.002663         125.201
     2               0.002663         125.951
     3               0.002663         127.591
     4               0.002663         127.461
     …                  …             …
     1498            0.052849         110.264
     1499            0.052849         109.254
     1500            0.052849         106.604
     1501            0.052849         106.224
     1502            0.052849         104.204

     [1503 rows x 6 columns]
```

1.4.1 Part A - Analyze the data visually

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

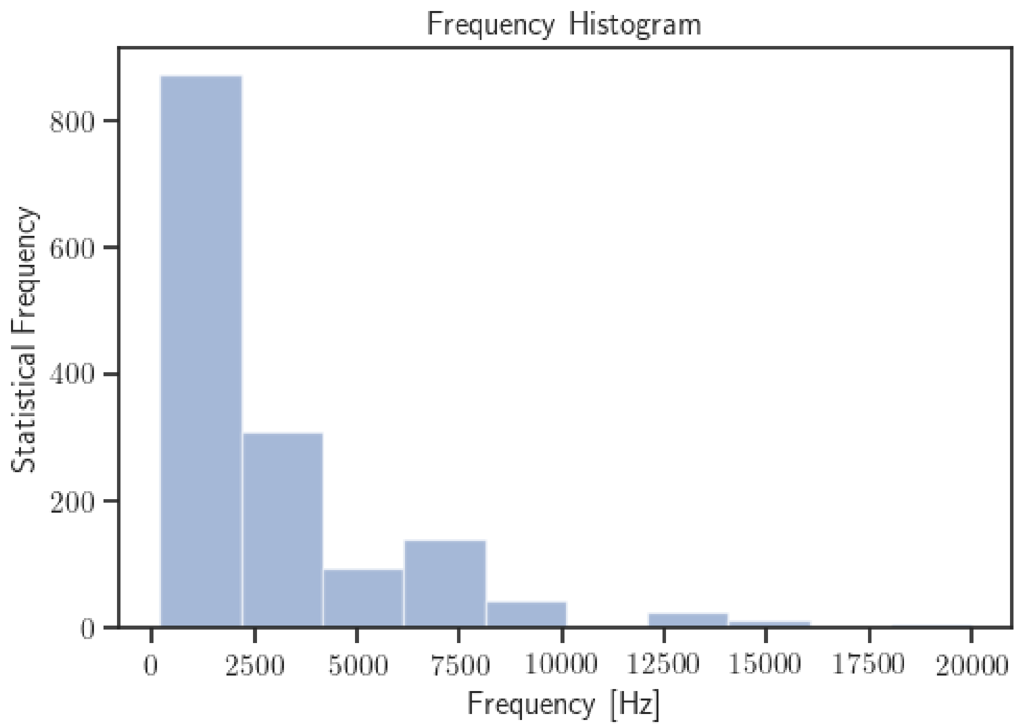
Part A.I - Do the histogtrams of all variables Use as many code segments you need below to plot the histogram of each variable (all inputs and the output in separate plots) Discuss whether or not you need to standarize the data before moving to regression.

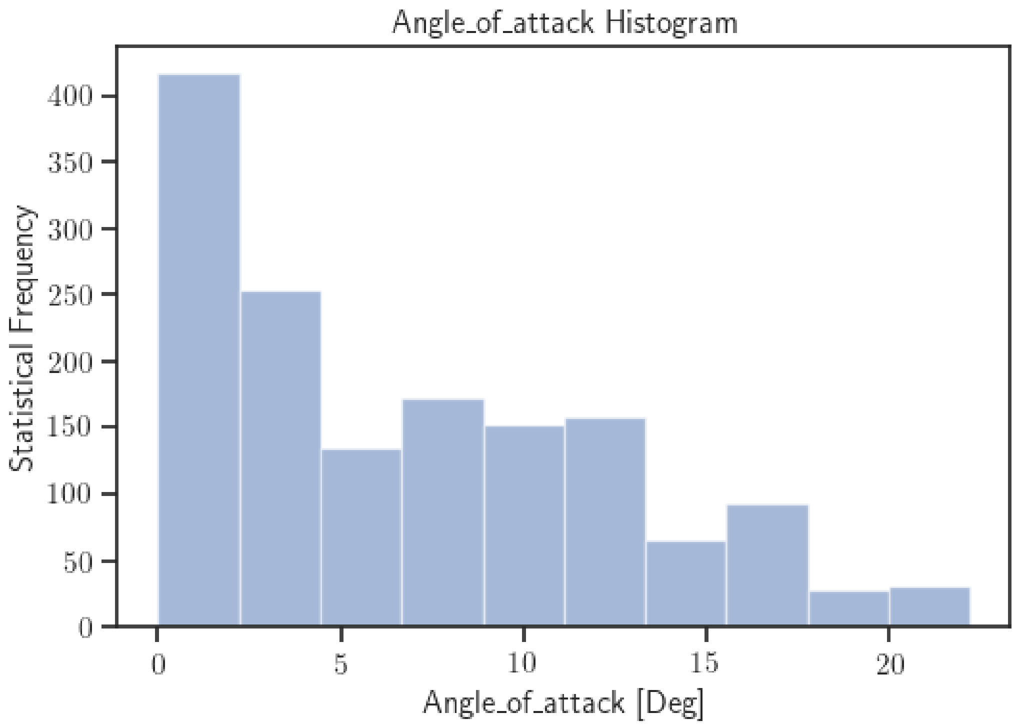
Answer:

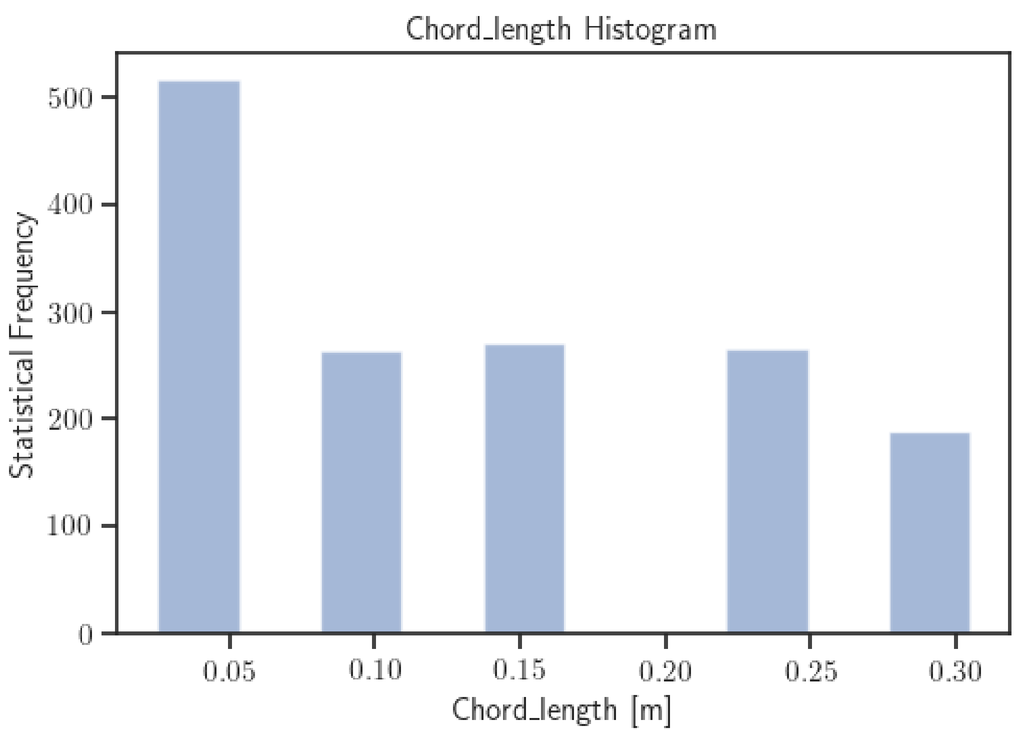
The data does need to be standarized before moving to regression. This is because the set of variables have substantially different scales with respect to one another, which could cause problems during regression.

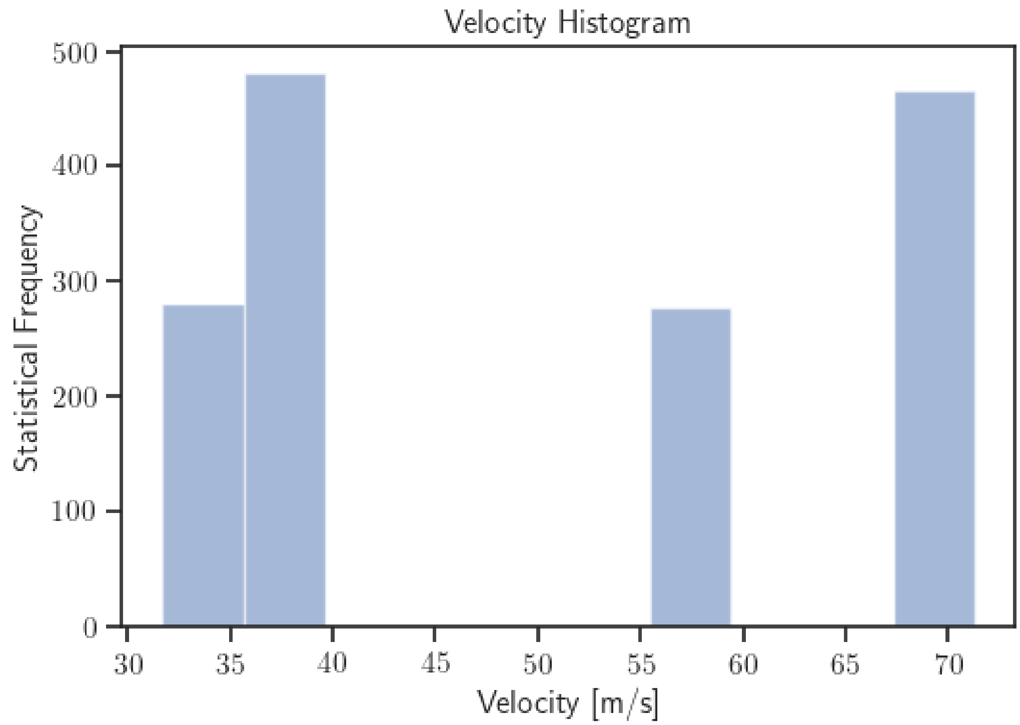
```
[ ]: # define new array to store label names of the data
     column_names = ['Frequency', 'Angle\_of\_attack', 'Chord\_length',
                     'Velocity', 'Suction\_thickness', 'Sound\_pressure']
     # define new array to store units of the data
     column_units = ['Hz', 'Deg', 'm', 'm/s', 'm', 'dB']
```

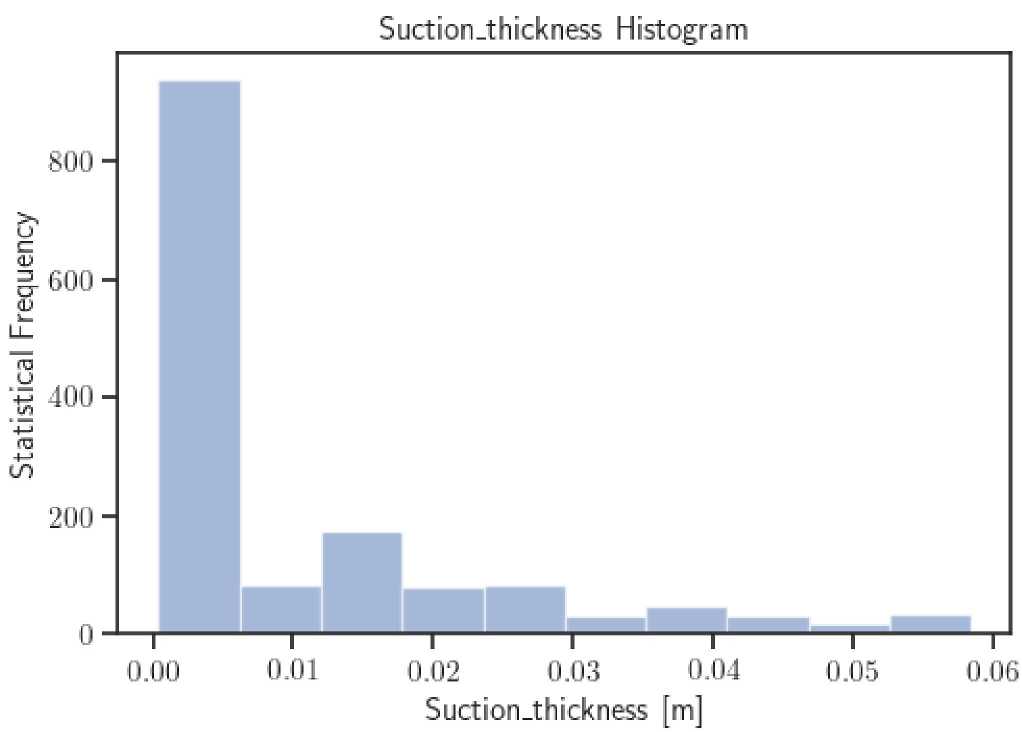
```
# generating individual histogram for each variable using function
for i in range(len(column_names)):
  make_hist(df, column_names, column_units, i)
```

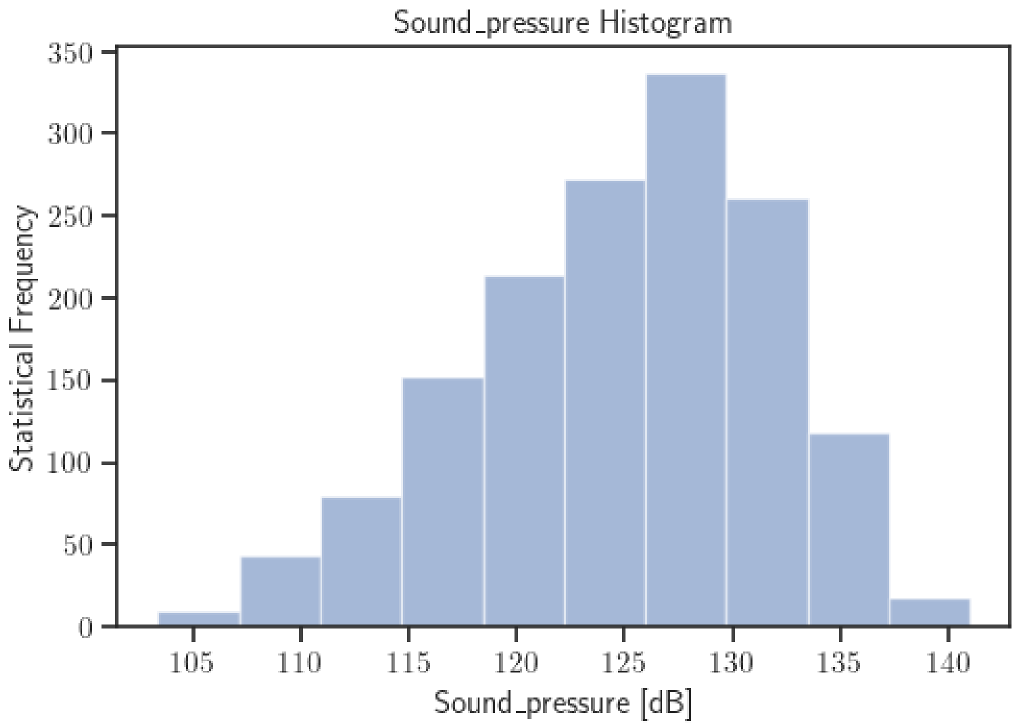



Angle_of_attack Histogram
Statistical Frequency
0
50
100
150
200
250
300
350
400
0
5
10
15
20
Angle_of_attack [Deg]


Chord_length Histogram
Statistical Frequency
0
100
200
300
400
500
0.05
0.10
0.15
0.20
0.25
0.30
Chord_length [m]


Velocity Histogram
Statistical Frequency
Velocity [m/s]
0
100
200
300
400
500
30
35
40
45
50
55
60
65
70


Suction_thickness Histogram
Statistical Frequency
Suction_thickness [m]
0
200
400
600
800
0.00
0.01
0.02
0.03
0.04
0.05
0.06

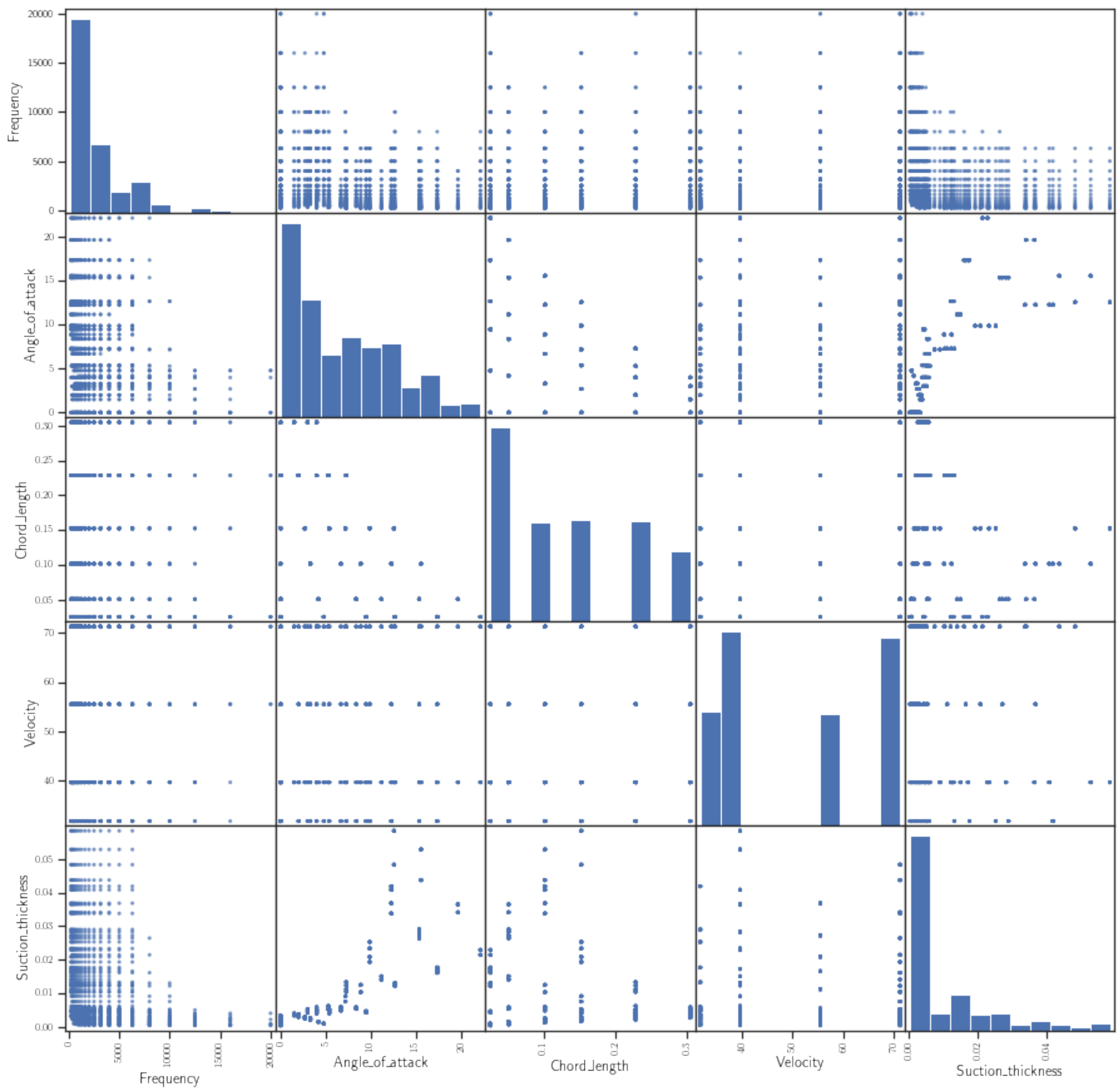


Part A.II - Do the scatter plots between all input variables Do the scatter plot between all input variables. This will give you an idea of the range of experimental conditions. Whatever model you build will only be valid inside the domain implicitly defined with your experimental conditions. Are there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

Answer:

There are multiple holes in the dataset. In fact, each variable appears to have some holes in its data. The most noticeable holes are present in the chord length and velocity variable data.

[ ]:
```
# Using the scatter_matrix() function from pandas for convenience
# remove the output column from the dataframe
input_df = df.drop(columns='Sound\_pressure')
# generate all scatter plot combinations of the input variables
pd.plotting.scatter_matrix(input_df, alpha=0.75, figsize=(15, 15));
# The diagonal of the scatter matrix includes the histograms of the input␣
↪variables as seen above
```



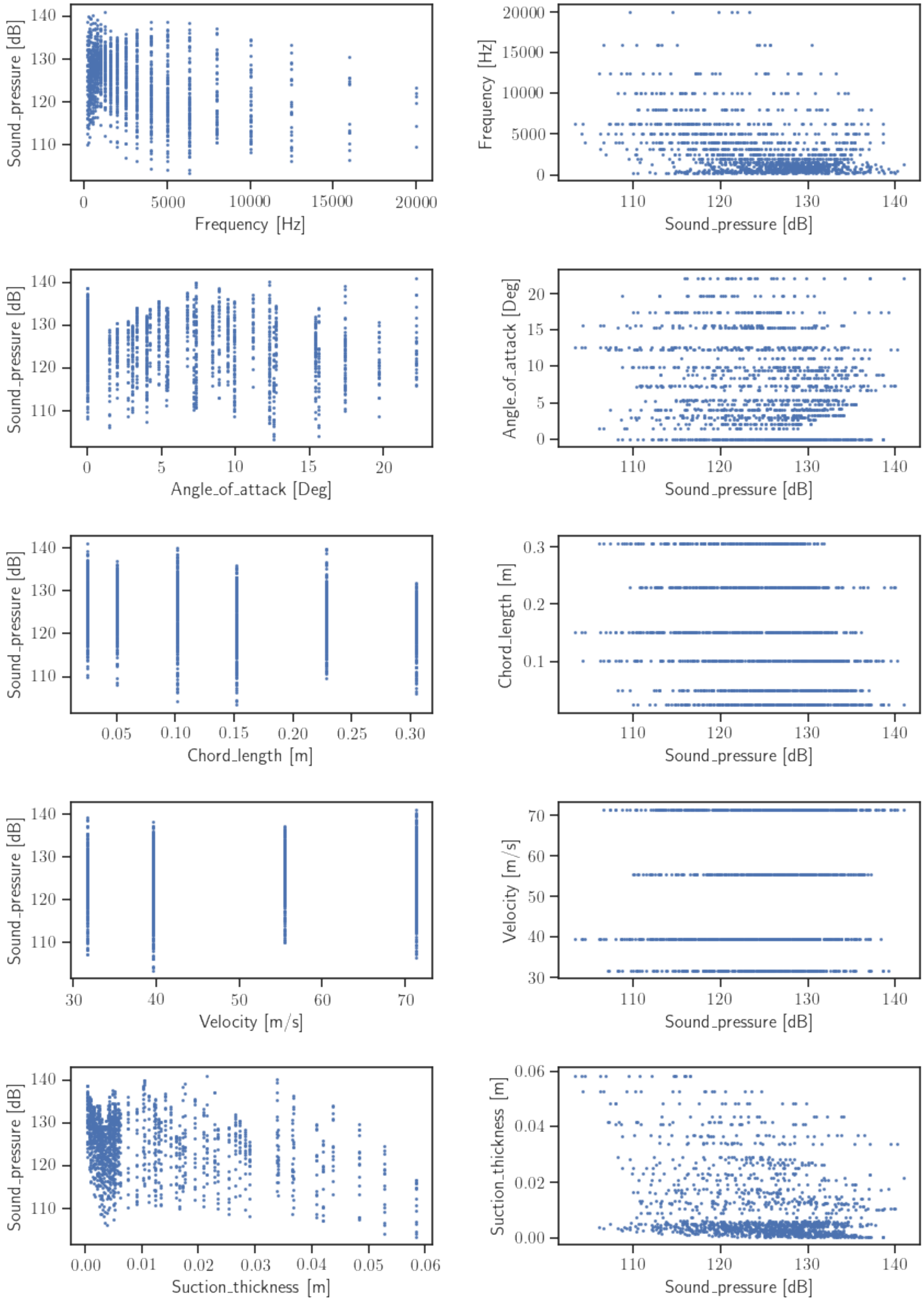
Part A.III - Do the scatter plots between each input and the output Do the scatter plot between each input variable and the output. This will give you an idea of the functional relationship between the two. Do you observe any obvious patterns?

Answer:

It appears that sound pressure data was collected at discrete selections for frequency, angle of attack, chord length, velocity, and suction thickness. If possible, it may be beneficial to collect sound pressure data for a more dense set of input variables to further improve regression results.

[ ]:
```
# generate a matplotlib.pyplot object of the correct dimensions
nrows = 5
ncols = 2
```

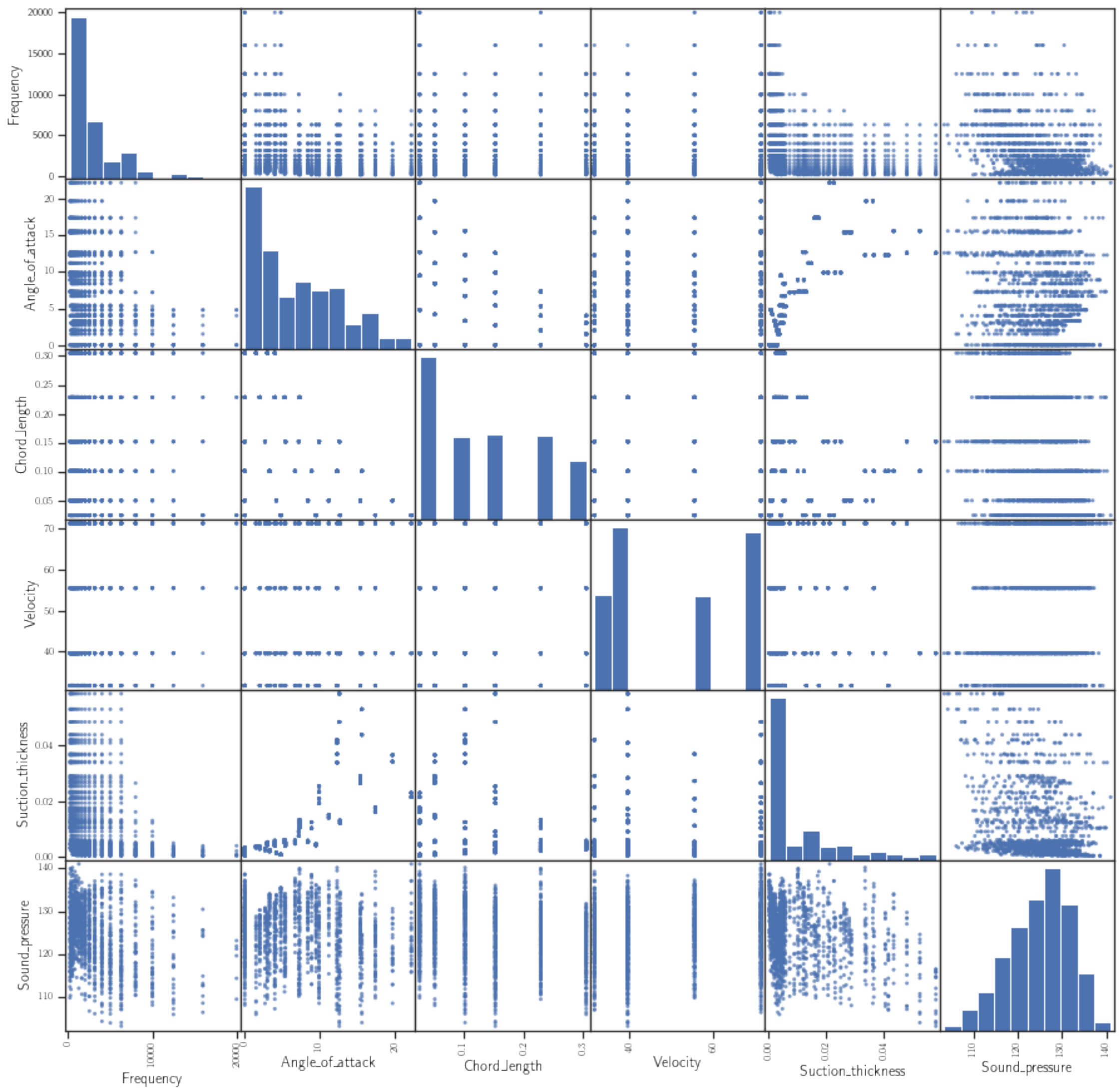
```python
fig, axs = plt.subplots(nrows=nrows, ncols=ncols, squeeze=False,␣
↪figsize=(10,15), dpi=100)
fig.subplots_adjust(wspace=0.35, hspace=0.5)
# generate a pair of scatter plots for each input variable and the output␣
↪variable using function
for i in range(nrows):
  make_scatter(df, column_names, column_units, i, -1, axs[i, 0])
  make_scatter(df, column_names, column_units, -1, i, axs[i, 1])
```



Personal addition:

Plotting all of the data combinations together, which encompasses all of the information for the solutions to Part A.II and A.III. This is also a convenient way to verify that the above plots are correct.

[ ]:
```
pd.plotting.scatter_matrix(df, alpha=0.75, figsize=(15, 15));
```



1.4.2 Part B - Use DNN to do regression

Let start by separating inputs and outputs for you:

[ ]:
```
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:

[ ]:
```
import torch
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()
```

[ ]:
```
# 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: 2.33
What you should be getting: 2.33
```

[ ]:
```
# 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
```

[ ]:
```
# 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.
    """
    J = mse_loss(y_obs, y_pred) + reg_weight * l2_reg_loss(params)
    return J
# reference(s): lecture 25 reading
```

[ ]:
```
# 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: 2.33
This should be the same as this: 2.33
The loss with regularization: 2.41
```

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:

```
[ ]: # 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
```

[ ]:
```
# 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`
    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
            # zero out the gradient buffers
            optimizer.zero_grad()
            # make predictions
            y_pred = net(X_batch)
            # evaluate the loss
            loss = loss_func(y_batch, y_pred, reg_weight, net.parameters())
```

```
            # evaluate the derivative of the loss w.r.t. all parameters
            loss.backward()
            # make step
            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
# reference(s): hands-on activity 24
```

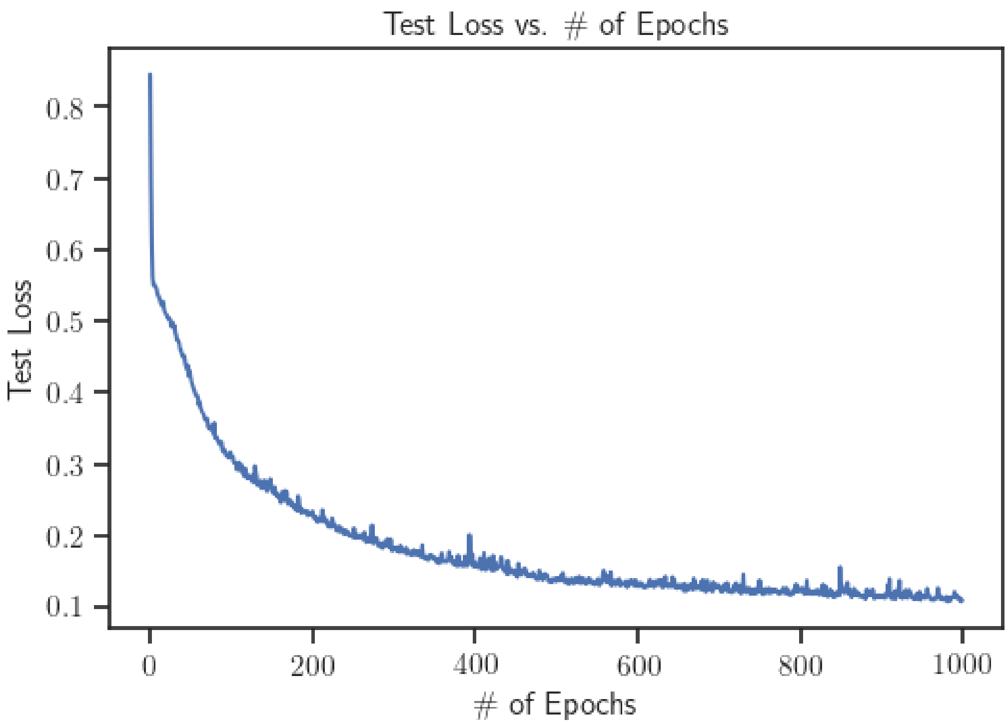
Use this to test your code:

[ ]:
```
# A simple one-layer network with 20 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 [00:14<00:00, 67.93it/s]
```

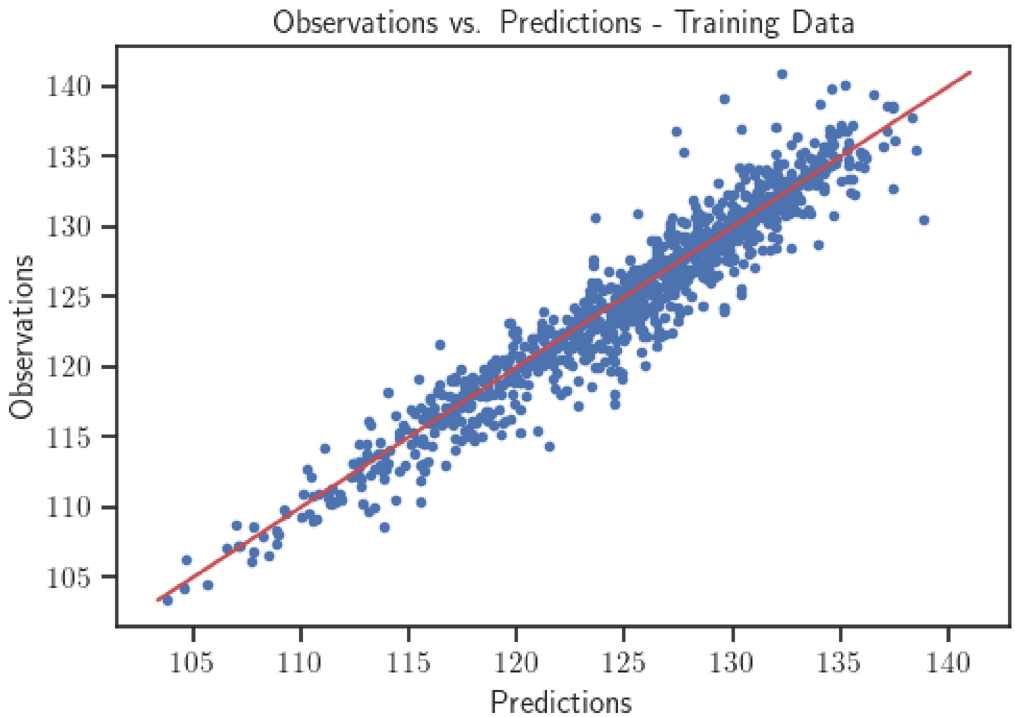
There are a few more things for you to do here. First, plot the evolution of the test loss as a function of the number of epochs:

[ ]:
```
fig, ax = plt.subplots(dpi=100)
# generate array of epoch counts (necessary for plotting purposes)
epoch_arr = [i for i in range(epochs)]
# plot results
ax.plot(epoch_arr, test_loss)
ax.set_xlabel('\# of Epochs')
ax.set_ylabel('Test Loss')
ax.set_title('Test Loss vs. \# of Epochs');
```



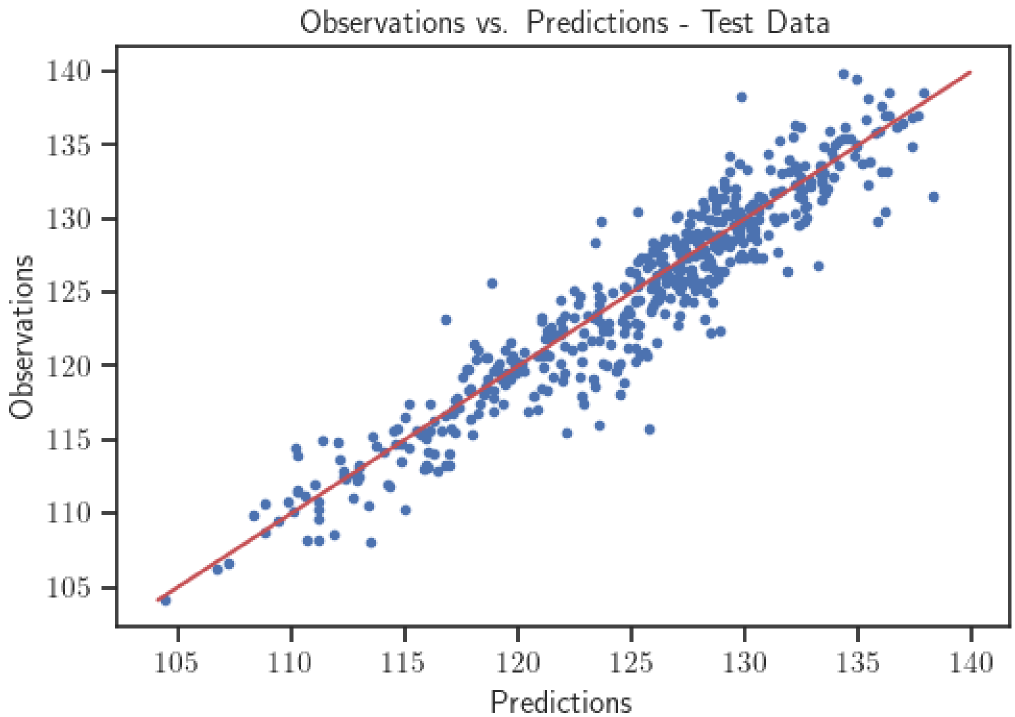
Now plot the observations vs predictions plot for the training data:

[ ]:
```
# Observations vs. Predictions Plot for Training Data
fig, ax = plt.subplots(dpi=100)
ax.plot(model(X_train), y_train, '.')
yys = np.linspace(y_train.min(), y_train.max(), 10)
ax.plot(yys, yys, 'r')
ax.set_xlabel('Predictions')
ax.set_ylabel('Observations')
ax.set_title('Observations vs. Predictions - Training Data');
```



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

[ ]:
```
# Observations vs. Predictions Plot for Test Data
fig, ax = plt.subplots(dpi=100)
ax.plot(model(X_test), y_test, '.')
yys = np.linspace(y_test.min(), y_test.max(), 10)
ax.plot(yys, yys, 'r')
ax.set_xlabel('Predictions')
ax.set_ylabel('Observations')
ax.set_title('Observations vs. Predictions - Test Data');
```



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?

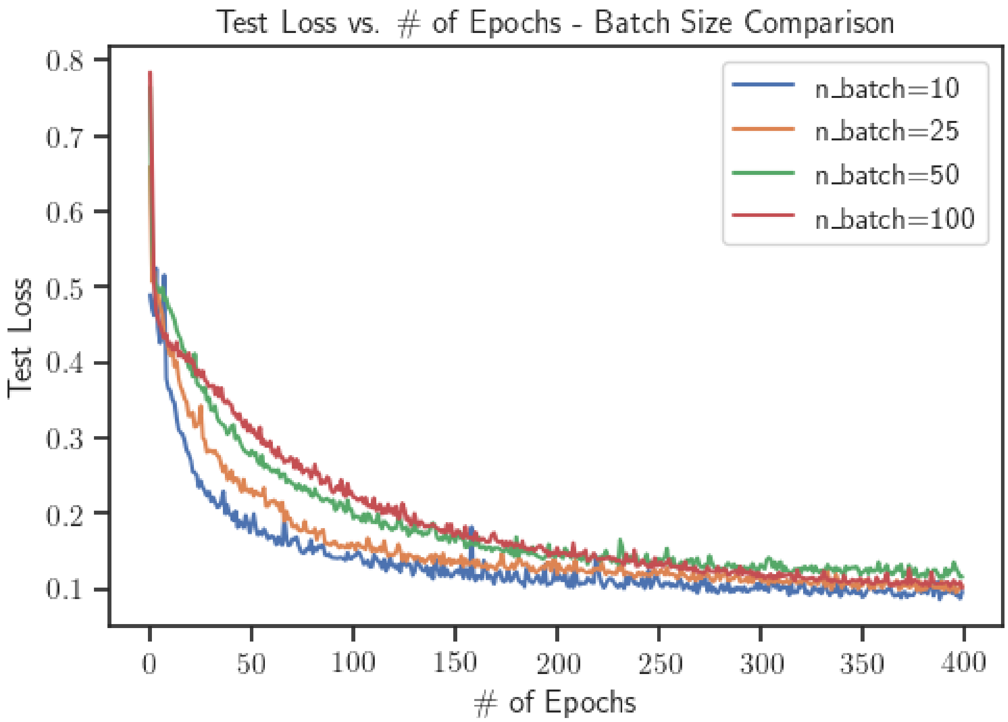
Answer:

[ ]:
```
# set parameters, with batch size as a vector
epochs = 400
lr = 0.01
reg_weight = 1e-12
test_losses = []
models = []
batches = [10, 25, 50, 100]
# loop through the different batch sizes and train the network using each
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)
```

```
Training n_batch: 10

100%|          | 400/400 [00:38<00:00, 10.39it/s]

Training n_batch: 25

100%|          | 400/400 [00:18<00:00, 21.98it/s]

Training n_batch: 50

100%|          | 400/400 [00:09<00:00, 42.03it/s]

Training n_batch: 100

100%|          | 400/400 [00:05<00:00, 68.54it/s]
```

[ ]:
```
# plotting results from adjusting the batch size
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('\# of Epochs')
ax.set_ylabel('Test Loss')
ax.set_title('Test Loss vs. \# of Epochs - Batch Size Comparison')
plt.legend(loc='best');
```



Part C.I Discussion:

Among the batch sizes that were considered, the batch size that leads to the fastest training time is also the largest, at 100. In fact, based on the above progress bars it is evident that batch size and computation time are inversely related. This is because larger batch sizes mean that more training data is passed in to the model at one time and used to update internal model parameters. As a result a single epoch, which is completed when all of the training data is passed through the network, is completed more rapidly. However, using larger batch sizes leads to smaller changes in test loss between individual epochs because less updates to the model parameters are being made during each epoch. Intuitively, the second fastest training sequence used a batch size of 50, and so on.

The optimal selection for batch size is subjective to a degree, but it should depend jointly on test loss convergence and computation time using a set number of epochs. Using a batch size of 10 leads to the most rapid test loss convergence. **It is arguable that a batch size of 50 is optimal**, because it converges to a similar test loss value and results in significantly shorter computation time compared to a batch size of 10 given 400 epochs. Of course, other arguments could be made about how other batch sizes are optimal based on more detailed test loss convergence and computational time requirements.

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?

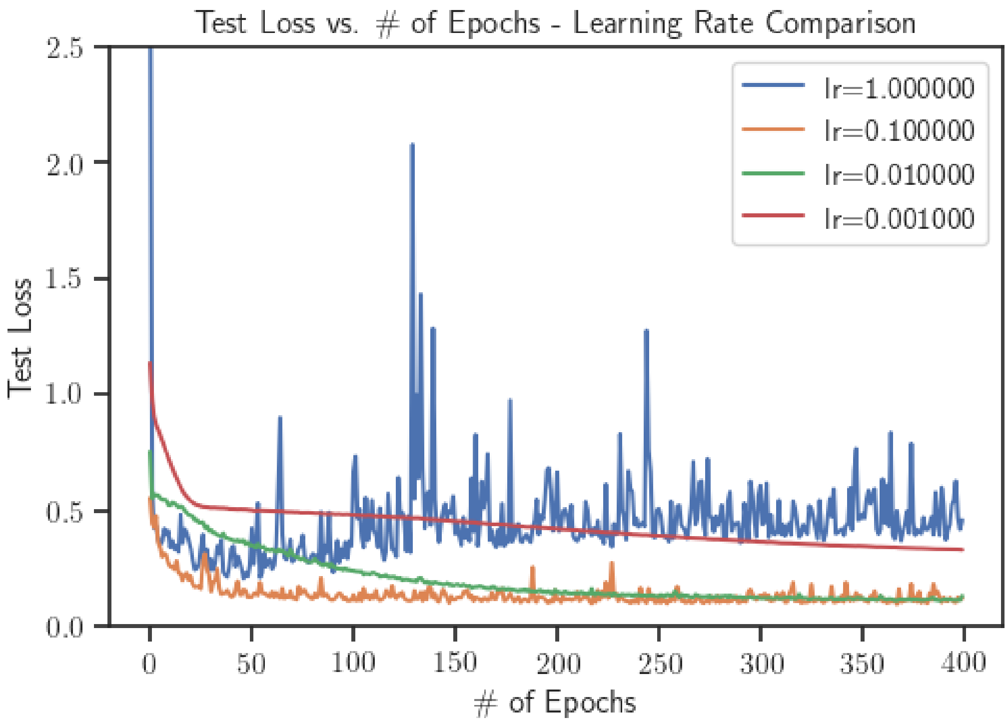
Answer:

[ ]:
```
# set parameters, with learning rate as a vector
epochs = 400
lrs = [1, 0.1, 0.01, 0.001]
reg_weight = 1e-12
test_losses = []
models = []
batch = 50
# loop through the different learning rates and train the network using each
for n_learn in lrs:
    print('Training n_learn: {0:f}'.format(n_learn))
    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,
        batch,
        epochs,
        n_learn
    )
    test_losses.append(test_loss)
    models.append(model)
```

```
Training n_learn: 1.000000

100%|      | 400/400 [00:09<00:00, 41.26it/s]

Training n_learn: 0.100000

100%|      | 400/400 [00:09<00:00, 41.71it/s]

Training n_learn: 0.010000

100%|      | 400/400 [00:09<00:00, 41.61it/s]

Training n_learn: 0.001000

100%|      | 400/400 [00:10<00:00, 38.08it/s]
```

[ ]:
```
# plotting results from adjusting the learning rate
fig, ax = plt.subplots(dpi=100)
for tl, n_learn in zip(test_losses, lrs):
    ax.plot(tl, label='lr={0:f}'.format(n_learn))
ax.set_xlabel('\# of Epochs')
ax.set_ylabel('Test Loss')
ax.set_title('Test Loss vs. \# of Epochs - Learning Rate Comparison')
ax.set_ylim(bottom=0, top=2.5)
```

```
plt.legend(loc='best');
```



Part C.II Discussion:

Based on the results above, the algorithm does not appear to converge for a learning rate of 1.0. A slight decreasing trend is noticeable in the test loss for a learning rate of 0.001 as the number of epochs increases, but a much higher number of epochs will be required for any sort of test loss convergence similar to the ones demonstrated using learning rates of 0.1 and 0.01. Although a learning rate of 0.1 demonstrates a slightly more rapid test loss convergence compared to a learning rate of 0.01, their final corresponding test loss values at 400 epochs are quite similar as seen above. Between the two, **a learning rate of 0.01 is arguably the optimal choice** because it's corresponding test loss vs. number of epochs curve has less high frequency noise.

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?

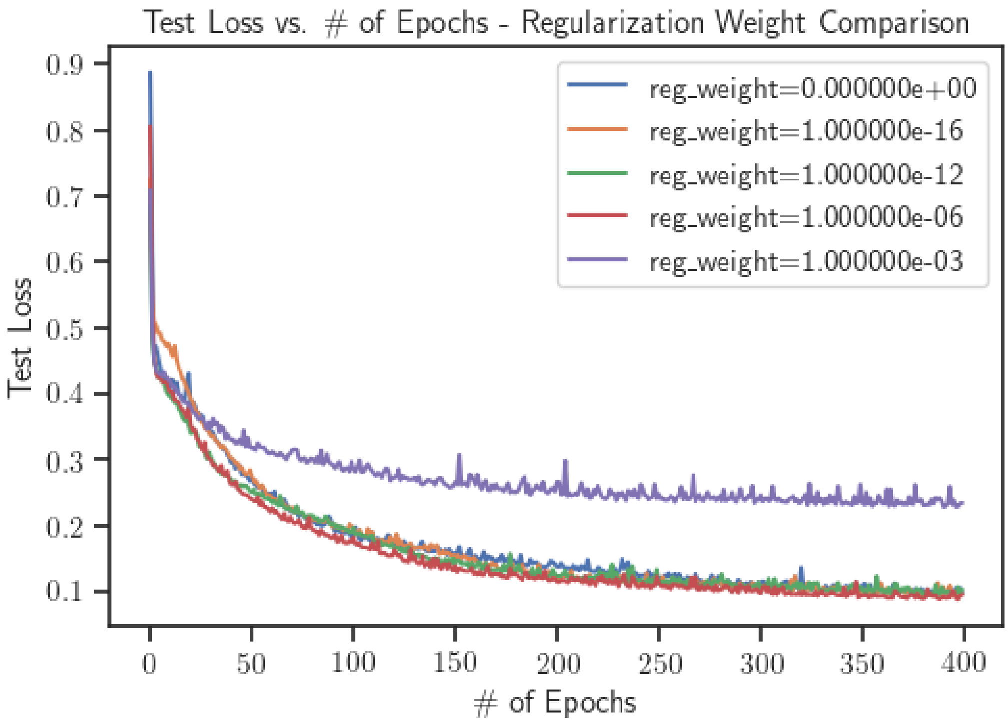
Answer:

[ ]:
```
# set parameters, with regularization weight as a vector
epochs = 400
lr = 0.01
```

```
reg_weights = [0, 1e-16, 1e-12, 1e-6, 1e-3]
test_losses = []
models = []
batch = 50
# loop through the different regularization weights and train the network using␣
↪each
for n_reg in reg_weights:
    print('Training n_reg: {0:e}'.format(n_reg))
    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,
        n_reg,
        batch,
        epochs,
        lr
    )
    test_losses.append(test_loss)
    models.append(model)
```

```
Training n_reg: 0.000000e+00

100%|      | 400/400 [00:09<00:00, 40.71it/s]

Training n_reg: 1.000000e-16

100%|      | 400/400 [00:09<00:00, 41.48it/s]

Training n_reg: 1.000000e-12

100%|      | 400/400 [00:09<00:00, 41.35it/s]

Training n_reg: 1.000000e-06

100%|      | 400/400 [00:09<00:00, 41.53it/s]

Training n_reg: 1.000000e-03

100%|      | 400/400 [00:09<00:00, 41.28it/s]
```

[ ]:
```
# plotting results from adjusting the regularization weight
fig, ax = plt.subplots(dpi=100)
for tl, n_reg in zip(test_losses, reg_weights):
    ax.plot(tl, label='reg\_weight={0:e}'.format(n_reg))
ax.set_xlabel('\# of Epochs')
ax.set_ylabel('Test Loss')
ax.set_title('Test Loss vs. \# of Epochs - Regularization Weight Comparison')
plt.legend(loc='best');
```



Part C.III Discussion:

Based on the results obove, the regularization weights that appear to be the best are the ones closer to zero. Take the following equation:

$$J(\theta) = L(\theta) + \lambda R(\theta),$$

where $J(\theta)$ is to be minimized, $L(\theta)$ is the MSE loss, λ is the regularization weight, and $R(\theta)$ is the sum of the L2 norms of the parameters. When λ is larger, more penalization is put on large paremetric weights within the model and the resulting minimization of $J(\theta)$ is larger. As a result, the lower values for λ result in convergence to a lower test loss value. However, if for any reason large paremetric model weights are undesirable, using a larger value for λ may be preferable even though test loss may converge to a slightly larger value.

In this situation, since the significance of large parametric model weights is ambiguous, **it is arguable that 1e-12 is the optimal choice for the regularization weight** since it adds slight penalization for large parametric model weights and also yields acceptable test loss convergence.

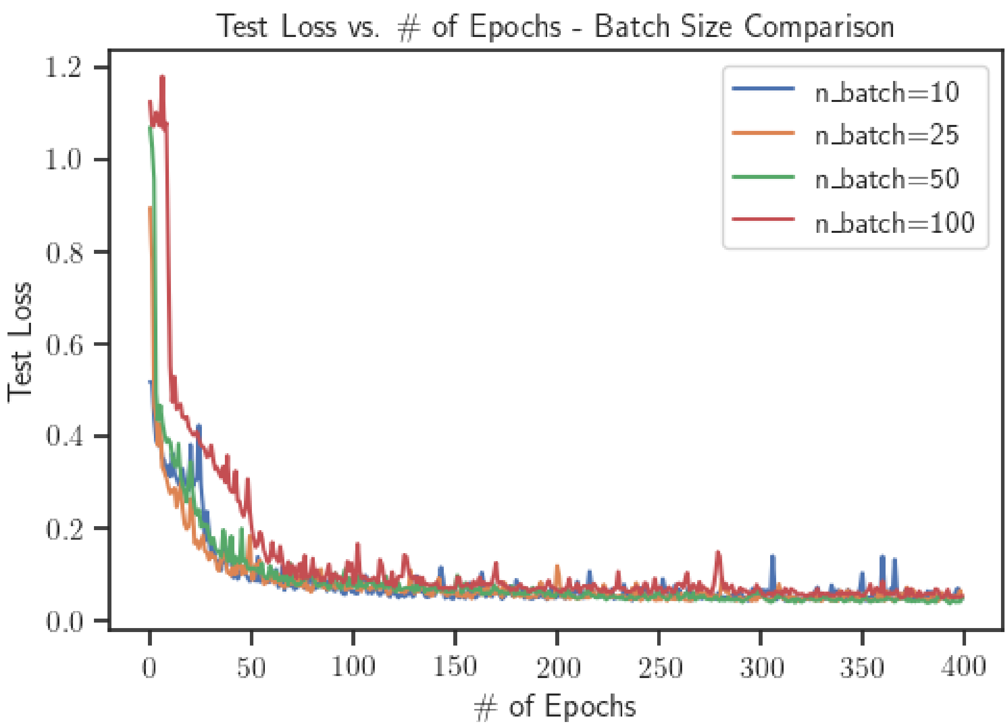
Part D.I - Train a bigger network Now that you have developed some intuition about the parameters involved in training a network, train a larger one. In particular, use a 5-layer deep network with 100 neurons per layer. You can use the sigmoid activation function or you can change it to something else.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

For the bigger network, the first step is to re-perform the tests above to determine the optimal selections for the hyperparameters. A similar methodology is adopted from previously to select the optimal hyperparameters based on a combination of best test loss convergence, test loss convergence rate, computation time, and mild penalization of large parametric model weights.

[ ]:
```
# re-performing previous operation to determine the best hyperparameter␣
↪selections
# batch size *****************************************************************
# set parameters, with batch size as a vector
epochs = 400
lr = 0.01
reg_weight = 1e-6
test_losses = []
models = []
batches = [10, 25, 50, 100]
# loop through the different batch sizes and train the network using each
for n_batch in batches:
    print('Training n_batch: {0:d}'.format(n_batch))
    net = 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))
    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)
# plotting results from adjusting the batch size
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('\# of Epochs')
ax.set_ylabel('Test Loss')
ax.set_title('Test Loss vs. \# of Epochs - Batch Size Comparison')
plt.legend(loc='best');
```

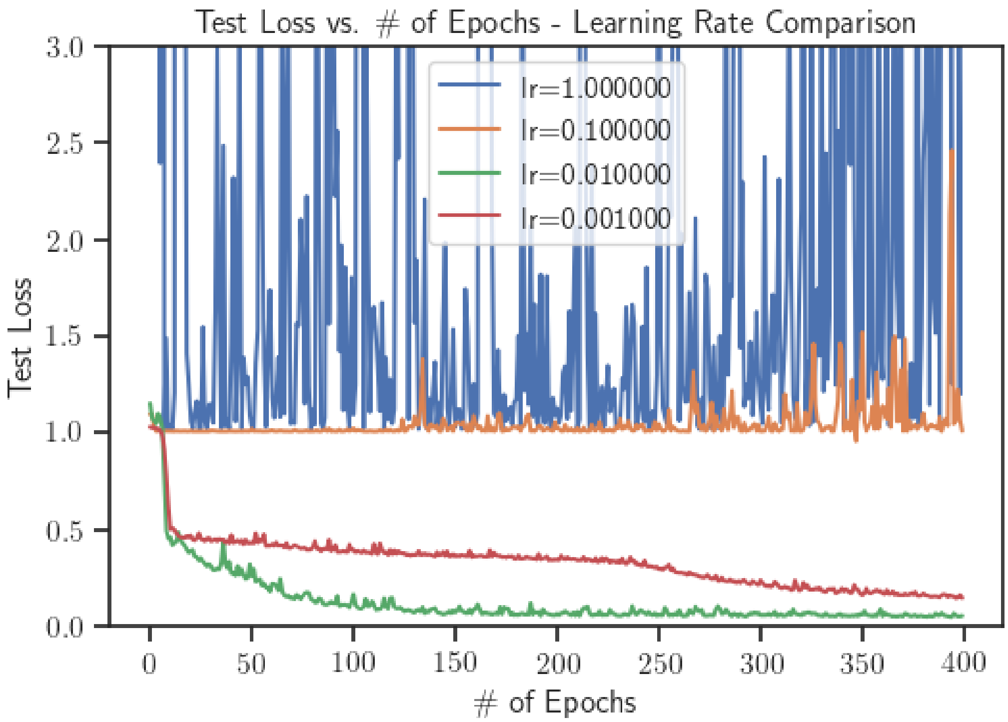
```
Training n_batch: 10
100%|          | 400/400 [01:15<00:00,  5.29it/s]
Training n_batch: 25
100%|          | 400/400 [00:33<00:00, 11.93it/s]
Training n_batch: 50
100%|          | 400/400 [00:18<00:00, 21.33it/s]
Training n_batch: 100
100%|          | 400/400 [00:11<00:00, 34.62it/s]
```



[ ]:
```python
# learning rate ****************************************************************
# set parameters, with learning rate as a vector
epochs = 400
lrs = [1, 0.1, 0.01, 0.001]
reg_weight = 1e-6
test_losses = []
models = []
batch = 100
# loop through the different learning rates and train the network using each
for n_learn in lrs:
    print('Training n_learn: {0:f}'.format(n_learn))
```

```python
    net = 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))
    model, test_loss, X_train, y_train, X_test, y_test = train_net(
        X,
        y,
        net,
        reg_weight,
        batch,
        epochs,
        n_learn
    )
    test_losses.append(test_loss)
    models.append(model)
# plotting results from adjusting the learning rate
fig, ax = plt.subplots(dpi=100)
for tl, n_learn in zip(test_losses, lrs):
    ax.plot(tl, label='lr={0:f}'.format(n_learn))
ax.set_xlabel('\# of Epochs')
ax.set_ylabel('Test Loss')
ax.set_title('Test Loss vs. \# of Epochs - Learning Rate Comparison')
ax.set_ylim(bottom=0, top=3)
plt.legend(loc='best');
```

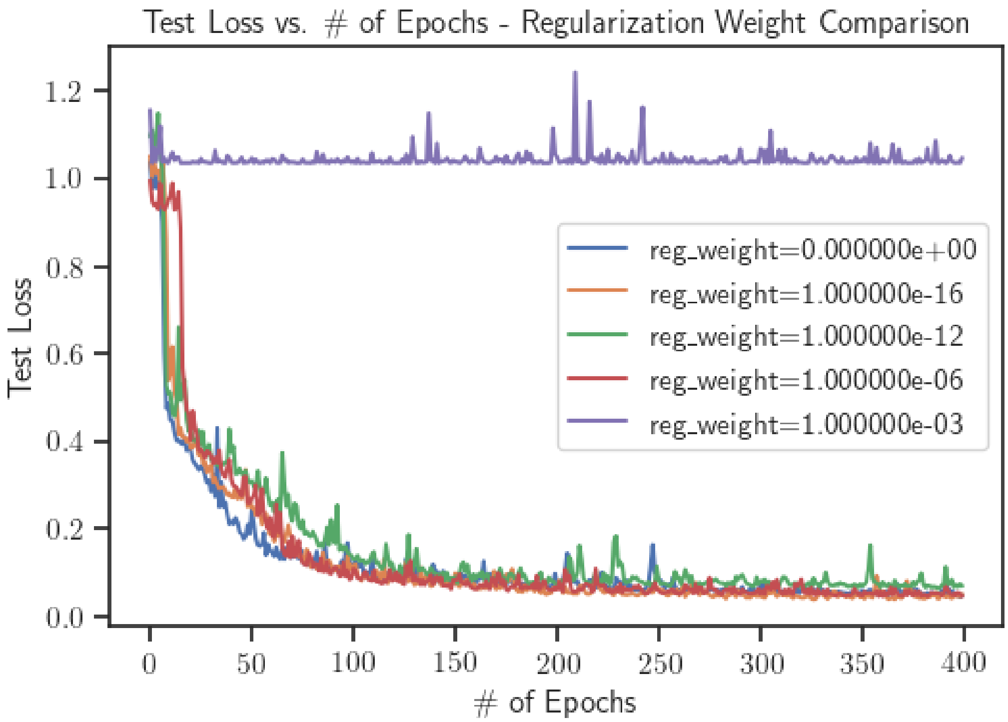
```
Training n_learn: 1.000000

100%|      | 400/400 [00:29<00:00, 13.50it/s]

Training n_learn: 0.100000

100%|      | 400/400 [00:16<00:00, 24.64it/s]

Training n_learn: 0.010000

100%|      | 400/400 [00:11<00:00, 34.48it/s]

Training n_learn: 0.001000

100%|      | 400/400 [00:11<00:00, 35.26it/s]
```



[ ]:
```
# regularization weight ********************************************************
# set parameters, with regularization weight as a vector
epochs = 400
lr = 0.01
reg_weights = [0, 1e-16, 1e-12, 1e-6, 1e-3]
test_losses = []
models = []
batch = 100
# loop through the different regularization weights and train the network using␣
 ↪each
for n_reg in reg_weights:
    print('Training n_reg: {0:e}'.format(n_reg))
    net = 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))
    model, test_loss, X_train, y_train, X_test, y_test = train_net(
        X,
```

```python
        y,
        net,
        n_reg,
        batch,
        epochs,
        lr
    )
    test_losses.append(test_loss)
    models.append(model)
# plotting results from adjusting the regularization weight
fig, ax = plt.subplots(dpi=100)
for tl, n_reg in zip(test_losses, reg_weights):
    ax.plot(tl, label='reg\_weight={0:e}'.format(n_reg))
ax.set_xlabel('\# of Epochs')
ax.set_ylabel('Test Loss')
ax.set_title('Test Loss vs. \# of Epochs - Regularization Weight Comparison')
plt.legend(loc='best');
```

```
Training n_reg: 0.000000e+00

100%|      | 400/400 [00:16<00:00, 24.91it/s]

Training n_reg: 1.000000e-16

100%|      | 400/400 [00:11<00:00, 34.93it/s]

Training n_reg: 1.000000e-12

100%|      | 400/400 [00:11<00:00, 35.06it/s]

Training n_reg: 1.000000e-06

100%|      | 400/400 [00:11<00:00, 35.21it/s]

Training n_reg: 1.000000e-03

100%|      | 400/400 [00:11<00:00, 35.19it/s]
```



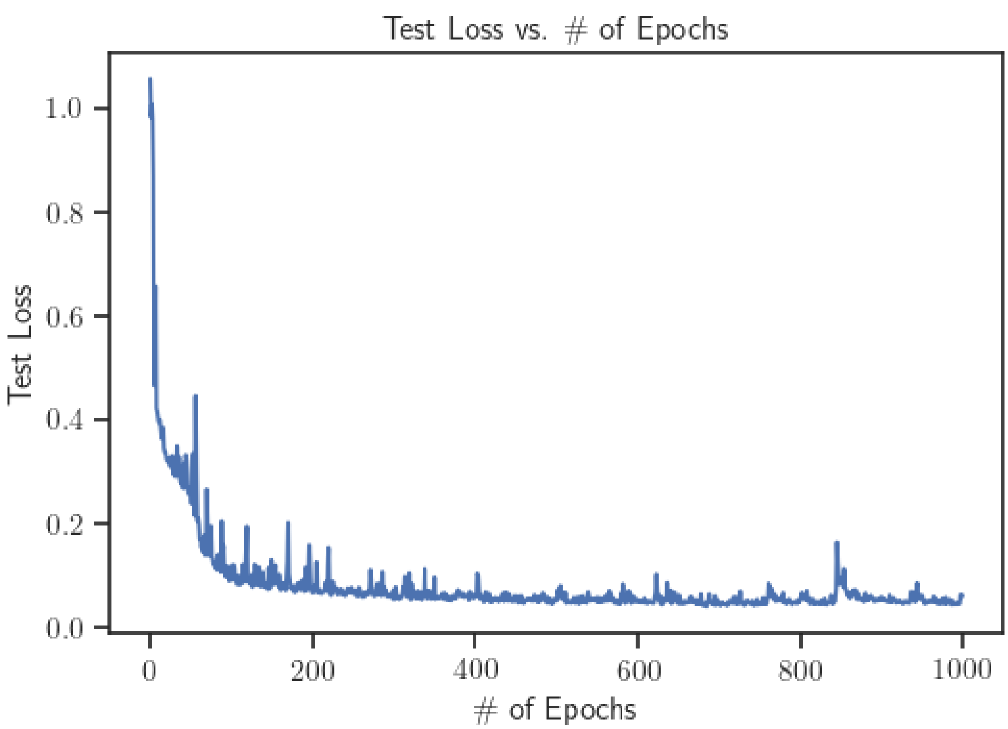
Now that the hyperparameters have been methodically selected, train a model using them:

```
[ ]: # use an even larger number of epochs
epochs = 1000
# optimal learning rate
lr = 0.01
# optimal regularization weight
reg_weight = 1e-6
# optimal batch size
batch = 100
# re-define the bigger network
net = 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))
# train model
model, test_loss, X_train, y_train, X_test, y_test = train_net(
    X,
    y,
```

```
    net,
    reg_weight,
    batch,
    epochs,
    lr
)
```

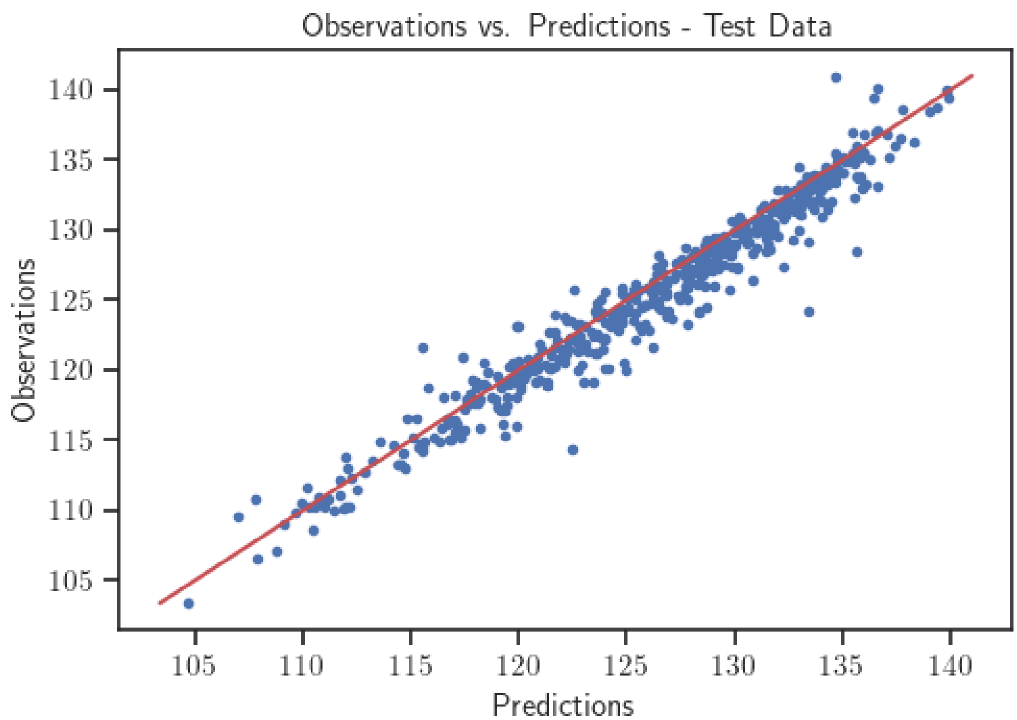
```
100%|          | 1000/1000 [00:28<00:00, 34.77it/s]
```

[ ]:
```
fig, ax = plt.subplots(dpi=100)
# generate array of epoch counts (necessary for plotting purposes)
epoch_arr = [i for i in range(epochs)]
# plot results
ax.plot(epoch_arr, test_loss)
ax.set_xlabel('\# of Epochs')
ax.set_ylabel('Test Loss')
ax.set_title('Test Loss vs. \# of Epochs');
```



[ ]:
```
# Observations vs. Predictions Plot for Test Data
fig, ax = plt.subplots(dpi=100)
ax.plot(model(X_test), y_test, '.')
yys = np.linspace(y_test.min(), y_test.max(), 10)
ax.plot(yys, yys, 'r')
```

```
ax.set_xlabel('Predictions')
ax.set_ylabel('Observations')
ax.set_title('Observations vs. Predictions - Test Data');
```



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.

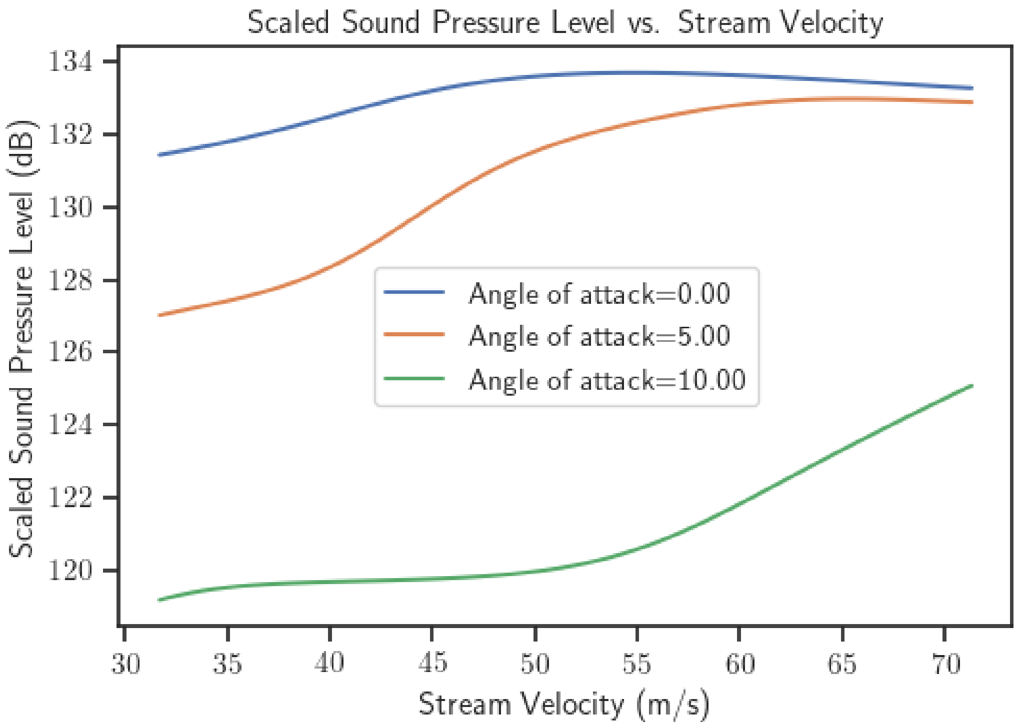
Answer:

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.

[ ]:
```
best_model = 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('Stream Velocity (m/s)')
    ax.set_ylabel('Scaled Sound Pressure Level (dB)')
    ax.set_title('Scaled Sound Pressure Level vs. Stream Velocity')
```

[ ]:
```
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');
```



1.5 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:

```
[ ]: !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
100  203M  100  203M    0     0  40.7M      0  0:00:04  0:00:04 --:--:-- 54.0M
```

Here is how to load the data:

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

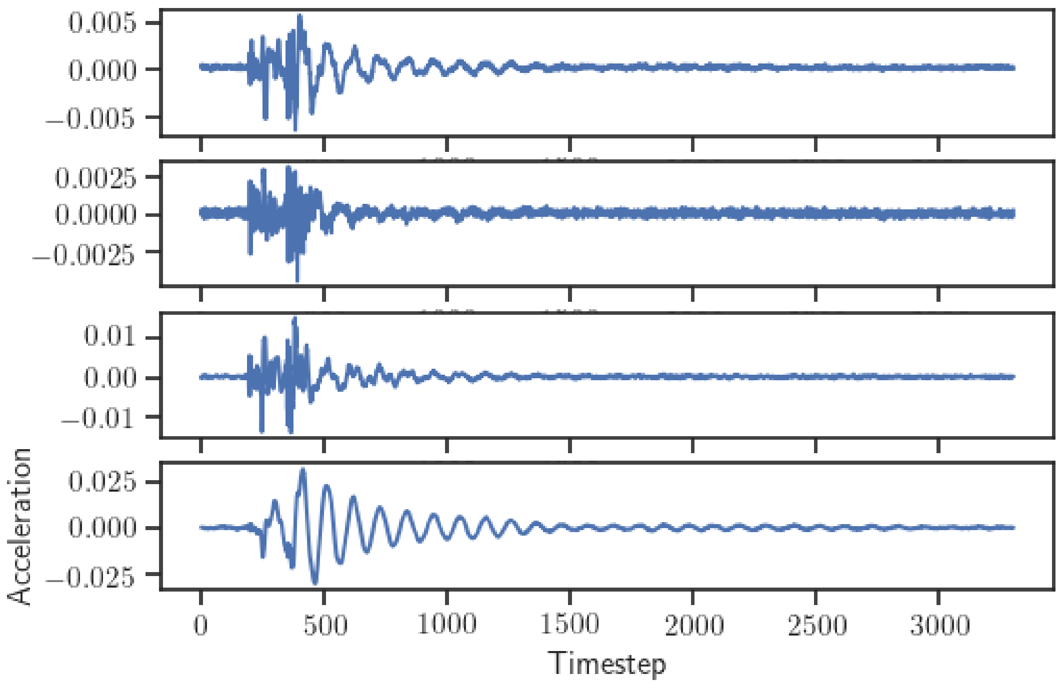
This is a Python dictionary that contains the following entries:

```
[ ]: 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:

```
[ ]: 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');
```

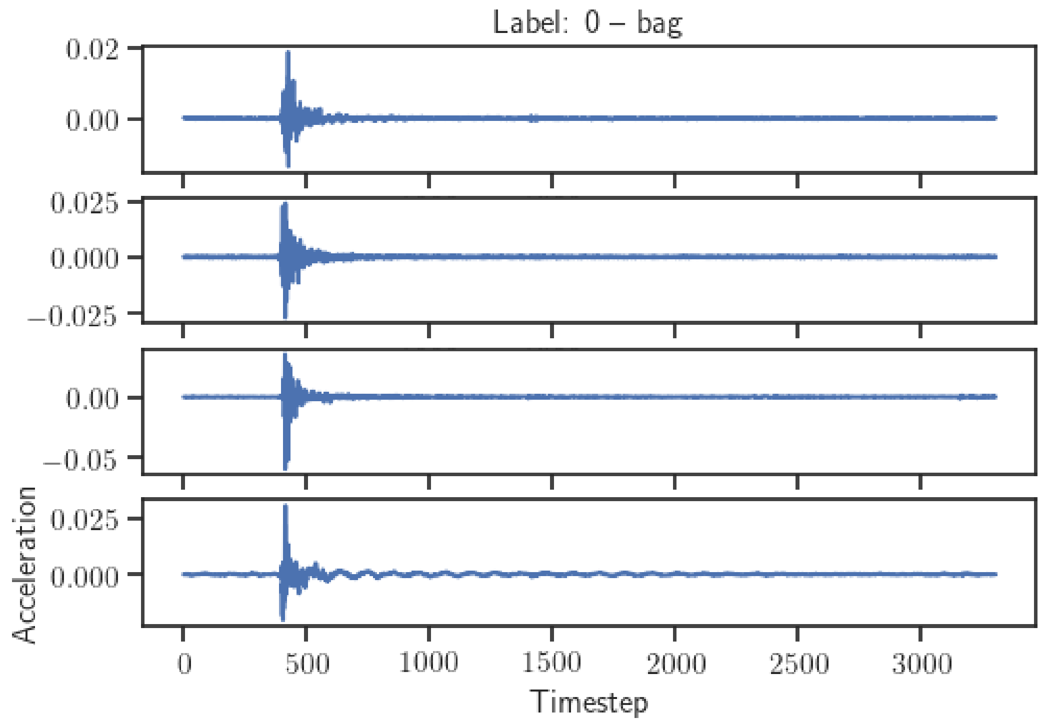


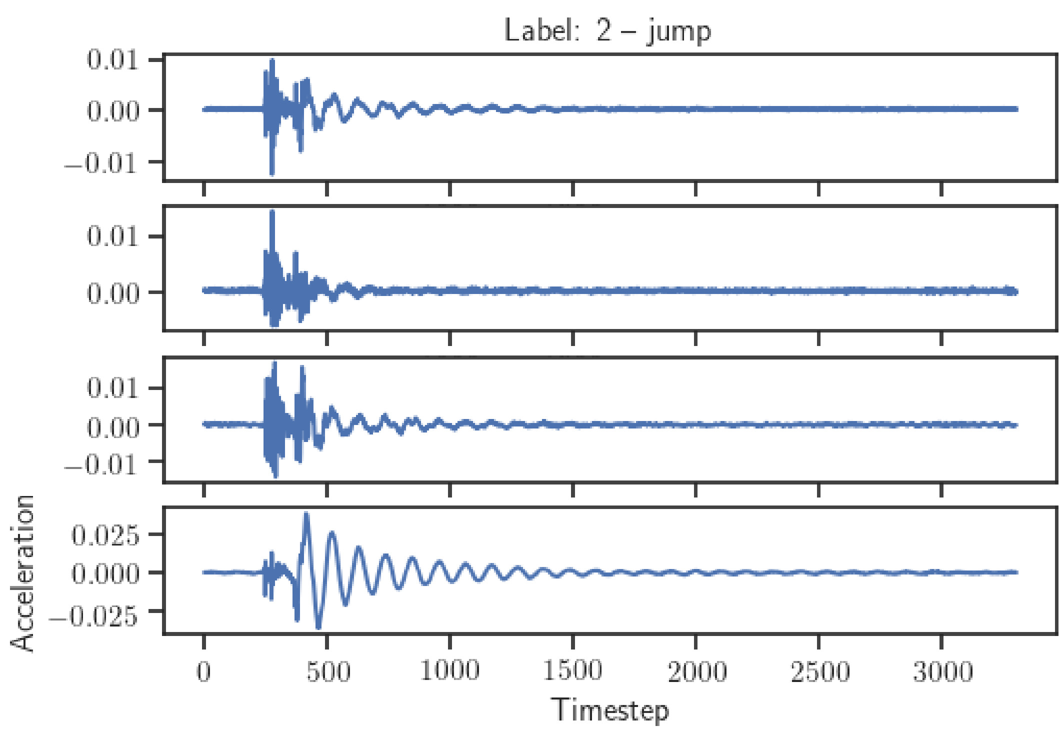
The second key, `labels_1`, is a bunch of integers ranging from 0 to 2 indicating whether the entry corresponds to a "bag," a "ball" or a "jump." For your reference, the correspondence is:

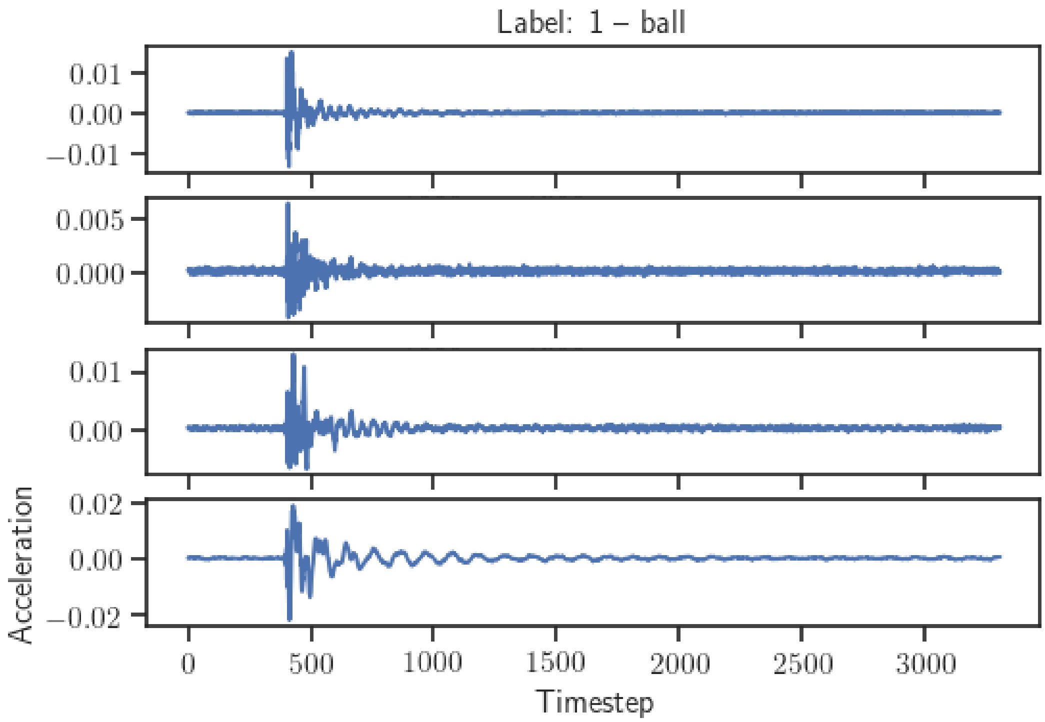
```
[ ]: LABELS_1_TO_TEXT = {
         0: 'bag',
         1: 'ball',
         2: 'jump'
     }
```

And here are a few examples:

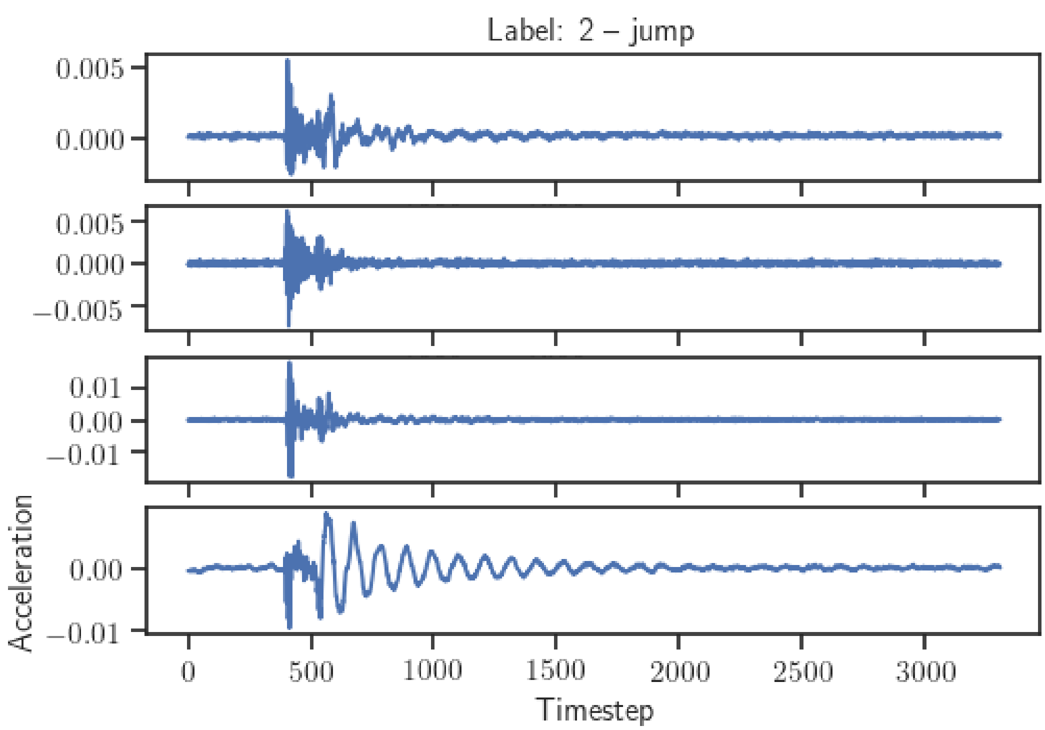
```
[ ]: for _ in range(5):
         i = np.random.randint(0, data['features'].shape[0])
         fig, ax = plt.subplots(4, 1, dpi=100)
         for j in range(4):
             ax[j].plot(data['features'][i, j])
         ax[-1].set_xlabel('Timestep')
         ax[-1].set_ylabel('Acceleration')
         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
0.02
0.00
0.025
0.000
−0.025
0.00
−0.05
0.025
0.000
Acceleration
0
500
1000
1500
2000
2500
3000
Timestep

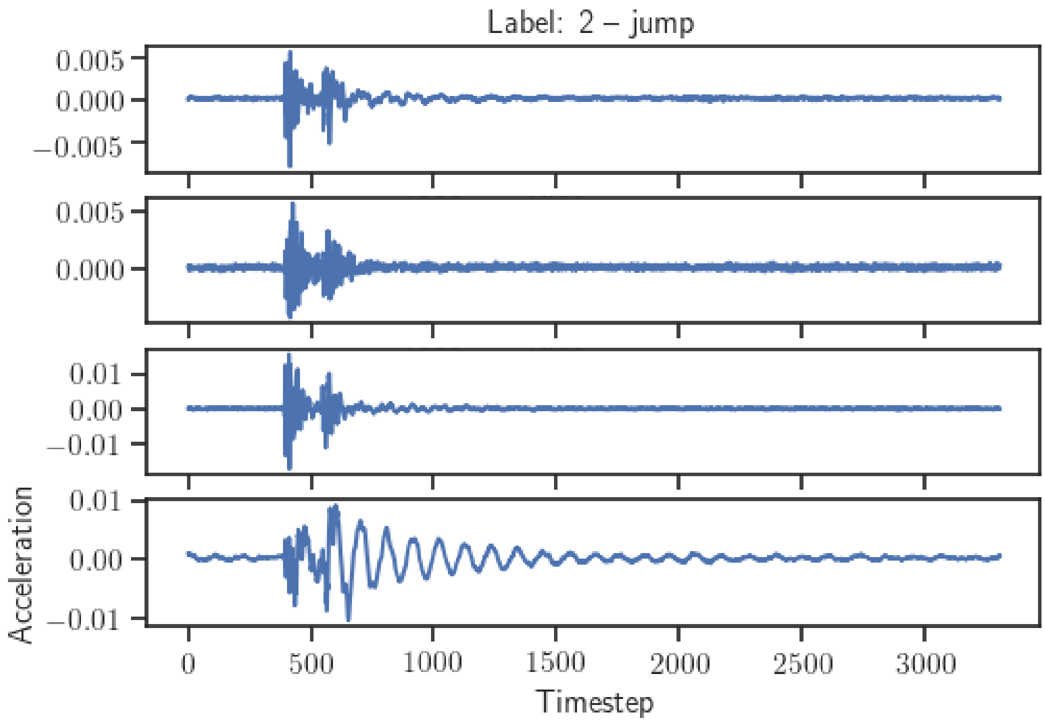

Label: 2 – jump
0.01
0.00
−0.01
0.01
0.00
0.01
0.00
−0.01
0.025
0.000
−0.025
Acceleration
0
500
1000
1500
2000
2500
3000
Timestep



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



Label: 2 – jump
0.005
0.000
0.005
0.000
−0.005
0.01
0.00
−0.01
0.00
−0.01
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:

```
[ ]: 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:

```
[ ]: # 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']
```

1.5.1 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:

[ ]:
```
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
```

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

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

[ ]:
```
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.

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

def train_cnn(X, y, net, n_batch, epochs, lr, test_size=0.33, dispAcc=True):
    """
    A function that trains a regression neural network using stochastic 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 = torch.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
            # cnn_loss_func(y_batch, y_pred, reg_weight)
            # Hint 1: You have defined cnn_loss_func() already
            # Hint 2: Consult the hands-on activities for an example
            # zero out the gradient buffers
            optimizer.zero_grad()
            # make predictions
            y_pred = net(X_batch)
            # evaluate the loss
            loss = cnn_loss_func(y_pred, y_batch)
            # evaluate the derivative of the loss w.r.t. all parameters
            loss.backward()
            # make step
            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])
        if dispAcc==True:
          # Print something about the accuracy
          print('Epoch {0:d}: accuracy = {1:1.5f}%'.format(e+1, accuracy[-1]))
    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
# reference(s): hands-on activity 24, hands-on activity 25
```

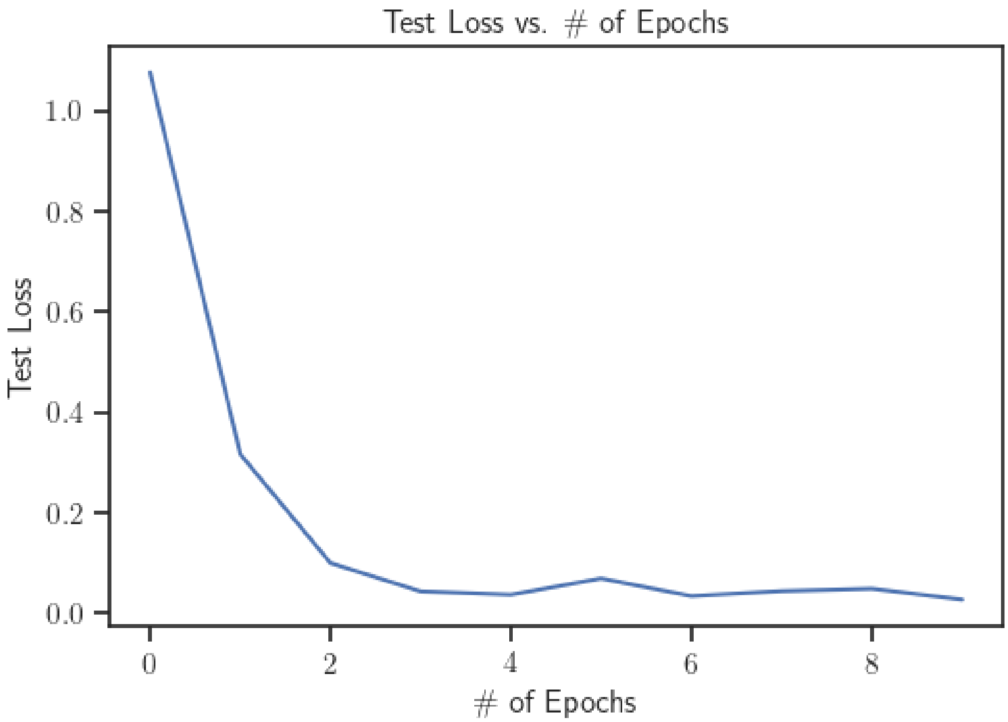
Now experiment with the epochs, the learning rate, and the batch size until this works.

[ ]:
```
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 = 0.43115%
Epoch 2: accuracy = 0.86907%
Epoch 3: accuracy = 0.97366%
Epoch 4: accuracy = 0.98796%
Epoch 5: accuracy = 0.99398%
Epoch 6: accuracy = 0.98646%
Epoch 7: accuracy = 0.99624%
Epoch 8: accuracy = 0.98871%
Epoch 9: accuracy = 0.99097%
Epoch 10: accuracy = 0.99774%
```

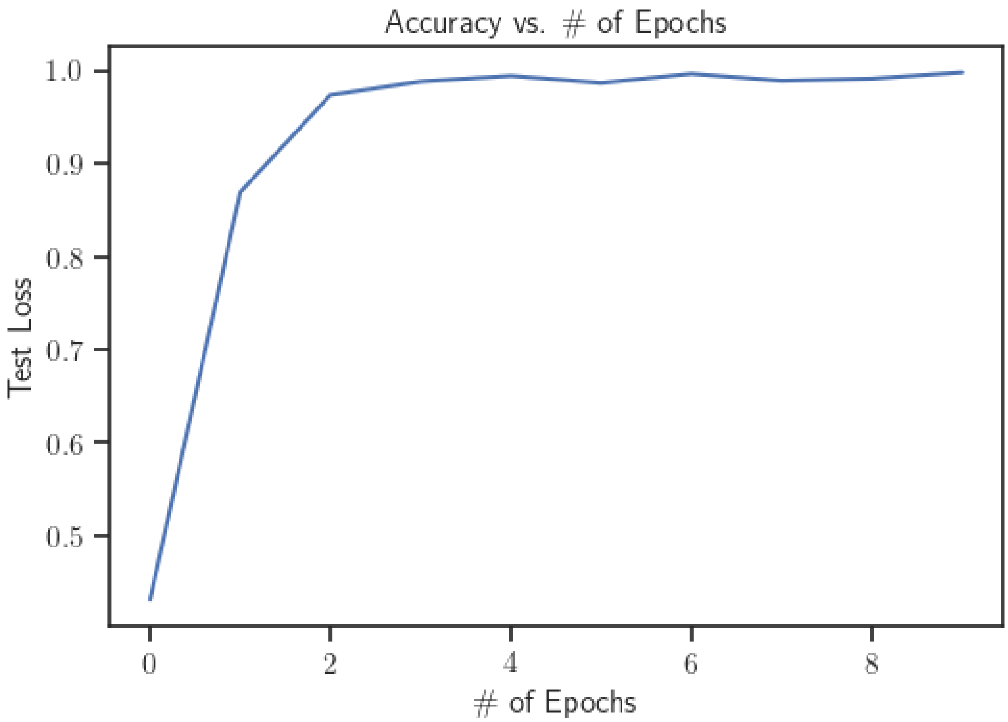
Plot the evolution of the test loss as a function of epochs.

[ ]:
```
fig, ax = plt.subplots(dpi=100)
ax.plot(test_loss)
ax.set_xlabel('\# of Epochs')
ax.set_ylabel('Test Loss')
ax.set_title('Test Loss vs. \# of Epochs');
```



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

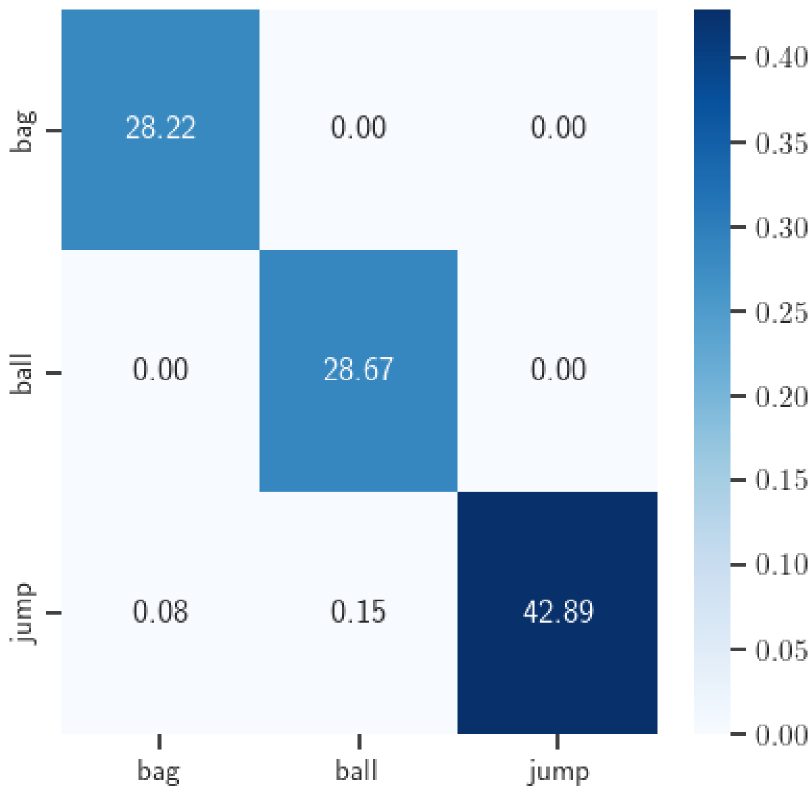
[ ]:
```
fig, ax = plt.subplots(dpi=100)
ax.plot(accuracy)
ax.set_xlabel('\# of Epochs')
ax.set_ylabel('Test Loss')
ax.set_title('Accuracy vs. \# of Epochs');
```



Plot the confusion matrix.

```
[ ]: 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)
```

```
[ ]: fig, ax = plt.subplots(figsize=(5,5))
     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());
```



1.5.2 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`.

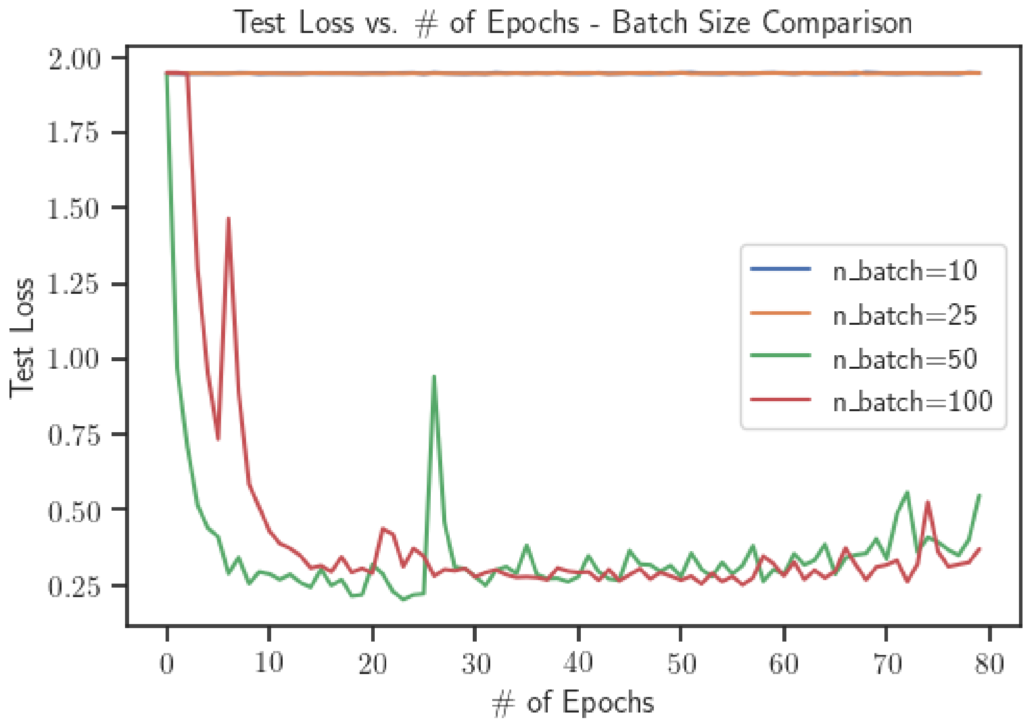
Answer:

For this model, the first step is to re-perform the same hyperparemeter selection tests as seen further above in this document. A similar methodology is adopted from previously to select the optimal hyperparameters based on a combination of best test loss convergence, test loss convergence rate, computation time, and mild penalization of large parametric model weights.

```
[ ]: # re-performing previous operation to determine the best hyperparameter␣
      ↪selections
     # batch size ******************************************************************
     # set parameters, with batch size as a vector
     epochs = 80
     lr = 0.01
     test_losses = []
     models = []
     batches = [10, 25, 50, 100]
     # loop through the different batch sizes and train the network using each
```

```python
for n_batch in batches:
    print('Training n_batch: {0:d}'.format(n_batch))
    net = Net(7)
    model, test_loss, accuracy, X_train, y_train, X_test, y_test = train_cnn(
        X,
        y2,
        net,
        n_batch,
        epochs,
        lr,
        dispAcc=False
    )
    test_losses.append(test_loss)
    models.append(model)
# plotting results from adjusting the batch size
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('\# of Epochs')
ax.set_ylabel('Test Loss')
ax.set_title('Test Loss vs. \# of Epochs - Batch Size Comparison')
plt.legend(loc='best');
```

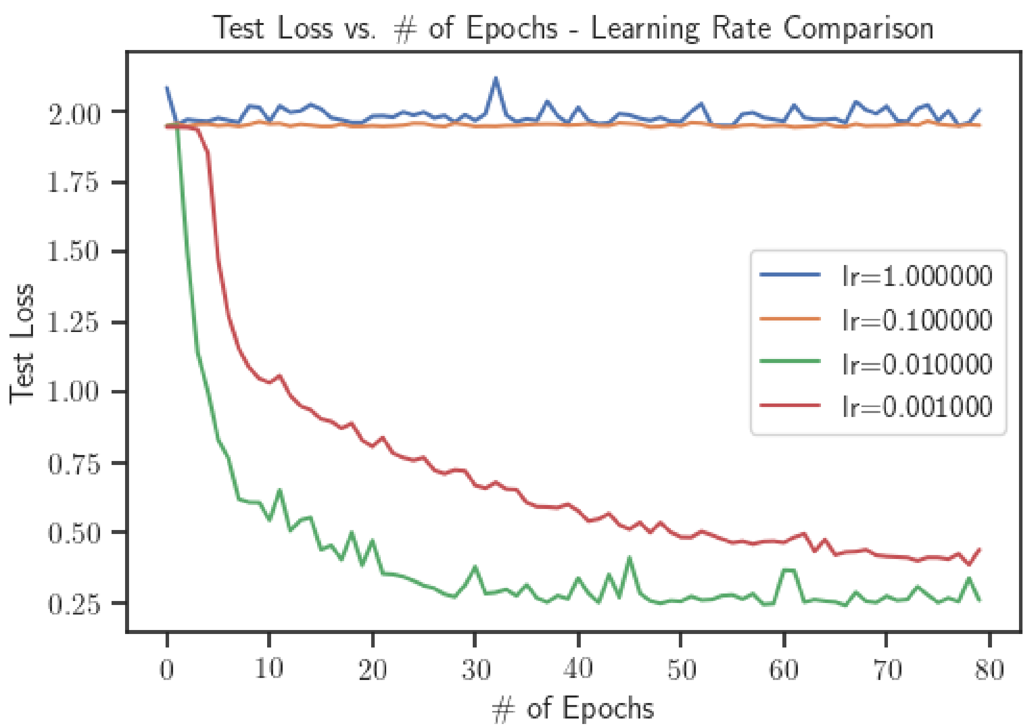
```
Training n_batch: 10
Training n_batch: 25
Training n_batch: 50
Training n_batch: 100
```



[ ]:
```
# learning rate ******************************************************************
# set parameters, with learning rate as a vector
epochs = 80
lrs = [1, 0.1, 0.01, 0.001]
test_losses = []
models = []
batch = 100
# loop through the different learning rates and train the network using each
for n_learn in lrs:
    print('Training n_learn: {0:f}'.format(n_learn))
    net = Net(7)
    model, test_loss, accuracy, X_train, y_train, X_test, y_test = train_cnn(
        X,
        y2,
        net,
        batch,
        epochs,
        n_learn,
        dispAcc=False
    )
    test_losses.append(test_loss)
    models.append(model)
# plotting results from adjusting the learning rate
```

```
fig, ax = plt.subplots(dpi=100)
for tl, n_learn in zip(test_losses, lrs):
    ax.plot(tl, label='lr={0:f}'.format(n_learn))
ax.set_xlabel('\# of Epochs')
ax.set_ylabel('Test Loss')
ax.set_title('Test Loss vs. \# of Epochs - Learning Rate Comparison')
plt.legend(loc='best');
```

```
Training n_learn: 1.000000
Training n_learn: 0.100000
Training n_learn: 0.010000
Training n_learn: 0.001000
```




```
[ ]: # use slightly more epochs than what was used for hyperparemeter selection tests
epochs = 100
# optimal learning rate, based on results above
lr = 0.01
# optimal batch size, based on results above
n_batch = 100
# re-define the network
net = Net(7)
# train model
```

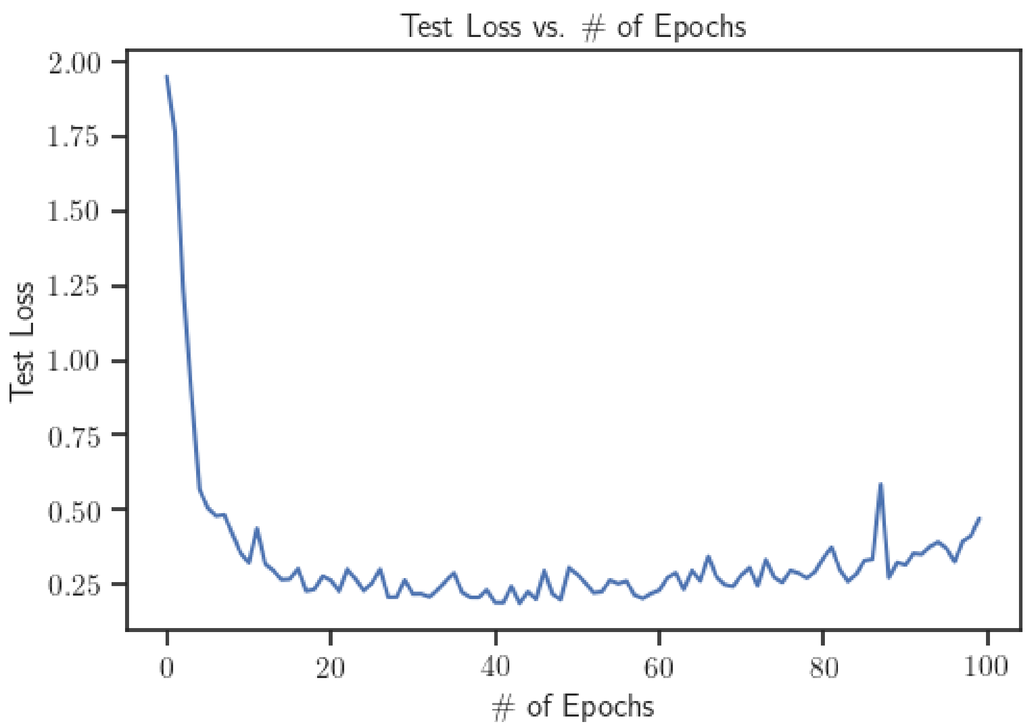
```
trained_model, test_loss, accuracy, X_train, y_train, X_test, y_test =␣
↪train_cnn(X, y2, net, n_batch, epochs, lr)
```

```
Epoch 1: accuracy = 0.13469%
Epoch 2: accuracy = 0.26411%
Epoch 3: accuracy = 0.34011%
Epoch 4: accuracy = 0.58766%
Epoch 5: accuracy = 0.73890%
Epoch 6: accuracy = 0.78330%
Epoch 7: accuracy = 0.78405%
Epoch 8: accuracy = 0.78631%
Epoch 9: accuracy = 0.78932%
Epoch 10: accuracy = 0.80813%
Epoch 11: accuracy = 0.83747%
Epoch 12: accuracy = 0.78254%
Epoch 13: accuracy = 0.83822%
Epoch 14: accuracy = 0.85553%
Epoch 15: accuracy = 0.87434%
Epoch 16: accuracy = 0.86832%
Epoch 17: accuracy = 0.84725%
Epoch 18: accuracy = 0.88789%
Epoch 19: accuracy = 0.88563%
Epoch 20: accuracy = 0.85026%
Epoch 21: accuracy = 0.89240%
Epoch 22: accuracy = 0.89165%
Epoch 23: accuracy = 0.85628%
Epoch 24: accuracy = 0.88111%
Epoch 25: accuracy = 0.88638%
Epoch 26: accuracy = 0.87058%
Epoch 27: accuracy = 0.86456%
Epoch 28: accuracy = 0.90444%
Epoch 29: accuracy = 0.90444%
Epoch 30: accuracy = 0.88262%
Epoch 31: accuracy = 0.89014%
Epoch 32: accuracy = 0.90293%
Epoch 33: accuracy = 0.89767%
Epoch 34: accuracy = 0.89541%
Epoch 35: accuracy = 0.88187%
Epoch 36: accuracy = 0.86682%
Epoch 37: accuracy = 0.90218%
Epoch 38: accuracy = 0.89691%
Epoch 39: accuracy = 0.89917%
Epoch 40: accuracy = 0.89391%
Epoch 41: accuracy = 0.91121%
Epoch 42: accuracy = 0.91723%
Epoch 43: accuracy = 0.88939%
Epoch 44: accuracy = 0.91347%
```

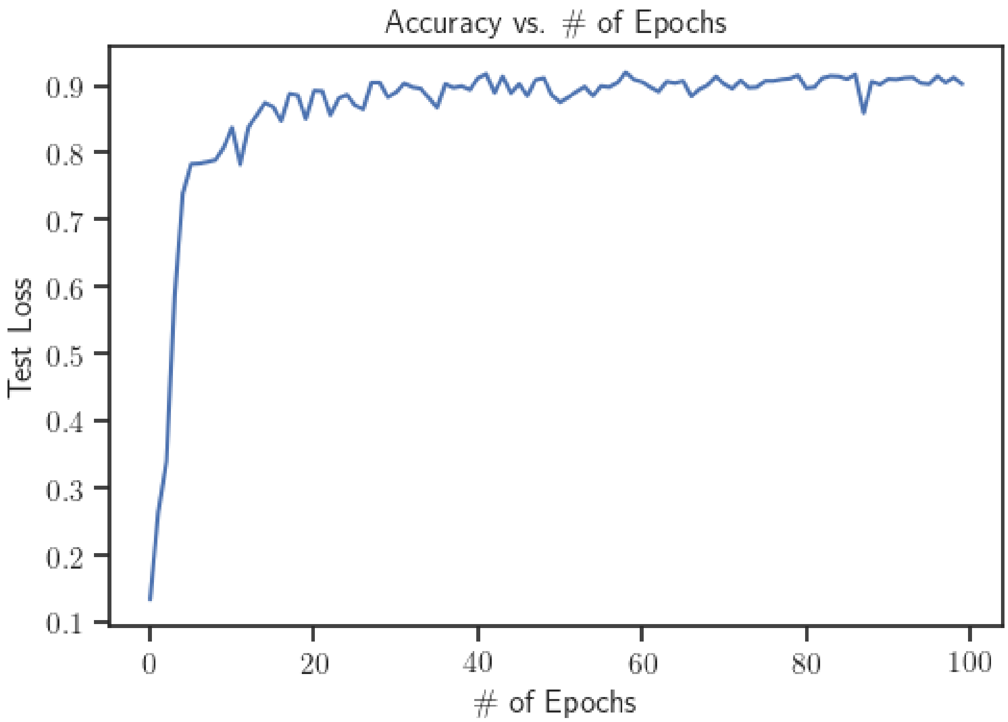
```
Epoch 45: accuracy = 0.88864%
Epoch 46: accuracy = 0.90218%
Epoch 47: accuracy = 0.88488%
Epoch 48: accuracy = 0.90820%
Epoch 49: accuracy = 0.91121%
Epoch 50: accuracy = 0.88563%
Epoch 51: accuracy = 0.87509%
Epoch 52: accuracy = 0.88262%
Epoch 53: accuracy = 0.89090%
Epoch 54: accuracy = 0.89842%
Epoch 55: accuracy = 0.88488%
Epoch 56: accuracy = 0.89917%
Epoch 57: accuracy = 0.89767%
Epoch 58: accuracy = 0.90444%
Epoch 59: accuracy = 0.91949%
Epoch 60: accuracy = 0.90895%
Epoch 61: accuracy = 0.90519%
Epoch 62: accuracy = 0.89767%
Epoch 63: accuracy = 0.89090%
Epoch 64: accuracy = 0.90594%
Epoch 65: accuracy = 0.90369%
Epoch 66: accuracy = 0.90670%
Epoch 67: accuracy = 0.88412%
Epoch 68: accuracy = 0.89466%
Epoch 69: accuracy = 0.90068%
Epoch 70: accuracy = 0.91347%
Epoch 71: accuracy = 0.90218%
Epoch 72: accuracy = 0.89541%
Epoch 73: accuracy = 0.90745%
Epoch 74: accuracy = 0.89691%
Epoch 75: accuracy = 0.89767%
Epoch 76: accuracy = 0.90670%
Epoch 77: accuracy = 0.90670%
Epoch 78: accuracy = 0.90895%
Epoch 79: accuracy = 0.91046%
Epoch 80: accuracy = 0.91497%
Epoch 81: accuracy = 0.89616%
Epoch 82: accuracy = 0.89767%
Epoch 83: accuracy = 0.91121%
Epoch 84: accuracy = 0.91422%
Epoch 85: accuracy = 0.91347%
Epoch 86: accuracy = 0.90971%
Epoch 87: accuracy = 0.91648%
Epoch 88: accuracy = 0.85929%
Epoch 89: accuracy = 0.90594%
Epoch 90: accuracy = 0.90143%
Epoch 91: accuracy = 0.90971%
Epoch 92: accuracy = 0.90895%
```

```
Epoch 93: accuracy = 0.91121%
Epoch 94: accuracy = 0.91196%
Epoch 95: accuracy = 0.90369%
Epoch 96: accuracy = 0.90218%
Epoch 97: accuracy = 0.91422%
Epoch 98: accuracy = 0.90444%
Epoch 99: accuracy = 0.91196%
Epoch 100: accuracy = 0.90218%
```

[ ]:
```
fig, ax = plt.subplots(dpi=100)
ax.plot(test_loss)
ax.set_xlabel('\# of Epochs')
ax.set_ylabel('Test Loss')
ax.set_title('Test Loss vs. \# of Epochs');
```

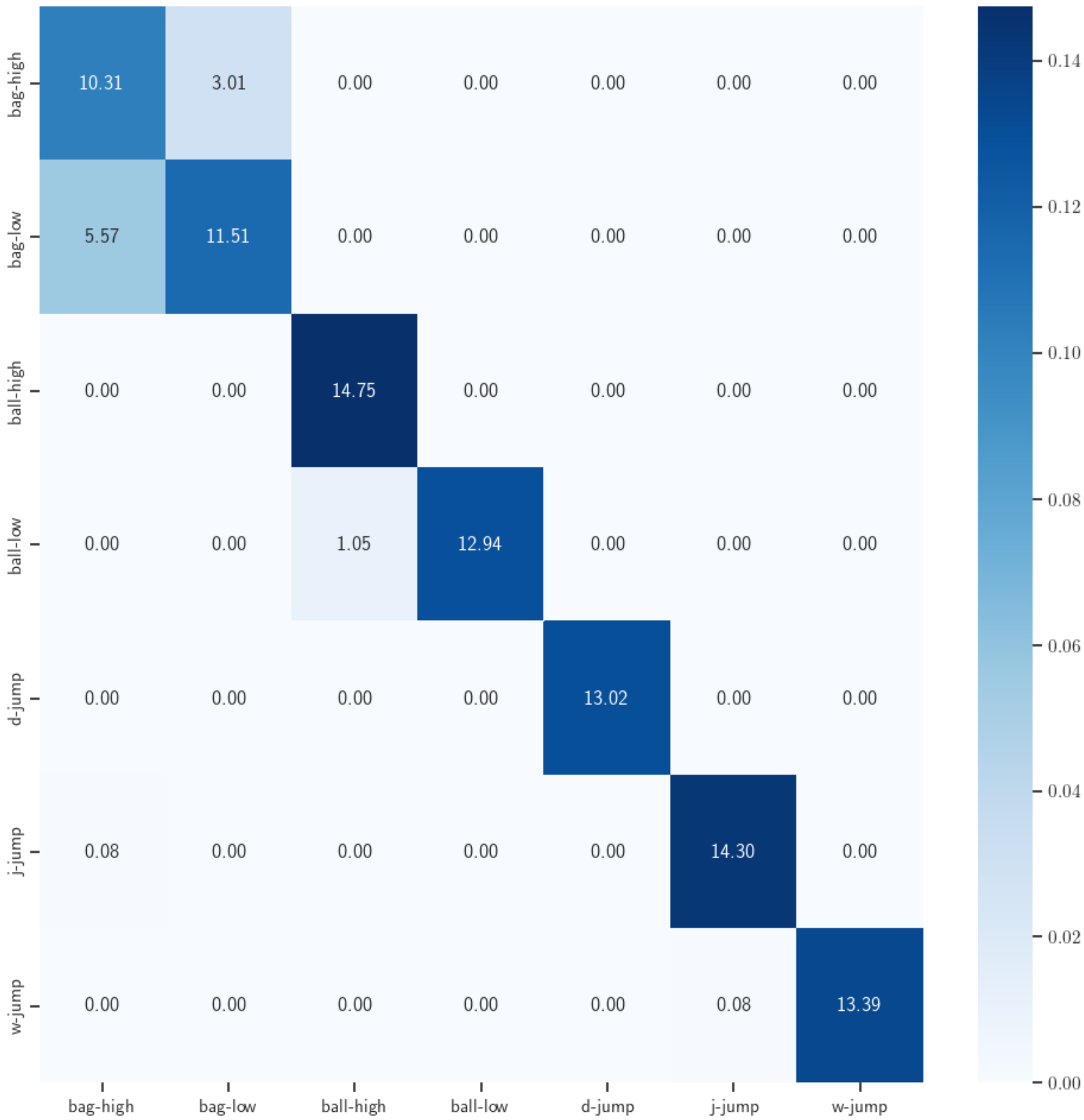


[ ]:
```
fig, ax = plt.subplots(dpi=100)
ax.plot(accuracy)
ax.set_xlabel('\# of Epochs')
ax.set_ylabel('Test Loss')
ax.set_title('Accuracy vs. \# of Epochs');
```




```
[ ]: # 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)
```

```
[ ]: fig, ax = plt.subplots(figsize=(12,12))
     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());
```


bag-high
bag-low
ball-high
ball-low
d-jump
j-jump
w-jump
10.31
3.01
0.00
5.57
11.51
14.75
1.05
12.94
13.02
0.08
14.30
13.39
0.14
0.12
0.10
0.08
0.06
0.04
0.02
0.00