Homework 7 - Part A

Note that there are two different notebooks for HW assignment 7. This is part A. There will be two different assignments in gradescope for each part. The deadlines are the same for both parts.

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 [ ]:
```python
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import matplotlib_inline
matplotlib_inline.backend_inline.set_matplotlib_formats('svg')
import seaborn as sns
sns.set_context("paper")
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:** Kyle
- **Last Name:** Illenden
- **Email:** killende@purdue.edu

In this problem, you must 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 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. The 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.

In [ ]:
```
#!curl -O --insecure "https://archive.ics.uci.edu/ml/machine-learning-databases/00291/airfoil_self_noise.dat"
```

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

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

Out[ ]:
```
array([[8.00000e+02, 0.00000e+00, 3.04800e-01, 7.13000e+01, 2.66337e-03,
        1.26201e+02],
       [1.00000e+03, 0.00000e+00, 3.04800e-01, 7.13000e+01, 2.66337e-03,
        1.25201e+02],
       [1.25000e+03, 0.00000e+00, 3.04800e-01, 7.13000e+01, 2.66337e-03,
        1.25951e+02],
       ...,
       [4.00000e+03, 1.56000e+01, 1.01600e-01, 3.96000e+01, 5.28487e-02,
        1.06604e+02],
       [5.00000e+03, 1.56000e+01, 1.01600e-01, 3.96000e+01, 5.28487e-02,
        1.06224e+02],
       [6.30000e+03, 1.56000e+01, 1.01600e-01, 3.96000e+01, 5.28487e-02,
        1.04204e+02]])
```

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

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

Out[ ]:

	Frequency	Angle_of_attack	Chord_length	Velocity	Suction_thickness	Sound_pressure
0	800.0	0.0	0.3048	71.3	0.002663	126.201
1	1000.0	0.0	0.3048	71.3	0.002663	125.201
2	1250.0	0.0	0.3048	71.3	0.002663	125.951
3	1600.0	0.0	0.3048	71.3	0.002663	127.591
4	2000.0	0.0	0.3048	71.3	0.002663	127.461
...	...	...	...	...	...	...
1498	2500.0	15.6	0.1016	39.6	0.052849	110.264
1499	3150.0	15.6	0.1016	39.6	0.052849	109.254
1500	4000.0	15.6	0.1016	39.6	0.052849	106.604
1501	5000.0	15.6	0.1016	39.6	0.052849	106.224
1502	6300.0	15.6	0.1016	39.6	0.052849	104.204

1503 rows × 6 columns

Part A - Analyze the data visually

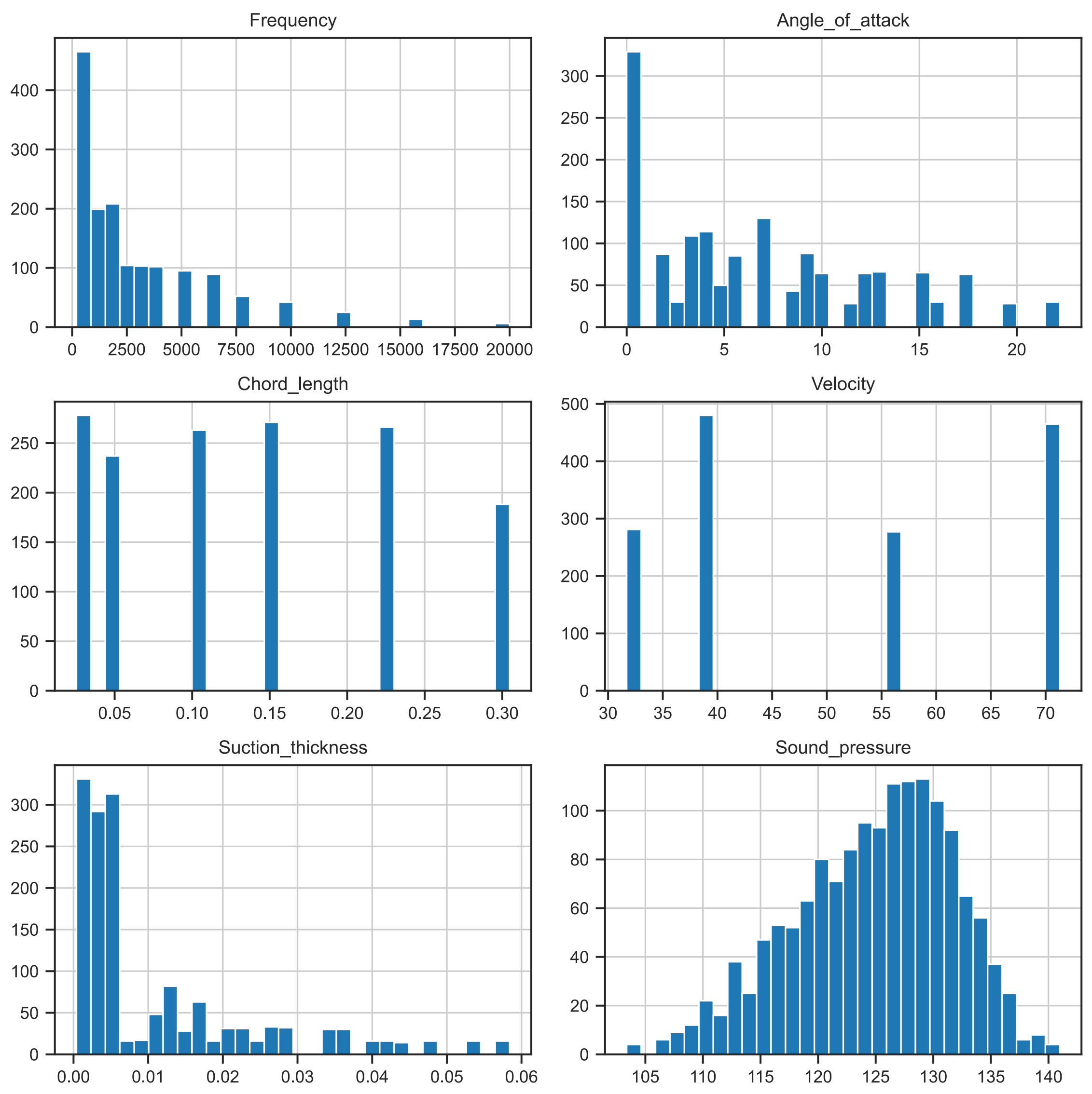
It is always a good idea to visualize the data before you start doing anything with them.

Part A.I. - Do the histograms of all variables

Use as many code segments as 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 standardize the data before moving to regression.

Answer:

In [ ]:
```
# Your code here

df.hist(bins=30, figsize=(8, 8))
plt.tight_layout()
```



Since the data has a meaningful scale, so the y-value in the histograms contains the total counts of the x-value, we do not need to standardize the data.

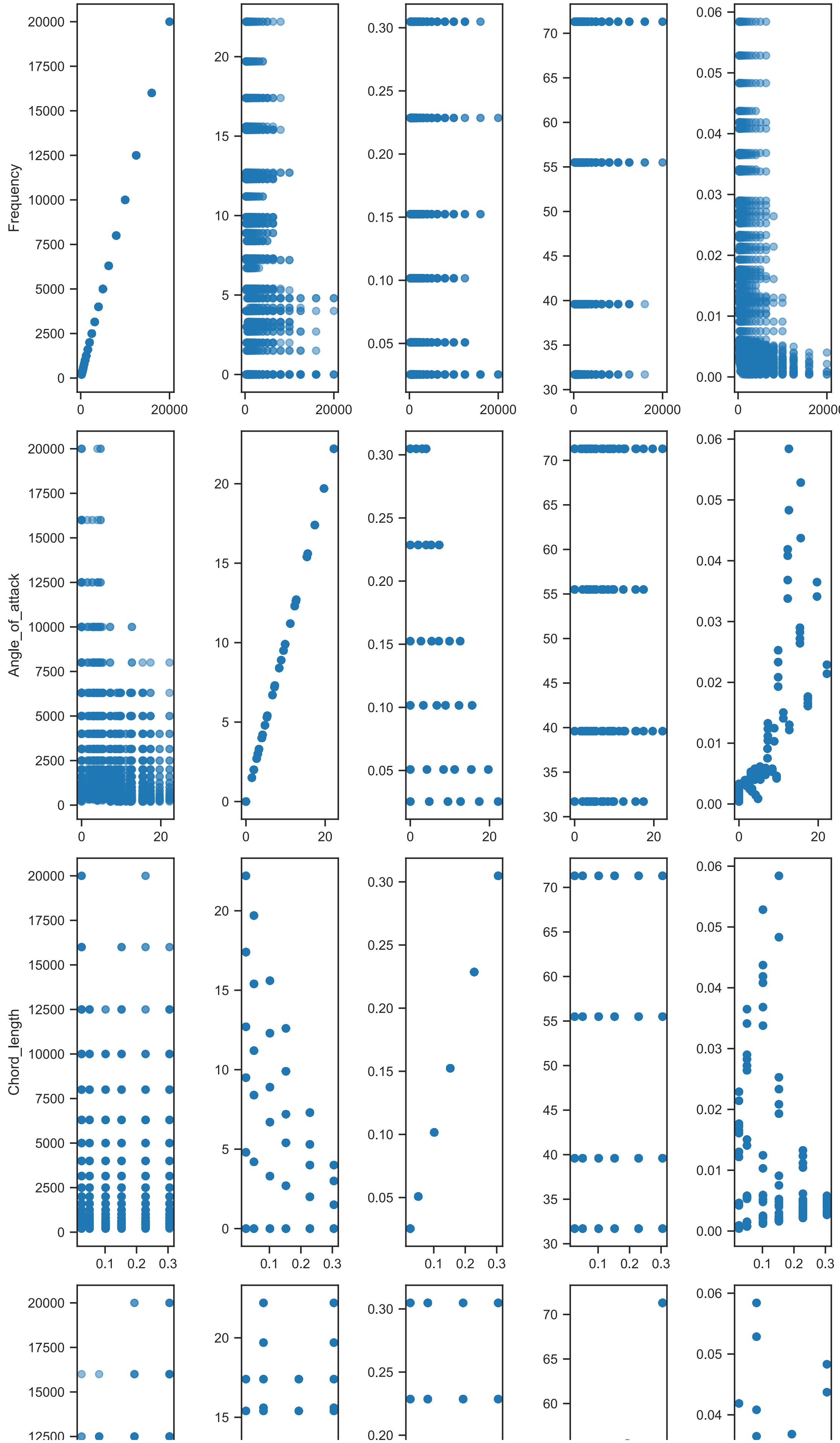
Part A.II - Do the scatter plots between all input variables

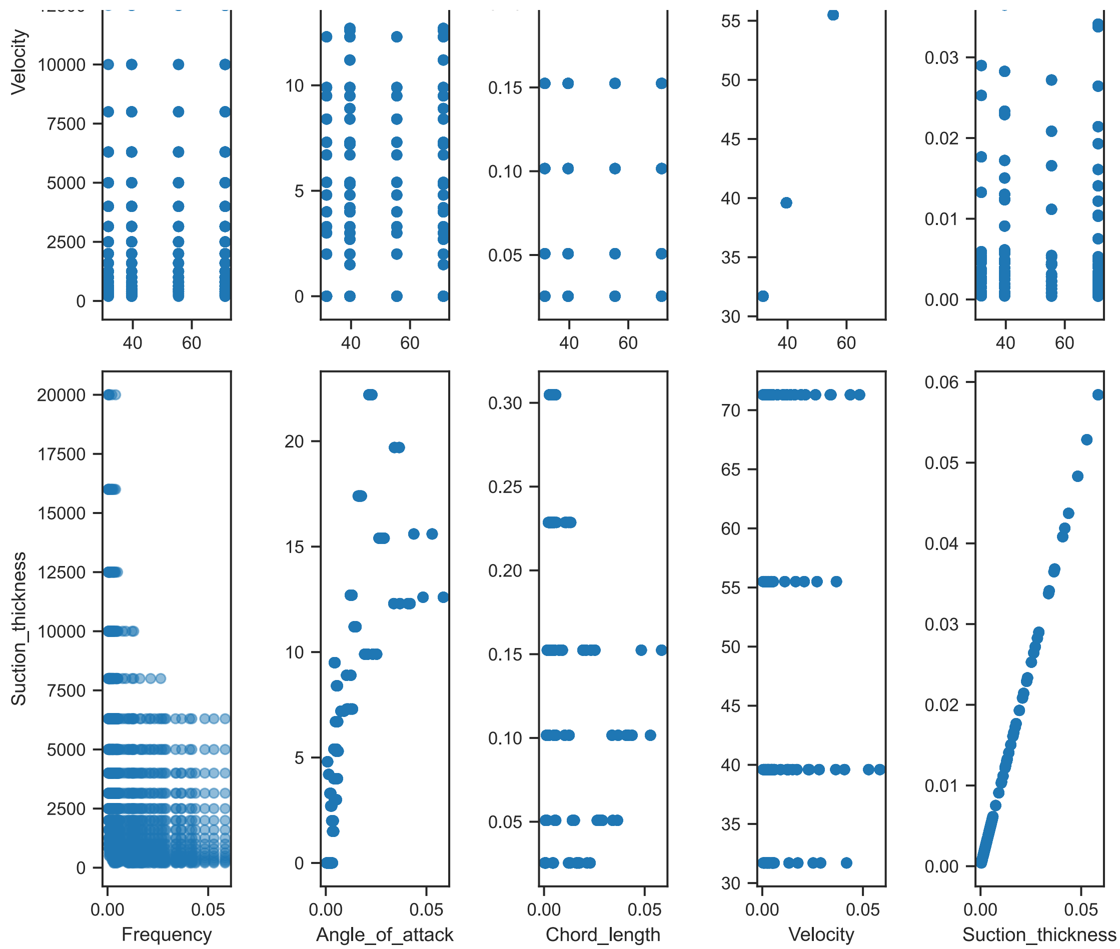
Do the scatter plot between all input variables. This will give you an idea of the range of experimental conditions. Whatever model you build will only be valid inside the domain implicitly defined with your experimental conditions. Are there any holes in the dataset, i.e., places where you have no data?

Answer:

In [ ]:

```
# Your code here
input_columns = df.columns[:5]

num_inputs = len(input_columns)

fig, axes = plt.subplots(nrows=num_inputs, ncols=num_inputs, figsize=(8, num_inputs * 4))

for i in range(num_inputs):
    for j in range(num_inputs):
        axes[i, j].scatter(df[input_columns[i]], df[input_columns[j]], alpha=0.5)


        if i == num_inputs - 1:
            axes[i, j].set_xlabel(input_columns[j])
        if j == 0:
            axes[i, j].set_ylabel(input_columns[i])



plt.tight_layout()
```


Frequency
Angle_of_attack
Chord_length



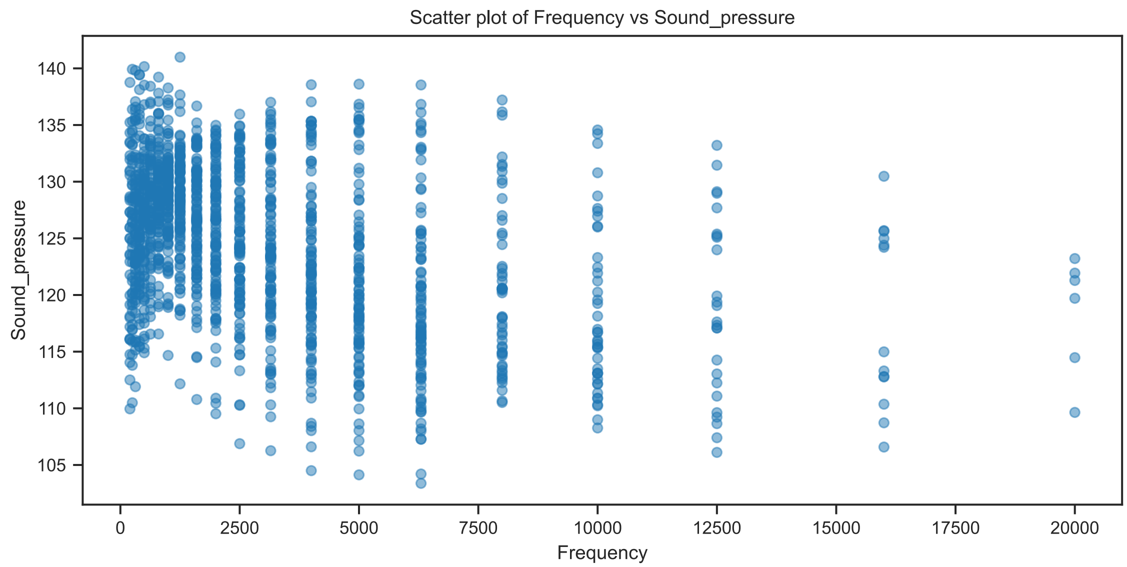
There are a few holes in which we have no data, particularly where the values fall in the same x or y value (depending on which graph is being focused on). For instance, all of the velocity plots have data at the same x-values, just different velocity measurements (if we are looking at the velocity graph row).

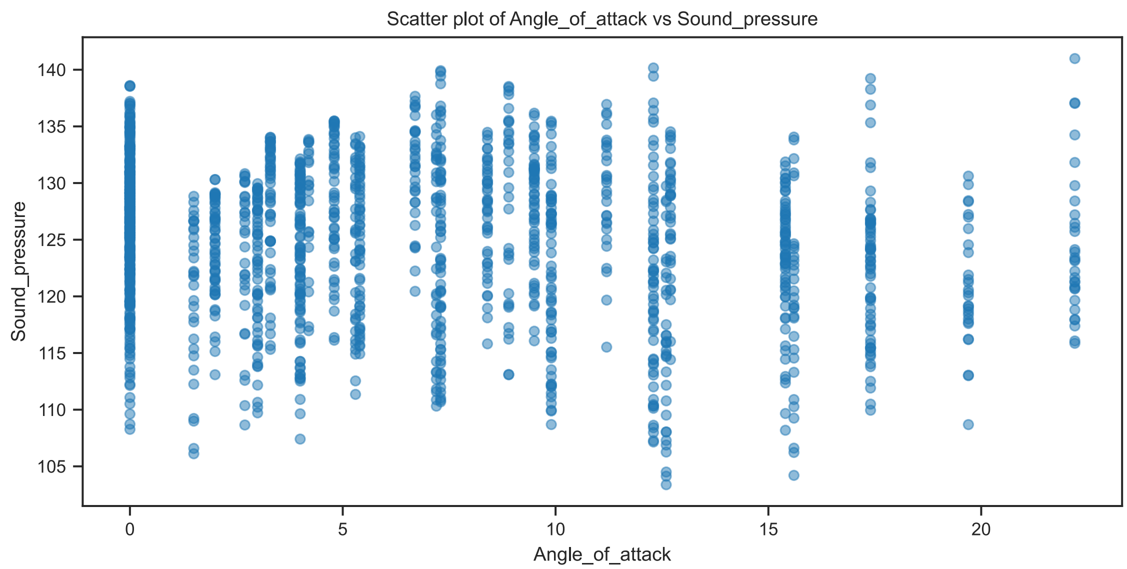
Part A.III - Do the scatter plots between each input and the output

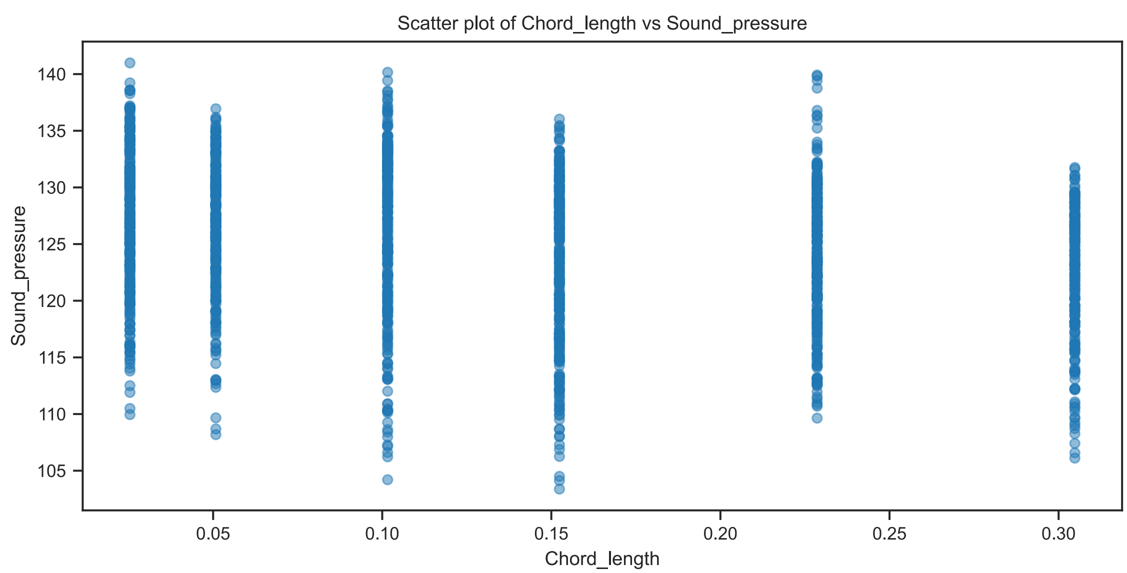
Do the scatter plot between each input variable and the output. This will give you an idea of the functional relationship between the two. Do you observe any obvious patterns?

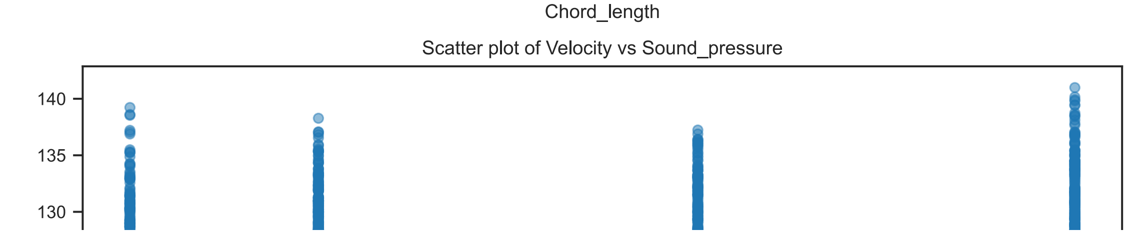
Answer:

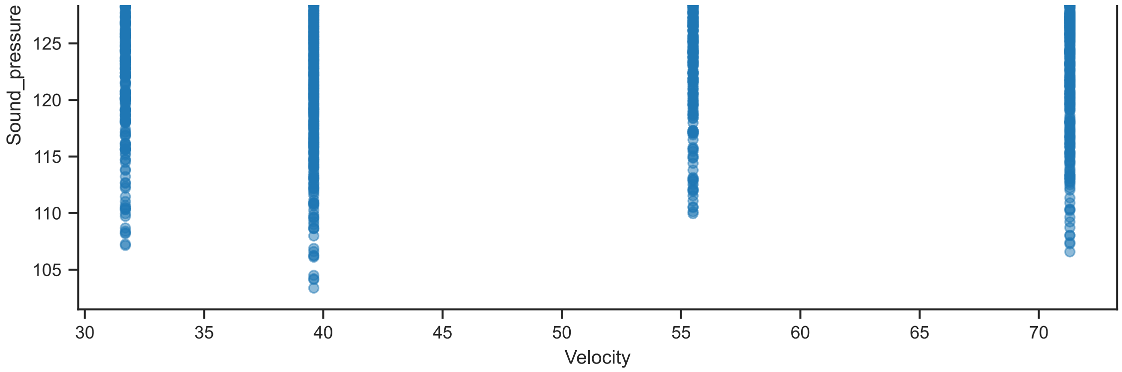
In [ ]:
```
# Your code here

input_columns = df.columns[:-1]
output_column = df.columns[-1]

num_inputs = len(input_columns)

fig, axes = plt.subplots(nrows=num_inputs, ncols=1, figsize=(8, num_inputs * 4))

for i in range(num_inputs):
    axes[i].scatter(df[input_columns[i]], df[output_column], alpha=0.5)
    axes[i].set_xlabel(input_columns[i])
    axes[i].set_ylabel(output_column)
    axes[i].set_title(f'Scatter plot of {input_columns[i]} vs {output_column}')

plt.tight_layout()
```


Scatter plot of Frequency vs Sound_pressure
Sound_pressure
Frequency

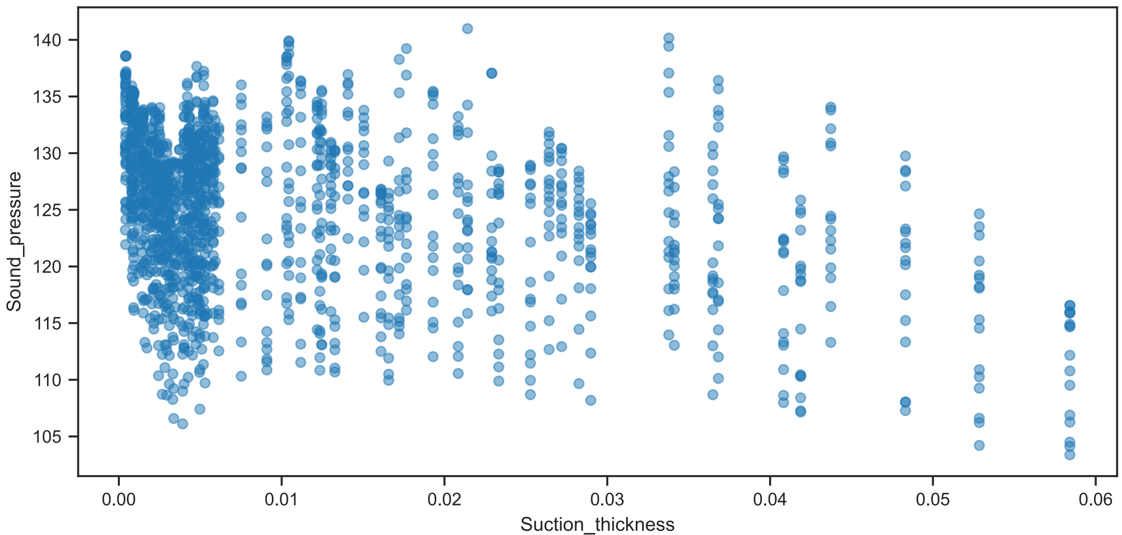

Scatter plot of Angle_of_attack vs Sound_pressure
Sound_pressure
Angle_of_attack


Scatter plot of Chord_length vs Sound_pressure
Sound_pressure
Chord_length


Scatter plot of Velocity vs Sound_pressure



Scatter plot of Suction_thickness vs Sound_pressure



Again, as mentioned in the last part, there are obvious patterns. Specifically looking at the x-values for each data point. The output value (sound pressure) follows a continuous pattern, but the input values follow a discrete behavior.

Part B - Use DNN to do regression

Let start by separating inputs and outputs for you:

In [ ]:
```
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 biases of all network parameters (remember that the L2 regularization is like putting a Gaussian prior on each parameter). Follow the instructions below and fill in the missing code.

Answer:

In [ ]:
```
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.
def mse_loss(obs, pred):
    # Create an instance of the MSELoss class
    criterion = nn.MSELoss()
    # Compute the loss
    return criterion(obs, pred)
```

In [ ]:
```
# 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: 1.95
What you should be getting: 1.95
```

In [ ]:
```
# 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 [ ]:
```
# 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.
    """
    # Your code here
    mse_loss_var = mse_loss(y_obs, y_pred)
    # Compute the regularization term
    reg_loss = sum(torch.sum(param ** 2) for param in params)

    # Compute the total loss
    total_loss = mse_loss_var + reg_weight * reg_loss

    return total_loss
```

In [ ]:
```
# 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: 1.95
This should be the same as this: 1.95
The loss with regularization: 2.03
```

Part B.III - Write flexible code to perform regression

When training neural networks, you must hand-pick many parameters, from the network structure to the activation functions to the regularization parameters to the details of the stochastic optimization. Instead of mindlessly 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 train networks with all different parameter variations repeatedly. In what follows, I will guide you through writing code for training an arbitrary regression network having the flexibility to:

- standardize the inputs and output or not
- experiment with various levels of regularization
- change the learning rate of the stochastic 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 [ ]:
```
# We will start by creating a class that encapsulates a regression
# network so that we can turn on or off input/output standardization
# without too much fuss.
# The class will represent a trained network model.
# It will "know" whether or not during training we standardized the data.
# I am not asking you to do anything here, so you can run this code segment
# or read through it if you want to know 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 standardization
    process in an excellent way.

    Parameters:

    net             -    A network.
    standardized -    True if the network expects standardized features and outputs
                         standardized 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, standardized=False, feature_scaler=None, target_scaler=None):
        self.net = net
        self.standardized = standardized
        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.standardized:
            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 [ ]:
```
# 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,
              standardize=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`.

    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).
    standardize -    Whether or not you want to standardize 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)

    # Standardize the data
    if standardize:
        # 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 magic 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`

    # Let's see now if a stochastic optimizer makes a difference
    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 the gradients
            optimizer.zero_grad()
            # Forward pass
            y_pred = net(X_batch)
            # Compute the loss
            loss = loss_func(y_batch, y_pred, reg_weight, net.parameters())
            # Backward pass
            loss.backward()
            # Update weights
            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, standardized=standardize,
                                 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
```

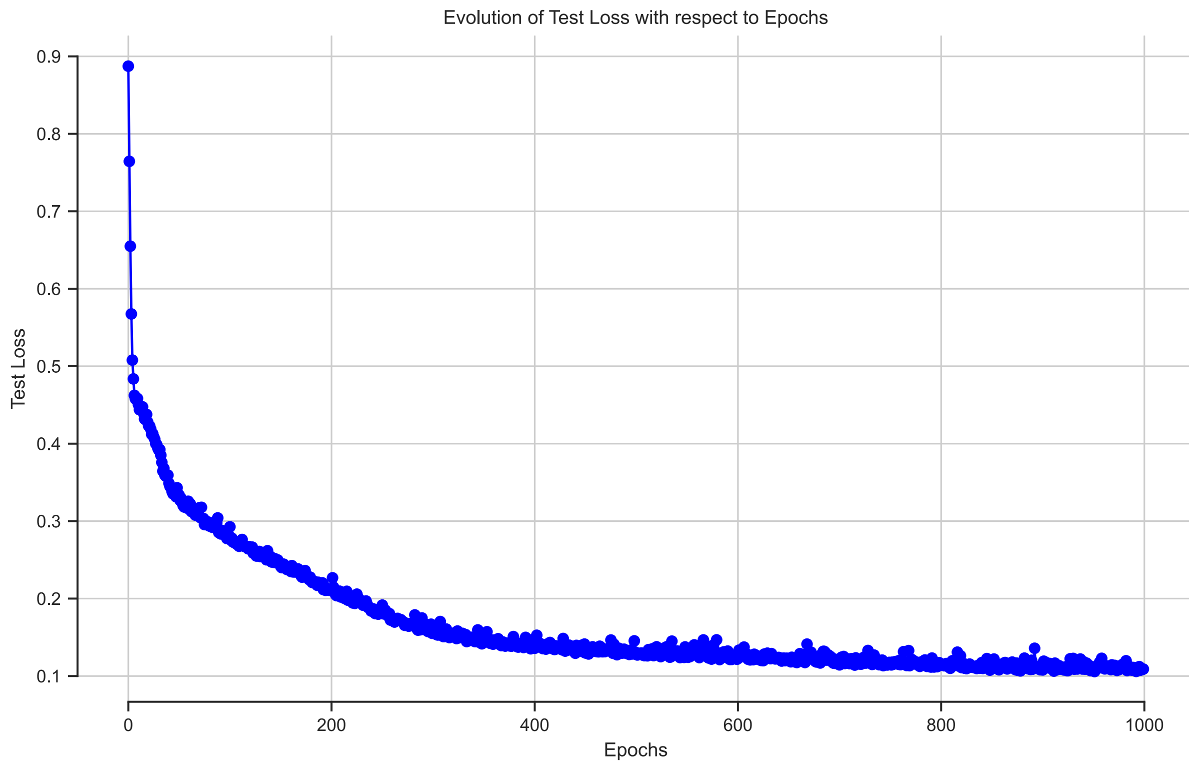
Use this to test your code:

In [ ]:
```
# 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 [00:19<00:00, 50.59it/s]
```

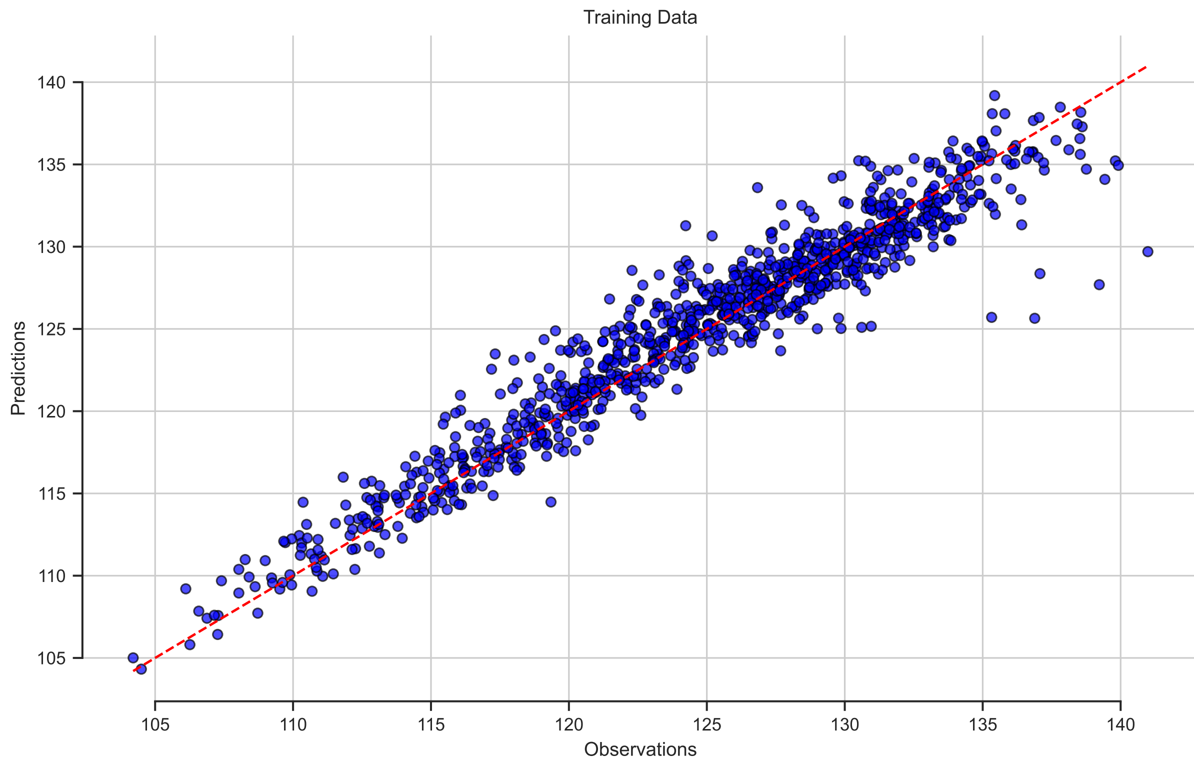
There are a few more things for you to do here. First, plot the evolution of the test loss as a function of the number of epochs:

In [ ]:
```
# Your code here
plt.figure(figsize=(10, 6))
plt.plot(range(epochs), test_loss, marker='o', linestyle='-', color='b')
plt.xlabel('Epochs')
plt.ylabel('Test Loss')
plt.title('Evolution of Test Loss with respect to Epochs')
plt.grid(True)
sns.despine(trim=True);
```



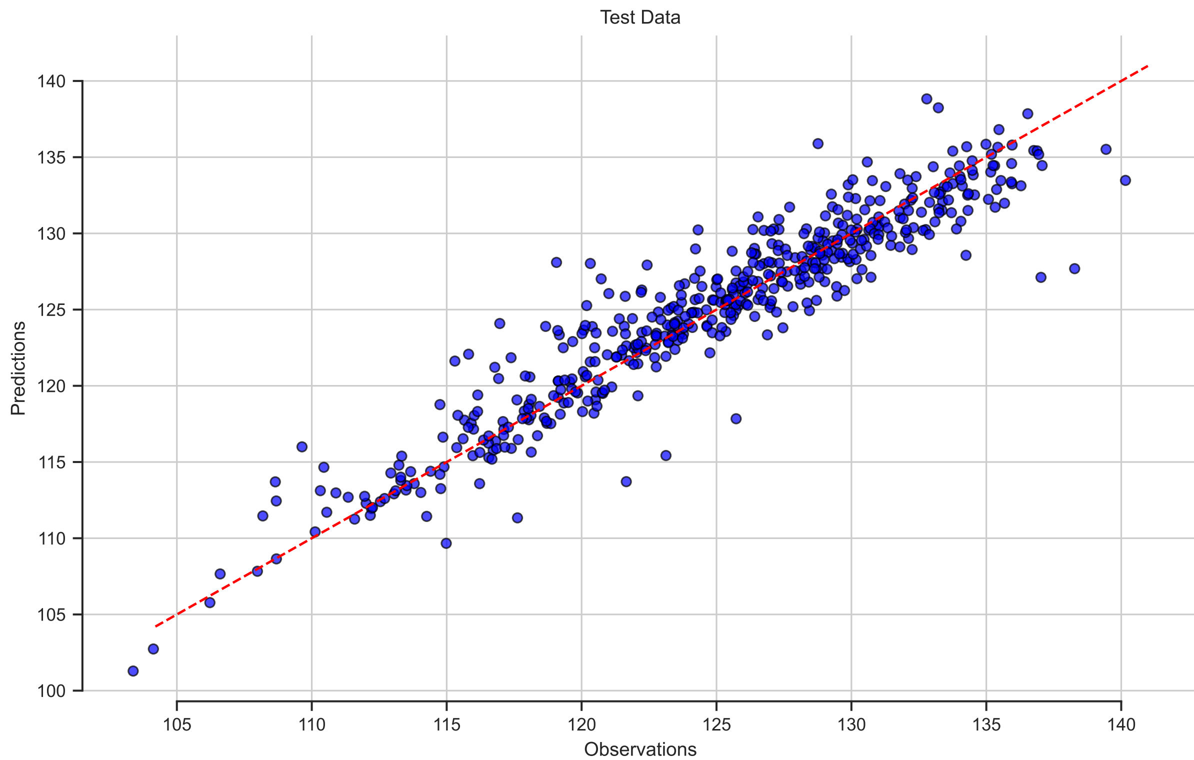
Now plot the observations vs predictions plot for the training data:

In [ ]:
```
# Your code here


X_train_tensor = torch.Tensor(X_train)


with torch.no_grad():
    y_pred = model(X_train_tensor)


plt.figure(figsize=(10, 6))
plt.scatter(y_train, y_pred, color='b', alpha=0.7, edgecolors='k')
plt.xlabel('Observations')
plt.ylabel('Predictions')
plt.title('Training Data')
plt.grid(True)
plt.plot([min(y_train), max(y_train)], [min(y_train), max(y_train)], color='r', linestyle='--')
sns.despine(trim=True);
```



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

```
In [ ]: # Your code here

X_test_tensor = torch.Tensor(X_test)

with torch.no_grad():
    y_pred = model(X_test_tensor)


plt.figure(figsize=(10, 6))
plt.scatter(y_test, y_pred, color='b', alpha=0.7, edgecolors='k')
plt.xlabel('Observations')
plt.ylabel('Predictions')
plt.title('Test Data')
plt.grid(True)
plt.plot([min(y_train), max(y_train)], [min(y_train), max(y_train)], color='r', linestyle='--')
sns.despine(trim=True);
```



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?

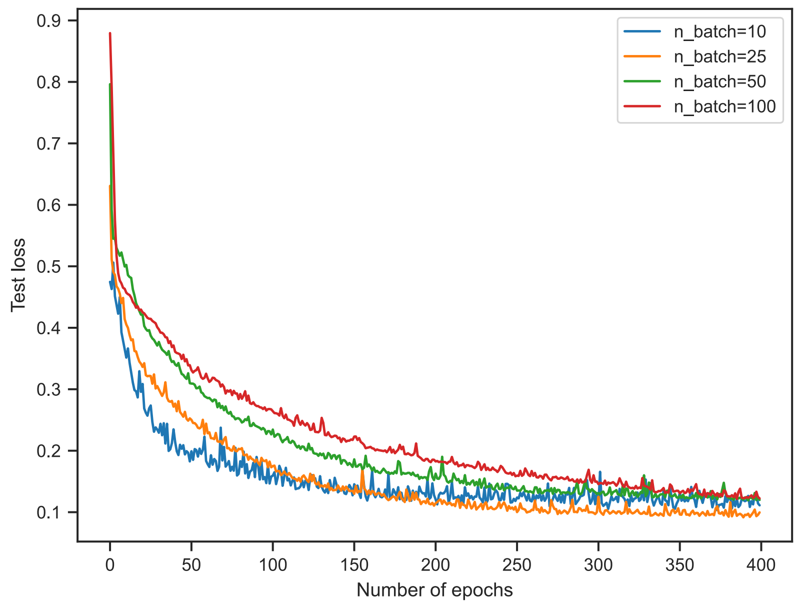
Answer:

In [ ]:
```
epochs = 400
lr = 0.01
reg_weight = 0
test_losses = []
models = []
batches = [10, 25, 50, 100]
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 [00:45<00:00,  8.78it/s]
Training n_batch: 25
100%|██████████| 400/400 [00:19<00:00, 20.71it/s]
Training n_batch: 50
100%|██████████| 400/400 [00:11<00:00, 35.54it/s]
Training n_batch: 100
100%|██████████| 400/400 [00:07<00:00, 56.76it/s]
```

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



Write your obsevsations about the batch size here

The larger the batch size the less change of test loss as the number of epochs increased. With a batch size of 10, the test loss reached 0.1 in 150 epochs, whereas a batch size of 100 only reached a test loss of 0.2 at 400 epochs. I will choose a batch size of 50 to move forward with my network. This batch size seems to get the benefits of reduced test loss.

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

Fix the batch size to the 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:

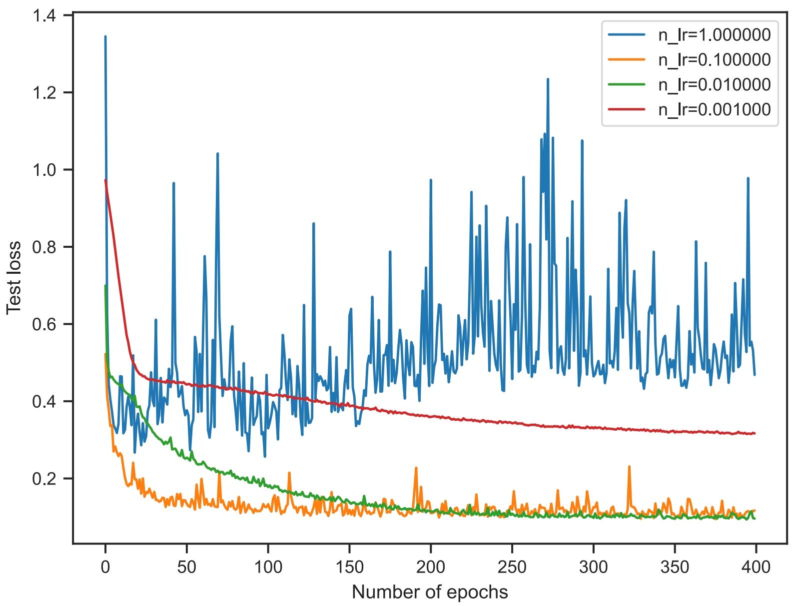
In [ ]:
```
# your code here
epochs = 400
lr = [1, 0.1, 0.01, 0.001]
reg_weight = 0
test_losses = []
models = []
batches = 50
for n_lr in lr:
    print('Training lr: {0:f}'.format(n_lr))
    net = nn.Sequential(nn.Linear(5, 20),
                    nn.Sigmoid(),
                    nn.Linear(20, 1))
    model, test_loss, X_train, y_train, X_test, y_test = train_net(
        X,
        y,
        net,
        reg_weight,
        batches,
        epochs,
        n_lr
    )
    test_losses.append(test_loss)
    models.append(model)
```

```
Training lr: 1.000000
100%|██████████| 400/400 [00:12<00:00, 32.08it/s]
Training lr: 0.100000
100%|██████████| 400/400 [00:11<00:00, 35.97it/s]
```

```
Training lr: 0.010000
100%|██████████| 400/400 [00:11<00:00, 34.93it/s]
Training lr: 0.001000
100%|██████████| 400/400 [00:11<00:00, 35.58it/s]
```

In [ ]:

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



Write your observsations about the learning rate here

The larger the learning rate the less change of test loss as the number of epochs increased. I will choose a learning rate of 0.01 to move forward with my network. This rate seems to be a less noisy value compared to the others, as it is a smooth curve and has the least amount of test loss.

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:

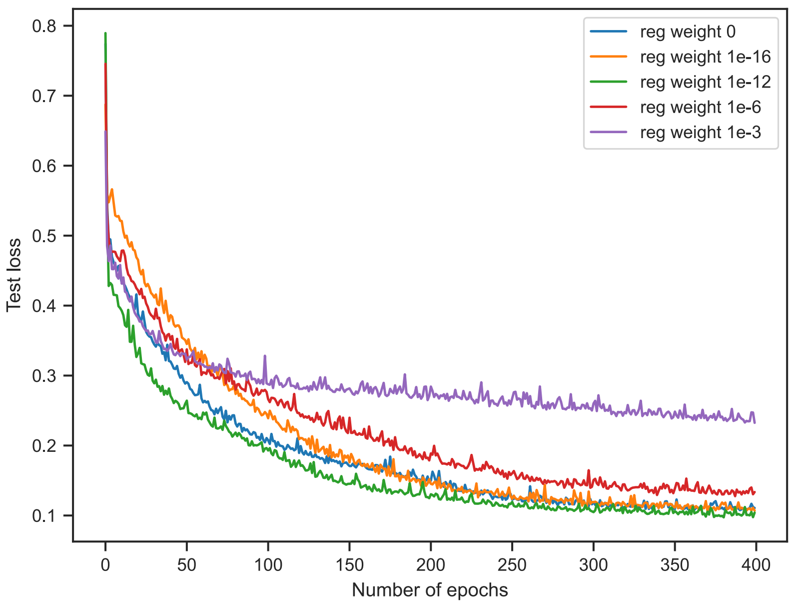
In [ ]:

```python
# Your code here
epochs = 400
lr = 0.01
reg_weight = [0, 1e-16, 1e-12, 1e-6, 1e-3]
test_losses = []
models = []
batches = 50
for n_reg_weight in reg_weight:
    print('Training lr: {0:f}'.format(n_reg_weight))
    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_weight,
        batches,
        epochs,
        lr
```

```
    )
    test_losses.append(test_loss)
    models.append(model)
```

```
Training lr: 0.000000
100%|██████████| 400/400 [00:12<00:00, 31.82it/s]
Training lr: 0.000000
100%|██████████| 400/400 [00:12<00:00, 32.51it/s]
Training lr: 0.000000
100%|██████████| 400/400 [00:11<00:00, 34.08it/s]
Training lr: 0.000001
100%|██████████| 400/400 [00:11<00:00, 33.36it/s]
Training lr: 0.001000
100%|██████████| 400/400 [00:12<00:00, 33.09it/s]
```

In [ ]:
```
fig, ax = plt.subplots(dpi=100)
labels = ['0', '1e-16', '1e-12', '1e-6', '1e-3']
i = 0
for tl, n_reg_weight in zip(test_losses, reg_weight):
    ax.plot(tl, label='reg weight '+str(labels[i]))
    i += 1
ax.set_xlabel('Number of epochs')
ax.set_ylabel('Test loss')
plt.legend(loc='best');
```



Write your observsations about the regularization weights here

The larger the regularization weight the less change of test loss as the number of epochs increased. I will choose a regularization weight of 1e-12 to move forward with my network. All of the weights, besides a weight of 0.001 seems to follow a similar curve. I am proceeding with this weight since it is in the middle of the other values.

Part D.I - Train a bigger network

You have developed some intuition about the parameters involved in training a network. Now, let's 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 as a function of the epochs
- the observations vs predictions plot for the test data

Answer:

In [ ]:
```
# your code here
# A simple one-layer network with 10 neurons
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, 100),
                    nn.Sigmoid(),
                    nn.Linear(100, 100),
                    nn.Sigmoid(),
                    nn.Linear(100, 1))
epochs = 1000
lr = 0.01
reg_weight = 1e-12
n_batch = 50
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 [01:49<00:00,  9.11it/s]
```

Part D.II - Make a prediction

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

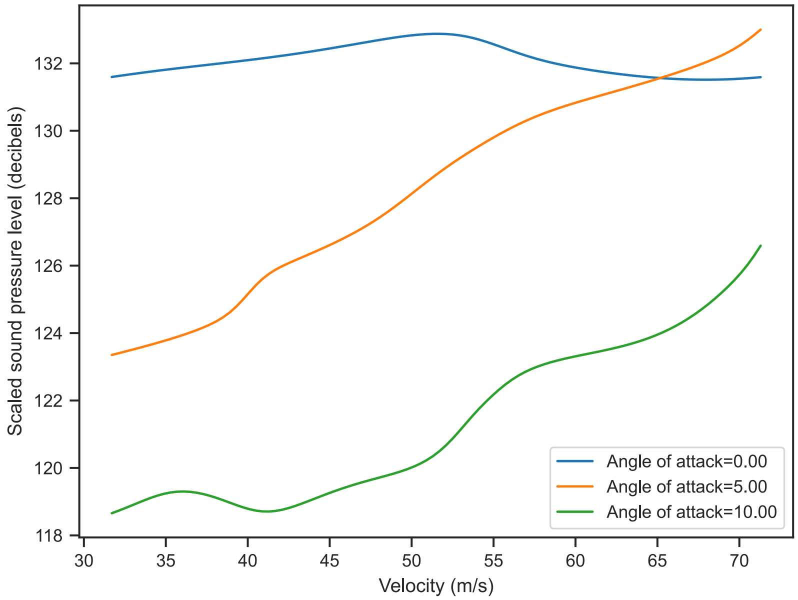
Answer:

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

In [ ]:
```
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('Velocity (m/s)')
    ax.set_ylabel('Scaled sound pressure level (decibels)')
```

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



Problem 2 - Classification with DNNs

Dr. Ali Lenjani kindly provided this homework problem. 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 inside 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 will 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 other 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 five different building locations. The original data are here, but I have repackaged them for you in a more convenient format. Let's download them:

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

Here is how to load the data:

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

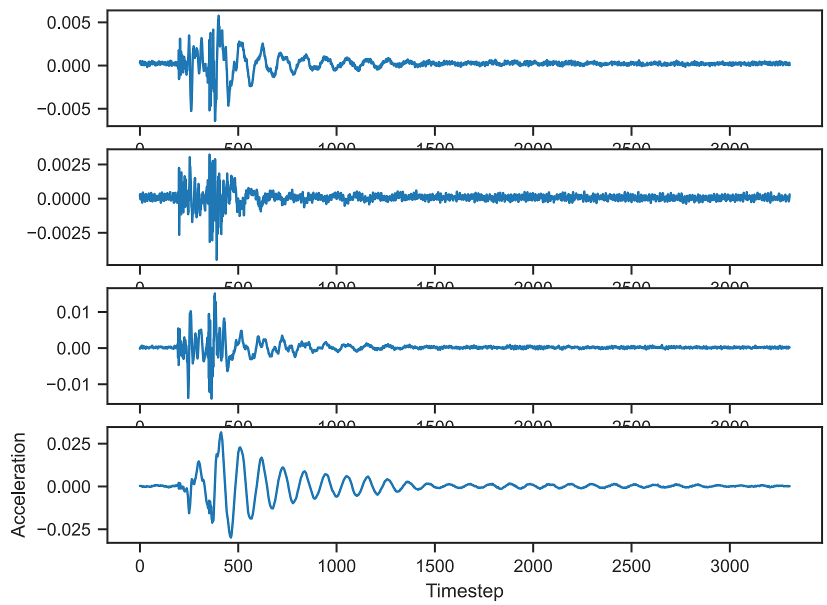
This is a Python dictionary that contains the following entries:

In [ ]:
```
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 [ ]:
```python
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');
```

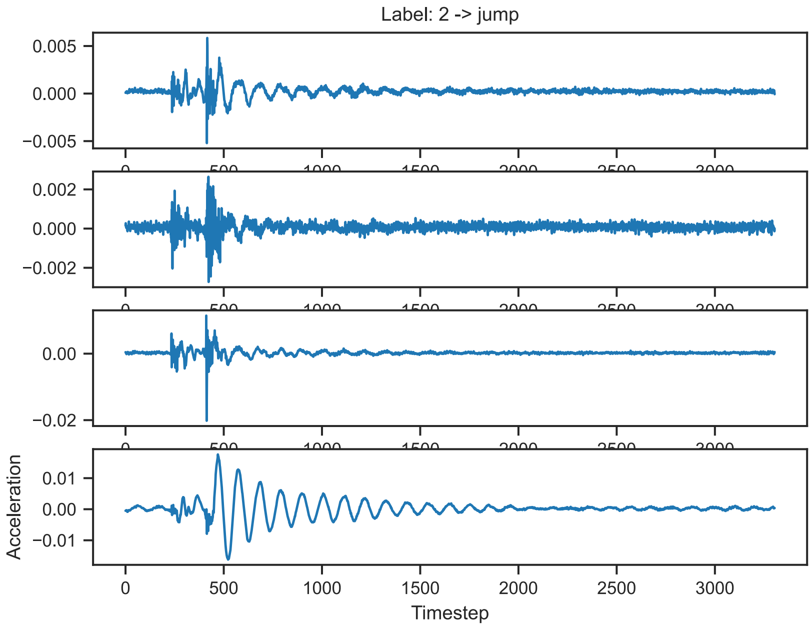


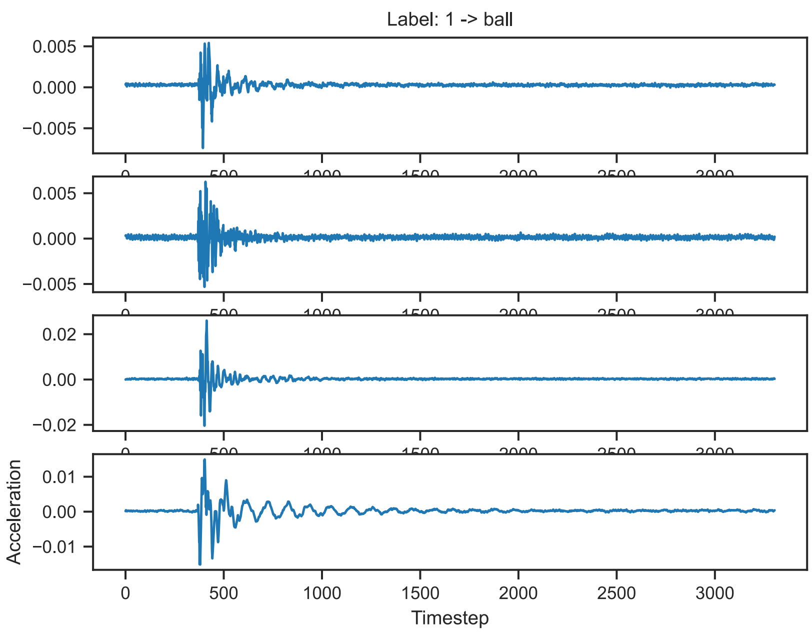
The second key, `labels_1`, is a bunch of integers ranging from 0 to 2 indicating whether the entry corresponds to a "bag," a "ball" or a "jump." For your reference, the correspondence is:

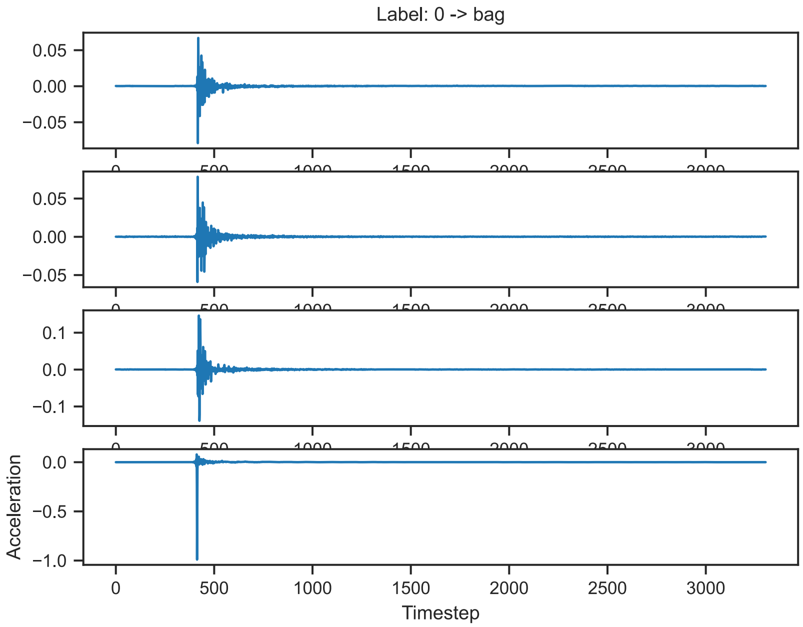
In [ ]:
```python
LABELS_1_TO_TEXT = {
    0: 'bag',
    1: 'ball',
    2: 'jump'
}
```

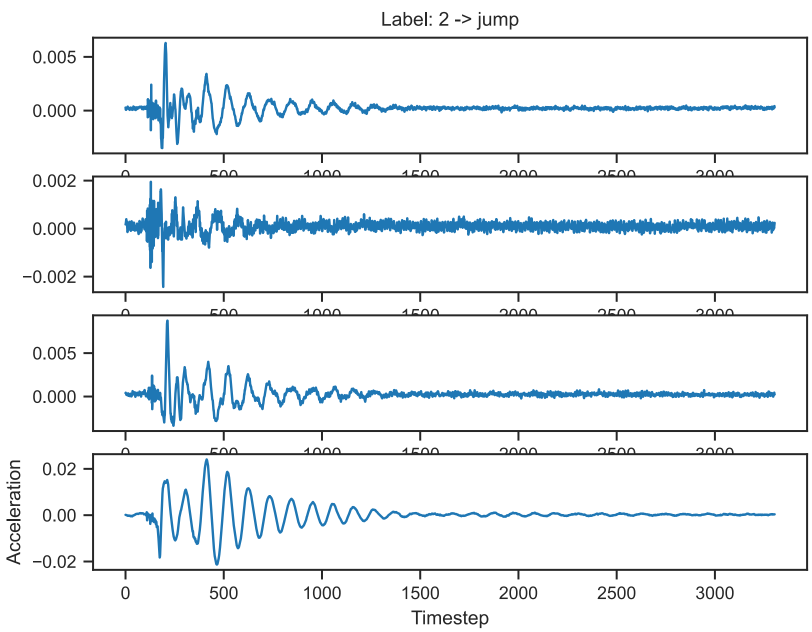
And here are a few examples:

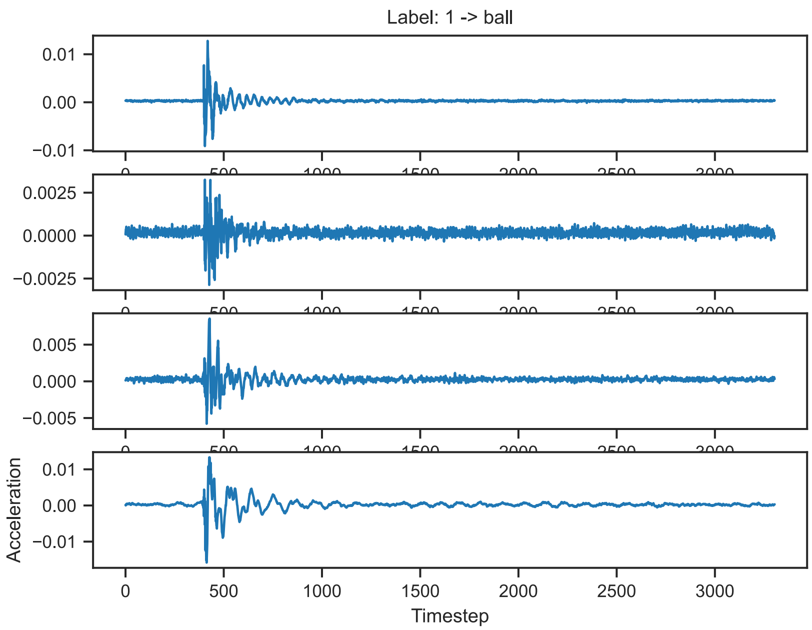
In [ ]:
```python
for _ in range(5):
    i = np.random.randint(0, data['features'].shape[0])
    fig, ax = plt.subplots(4, 1, dpi=100)
    for j in range(4):
        ax[j].plot(data['features'][i, j])
    ax[-1].set_xlabel('Timestep')
    ax[-1].set_ylabel('Acceleration')
    ax[0].set_title('Label: {0:d} -> {1:s}'.format(data['labels_1'][i],
                                                   LABELS_1_TO_TEXT[data['labels_1'][i]]))
```


Label: 2 -> jump
0.005
0.000
−0.005
0.002
0.000
−0.002
0.00
−0.02
0.01
0.00
−0.01
Acceleration
0
500
1000
1500
2000
2500
3000
Timestep


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


Label: 0 -> bag
Acceleration
Timestep
0
500
1000
1500
2000
2500
3000


Label: 2 -> jump
Acceleration
Timestep
0
500
1000
1500
2000
2500
3000



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 [ ]:
```
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 [ ]:
```
# 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 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 will take you from the four acceleration sensor data to the high-level type of each observation. You can keep the network structure 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. Let's 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 that the convolutional layers are invariant to small translations of the acceleration signal (just like the labels are). Here is what I propose:

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

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

In [ ]:
```
import torch.nn.functional as F

def cnn_loss_func(y_pred, y_batch):

    # Your code here
    y_batch = y_batch.long()

    cross_entropy_loss = F.cross_entropy(y_pred,y_batch)


    return cross_entropy_loss
```

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 [ ]:
```
def train_cnn(X, y, net, n_batch, epochs, lr, test_size=0.33):
    """
    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
            # your code here
            # Run the optimizer
            optimizer.zero_grad()
            y_pred = net(X_batch)
            loss = cnn_loss_func(y_pred, y_batch)
            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]))
    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.

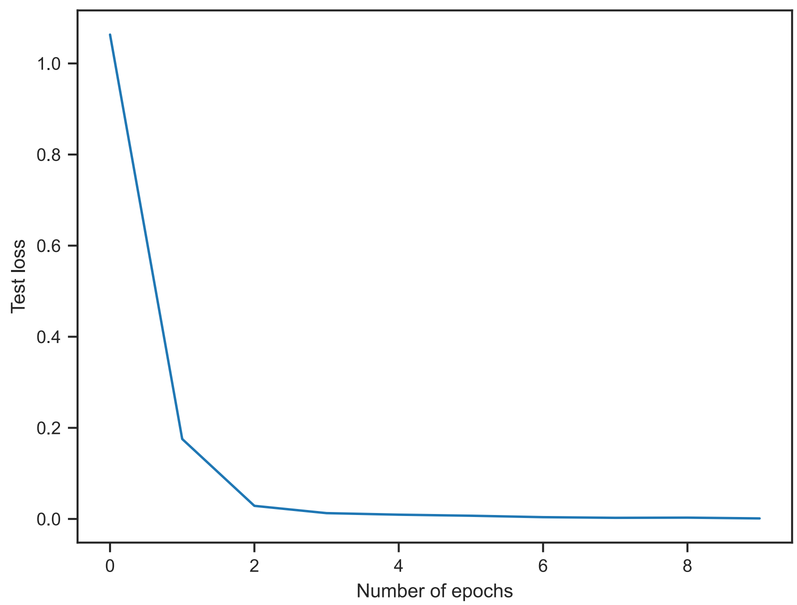
In [ ]:

```
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 = 0.41309%
Epoch 2: accuracy = 0.90895%
Epoch 3: accuracy = 0.99323%
Epoch 4: accuracy = 0.99549%
Epoch 5: accuracy = 0.99925%
Epoch 6: accuracy = 0.99925%
Epoch 7: accuracy = 1.00000%
Epoch 8: accuracy = 1.00000%
Epoch 9: accuracy = 1.00000%
Epoch 10: accuracy = 1.00000%
```

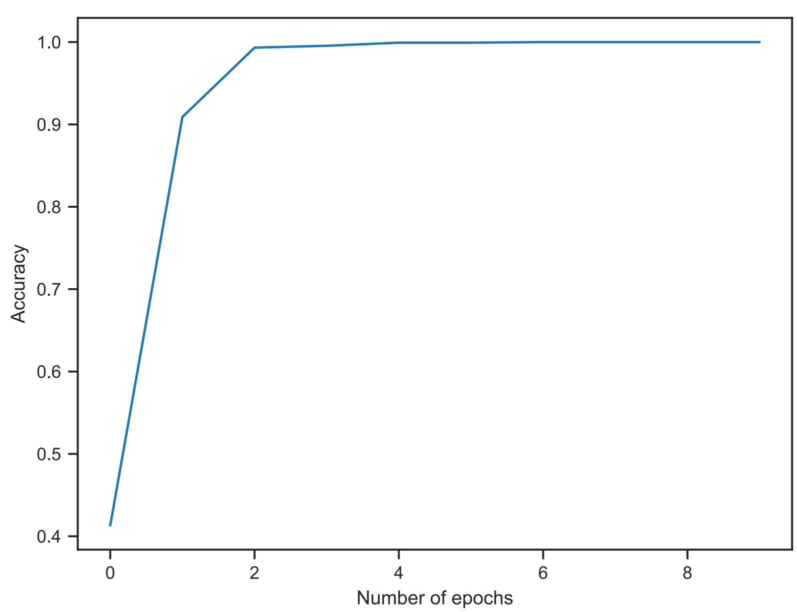
Plot the evolution of the test loss as a function of epochs.

In [ ]:
```
fig, ax = plt.subplots(dpi=100)
ax.plot(test_loss)
ax.set_xlabel('Number of epochs')
ax.set_ylabel('Test loss');
```



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

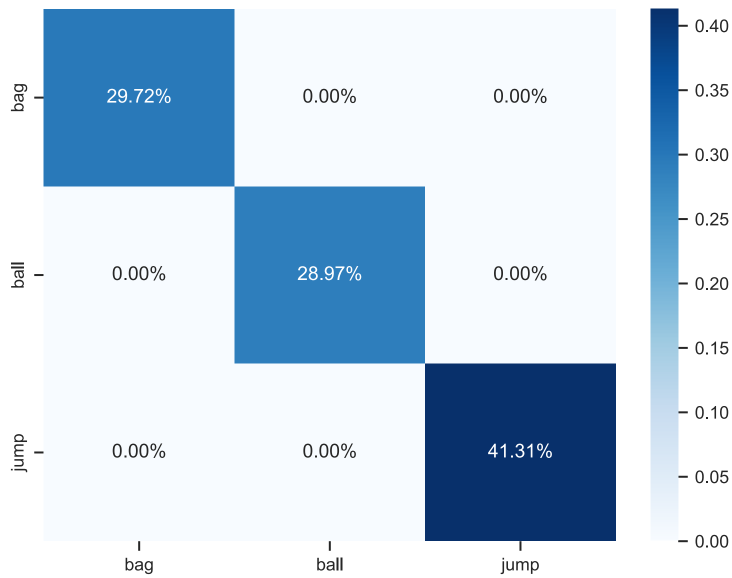
In [ ]:
```
fig, ax = plt.subplots(dpi=100)
ax.plot(accuracy)
ax.set_xlabel('Number of epochs')
ax.set_ylabel('Accuracy');
```



Plot the confusion matrix.

```
In [ ]: 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 [ ]: 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 [ ]: # your code here
        net_y2 = Net(7)

        epochs = 300
        lr = 0.001
        n_batch = 150
        trained_model, test_loss, accuracy, X_train, y_train, X_test, y_test = train_cnn(X, y2, net_y2, n_batch, epochs, lr)
```

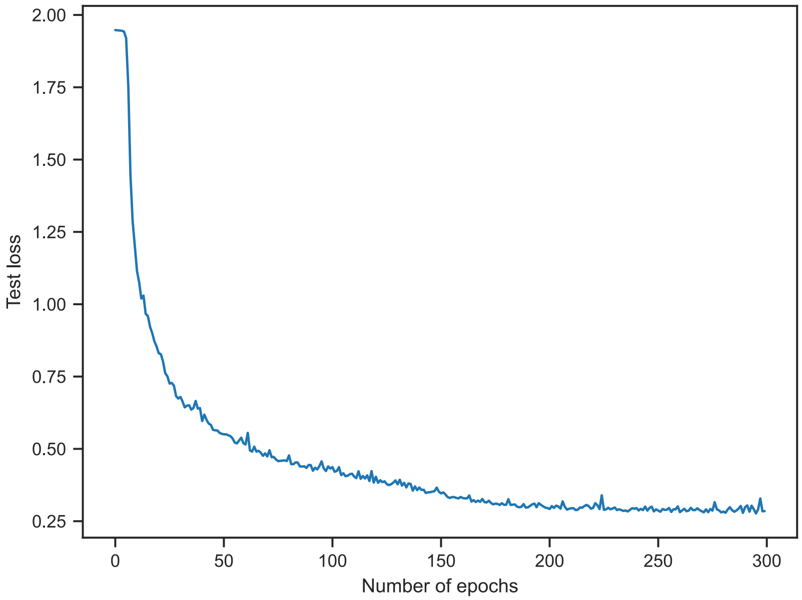
```
Epoch 1: accuracy = 0.13318%
Epoch 2: accuracy = 0.13318%
Epoch 3: accuracy = 0.13920%
Epoch 4: accuracy = 0.23627%
Epoch 5: accuracy = 0.25132%
Epoch 6: accuracy = 0.13920%
Epoch 7: accuracy = 0.27615%
Epoch 8: accuracy = 0.37547%
Epoch 9: accuracy = 0.31753%
Epoch 10: accuracy = 0.42438%
Epoch 11: accuracy = 0.49210%
Epoch 12: accuracy = 0.49586%
Epoch 13: accuracy = 0.55004%
Epoch 14: accuracy = 0.47780%
Epoch 15: accuracy = 0.54929%
Epoch 16: accuracy = 0.52746%
Epoch 17: accuracy = 0.61550%
Epoch 18: accuracy = 0.57487%
Epoch 19: accuracy = 0.56584%
Epoch 20: accuracy = 0.61023%
Epoch 21: accuracy = 0.58239%
Epoch 22: accuracy = 0.60271%
Epoch 23: accuracy = 0.66817%
Epoch 24: accuracy = 0.63807%
Epoch 25: accuracy = 0.67871%
Epoch 26: accuracy = 0.71257%
Epoch 27: accuracy = 0.70128%
Epoch 28: accuracy = 0.67043%
Epoch 29: accuracy = 0.72686%
Epoch 30: accuracy = 0.71859%
Epoch 31: accuracy = 0.67645%
Epoch 32: accuracy = 0.70805%
Epoch 33: accuracy = 0.72912%
Epoch 34: accuracy = 0.70805%
Epoch 35: accuracy = 0.72611%
Epoch 36: accuracy = 0.72084%
Epoch 37: accuracy = 0.69375%
Epoch 38: accuracy = 0.74266%
Epoch 39: accuracy = 0.75169%
Epoch 40: accuracy = 0.71106%
Epoch 41: accuracy = 0.74417%
Epoch 42: accuracy = 0.72160%
Epoch 43: accuracy = 0.75094%
Epoch 44: accuracy = 0.75696%
Epoch 45: accuracy = 0.76749%
Epoch 46: accuracy = 0.76223%
Epoch 47: accuracy = 0.75997%
Epoch 48: accuracy = 0.77276%
Epoch 49: accuracy = 0.75771%
Epoch 50: accuracy = 0.76674%
Epoch 51: accuracy = 0.76524%
Epoch 52: accuracy = 0.77351%
Epoch 53: accuracy = 0.75621%
Epoch 54: accuracy = 0.76674%
Epoch 55: accuracy = 0.75847%
Epoch 56: accuracy = 0.76975%
Epoch 57: accuracy = 0.77126%
Epoch 58: accuracy = 0.77577%
Epoch 59: accuracy = 0.75320%
Epoch 60: accuracy = 0.77351%
Epoch 61: accuracy = 0.78104%
Epoch 62: accuracy = 0.75245%
Epoch 63: accuracy = 0.78405%
Epoch 64: accuracy = 0.77728%
Epoch 65: accuracy = 0.76900%
Epoch 66: accuracy = 0.79308%
Epoch 67: accuracy = 0.79157%
Epoch 68: accuracy = 0.78104%
Epoch 69: accuracy = 0.79082%
Epoch 70: accuracy = 0.78555%
Epoch 71: accuracy = 0.78932%
Epoch 72: accuracy = 0.77728%
Epoch 73: accuracy = 0.79910%
Epoch 74: accuracy = 0.79910%
Epoch 75: accuracy = 0.79684%
```

```
Epoch 76: accuracy = 0.79834%
Epoch 77: accuracy = 0.78932%
Epoch 78: accuracy = 0.80211%
Epoch 79: accuracy = 0.77728%
Epoch 80: accuracy = 0.79458%
Epoch 81: accuracy = 0.79458%
Epoch 82: accuracy = 0.79759%
Epoch 83: accuracy = 0.80135%
Epoch 84: accuracy = 0.80286%
Epoch 85: accuracy = 0.80361%
Epoch 86: accuracy = 0.80135%
Epoch 87: accuracy = 0.80813%
Epoch 88: accuracy = 0.80060%
Epoch 89: accuracy = 0.80813%
Epoch 90: accuracy = 0.80963%
Epoch 91: accuracy = 0.80060%
Epoch 92: accuracy = 0.80587%
Epoch 93: accuracy = 0.80060%
Epoch 94: accuracy = 0.80662%
Epoch 95: accuracy = 0.80662%
Epoch 96: accuracy = 0.77351%
Epoch 97: accuracy = 0.80662%
Epoch 98: accuracy = 0.80963%
Epoch 99: accuracy = 0.78330%
Epoch 100: accuracy = 0.80587%
Epoch 101: accuracy = 0.81791%
Epoch 102: accuracy = 0.81415%
Epoch 103: accuracy = 0.81189%
Epoch 104: accuracy = 0.80060%
Epoch 105: accuracy = 0.81189%
Epoch 106: accuracy = 0.80211%
Epoch 107: accuracy = 0.82017%
Epoch 108: accuracy = 0.81114%
Epoch 109: accuracy = 0.81640%
Epoch 110: accuracy = 0.82167%
Epoch 111: accuracy = 0.81791%
Epoch 112: accuracy = 0.82167%
Epoch 113: accuracy = 0.81415%
Epoch 114: accuracy = 0.81640%
Epoch 115: accuracy = 0.81716%
Epoch 116: accuracy = 0.81716%
Epoch 117: accuracy = 0.80587%
Epoch 118: accuracy = 0.82468%
Epoch 119: accuracy = 0.82242%
Epoch 120: accuracy = 0.82092%
Epoch 121: accuracy = 0.80060%
Epoch 122: accuracy = 0.82468%
Epoch 123: accuracy = 0.81866%
Epoch 124: accuracy = 0.82769%
Epoch 125: accuracy = 0.82242%
Epoch 126: accuracy = 0.82468%
Epoch 127: accuracy = 0.82318%
Epoch 128: accuracy = 0.82769%
Epoch 129: accuracy = 0.82017%
Epoch 130: accuracy = 0.82995%
Epoch 131: accuracy = 0.82995%
Epoch 132: accuracy = 0.82694%
Epoch 133: accuracy = 0.82468%
Epoch 134: accuracy = 0.81866%
Epoch 135: accuracy = 0.82769%
Epoch 136: accuracy = 0.82619%
Epoch 137: accuracy = 0.82318%
Epoch 138: accuracy = 0.83220%
Epoch 139: accuracy = 0.83446%
Epoch 140: accuracy = 0.82543%
Epoch 141: accuracy = 0.83145%
Epoch 142: accuracy = 0.83898%
Epoch 143: accuracy = 0.83672%
Epoch 144: accuracy = 0.84048%
Epoch 145: accuracy = 0.84048%
Epoch 146: accuracy = 0.84199%
Epoch 147: accuracy = 0.84274%
Epoch 148: accuracy = 0.83898%
Epoch 149: accuracy = 0.83521%
Epoch 150: accuracy = 0.83822%
```

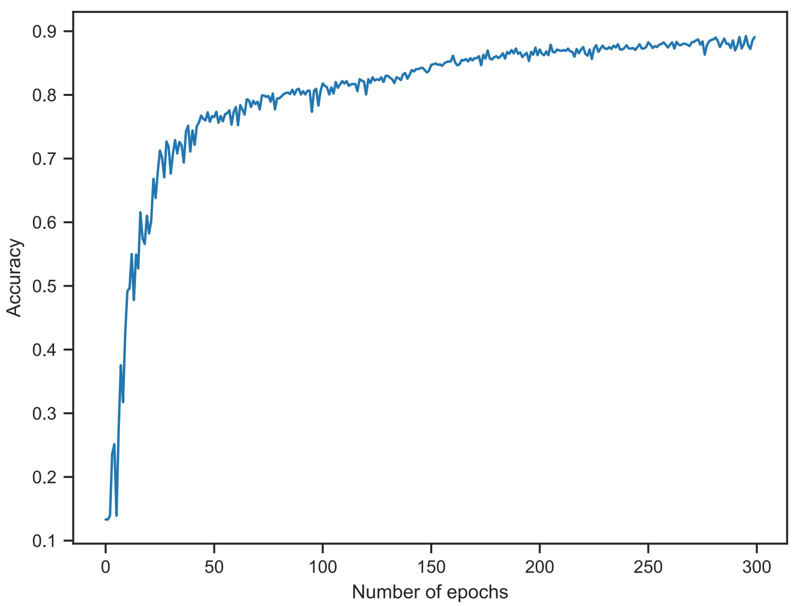
```
Epoch 151: accuracy = 0.84725%
Epoch 152: accuracy = 0.84801%
Epoch 153: accuracy = 0.84951%
Epoch 154: accuracy = 0.84725%
Epoch 155: accuracy = 0.84801%
Epoch 156: accuracy = 0.84575%
Epoch 157: accuracy = 0.84951%
Epoch 158: accuracy = 0.85177%
Epoch 159: accuracy = 0.85252%
Epoch 160: accuracy = 0.85252%
Epoch 161: accuracy = 0.86155%
Epoch 162: accuracy = 0.85102%
Epoch 163: accuracy = 0.84650%
Epoch 164: accuracy = 0.84801%
Epoch 165: accuracy = 0.85478%
Epoch 166: accuracy = 0.85403%
Epoch 167: accuracy = 0.85628%
Epoch 168: accuracy = 0.85252%
Epoch 169: accuracy = 0.85779%
Epoch 170: accuracy = 0.85403%
Epoch 171: accuracy = 0.85779%
Epoch 172: accuracy = 0.85779%
Epoch 173: accuracy = 0.86080%
Epoch 174: accuracy = 0.84650%
Epoch 175: accuracy = 0.86305%
Epoch 176: accuracy = 0.85704%
Epoch 177: accuracy = 0.86983%
Epoch 178: accuracy = 0.85704%
Epoch 179: accuracy = 0.85553%
Epoch 180: accuracy = 0.85929%
Epoch 181: accuracy = 0.86080%
Epoch 182: accuracy = 0.85779%
Epoch 183: accuracy = 0.86080%
Epoch 184: accuracy = 0.86531%
Epoch 185: accuracy = 0.85704%
Epoch 186: accuracy = 0.86757%
Epoch 187: accuracy = 0.86456%
Epoch 188: accuracy = 0.87058%
Epoch 189: accuracy = 0.86531%
Epoch 190: accuracy = 0.87284%
Epoch 191: accuracy = 0.86456%
Epoch 192: accuracy = 0.86682%
Epoch 193: accuracy = 0.85929%
Epoch 194: accuracy = 0.86305%
Epoch 195: accuracy = 0.86606%
Epoch 196: accuracy = 0.85327%
Epoch 197: accuracy = 0.86757%
Epoch 198: accuracy = 0.86305%
Epoch 199: accuracy = 0.87434%
Epoch 200: accuracy = 0.86230%
Epoch 201: accuracy = 0.87133%
Epoch 202: accuracy = 0.86456%
Epoch 203: accuracy = 0.86230%
Epoch 204: accuracy = 0.86832%
Epoch 205: accuracy = 0.86230%
Epoch 206: accuracy = 0.87886%
Epoch 207: accuracy = 0.86757%
Epoch 208: accuracy = 0.86682%
Epoch 209: accuracy = 0.87133%
Epoch 210: accuracy = 0.86983%
Epoch 211: accuracy = 0.86907%
Epoch 212: accuracy = 0.87058%
Epoch 213: accuracy = 0.86907%
Epoch 214: accuracy = 0.87284%
Epoch 215: accuracy = 0.86832%
Epoch 216: accuracy = 0.86757%
Epoch 217: accuracy = 0.86005%
Epoch 218: accuracy = 0.87208%
Epoch 219: accuracy = 0.86531%
Epoch 220: accuracy = 0.87133%
Epoch 221: accuracy = 0.87509%
Epoch 222: accuracy = 0.86456%
Epoch 223: accuracy = 0.86155%
Epoch 224: accuracy = 0.87133%
Epoch 225: accuracy = 0.85628%
```

```
Epoch 226: accuracy = 0.87284%
Epoch 227: accuracy = 0.87810%
Epoch 228: accuracy = 0.86757%
Epoch 229: accuracy = 0.87284%
Epoch 230: accuracy = 0.87735%
Epoch 231: accuracy = 0.87284%
Epoch 232: accuracy = 0.87208%
Epoch 233: accuracy = 0.87509%
Epoch 234: accuracy = 0.87208%
Epoch 235: accuracy = 0.87735%
Epoch 236: accuracy = 0.87434%
Epoch 237: accuracy = 0.87961%
Epoch 238: accuracy = 0.87133%
Epoch 239: accuracy = 0.87133%
Epoch 240: accuracy = 0.87359%
Epoch 241: accuracy = 0.87810%
Epoch 242: accuracy = 0.87284%
Epoch 243: accuracy = 0.87284%
Epoch 244: accuracy = 0.87359%
Epoch 245: accuracy = 0.87058%
Epoch 246: accuracy = 0.87509%
Epoch 247: accuracy = 0.87961%
Epoch 248: accuracy = 0.87284%
Epoch 249: accuracy = 0.87284%
Epoch 250: accuracy = 0.87509%
Epoch 251: accuracy = 0.88262%
Epoch 252: accuracy = 0.87886%
Epoch 253: accuracy = 0.87359%
Epoch 254: accuracy = 0.87660%
Epoch 255: accuracy = 0.87509%
Epoch 256: accuracy = 0.87886%
Epoch 257: accuracy = 0.87961%
Epoch 258: accuracy = 0.88262%
Epoch 259: accuracy = 0.87886%
Epoch 260: accuracy = 0.87434%
Epoch 261: accuracy = 0.87886%
Epoch 262: accuracy = 0.88262%
Epoch 263: accuracy = 0.87284%
Epoch 264: accuracy = 0.88337%
Epoch 265: accuracy = 0.87886%
Epoch 266: accuracy = 0.87810%
Epoch 267: accuracy = 0.88036%
Epoch 268: accuracy = 0.88036%
Epoch 269: accuracy = 0.87886%
Epoch 270: accuracy = 0.87660%
Epoch 271: accuracy = 0.88262%
Epoch 272: accuracy = 0.88337%
Epoch 273: accuracy = 0.88563%
Epoch 274: accuracy = 0.88713%
Epoch 275: accuracy = 0.87886%
Epoch 276: accuracy = 0.88412%
Epoch 277: accuracy = 0.86305%
Epoch 278: accuracy = 0.87810%
Epoch 279: accuracy = 0.88412%
Epoch 280: accuracy = 0.88638%
Epoch 281: accuracy = 0.88713%
Epoch 282: accuracy = 0.89014%
Epoch 283: accuracy = 0.88488%
Epoch 284: accuracy = 0.87509%
Epoch 285: accuracy = 0.88187%
Epoch 286: accuracy = 0.88864%
Epoch 287: accuracy = 0.88036%
Epoch 288: accuracy = 0.88036%
Epoch 289: accuracy = 0.87359%
Epoch 290: accuracy = 0.88563%
Epoch 291: accuracy = 0.86983%
Epoch 292: accuracy = 0.87735%
Epoch 293: accuracy = 0.89090%
Epoch 294: accuracy = 0.87284%
Epoch 295: accuracy = 0.87961%
Epoch 296: accuracy = 0.89240%
Epoch 297: accuracy = 0.87810%
Epoch 298: accuracy = 0.87208%
Epoch 299: accuracy = 0.88563%
Epoch 300: accuracy = 0.89090%
```

In [ ]:
```
fig, ax = plt.subplots(dpi=100)
ax.plot(test_loss)
ax.set_xlabel('Number of epochs')
ax.set_ylabel('Test loss');
```

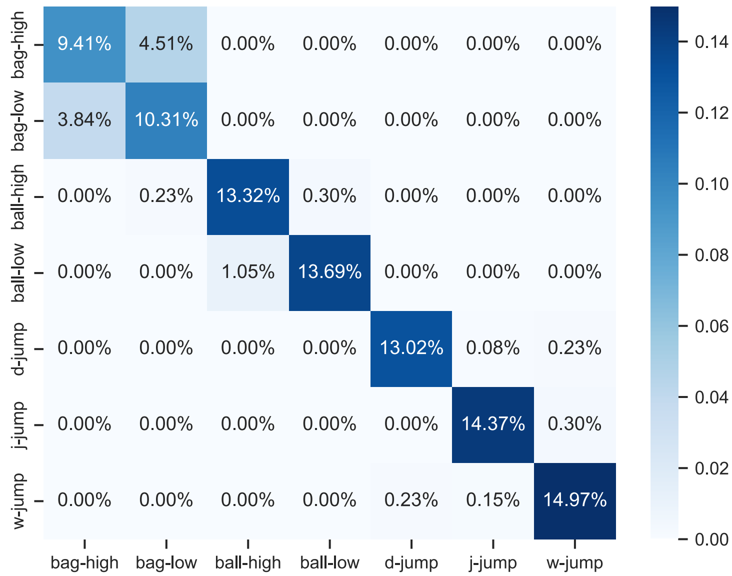


In [ ]:
```
fig, ax = plt.subplots(dpi=100)
ax.plot(accuracy)
ax.set_xlabel('Number of epochs')
ax.set_ylabel('Accuracy');
```



In [ ]:
```
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 [ ]:
```
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
9.41% 4.51% 0.00% 0.00% 0.00% 0.00% 0.00%
3.84% 10.31% 0.00% 0.00% 0.00% 0.00% 0.00%
0.00% 0.23% 13.32% 0.30% 0.00% 0.00% 0.00%
0.00% 0.00% 1.05% 13.69% 0.00% 0.00% 0.00%
0.00% 0.00% 0.00% 0.00% 13.02% 0.08% 0.23%
0.00% 0.00% 0.00% 0.00% 0.00% 14.37% 0.30%
0.00% 0.00% 0.00% 0.00% 0.23% 0.15% 14.97%
0.14
0.12
0.10
0.08
0.06
0.04
0.02
0.00