homework-07a

December 11, 2023

1 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.

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
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)
```

1.3 Student details

- **First Name:** Robert
- **Last Name:** Chandler
- **Email:** chandl71@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.

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

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--
0100 59984    0 59984    0     0   133k      0 --:--:-- --:--:-- --:--:--  133k
```

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

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

[ ]:
```
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:

[ ]:
```python
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  Suction_thickness  \
0         800.0              0.0        0.3048      71.3           0.002663
1        1000.0              0.0        0.3048      71.3           0.002663
2        1250.0              0.0        0.3048      71.3           0.002663
3        1600.0              0.0        0.3048      71.3           0.002663
4        2000.0              0.0        0.3048      71.3           0.002663
…           …                …            …         …                 …
1498     2500.0             15.6        0.1016      39.6           0.052849
1499     3150.0             15.6        0.1016      39.6           0.052849
1500     4000.0             15.6        0.1016      39.6           0.052849
1501     5000.0             15.6        0.1016      39.6           0.052849
1502     6300.0             15.6        0.1016      39.6           0.052849

      Sound_pressure
0            126.201
1            125.201
2            125.951
3            127.591
4            127.461
…               …
1498         110.264
1499         109.254
```

```
1500          106.604
1501          106.224
1502          104.204

[1503 rows x 6 columns]
```

1.3.1 Part A - Analyze the data visually

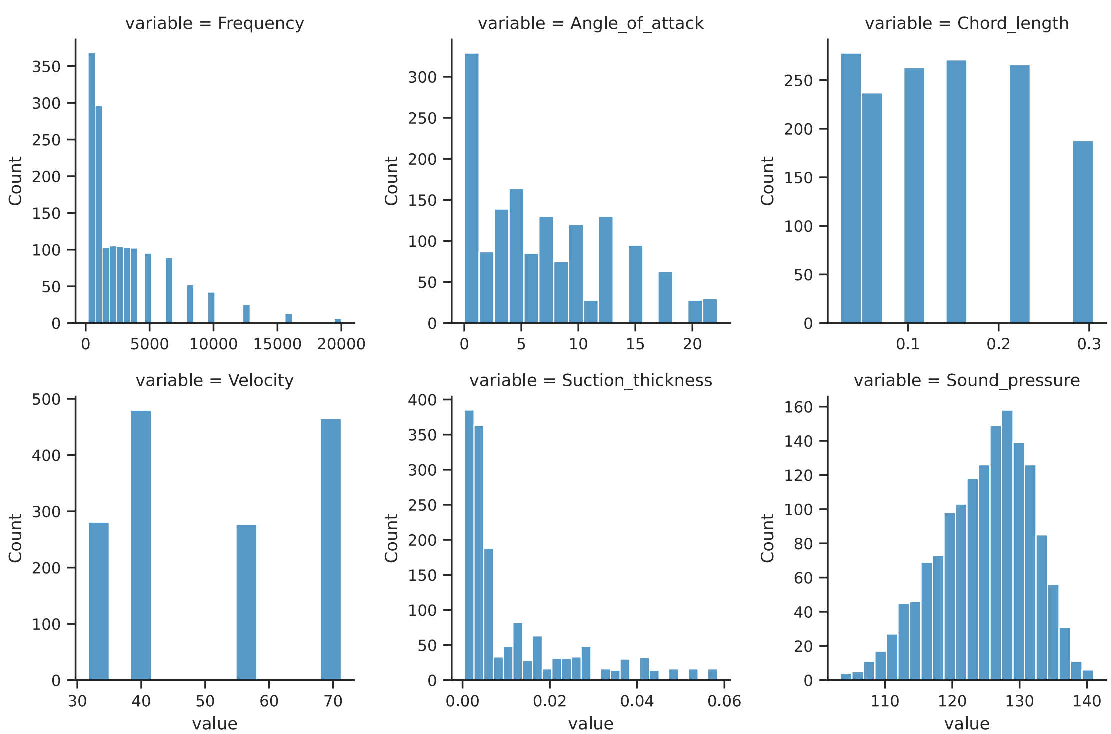
It is always a good idea to visualize the data before you start doing anything with them.

Part A.I. - Do the 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:

[ ]:
```python
grid = sns.FacetGrid(df.melt(), col="variable", col_wrap=3, sharex=False,␣
 ↪sharey=False)
grid.map(sns.histplot, "value");
```

```
/nix/store/m8m18kk2yg02wjngr2nlvycr8nxcrk32-python3-3.10.12-
env/lib/python3.10/site-packages/seaborn/axisgrid.py:118: UserWarning: The
figure layout has changed to tight
  self._figure.tight_layout(*args, **kwargs)
```

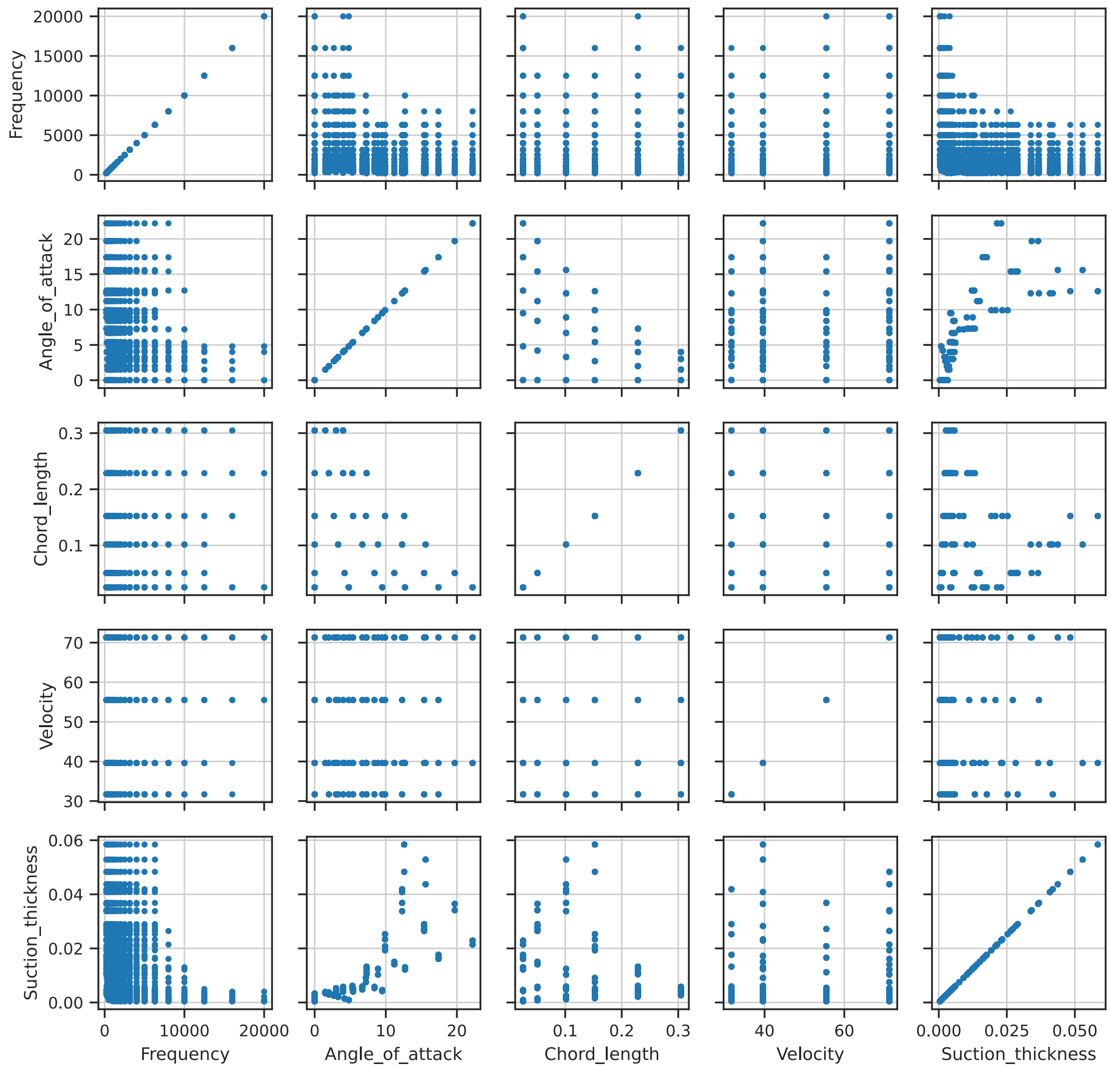


Given the wide range of scales present in each of the variables, it is evident that we need to standardize our data before moving onto regression.

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:

[ ]:
```
n_targets = 1
n_features = df.shape[1] - n_targets
fig, ax = plt.subplots(n_features, n_features)
fig.set_size_inches(10,10)
for i in range(n_features):
    for j in range(n_features):
        ax[i, j].plot(df.iloc[:, j], df.iloc[:, i], '.')
        ax[i, j].grid()

        if j == 0:
            ax[i, j].set_ylabel(df.columns[i])
        else:
            ax[i, j].yaxis.set_ticklabels([])

        if i == n_features - 1:
            ax[i, j].set_xlabel(df.columns[j])
        else:
            ax[i, j].xaxis.set_ticklabels([])
```



Most of the data has some kind of "stepping" going on where we tend to not have data in certain ranges, but some are worse than others. For example, Suction Thickness covers its supported range quite well, but velocity has extreme gaps in between each cluster of data points. In fact, velocity only exists in 4 discrete values, shown in the cell below:

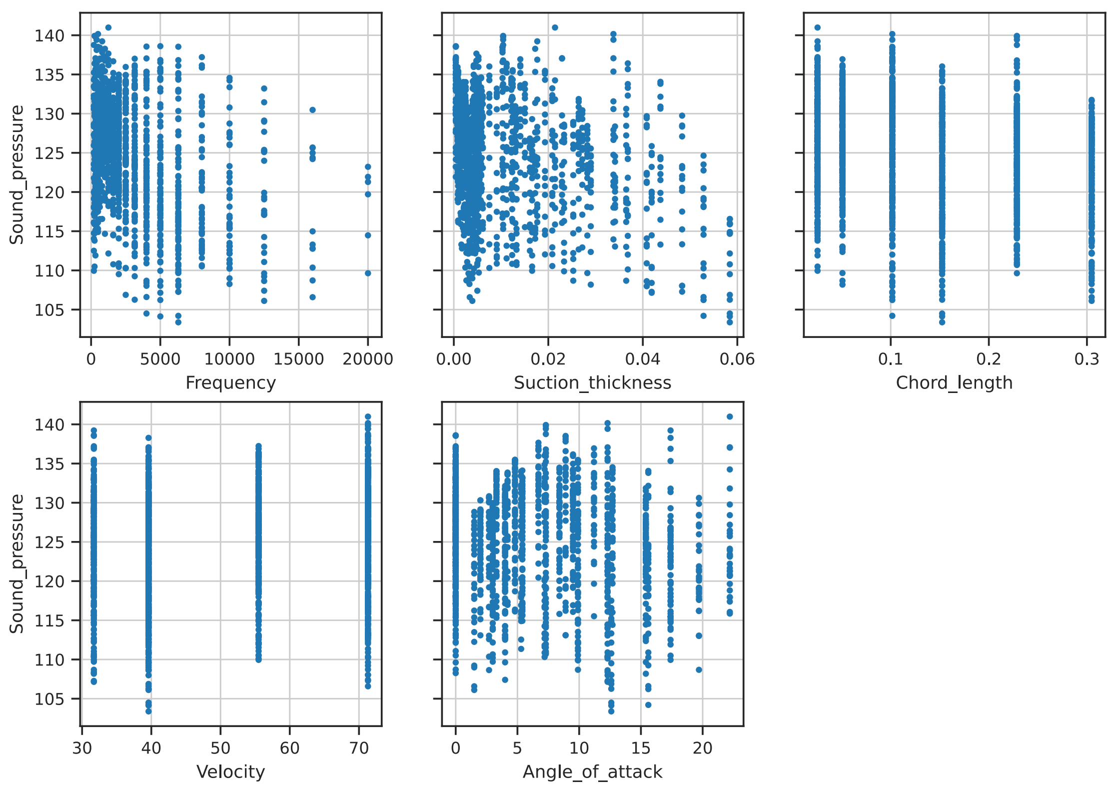
[ ]:
```
df["Velocity"].value_counts()
```

[ ]:
```
Velocity
39.6    480
71.3    465
31.7    281
55.5    277
Name: count, dtype: int64
```

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:

[ ]:
```
n_targets = 1
n_features = df.shape[1] - n_targets
fig, ax = plt.subplots(2, 3, sharey=True)
fig.set_size_inches(10,7)
for i in range(n_features):
    row = i % 2
    col = i % 3
    ax[row, col].plot(df.iloc[:, i], df.iloc[:, n_features], '.')
    ax[row, col].grid()

    ax[row, col].set_xlabel(df.columns[i])

    if col == 0:
        ax[row, col].set_ylabel(df.columns[n_features])

ax[-1, -1].axis("off");
```



There is no obvious pattern that we can discern between any single input and the output... the correlation values for these are probably all very weak

1.3.2 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 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:

[ ]:
```
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: 1.25
What you should be getting: 1.25
```

[ ]:
```
# 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(0.0)
    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.
    """
    return mse_loss(y_obs, y_pred) + reg_weight * l2_reg_loss(params)
```

[ ]:
```
# 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: 1.25
```

```
This should be the same as this: 1.25
The loss with regularization: 1.32
```

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:

```
[ ]: # 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:
        if self.feature_scaler is not None and self.target_scaler is not None:
            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
        else:
            raise TypeError("If standardized=True, then feature_scaler and␣
 ↪target_scaler must not be None")
```

[ ]:
```
# 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 fraction 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=test_size)

    # 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`
    optimizer = torch.optim.Adam(params=net.parameters(), lr=lr)
```

```python
    # 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 i, (X_batch, y_batch) in enumerate(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
            # Your code here
            optimizer.zero_grad()

            y_pred = net(X_batch)

            loss = loss_func(y_batch, y_pred, reg_weight=reg_weight, params=net.
↪parameters())

            loss.backward()

            optimizer.step()

            # test_loss.append(loss.item())

            # if i % 1000 == 0:
            #     print(f"epoch {e}, i {i}: loss = {loss.item():1.2e}")


        # 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
```

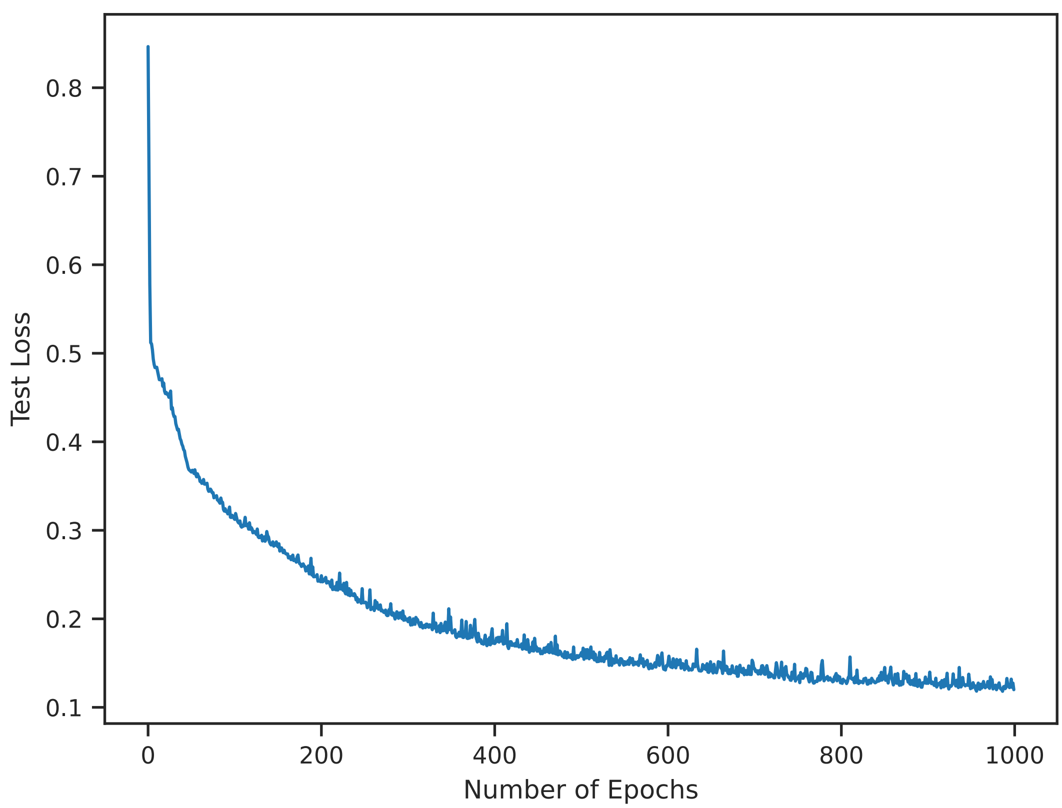
Use this to test your code:

[ ]:
```
# 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 [00:14<00:00, 68.32it/s]
```

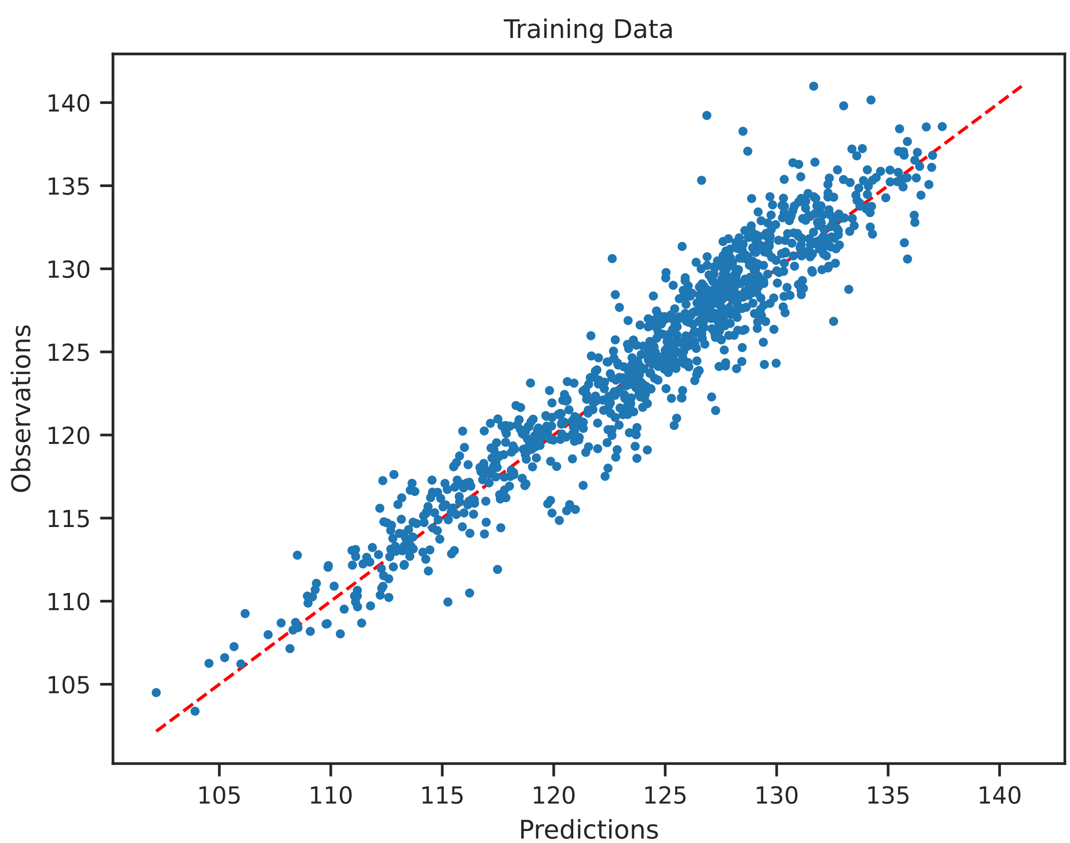
There are a few more things for you to do here. First, plot the evolution of the test loss as a function of the number of epochs:

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

ax.plot(test_loss)
ax.set_xlabel("Number of Epochs")
ax.set_ylabel("Test Loss");
```



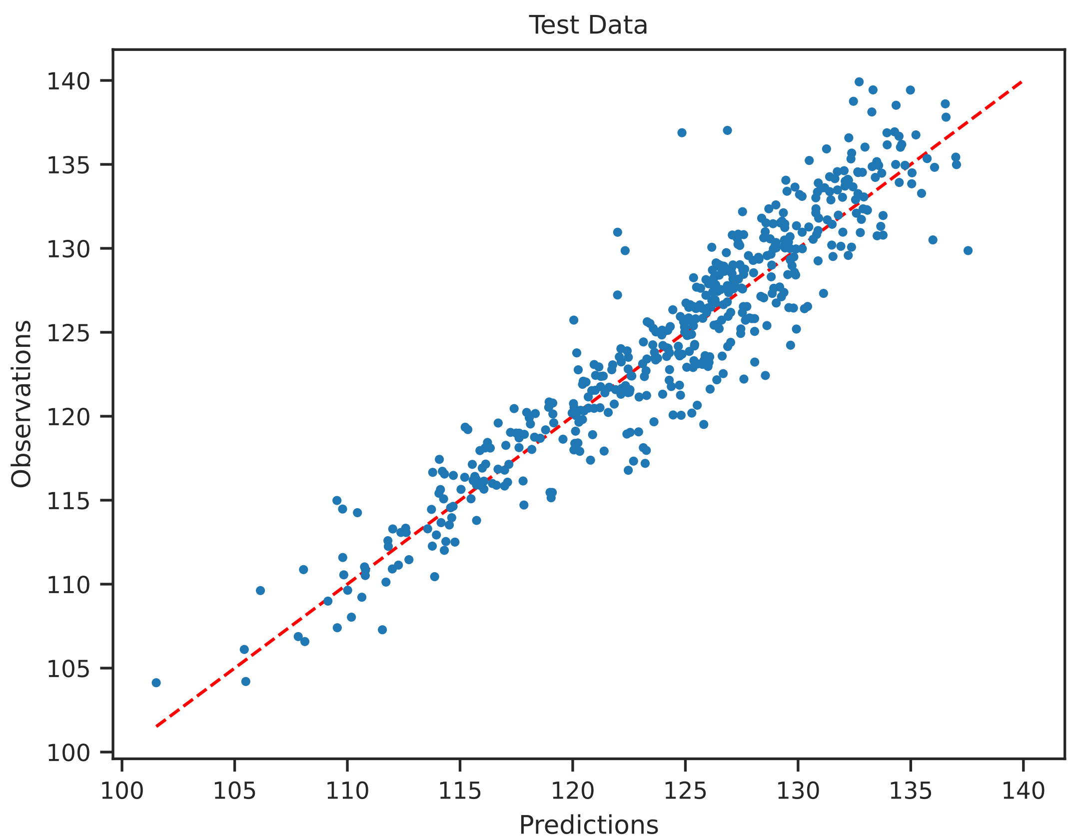
Now plot the observations vs predictions plot for the training data:

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

X_train_tensor = torch.Tensor(X_train)
y_pred_train = model(X_train_tensor)

# Plot the line of perfect fit
y_min = np.min(np.concatenate([y_pred_train, y_train]))
y_max = np.max(np.concatenate([y_pred_train, y_train]))
ax.plot([y_min, y_max], [y_min, y_max], 'r--')

ax.plot(y_pred_train, y_train, '.')

ax.set_title("Training Data")
ax.set_xlabel("Predictions")
ax.set_ylabel("Observations");
```



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

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

X_test_tensor = torch.Tensor(X_test)
y_pred_test = model(X_test_tensor)

# Plot the line of perfect fit
y_min = np.min(np.concatenate([y_pred_test, y_test]))
y_max = np.max(np.concatenate([y_pred_test, y_test]))
ax.plot([y_min, y_max], [y_min, y_max], 'r--')

ax.plot(y_pred_test, y_test, '.')

ax.set_title("Test Data")
ax.set_xlabel("Predictions")
ax.set_ylabel("Observations");
```



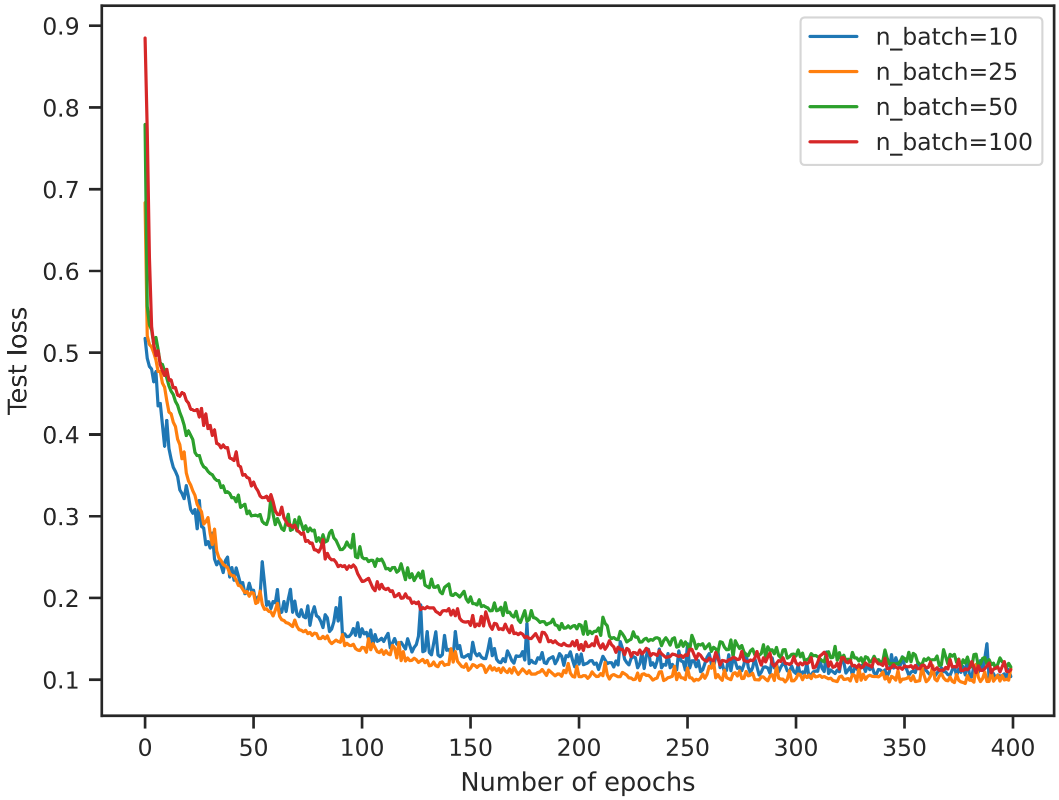
Part C.I - Investigate the effect of the batch size For the given network, try batch sizes of 10, 25, 50, and 100 for 400 epochs. In the sample plot, show the evolution of the test loss function for each case. Which batch sizes lead to faster training times and why? Which one would you choose?

Answer:

[ ]:
```
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 [00:38<00:00, 10.43it/s]

Training n_batch: 25

100%|      | 400/400 [00:16<00:00, 24.85it/s]

Training n_batch: 50

100%|      | 400/400 [00:08<00:00, 45.07it/s]

Training n_batch: 100

100%|      | 400/400 [00:06<00:00, 66.22it/s]
```

[ ]:
```
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");
```



The larger batch sizes take less time to train. This is probably due to a combination of factors. For one, PyTorch is designed to take advantage of vector operations and is good at parallelizing large vector operations (GPUs also perform best this way). When we choose a small batch size, though, we break up all of these large operations into a large number of smaller operations in series. This is slower to train. However, we observe that the model tends to converge faster and to a lower test loss value when the batch size is smaller. We assume that this translates to a more accurate model.

I would choose the batch size of 50 it is a good compromise between speed and accuracy, since it completed in less than a third of the time that 10 did and reached a minimum loss that was basically the same as the batch size of 25.

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:

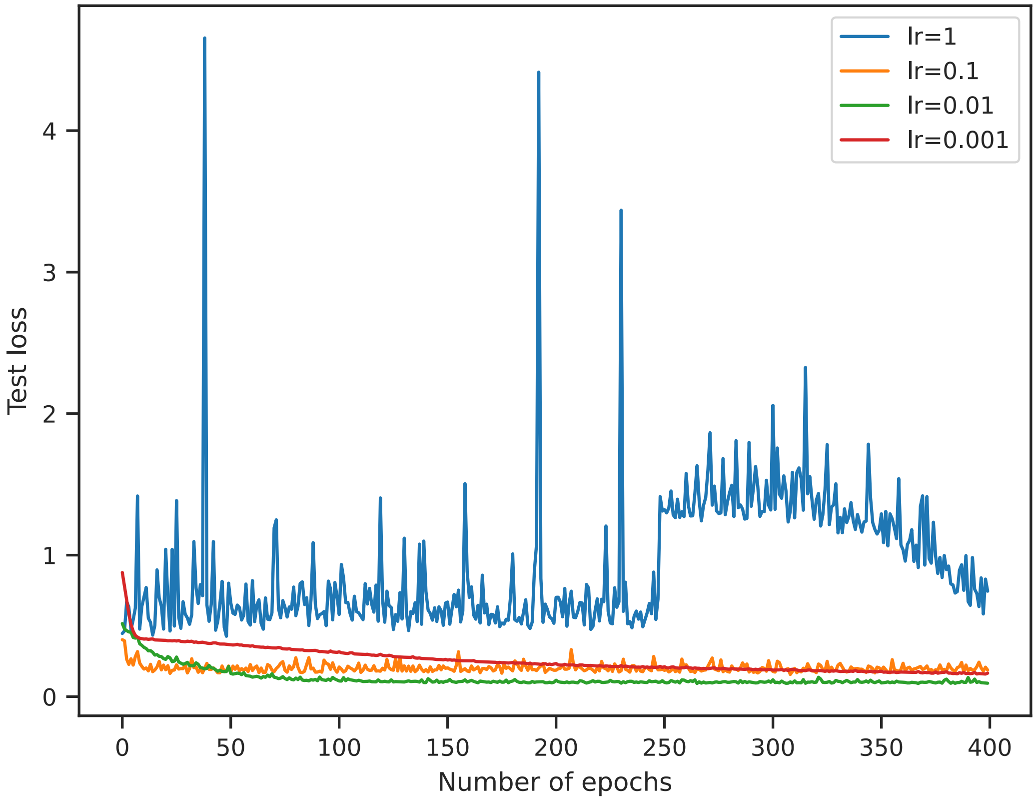
[ ]:
```
epochs = 400
lrs = [1, 0.1, 0.01, 0.001]
reg_weight = 0
test_losses = []
```

```python
models = []
n_batch = 10
for lr in lrs:
    print('Training lr: {}'.format(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,
        n_batch,
        epochs,
        lr
    )
    test_losses.append(test_loss)
    models.append(model)
```

```
Training lr: 1

100%|      | 400/400 [00:34<00:00, 11.45it/s]

Training lr: 0.1

100%|      | 400/400 [00:36<00:00, 10.81it/s]

Training lr: 0.01

100%|      | 400/400 [00:36<00:00, 11.05it/s]

Training lr: 0.001

100%|      | 400/400 [00:37<00:00, 10.76it/s]
```

[ ]:

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



The training times were all roughly equal for the different learning rates tested. The algorithm converged for all learning rates except for `lr=1`, which seemed to begin osciallating out of control as the number of epochs increased. The learning rate of 0.01 probably looks like the best choice here, since it leads to a more stable convergence at a low loss value that is basically equivalent to the value that 0.1 reaches, but 0.1 has more variation in the loss as a function of epoch number. 0.001 converges to a very stable value, but the loss is much higher, so this is too small of a value.

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?

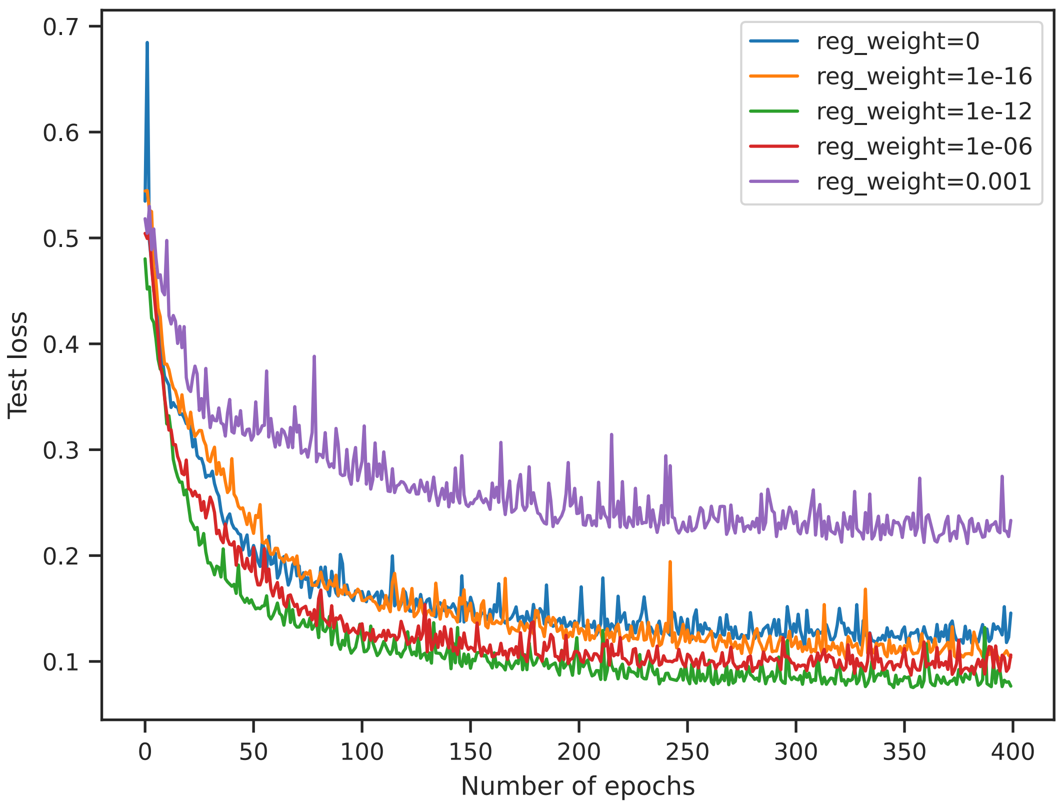
Answer:

[ ]:
```
epochs = 400
lr = 0.01
reg_weights = [0, 1e-16, 1e-12, 1e-6, 1e-3]
test_losses = []
models = []
for reg_weight in reg_weights:
    print('Training reg_weight: {}'.format(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,
        reg_weight,
        n_batch,
        epochs,
        lr
    )
    test_losses.append(test_loss)
    models.append(model)
```

```
Training reg_weight: 0

100%|      | 400/400 [00:35<00:00, 11.12it/s]

Training reg_weight: 1e-16

100%|      | 400/400 [00:36<00:00, 10.91it/s]

Training reg_weight: 1e-12

100%|      | 400/400 [00:36<00:00, 10.95it/s]

Training reg_weight: 1e-06

100%|      | 400/400 [00:37<00:00, 10.78it/s]

Training reg_weight: 0.001

100%|      | 400/400 [00:36<00:00, 10.90it/s]
```

[ ]:
```
fig, ax = plt.subplots(dpi=100)
for tl, reg_weight in zip(test_losses, reg_weights):
    ax.plot(tl, label="reg_weight={}".format(reg_weight))
ax.set_xlabel("Number of epochs")
ax.set_ylabel("Test loss")
plt.legend(loc="best");
```



Once again, each value takes about the same time to train. `reg_weight=1e-16` seems to perform the best in this case. It probably strikes the balance between penalizing large weights and thus overfitting without preventing the model from being able to learn. It seems that the model was not at great risk for overfitting in the first place, since the weight of `0` is somewhat comparable to this.

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

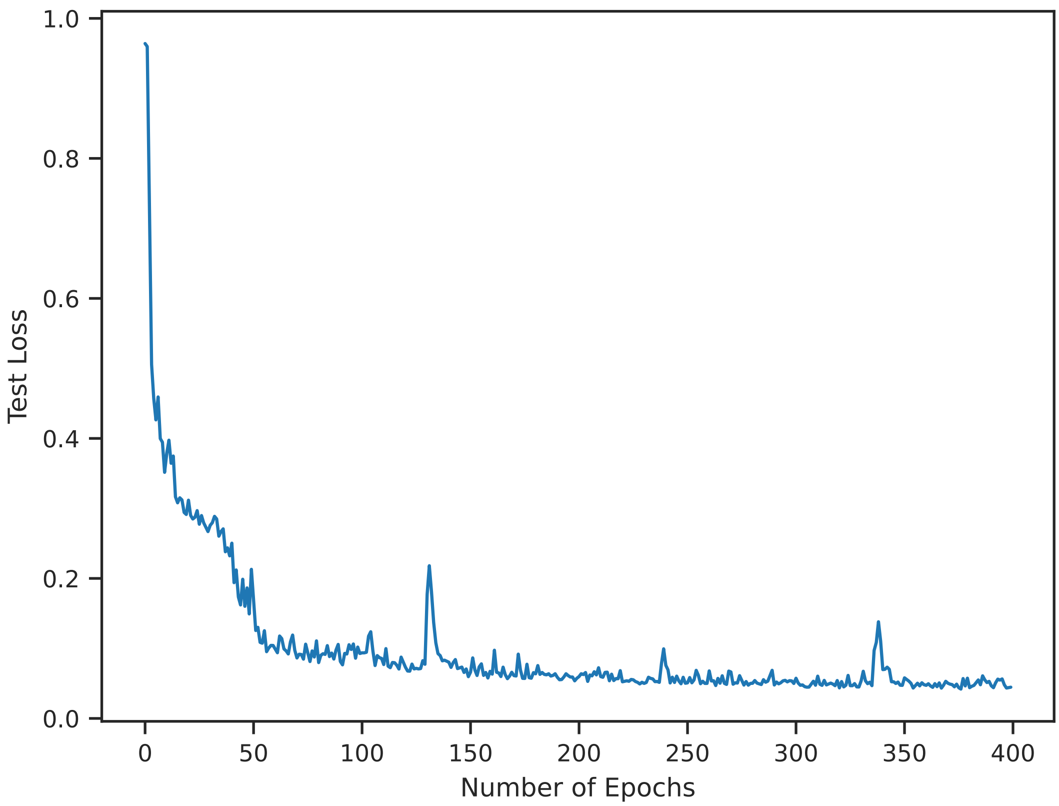
Answer:

[ ]:
```
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, 20), nn.Sigmoid(),
    nn.Linear(20, 1)
)
epochs = 400
```

```
lr = 0.01
reg_weight = 1e-16
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
)
```

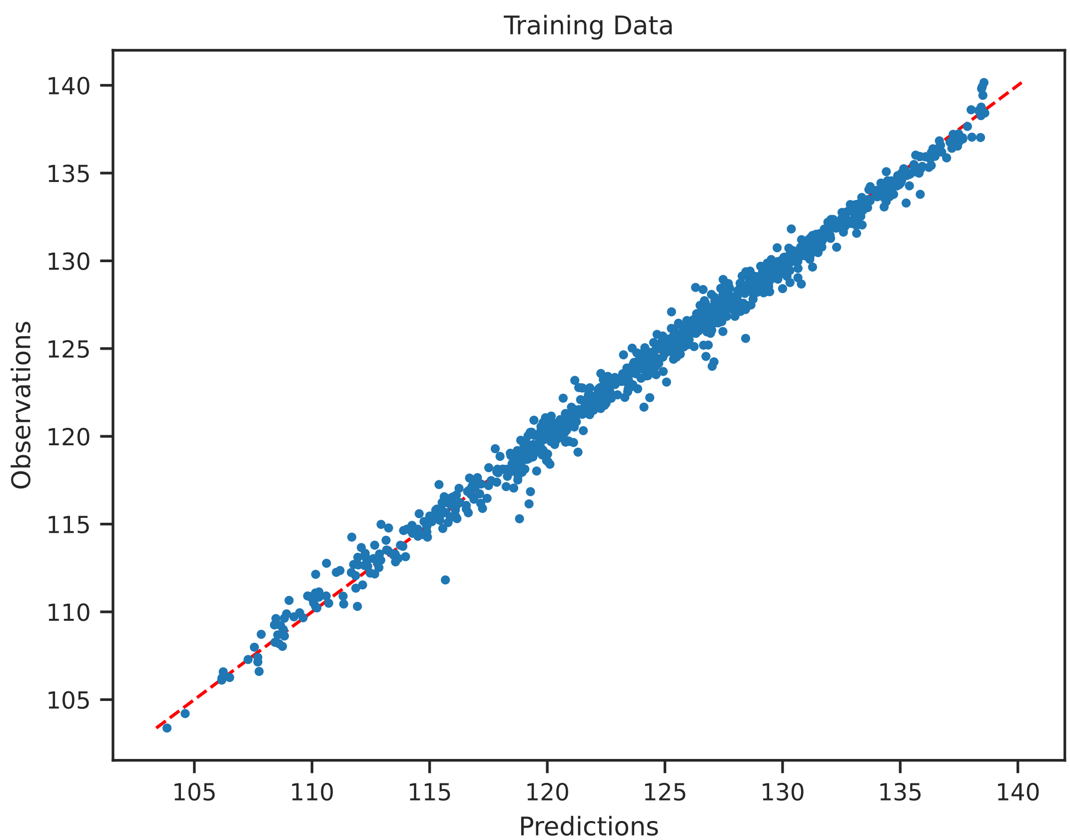
100%| | 400/400 [00:29<00:00, 13.42it/s]

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

ax.plot(test_loss)
ax.set_xlabel("Number of Epochs")
ax.set_ylabel("Test Loss");
```



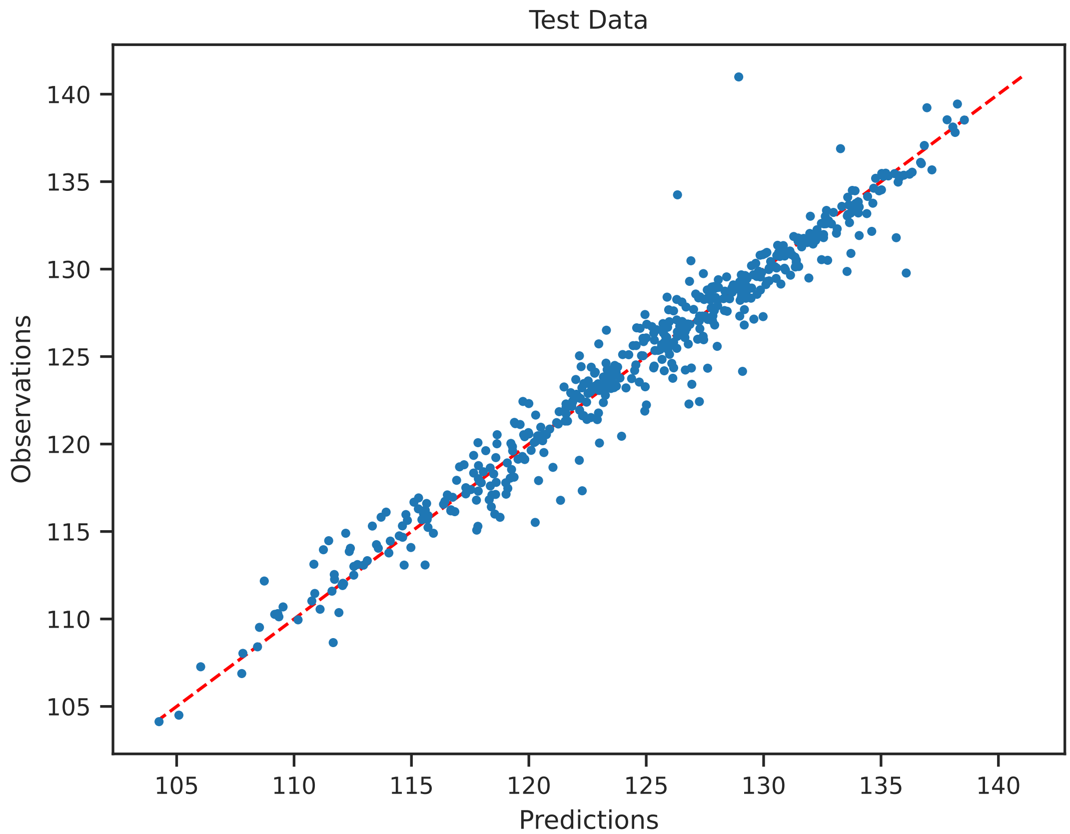
[ ]:
```
fig, ax = plt.subplots()

X_train_tensor = torch.Tensor(X_train)
y_pred_train = model(X_train_tensor)

# Plot the line of perfect fit
```

```
y_min = np.min(np.concatenate([y_pred_train, y_train]))
y_max = np.max(np.concatenate([y_pred_train, y_train]))
ax.plot([y_min, y_max], [y_min, y_max], 'r--')

ax.plot(y_pred_train, y_train, '.')

ax.set_title("Training Data")
ax.set_xlabel("Predictions")
ax.set_ylabel("Observations");
```



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

X_test_tensor = torch.Tensor(X_test)
y_pred_test = model(X_test_tensor)

# Plot the line of perfect fit
y_min = np.min(np.concatenate([y_pred_test, y_test]))
y_max = np.max(np.concatenate([y_pred_test, y_test]))
ax.plot([y_min, y_max], [y_min, y_max], 'r--')
```

```
ax.plot(y_pred_test, y_test, '.')

ax.set_title("Test Data")
ax.set_xlabel("Predictions")
ax.set_ylabel("Observations");
```



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.

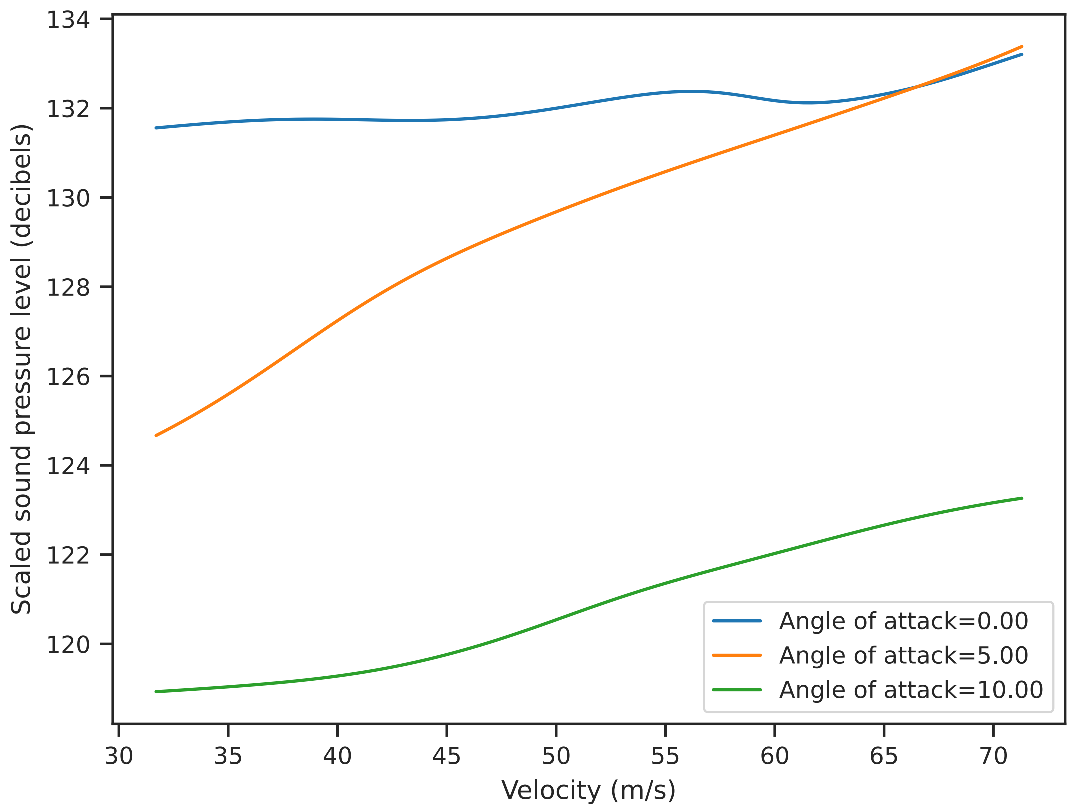
Answer:

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

[ ]:
```
best_model = model

def plot_sound_level_as_func_of_stream_vel(
    freq=2500,
    angle_of_attack=10,
    chord_length=0.1,
```

```python
    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)')
```

[ ]:
```python
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");
```



1.4 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:

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

Here is how to load the data:

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

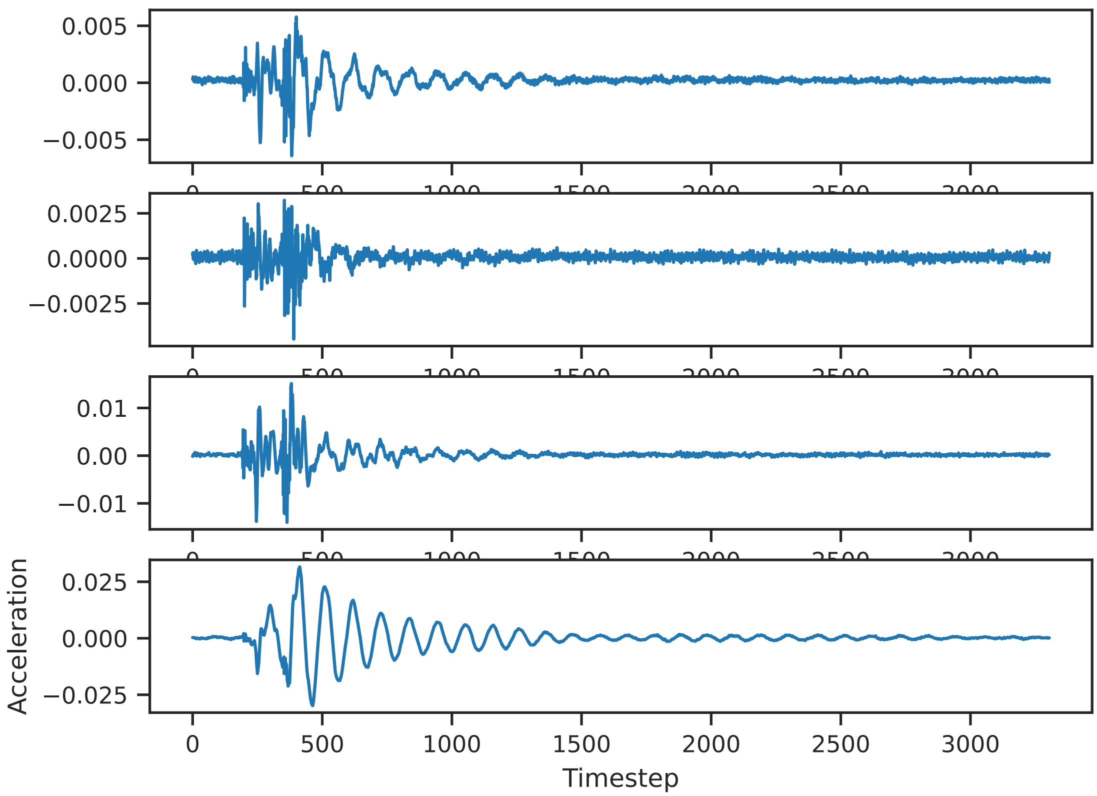
This is a Python dictionary that contains the following entries:

```
[ ]: 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");
```

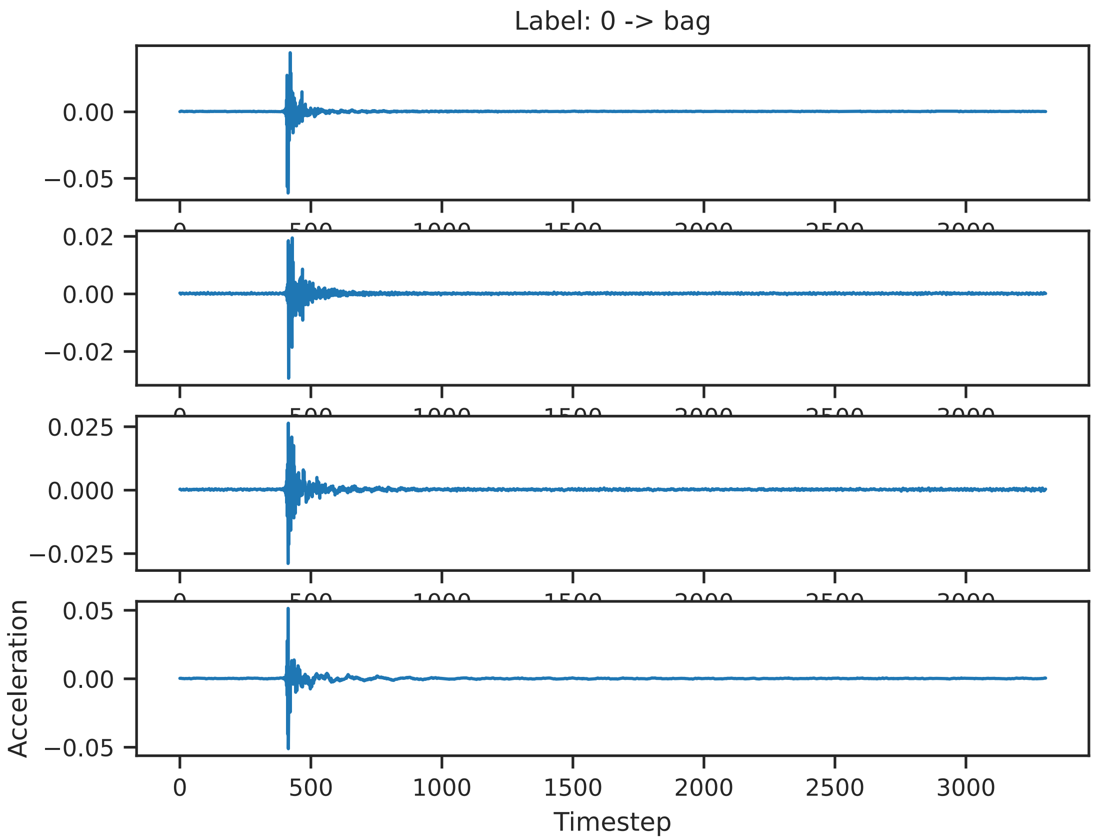


The second key, `labels_1`, is a bunch of integers ranging from 0 to 2 indicating whether the entry corresponds to a “bag,” a “ball” or a “jump.” For your reference, the correspondence is:

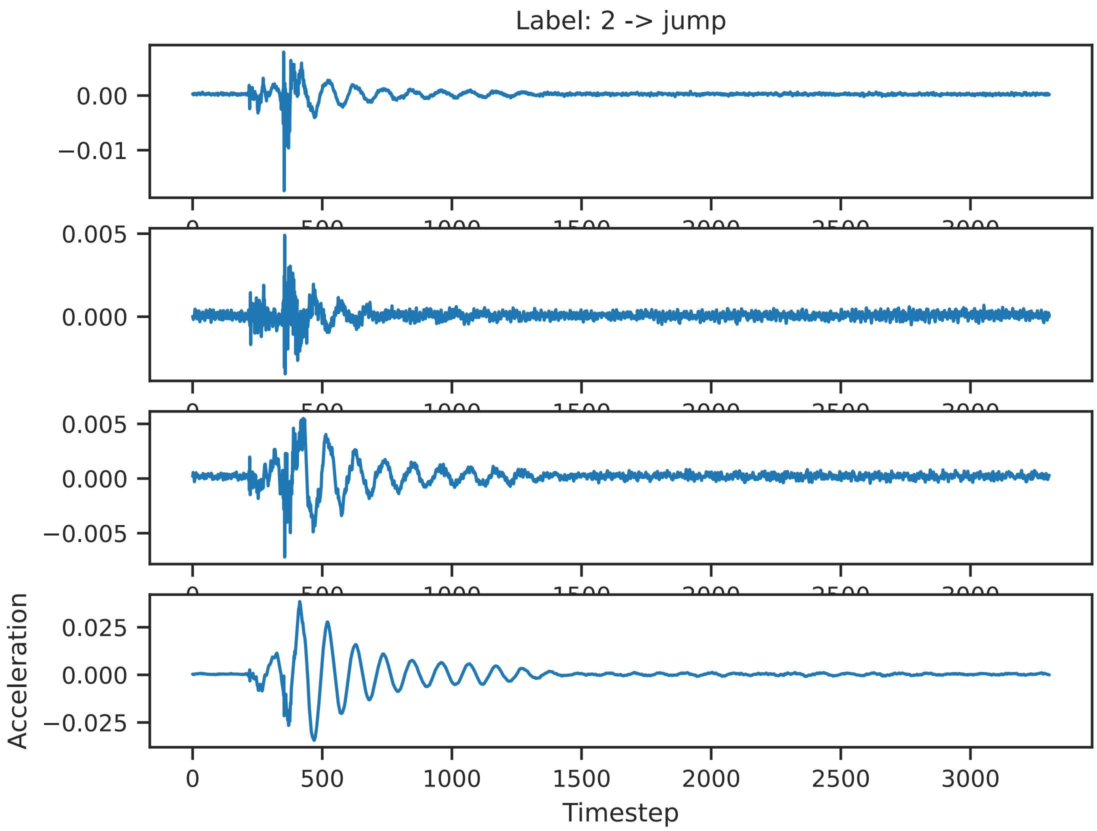
[ ]:
```
LABELS_1_TO_TEXT = {0: "bag", 1: "ball", 2: "jump"}
```

And here are a few examples:

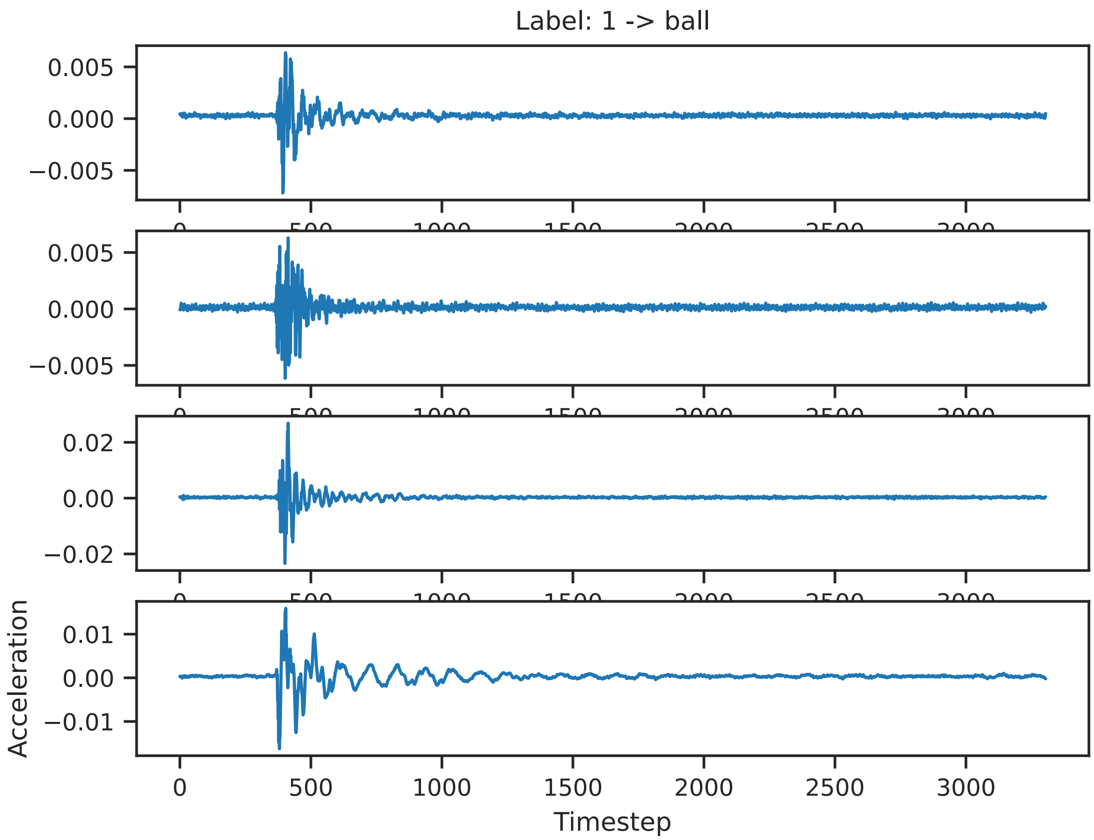
[ ]:
```
for _ in range(5):
    i = np.random.randint(0, data["features"].shape[0])
    fig, ax = plt.subplots(4, 1, dpi=100)
    for j in range(4):
        ax[j].plot(data["features"][i, j])
    ax[-1].set_xlabel("Timestep")
    ax[-1].set_ylabel("Acceleration")
    ax[0].set_title(
        "Label: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

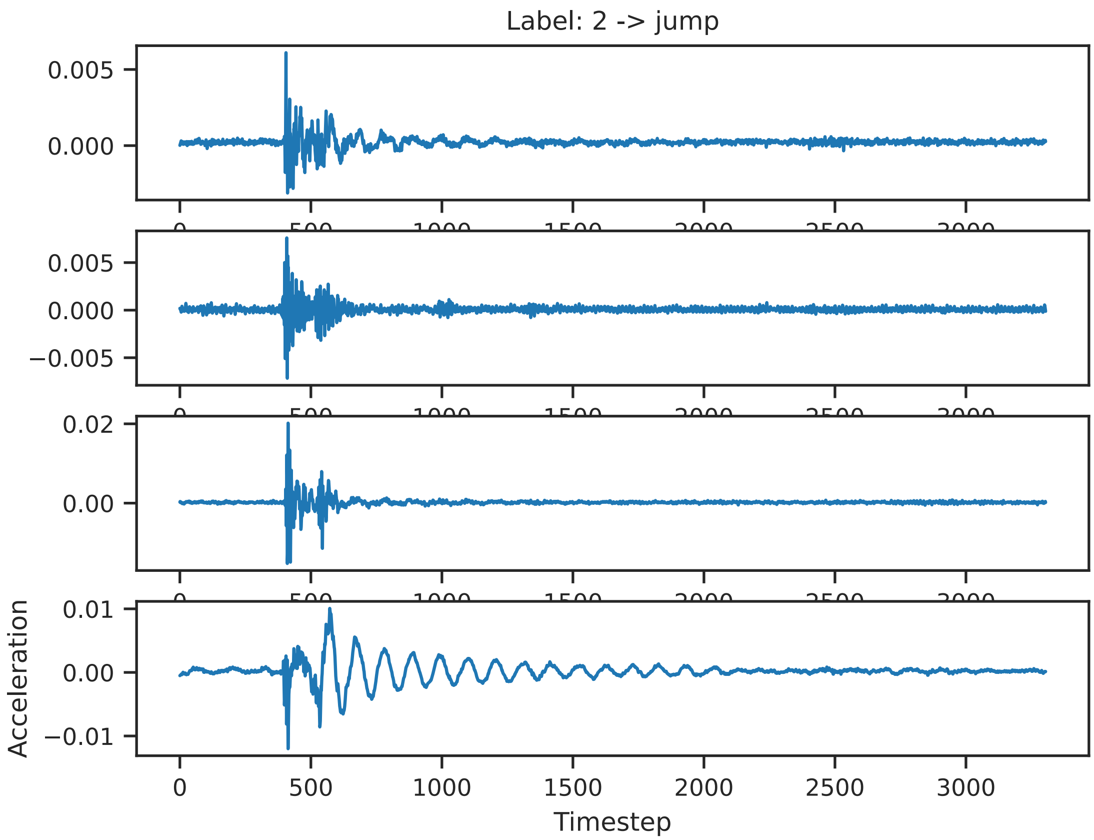


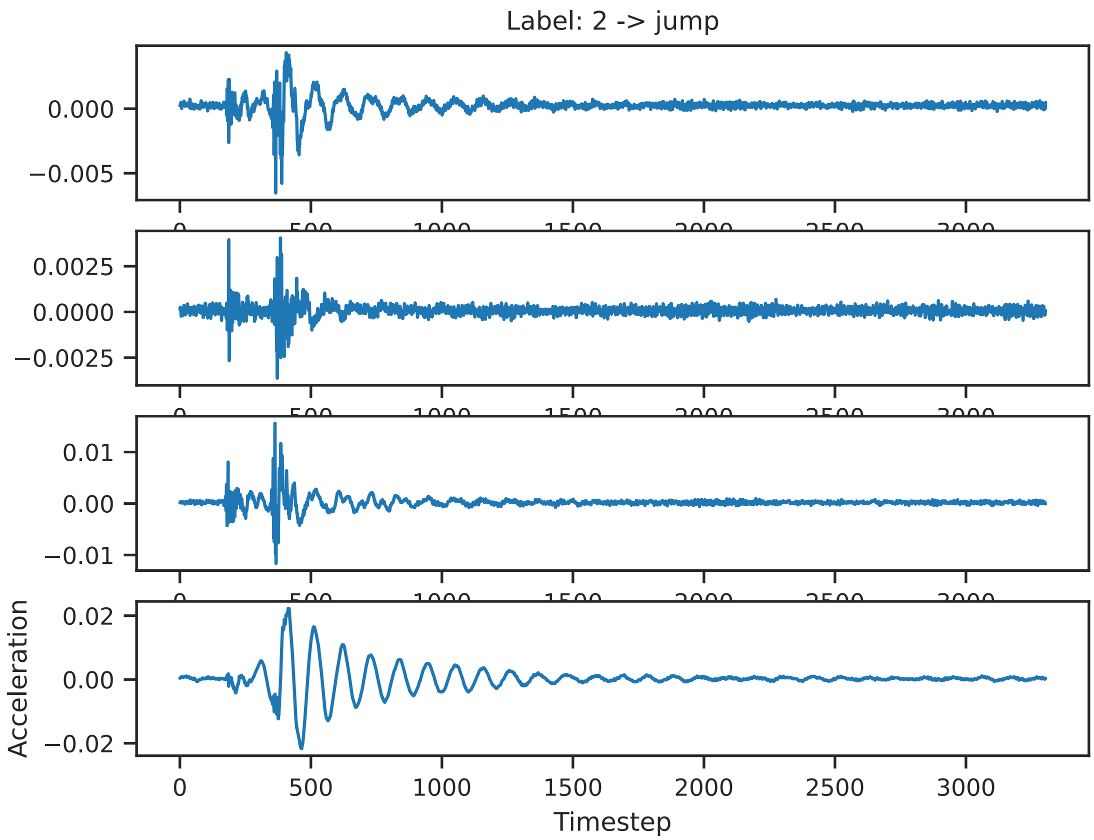
Label: 0 -> bag
0.00
−0.05
0.02
0.00
−0.02
0.025
0.000
−0.025
Acceleration
0.05
0.00
−0.05
0
500
1000
1500
2000
2500
3000
Timestep



Label: 2 -> jump
0.00
−0.01
0.005
0.000
0.005
0.000
−0.005
Acceleration
0.025
0.000
−0.025
0
500
1000
1500
2000
2500
3000
Timestep


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


Label: 2 -> jump
0.005
0.000
−0.005
0.02
0.00
0.01
−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"]
```

1.4.1 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:

[ ]:
```
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 through 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):
    """
    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=test_size)

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

```python
    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(params=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

            optimizer.zero_grad()

            y_pred = net(X_batch)

            loss = cnn_loss_func(y_pred, y_batch) # + reg_weight * l2_reg_loss(params)

            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(f'Epoch {e+1}: accuracy = {accuracy[-1]:1.5f}, test loss =␣
 ↪{ts_loss.item()}')
    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
```

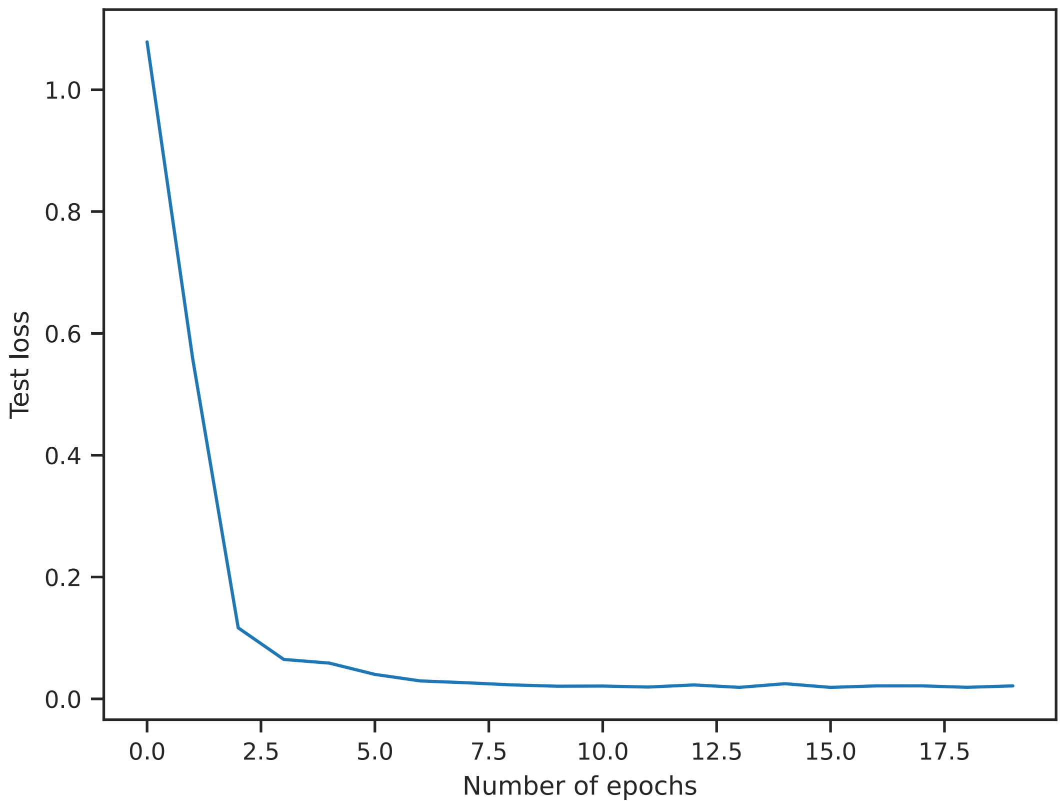
Now experiment with the epochs, the learning rate, and the batch size until this works.

```
[ ]: epochs = 20
lr = 0.001
n_batch = 30
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 = 0.42664, test loss = 1.0785764455795288
Epoch 2: accuracy = 0.72837, test loss = 0.5590016841888428
Epoch 3: accuracy = 0.97818, test loss = 0.11661700904369354
Epoch 4: accuracy = 0.98871, test loss = 0.06478121131658554
Epoch 5: accuracy = 0.98721, test loss = 0.0587134025990963
Epoch 6: accuracy = 0.99097, test loss = 0.04019084572792053
Epoch 7: accuracy = 0.99473, test loss = 0.029450371861457825
Epoch 8: accuracy = 0.99323, test loss = 0.026471437886357307
Epoch 9: accuracy = 0.99549, test loss = 0.02297958731651306
Epoch 10: accuracy = 0.99699, test loss = 0.020777974277734756
Epoch 11: accuracy = 0.99549, test loss = 0.021033180877566338
Epoch 12: accuracy = 0.99850, test loss = 0.019437361508607864
Epoch 13: accuracy = 0.99549, test loss = 0.02288350835442543
Epoch 14: accuracy = 0.99850, test loss = 0.018923873081803322
Epoch 15: accuracy = 0.99549, test loss = 0.024927642196416855
Epoch 16: accuracy = 0.99774, test loss = 0.018860435113310814
Epoch 17: accuracy = 0.99699, test loss = 0.021309170871973038
Epoch 18: accuracy = 0.99699, test loss = 0.02136903442442417
Epoch 19: accuracy = 0.99774, test loss = 0.019002089276909828
Epoch 20: accuracy = 0.99699, test loss = 0.021310392767190933
```

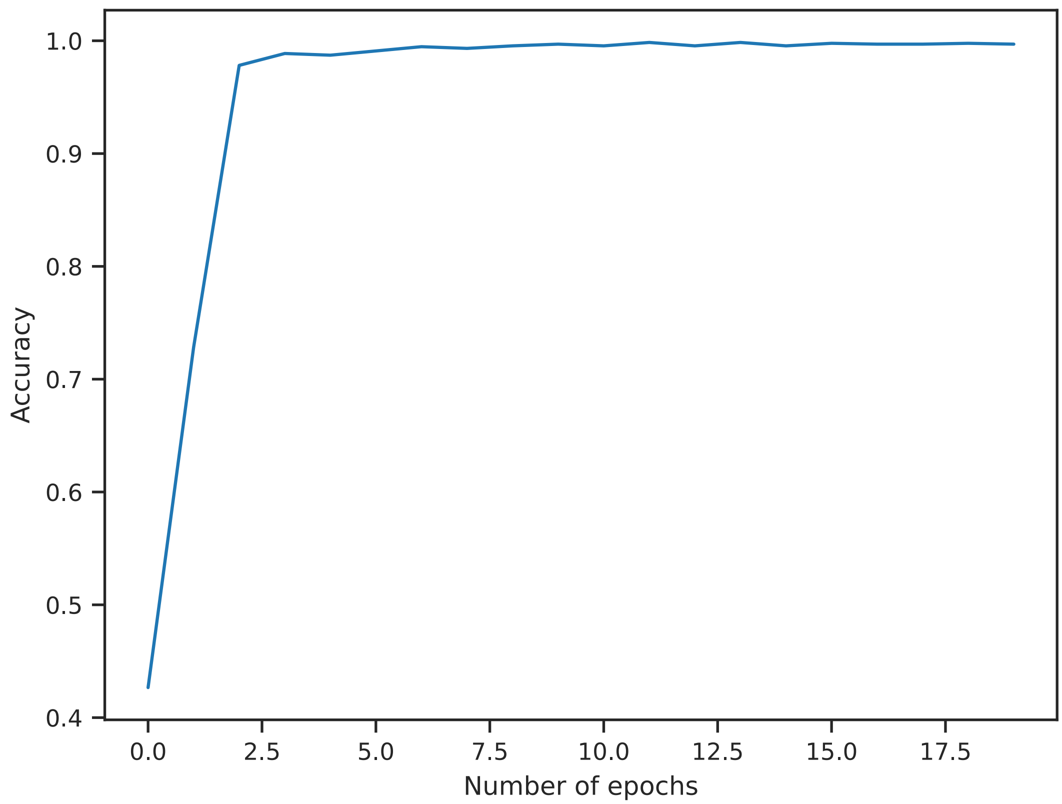
Plot the evolution of the test loss as a function of epochs.

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



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

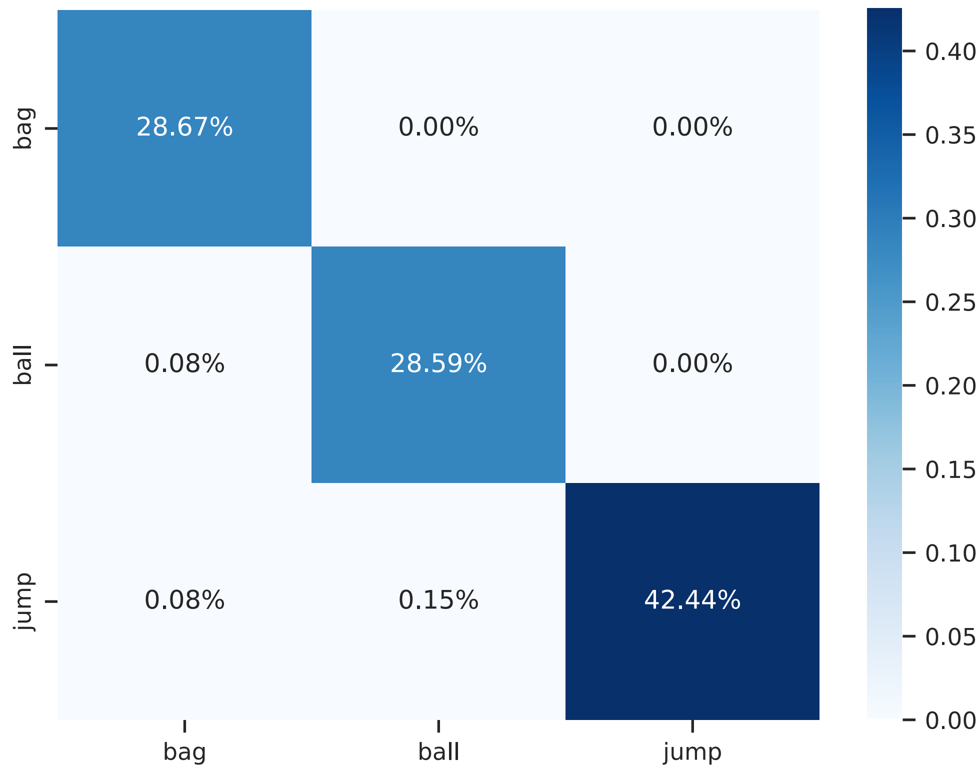
[ ]:
```
fig, ax = plt.subplots(dpi=100)
ax.plot(accuracy)
ax.set_xlabel("Number of epochs")
ax.set_ylabel("Accuracy");
```



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 probabilisticEpoch
# We need to simply pick the label with the highest probability:
_, y_pred_labels = torch.max(y_pred_test, 1)
# Here is the confusion matrix:
cf_matrix = confusion_matrix(y_test, y_pred_labels)
```

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



1.4.2 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:

[ ]:
```
net = Net(7)
```

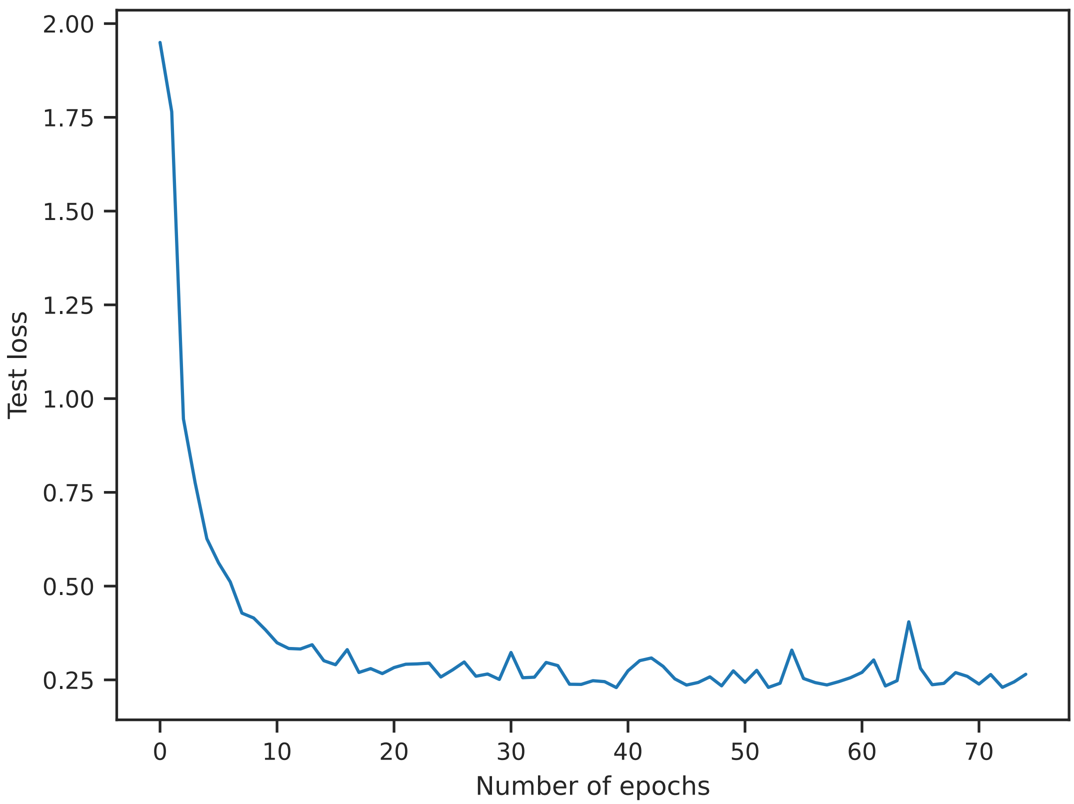
[ ]:
```
# 93%
epochs = 75
lr = 0.0011#5
n_batch = 30
trained_model, test_loss, accuracy1, X_train, y_train, X_test, y_test = train_cnn(
    X, y2, net, n_batch, epochs, lr
)

# epochs = 50
# lr = 0.001
# n_batch = 30
# trained_model, test_loss, accuracy, X_train, y_train, X_test, y_test = train_cnn(
```

```
#     X, y2, trained_model, n_batch, epochs, lr
# )

# epochs = 50
# lr = 0.0008
# n_batch = 30
# trained_model, test_loss, accuracy, X_train, y_train, X_test, y_test = train_cnn(
#     X, y2, trained_model, n_batch, epochs, lr
# )
```

```
Epoch 1: accuracy = 0.13093, test loss = 1.9496824741363525
Epoch 2: accuracy = 0.22423, test loss = 1.7648767232894897
Epoch 3: accuracy = 0.53047, test loss = 0.9458544850349426
Epoch 4: accuracy = 0.59142, test loss = 0.7757077813148499
Epoch 5: accuracy = 0.65839, test loss = 0.6264389753341675
Epoch 6: accuracy = 0.70429, test loss = 0.5621477961540222
Epoch 7: accuracy = 0.76373, test loss = 0.5113839507102966
Epoch 8: accuracy = 0.82092, test loss = 0.4279845356941223
Epoch 9: accuracy = 0.81941, test loss = 0.4150756597518921
Epoch 10: accuracy = 0.81941, test loss = 0.3838397264480591
Epoch 11: accuracy = 0.85403, test loss = 0.34898486733436584
Epoch 12: accuracy = 0.85854, test loss = 0.33369866013526917
Epoch 13: accuracy = 0.84725, test loss = 0.3323861360549927
Epoch 14: accuracy = 0.85327, test loss = 0.34370261430740356
Epoch 15: accuracy = 0.87133, test loss = 0.301111102104187
Epoch 16: accuracy = 0.88036, test loss = 0.29033809900283813
Epoch 17: accuracy = 0.86757, test loss = 0.33066239953041077
Epoch 18: accuracy = 0.88713, test loss = 0.26970377564430237
Epoch 19: accuracy = 0.88337, test loss = 0.2800876200199127
Epoch 20: accuracy = 0.88939, test loss = 0.2667470872402191
Epoch 21: accuracy = 0.87810, test loss = 0.28279614448547363
Epoch 22: accuracy = 0.87735, test loss = 0.291701078414917
Epoch 23: accuracy = 0.88036, test loss = 0.29264646768569946
Epoch 24: accuracy = 0.88412, test loss = 0.2946324646472931
Epoch 25: accuracy = 0.89014, test loss = 0.25764045119285583
Epoch 26: accuracy = 0.88187, test loss = 0.27653223276138306
Epoch 27: accuracy = 0.88488, test loss = 0.297753244638443
Epoch 28: accuracy = 0.89090, test loss = 0.2596733272075653
Epoch 29: accuracy = 0.88864, test loss = 0.2656620442867279
Epoch 30: accuracy = 0.88939, test loss = 0.25115013122558594
Epoch 31: accuracy = 0.86080, test loss = 0.3230888843536377
Epoch 32: accuracy = 0.89165, test loss = 0.25565099716186523
Epoch 33: accuracy = 0.89391, test loss = 0.25713035464286804
Epoch 34: accuracy = 0.87961, test loss = 0.296455442905426
Epoch 35: accuracy = 0.88412, test loss = 0.2880666255950928
Epoch 36: accuracy = 0.90068, test loss = 0.23832035064697266
Epoch 37: accuracy = 0.90068, test loss = 0.23792560398578644
```

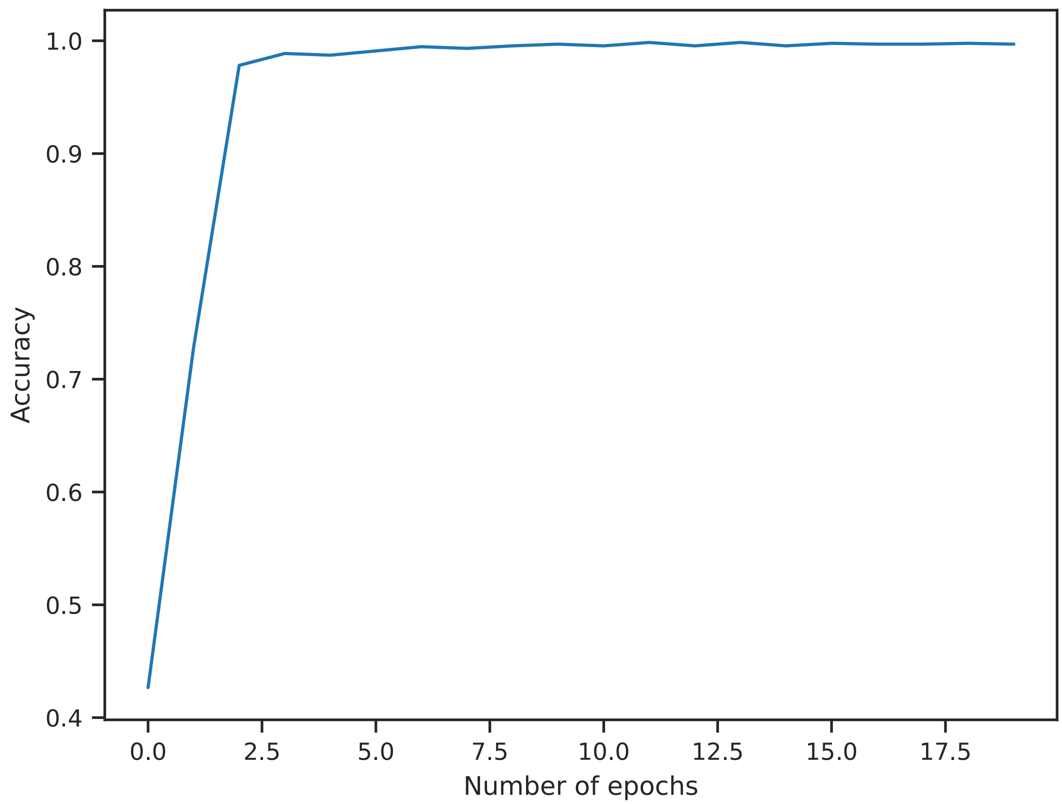
```
Epoch 38: accuracy = 0.89842, test loss = 0.24776367843151093
Epoch 39: accuracy = 0.89992, test loss = 0.2452692985534668
Epoch 40: accuracy = 0.90594, test loss = 0.22946150600910187
Epoch 41: accuracy = 0.89391, test loss = 0.27420902252197266
Epoch 42: accuracy = 0.88111, test loss = 0.30122965574264526
Epoch 43: accuracy = 0.88412, test loss = 0.3083687722682953
Epoch 44: accuracy = 0.89315, test loss = 0.2860420048236847
Epoch 45: accuracy = 0.90745, test loss = 0.25271111726760864
Epoch 46: accuracy = 0.90293, test loss = 0.2362031787633896
Epoch 47: accuracy = 0.90293, test loss = 0.2430894821882248
Epoch 48: accuracy = 0.90594, test loss = 0.25807514786720276
Epoch 49: accuracy = 0.90369, test loss = 0.23403804004192352
Epoch 50: accuracy = 0.89391, test loss = 0.2741195559501648
Epoch 51: accuracy = 0.90444, test loss = 0.243416428565979
Epoch 52: accuracy = 0.90143, test loss = 0.2754834294319153
Epoch 53: accuracy = 0.91046, test loss = 0.22989299893379211
Epoch 54: accuracy = 0.90971, test loss = 0.2410309761762619
Epoch 55: accuracy = 0.88187, test loss = 0.3293669521808624
Epoch 56: accuracy = 0.90519, test loss = 0.25362786650657654
Epoch 57: accuracy = 0.91196, test loss = 0.24274834990501404
Epoch 58: accuracy = 0.91046, test loss = 0.2364710420370102
Epoch 59: accuracy = 0.91196, test loss = 0.24519456923007965
Epoch 60: accuracy = 0.90519, test loss = 0.2555801570415497
Epoch 61: accuracy = 0.91046, test loss = 0.26999884843826294
Epoch 62: accuracy = 0.89014, test loss = 0.3031378388404846
Epoch 63: accuracy = 0.91573, test loss = 0.23372623324394226
Epoch 64: accuracy = 0.90519, test loss = 0.2478390634059906
Epoch 65: accuracy = 0.88187, test loss = 0.40484586358070374
Epoch 66: accuracy = 0.90895, test loss = 0.28060588240623474
Epoch 67: accuracy = 0.91874, test loss = 0.23709408938884735
Epoch 68: accuracy = 0.91874, test loss = 0.2406998574733734
Epoch 69: accuracy = 0.91046, test loss = 0.2693856954574585
Epoch 70: accuracy = 0.90895, test loss = 0.25943586230278015
Epoch 71: accuracy = 0.91573, test loss = 0.23877692222595215
Epoch 72: accuracy = 0.90820, test loss = 0.2642674446105957
Epoch 73: accuracy = 0.92099, test loss = 0.23007147014141083
Epoch 74: accuracy = 0.91798, test loss = 0.24462997913360596
Epoch 75: accuracy = 0.91347, test loss = 0.2649286091327667
```

[ ]:
```
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.

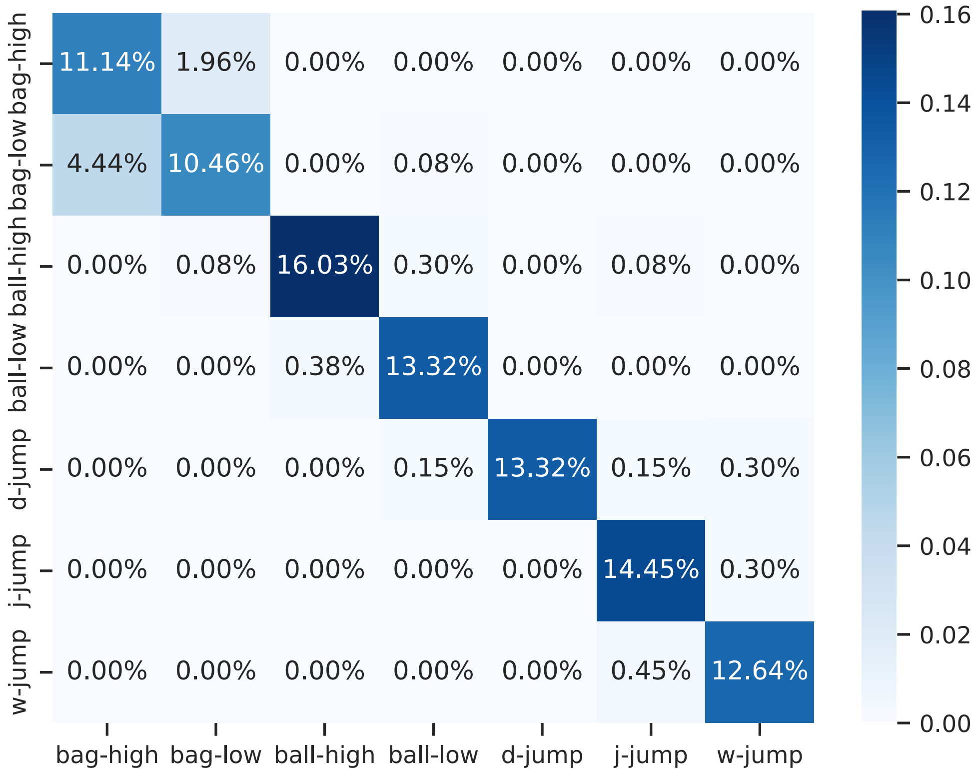
```
[ ]: fig, ax = plt.subplots(dpi=100)
ax.plot(accuracy)
ax.set_xlabel("Number of epochs")
ax.set_ylabel("Accuracy");
```



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 probabilisticEpoch
# We need to simply pick the label with the highest probability:
_, y_pred_labels = torch.max(y_pred_test, 1)
# Here is the confusion matrix:
cf_matrix = confusion_matrix(y_test, y_pred_labels)
```

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


bag-high
bag-low
ball-high
ball-low
d-jump
j-jump
w-jump
11.14%
1.96%
0.00%
0.00%
0.00%
0.00%
0.00%
4.44%
10.46%
0.00%
0.08%
0.00%
0.00%
0.00%
0.00%
0.08%
16.03%
0.30%
0.00%
0.08%
0.00%
0.00%
0.00%
0.38%
13.32%
0.00%
0.00%
0.00%
0.00%
0.00%
0.00%
0.15%
13.32%
0.15%
0.30%
0.00%
0.00%
0.00%
0.00%
0.00%
14.45%
0.30%
0.00%
0.00%
0.00%
0.00%
0.00%
0.45%
12.64%
0.16
0.14
0.12
0.10
0.08
0.06
0.04
0.02
0.00