homework_09

December 7, 2021

1 Homework 9

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 can either:
  - Type the answer using the built-in latex capabilities. In this case, simply export the notebook as a pdf and upload it on gradescope; or
  - You can print the notebook (after you are done with all the code), write your answers by hand, scan, turn your response to a single pdf, and upload on gradescope.
- The total homework points are 100. Please note that the problems are not weighed equally.

Note: Please match all the pages corresponding to each of the questions when you submit on gradescope.

1.3 Student details

- **First Name:** Alex
- **Last Name:** Shank
- **Email:** shank14@purdue.edu

[1]:
```
import matplotlib.pyplot as plt
%matplotlib inline
import numpy as np
import pandas as pd
import seaborn as sns
sns.set_context('paper')
sns.set_style('white')
import scipy.stats as st
# A helper function for downloading files
import requests
import os
```

```
def download(url, local_filename=None):
    """
    Downloads the file in the ``url`` and saves it in the current working␣
 ↪directory.
    """
    data = requests.get(url)
    if local_filename is None:
        local_filename = os.path.basename(url)
    with open(local_filename, 'wb') as fd:
        fd.write(data.content)
```

1.4 Problem 1 - Using DNNs to Analyze Experimental Data

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

> The NASA data set comprises different size NACA 0012 airfoils at various wind tunnel speeds and angles of attack. The span of the airfoil and the observer position were the same in all of the experiments.
>
> Attribute Information: This problem has the following inputs: 1. Frequency, in Hertzs. 2. Angle of attack, in degrees. 3. Chord length, in meters. 4. Free-stream velocity, in meters per second. 5. Suction side displacement thickness, in meters.
>
> The only output is: 6. Scaled sound pressure level, in decibels.

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

[3]:
```
# TODO Replace
url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/00291/
 ↪airfoil_self_noise.dat'
# download(url)
```

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

[4]:
```
data = np.loadtxt('airfoil_self_noise.dat')
data
```

[4]:
```
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:

[5]:
```
df = pd.DataFrame(data, columns=['Frequency', 'Angle_of_attack', 'Chord_length',
                                 'Velocity', 'Suction_thickness',␣
 ↪'Sound_pressure'])
df

# add these lists for plotting help
variable_names = ['Frequency', 'Angle_of_attack', 'Chord_length', 'Velocity',␣
 ↪'Suction_thickness', 'Sound_pressure']
variable_units = ['Hz', 'Degrees', 'm', 'm/s', 'm', 'decibels']
```

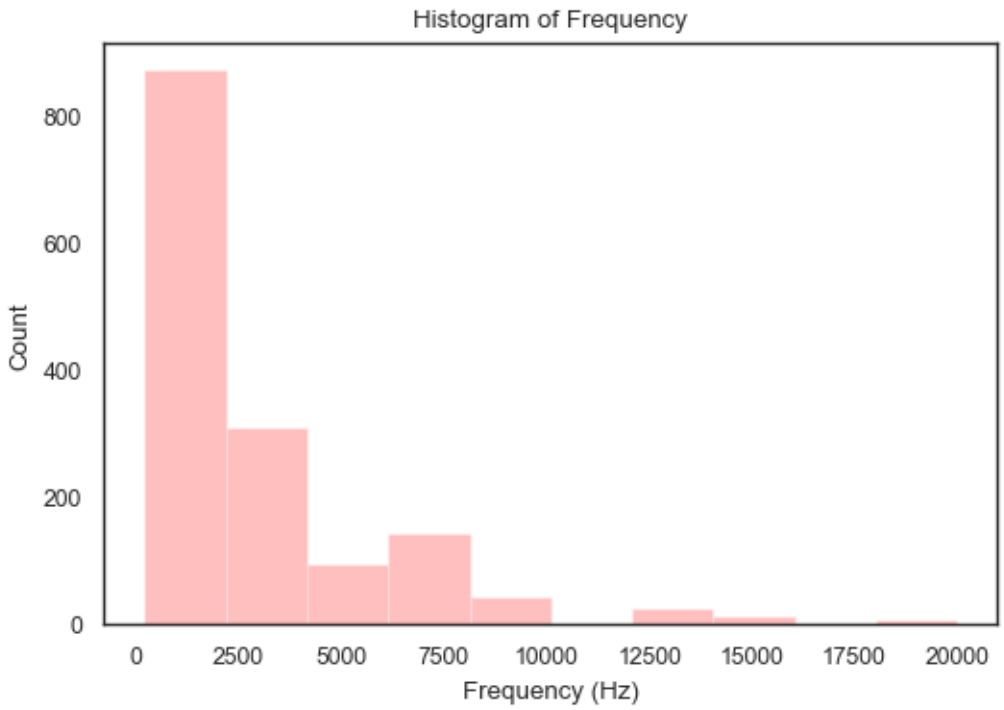
1.4.1 Part A - Analyze the data visually

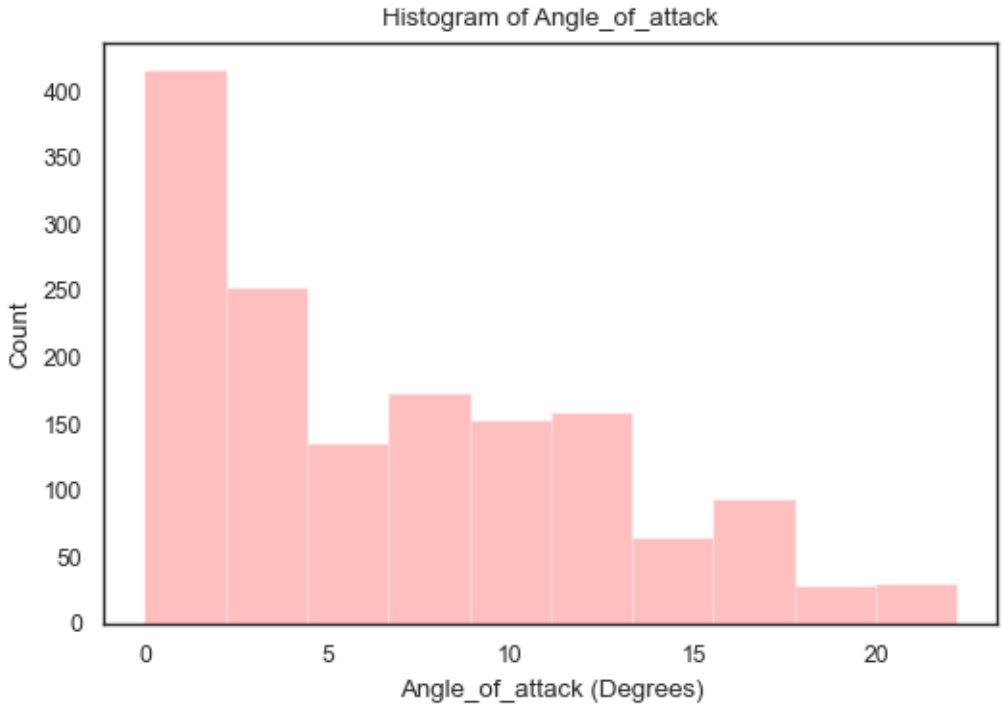
It is always a good idea to visualize the data before you start doing anything with them.

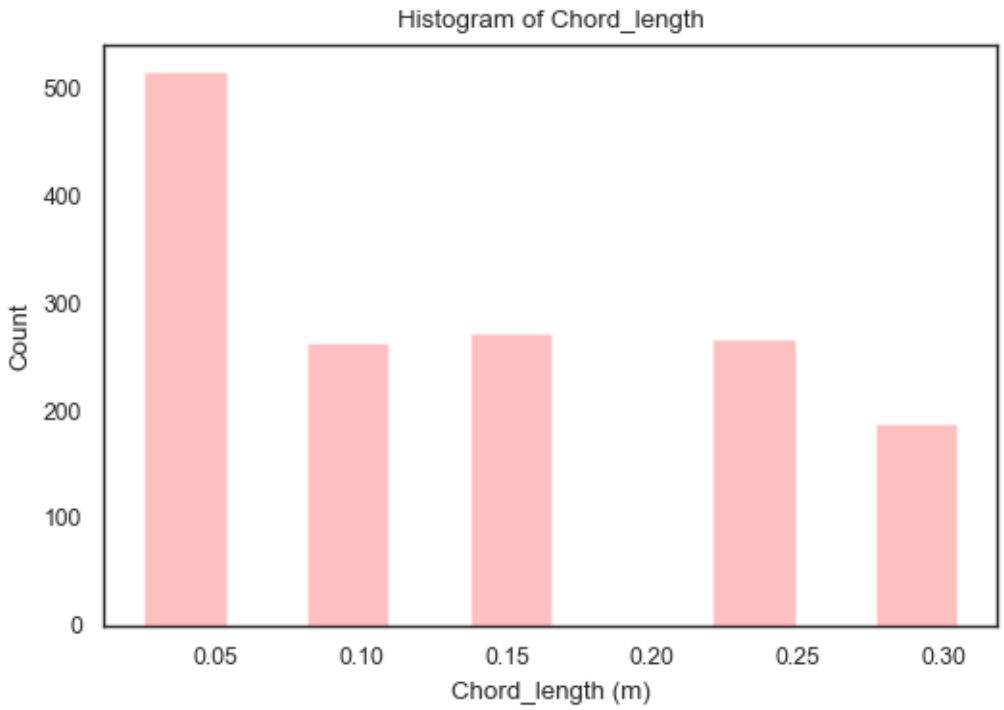
Part A.I - Do the histogtrams of all variables Use as many code segments you need below to plot the histogram of each variable (all inputs and the output in separate plots) Discuss whether or not you need to standarize the data before moving to regression.

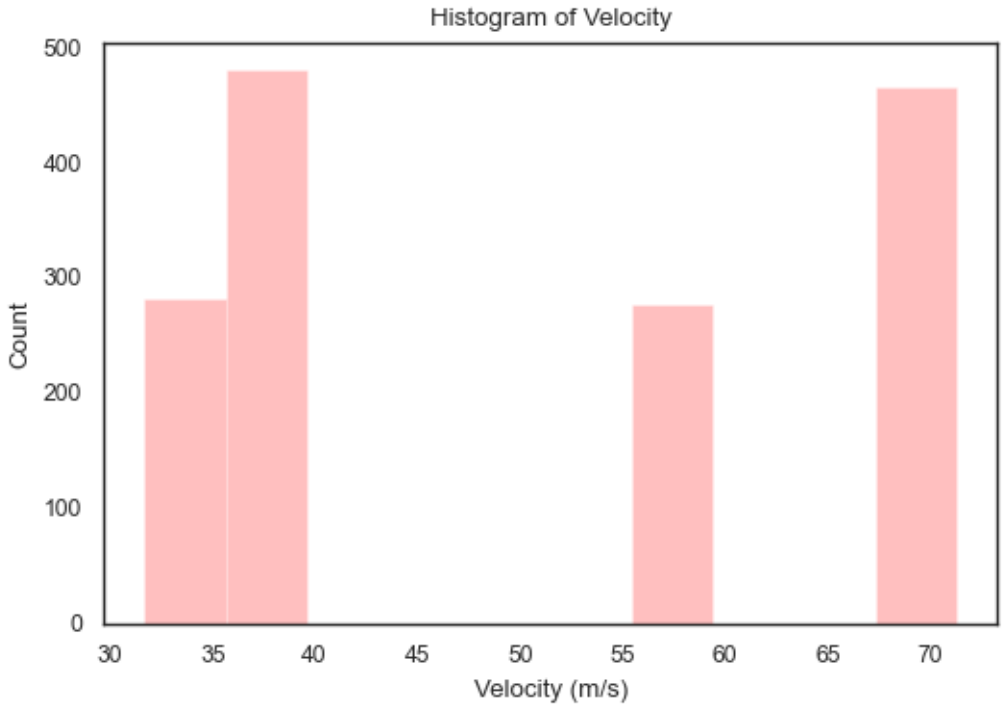
Answer: From the histograms below, we can see that the input and output variables have significantly different scales. **We should therefore normalize them for our regression task.**

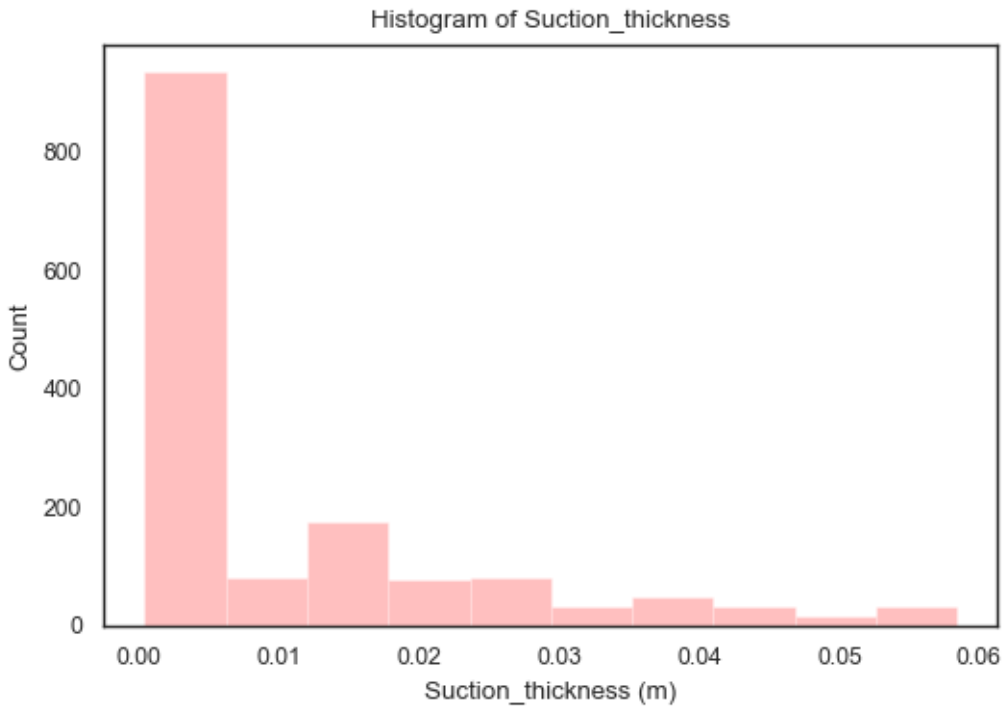
[7]:
```
def plotHistogram(variable_index):
    fig, ax = plt.subplots(dpi=100)
    ax.set_title('Histogram of ' + variable_names[variable_index])
    ax.set_xlabel(variable_names[i] + ' (' + variable_units[variable_index] +␣
 ↪')')
    ax.set_ylabel('Count')
    ax.hist(df[variable_names[i]], alpha=0.25, density=False, color='red',␣
 ↪label="Sample Histogram")

for i in range(len(variable_names)):
    plotHistogram(i)
```

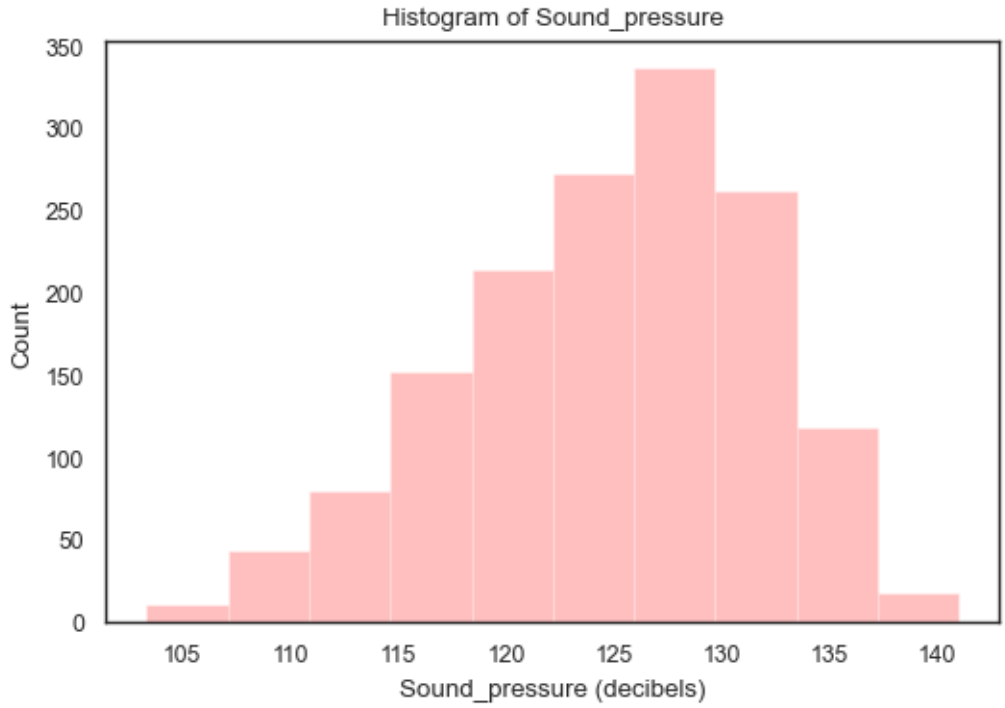

Histogram of Frequency
Count
800
600
400
200
0
0
2500
5000
7500
10000
12500
15000
17500
20000
Frequency (Hz)


Histogram of Angle_of_attack
Count
400
350
300
250
200
150
100
50
0
0
5
10
15
20
Angle_of_attack (Degrees)


Histogram of Chord_length
Count
0
100
200
300
400
500
0.05
0.10
0.15
0.20
0.25
0.30
Chord_length (m)


Histogram of Velocity
Count
0
100
200
300
400
500
30
35
40
45
50
55
60
65
70
Velocity (m/s)


Histogram of Suction_thickness
Count
800
600
400
200
0
0.00
0.01
0.02
0.03
0.04
0.05
0.06
Suction_thickness (m)


Histogram of Sound_pressure
Count
350
300
250
200
150
100
50
0
105
110
115
120
125
130
135
140
Sound_pressure (decibels)

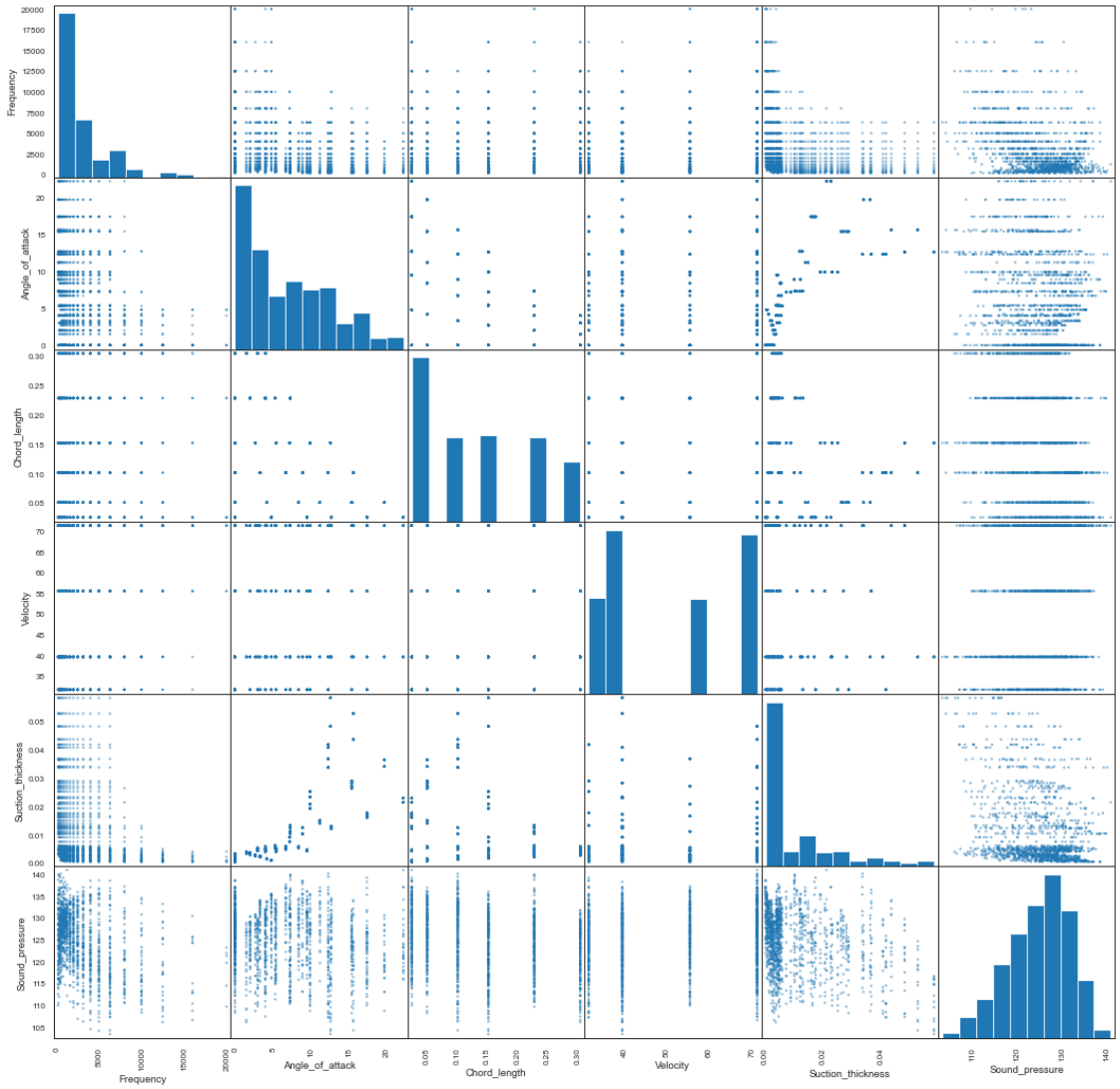
Part A.II - Do the scatter plots between all input variables Do the scatter plot between all input variables. This will give you an idea of the range of experimental conditions. Whatever model you build will only be valid inside the domain implicitly defined with your experimental conditions. Are there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

Answer: See Part A.III for complete matrix of input/output scatter plots.

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: Yes, it could be beneficial to gather more test data at input points that look sparce in the scatter plots and histograms below, but otherwise there are no immediate trends that are apparent.

[11]:
```
# we'll put the output on the x-axis
# will show us if we're missing data somewhere all the same
# plotInputScatter(5)
panda_df = pd.DataFrame(df);
pd.plotting.scatter_matrix(panda_df, figsize= (18,18));
```



1.4.2 Part B - Use DNN to do regression

Let start by separating inputs and outputs for you:

[12]:
```
X = data[:, :-1]
y = data[:, -1][:, None]
```

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

Answer:

[20]:
```
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()
```

[21]:
```
# Test your code here
y_obs_tmp = np.random.randn(100, 1)
y_pred_tmp = np.random.randn(100, 1)
print('Your mse_loss: {0:1.2f}'.format(mse_loss(torch.Tensor(y_obs_tmp),
                                                torch.Tensor(y_pred_tmp))))
print('What you should be getting: {0:1.2f}'.format(np.mean((y_obs_tmp -␣
 ↪y_pred_tmp) ** 2)))
```

```
Your mse_loss: 2.16
What you should be getting: 2.16
```

[22]:
```
# 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
```

[25]:
```
# 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)
```

[26]:
```
# You can try your final code here
# First, here is a dummy model
dummy_net = nn.Sequential(nn.Linear(10, 20),
                          nn.Sigmoid(),
                          nn.Linear(20, 1))
loss = loss_func(torch.Tensor(y_obs_tmp), torch.Tensor(y_pred_tmp),
                 0.0,
                 dummy_net.parameters())
print('The loss without regularization: {0:1.2f}'.format(loss.item()))
print('This should be the same as this: {0:1.2f}'.format(mse_loss(torch.
 ↪Tensor(y_obs_tmp), torch.Tensor(y_pred_tmp))))
loss = loss_func(torch.Tensor(y_obs_tmp), torch.Tensor(y_pred_tmp),
                 0.01,
                 dummy_net.parameters())
print('The loss with regularization: {0:1.2f}'.format(loss.item()))
```

```
The loss without regularization: 2.16
This should be the same as this: 2.16
The loss with regularization: 2.23
```

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

- standarize the inputs and output or not
- experiment with various levels of regularization
- change the learning rate of the stochatic optimization algorithm
- change the batch size of the optimization algorithm
- change the number of epochs (how many times the optimization algorithm does a complete sweep through all the data.

Answer:

[29]:
```
# We are going to start by creating a class that encapsulates a regression
# network so that we can turn on or off input/output standarization
# without too much fuss.
# The class will essentially represent a trained network model.
# It will "know" whether or not during training we standarized the data.
```

```
# I am not asking you to do anything here, so you may just run this code segment
# or read through if you want to know about the details.
from sklearn.preprocessing import StandardScaler

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

    Parameters:

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

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

    def __call__(self, X):
        """
        Evaluates the model at X.
        """
        # If not scaled, then the model is just net(X)
        if not self.standarized:
            return self.net(X)
        # Otherwise:
        # Scale X:
        X_scaled = self.feature_scaler.transform(X)
        # Evaluate the network output - which is also scaled:
        y_scaled = self.net(torch.Tensor(X_scaled))
        # Scale the output back:
        y = self.target_scaler.inverse_transform(y_scaled.detach().numpy())
        return y
```

[30]:
```
# Go through the code that follows and fill in the missing parts
from sklearn.model_selection import train_test_split
```

```
# We need this for a progress bar:
from tqdm import tqdm

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

    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).
    standarize -    Whether or not you want to standarize the features and the␣
↪targets.
    """
    # Split the data
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33)

    # Standarize the data
    if standarize:
        # Build the scalers
        feature_scaler = StandardScaler().fit(X)
        target_scaler = StandardScaler().fit(y)
        # Get scaled versions of the data
        X_train_scaled = feature_scaler.transform(X_train)
        y_train_scaled = target_scaler.transform(y_train)
        X_test_scaled = feature_scaler.transform(X_test)
        y_test_scaled = target_scaler.transform(y_test)
    else:
        feature_scaler = None
        target_scaler = None
        X_train_scaled = X_train
        y_train_scaled = y_train
        X_test_scaled = X_test
        y_test_scaled = y_test

    # Turn all the numpy arrays to torch tensors
```

```
    X_train_scaled = torch.Tensor(X_train_scaled)
    X_test_scaled = torch.Tensor(X_test_scaled)
    y_train_scaled = torch.Tensor(y_train_scaled)
    y_test_scaled = torch.Tensor(y_test_scaled)

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

    # Create an Adam optimizing object for the neural network `net`
    # with learning rate `lr`
    optimizer = torch.optim.Adam(net.parameters(), lr=lr)

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

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


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

    # Make a TrainedModel
    trained_model = TrainedModel(net, standarized=standarize,
```

```
                                feature_scaler=feature_scaler,
                                target_scaler=target_scaler)

    # Make sure that we return properly scaled

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

Use this to test your code:

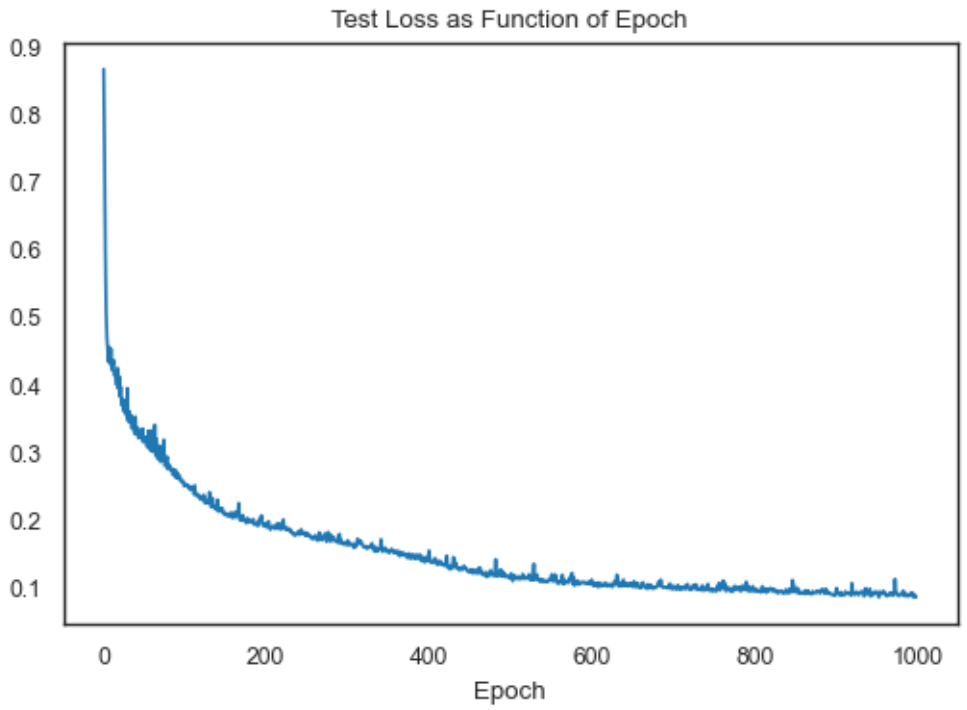
[31]:
```
# A simple one-layer network with 10 neurons
net = nn.Sequential(nn.Linear(5, 20),
                    nn.Sigmoid(),
                    nn.Linear(20, 1))
epochs = 1000
lr = 0.01
reg_weight = 0
n_batch = 100
model, test_loss, X_train, y_train, X_test, y_test = train_net(X, y, net,␣
 ↪reg_weight, n_batch, epochs, lr)
```

```
100%|      | 1000/1000 [00:11<00:00, 84.23it/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:

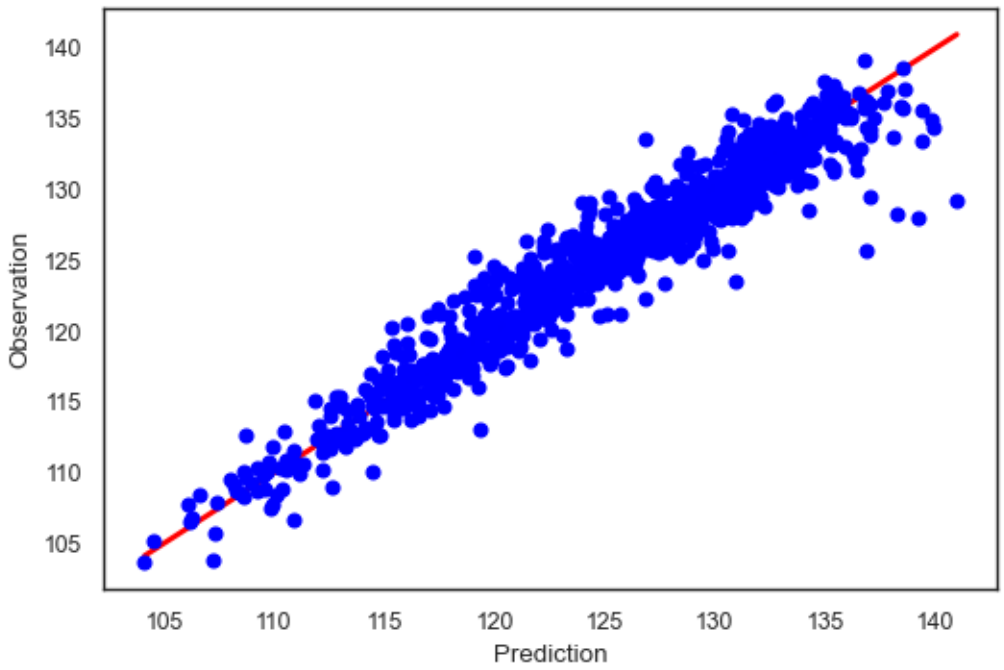
[32]:
```
fig, ax = plt.subplots(dpi=100)
ax.set_title('Test Loss as Function of Epoch')
ax.set_xlabel('Epoch')
ax.plot([i for i in range(epochs)], test_loss, label="Test Loss")
```

[32]:
```
[<matplotlib.lines.Line2D at 0x7f80185abe20>]
```



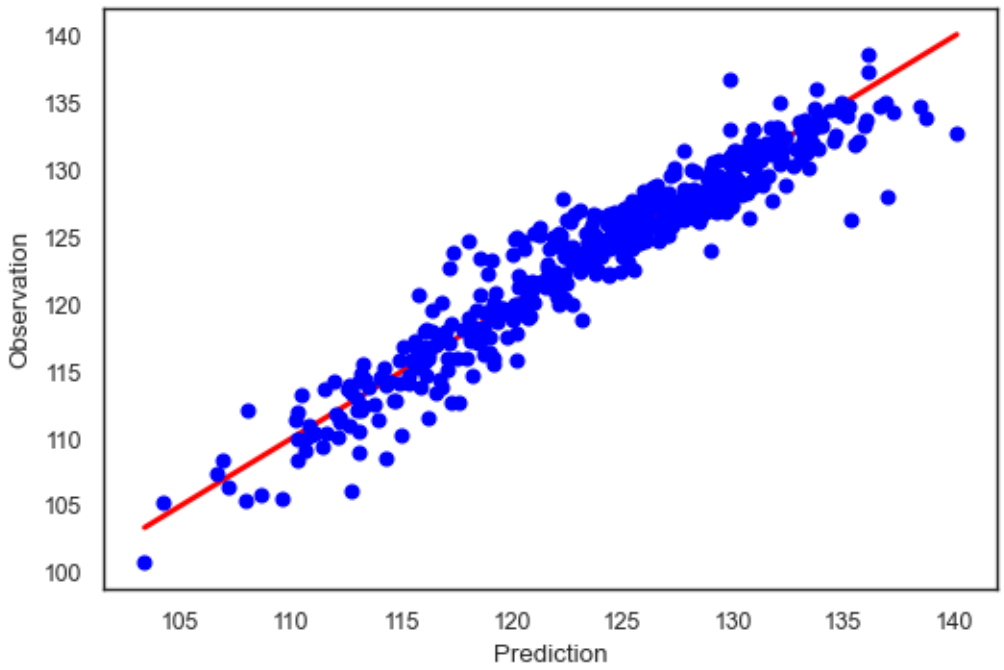
Now plot the observations vs predictions plot for the training data:

[33]:
```
fig, ax = plt.subplots(dpi=100)
y_range = np.linspace(y_train.min(), y_train.max(), 100)
ax.plot(y_range, y_range, 'r', lw=2)
ax.plot(y_train, model(X_train), 'bo')
ax.set_xlabel('Prediction')
ax.set_ylabel('Observation');
```



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

[34]:
```
fig, ax = plt.subplots(dpi=100)
y_range = np.linspace(y_test.min(), y_test.max(), 100)
ax.plot(y_range, y_range, 'r', lw=2)
ax.plot(y_test, model(X_test), 'bo')
ax.set_xlabel('Prediction')
ax.set_ylabel('Observation');
```



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?

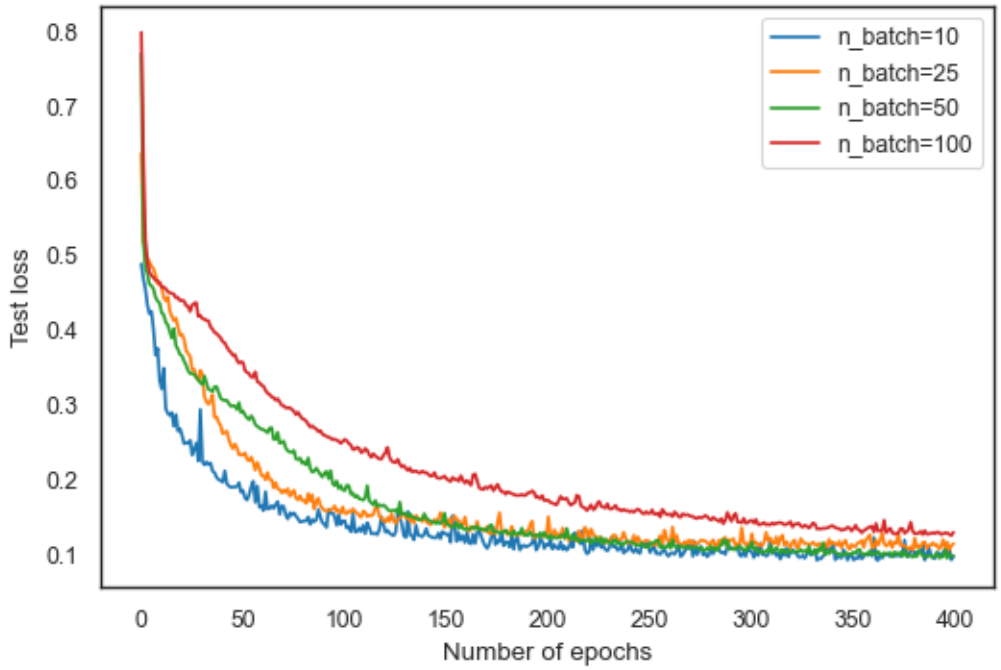
Answer:

[35]:
```
epochs = 400
lr = 0.01
reg_weight = 1e-12
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)
```

```
  0%|          | 2/400 [00:00<00:32, 12.15it/s]
Training n_batch: 10
```

```
100%|      | 400/400 [00:27<00:00, 14.64it/s]
  1%|          | 3/400 [00:00<00:13, 29.73it/s]

Training n_batch: 25

100%|      | 400/400 [00:12<00:00, 33.11it/s]
  2%|          | 6/400 [00:00<00:07, 52.37it/s]

Training n_batch: 50

100%|      | 400/400 [00:07<00:00, 56.66it/s]
  2%|          | 9/400 [00:00<00:04, 81.90it/s]

Training n_batch: 100

100%|      | 400/400 [00:04<00:00, 89.57it/s]
```

[36]:
```
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 observsations about the batch size here **Answer** Fastest training time is caused by **n_batch=100**. This is because the training process is using 100 training samples before it updates its parameters, which is less computationally expensive than updating every 10, 25, or 50 training samples. This allows the model to get through an epoch more quickly. I would choose a batch size

based on my target test loss and desired training time. A larger batch size lets the model train faster (get through an epoch more quickly), but the change in test loss is smaller between each epoch (because fewer updates are being made).

In this case, **n_batch=10** gives the smallest final test loss, so I will use that.

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

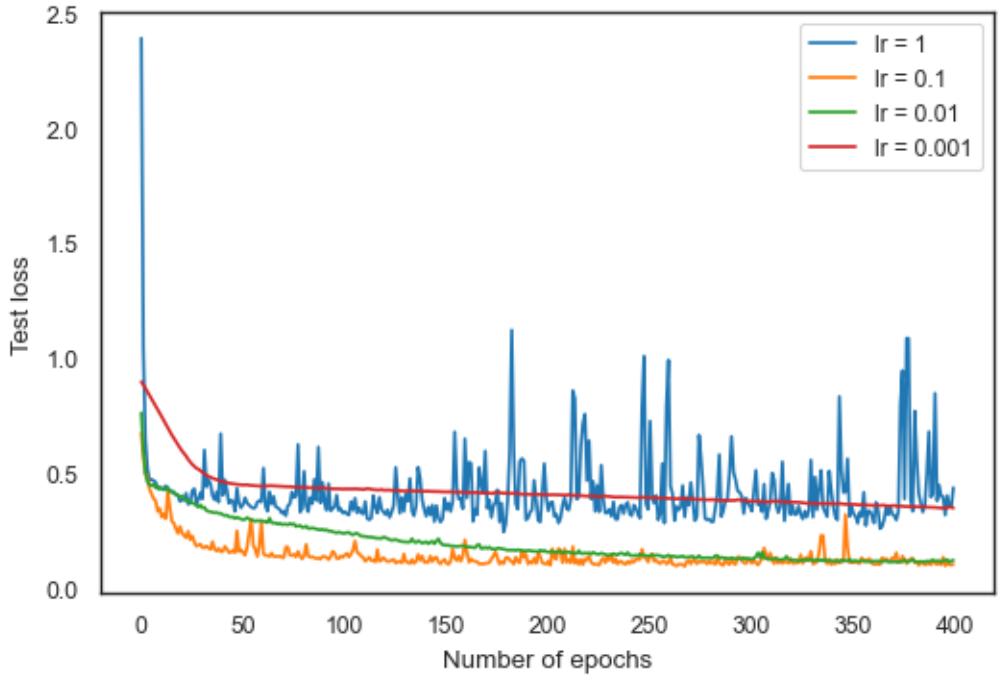
Answer

[37]:
```
epochs = 400
lrs = [1, 0.1, 0.01, 0.001]
reg_weight = 1e-12
test_losses = []
models = []
batches = 10
for lr in lrs:
    print('Training with learning rate: {}'.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)
```

```
  2%|          | 7/400 [00:00<00:05, 65.74it/s]

Training with learning rate: 1

100%|      | 400/400 [00:04<00:00, 81.40it/s]
  2%|          | 8/400 [00:00<00:05, 74.50it/s]

Training with learning rate: 0.1

100%|      | 400/400 [00:04<00:00, 82.91it/s]
  2%|          | 9/400 [00:00<00:04, 83.56it/s]

Training with learning rate: 0.01

100%|      | 400/400 [00:04<00:00, 82.14it/s]
  2%|          | 8/400 [00:00<00:05, 72.83it/s]

Training with learning rate: 0.001

100%|      | 400/400 [00:04<00:00, 82.89it/s]
```

[38]:
```
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');
```



Write your observsations about the learning rate here **Answer** The algorithm seems to converge for all but **lr = 1.0**. I would choose a learning rate **lr=0.01**, because it converges to the lowest training 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

[39]:
```
epochs = 400
lr = 0.01
reg_weights = [0, 1e-16, 1e-12, 1e-6, 1e-3]
test_losses = []
models = []
batches = 10
for reg_weight in reg_weights:
    print('Training with 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)
```

```
  2%|          | 7/400 [00:00<00:05, 67.04it/s]
```

```
Training with reg_weight: 0
```

```
100%|      | 400/400 [00:04<00:00, 80.69it/s]
  2%|          | 8/400 [00:00<00:04, 79.14it/s]
```

```
Training with reg_weight: 1e-16
```

```
100%|      | 400/400 [00:04<00:00, 88.94it/s]
  2%|          | 9/400 [00:00<00:04, 84.03it/s]
```

```
Training with reg_weight: 1e-12
```

```
100%|      | 400/400 [00:04<00:00, 87.98it/s]
  2%|          | 8/400 [00:00<00:05, 70.45it/s]
```

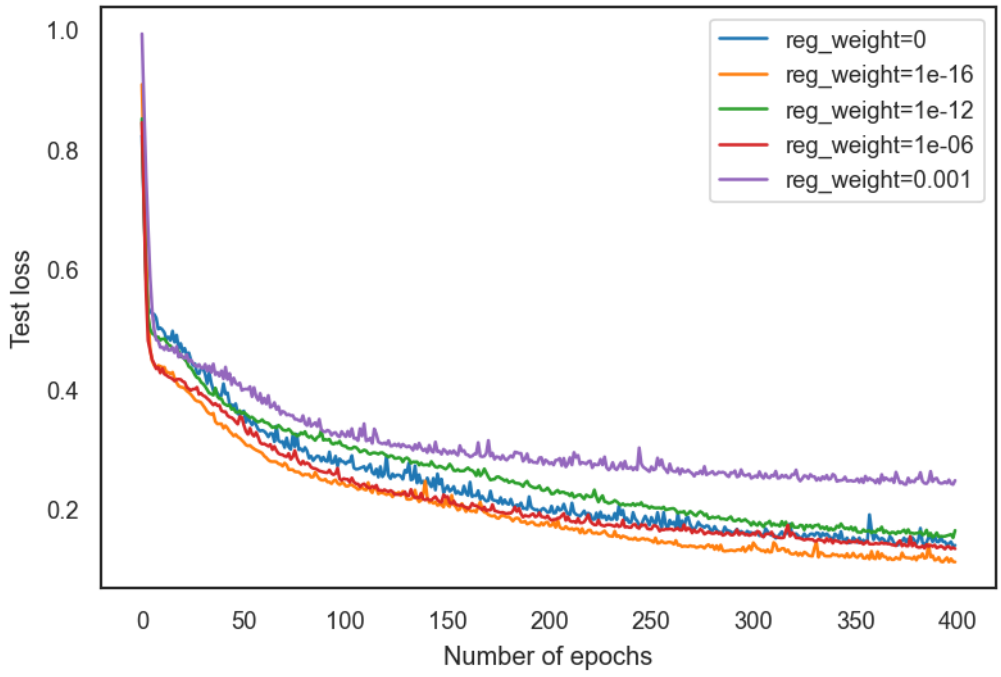
```
Training with reg_weight: 1e-06
```

```
100%|      | 400/400 [00:04<00:00, 82.59it/s]
  2%|          | 9/400 [00:00<00:04, 82.91it/s]
```

```
Training with reg_weight: 0.001
```

```
100%|      | 400/400 [00:04<00:00, 88.92it/s]
```

[40]:
```
fig, ax = plt.subplots(dpi=150)
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');
```



Write your observsations about the regularization weights here **Answer** Small regularization weights close to **reg_weight = 1e-16** seem to cause the lowest test loss. Large reg_weight values will be multiplied with the l2 norm of the hyperparameters and thus create a larger test loss, so small reg_weights perform better. We can see this with **reg_weight = 0.001**, which yields a relatively high final test loss. **reg_weight = 0** is also not ideal, because then there is no penalty for the hyperparameters becoming very large.

Part D.I - Train a bigger network Now that you have developed some intuition about the parameters involved in training a network, train a larger one. In particular, use a 5-layer deep network with 100 neurons per layer. You can use the sigmoid activation function or you can change it to something else. Make sure you plot: - the evolution of the test loss a a function of the epochs - the observations vs predictions plot for the test data

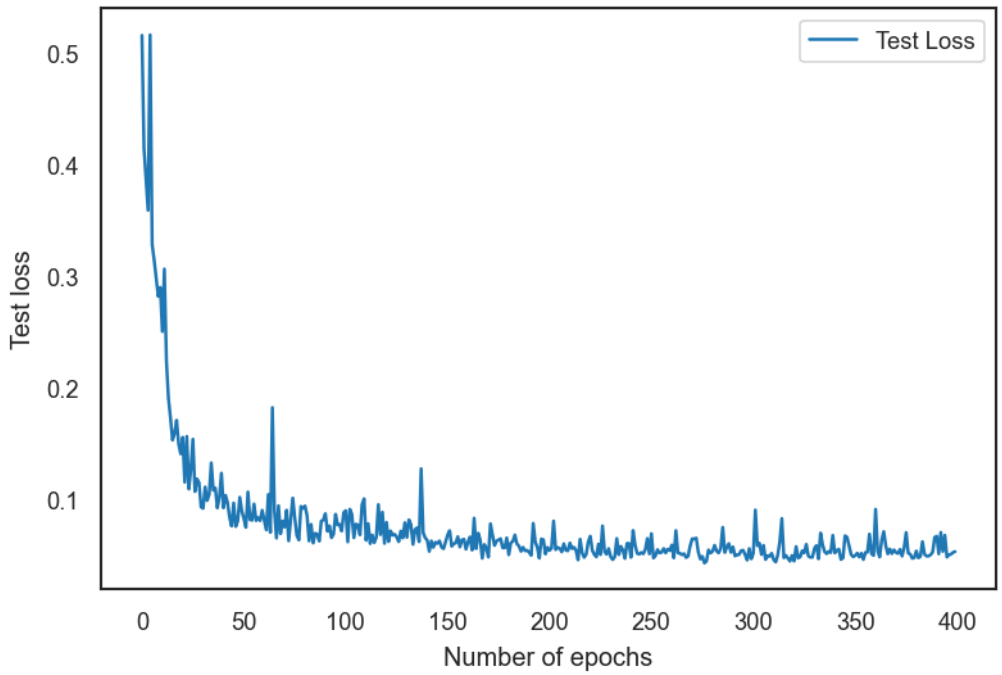
Answer:

[67]:
```
epochs = 400
lr = 0.01
reg_weight = 1e-16
batches = 10
net = nn.Sequential(nn.Linear(5, 100),
                    nn.Sigmoid(),
                    nn.Linear(100,100),
                    nn.Sigmoid(),
                    nn.Linear(100,100),
                    nn.Sigmoid(),
```

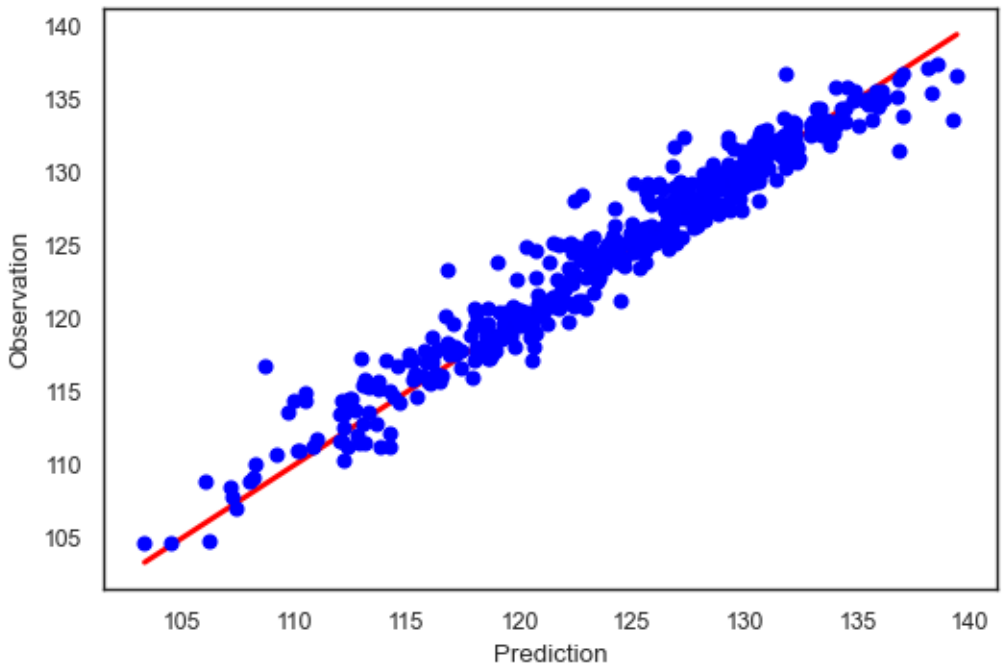
```
                    nn.Linear(100,100),
                    nn.Sigmoid(),
                    nn.Linear(100, 1))
model, test_loss, X_train, y_train, X_test, y_test = train_net(X, y, net,␣
 ↪reg_weight, batches, epochs, lr)
```

```
100%|      | 400/400 [01:10<00:00,  5.69it/s]
```

[68]:
```
fig, ax = plt.subplots(dpi=150)
ax.plot(test_loss, label='Test Loss')
ax.set_xlabel('Number of epochs')
ax.set_ylabel('Test loss')
plt.legend(loc='best');
```



[69]:
```
fig, ax = plt.subplots(dpi=100)
y_range = np.linspace(y_test.min(), y_test.max(), 100)
ax.plot(y_range, y_range, 'r', lw=2)
ax.plot(y_test, model(X_test), 'bo')
ax.set_xlabel('Prediction')
ax.set_ylabel('Observation');
```



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

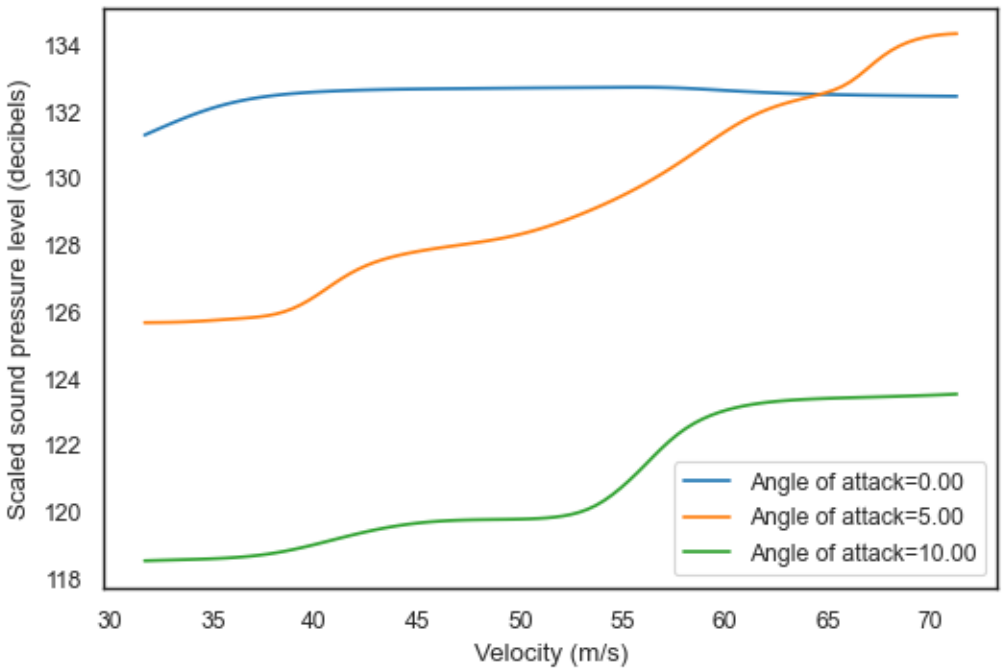
Answer:

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

[70]:
```
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)')
```

[71]:
```
fig, ax = plt.subplots(dpi=100)
for aofa in [0, 5, 10]:
    plot_sound_level_as_func_of_stream_vel(angle_of_attack=aofa, ax=ax,
                                           label='Angle of attack={0:1.2f}'.
 ↪format(aofa))
plt.legend(loc='best');
```



1.5 Problem 2 - Classification with DNNs

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

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

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

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

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

[72]:
```
# !curl -O 'https://dl.dropboxusercontent.com/s/n8dczk7t8bx0pxi/
↪human_activity_data.npz'
```

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  203M  100  203M    0     0   646k      0  0:05:21  0:05:21 --:--:--  460k
4 8807k    0     0   433k      0  0:07:59  0:00:20  0:07:39  323k 203M    7
15.7M    0     0   514k      0  0:06:44  0:00:31  0:06:13  688k 203M   18 38.5M
0     0   538k      0  0:06:26  0:01:13  0:05:13  781k203M   23 46.9M    0     0
569k      0  0:06:04  0:01:24  0:04:40  720k 203M   36 73.3M    0     0   609k
0  0:05:41  0:02:03  0:03:38  849k 39 80.5M    0     0   604k      0  0:05:44
0:02:16  0:03:28  456k5 91.8M    0     0   634k      0  0:05:27  0:02:28
0:02:59 1153k72  147M    0     0   700k      0  0:04:56  0:03:36  0:01:20  382kM
78  158M    0     0   703k      0  0:04:55  0:03:51  0:01:04  761k  0:01:02
742k  84  171M    0     0   692k      0  0:05:00  0:04:13  0:00:47  566k 88
180M    0     0   690k      0  0:05:01  0:04:27  0:00:34  778k
```

Here is how to load the data:

[73]:
```
data = np.load('human_activity_data.npz')
```

This is a Python dictionary that contains the following entries:

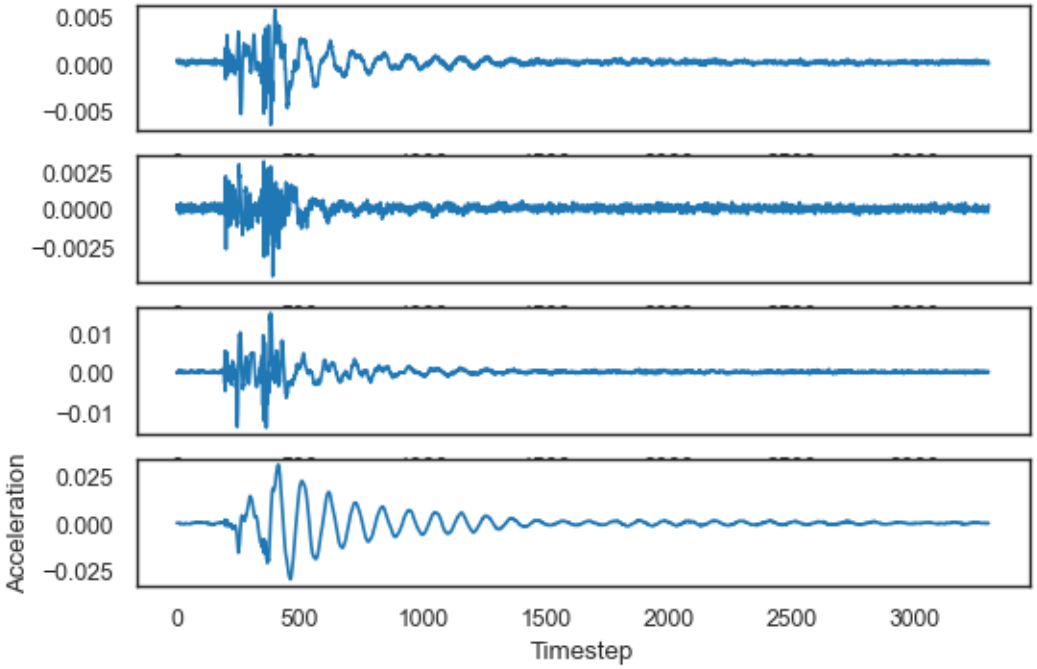
[74]:
```
for key in data.keys():
    print(key, ':', data[key].shape)
```

```
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:

[75]:
```
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');
```



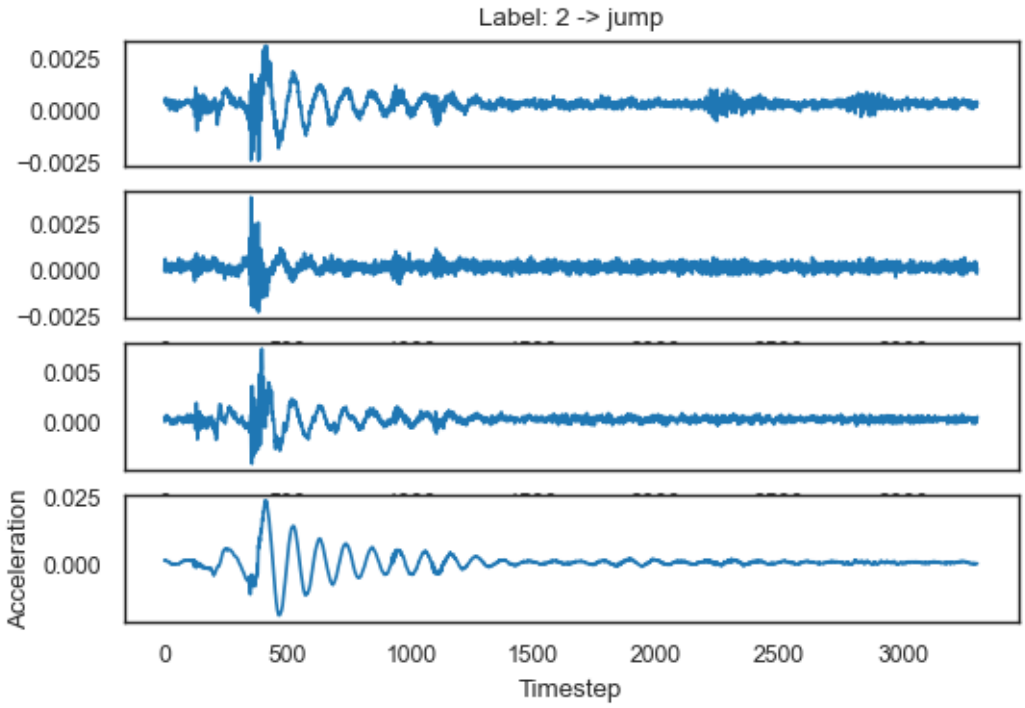
The second key, `labels_1`, is a bunch of integers ranging from 0 to 2 indicating whether the entry corresponds to a "bag," a "ball" or a "jump." For your reference, the correspondence is:

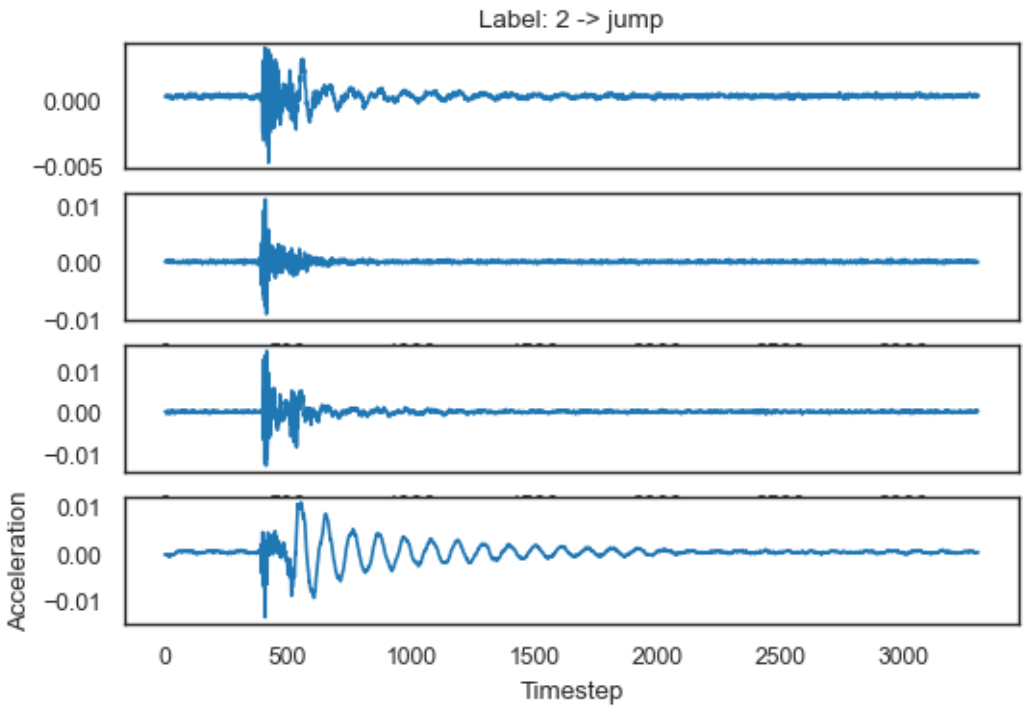
[76]:
```
LABELS_1_TO_TEXT = {
    0: 'bag',
    1: 'ball',
    2: 'jump'
}
```

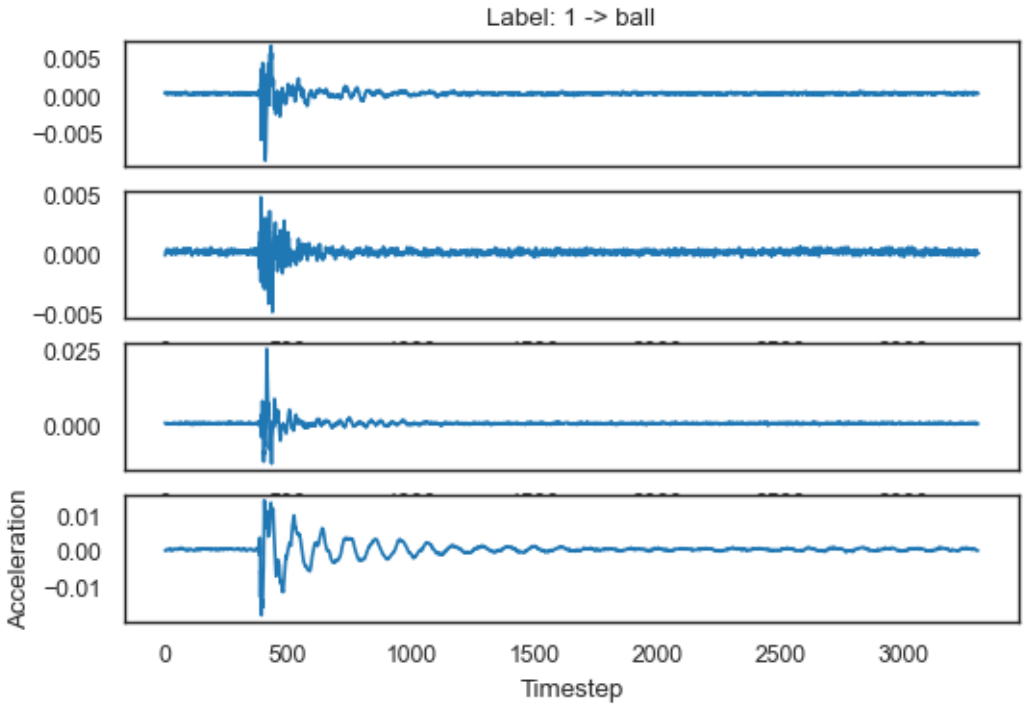
And here are a few examples:

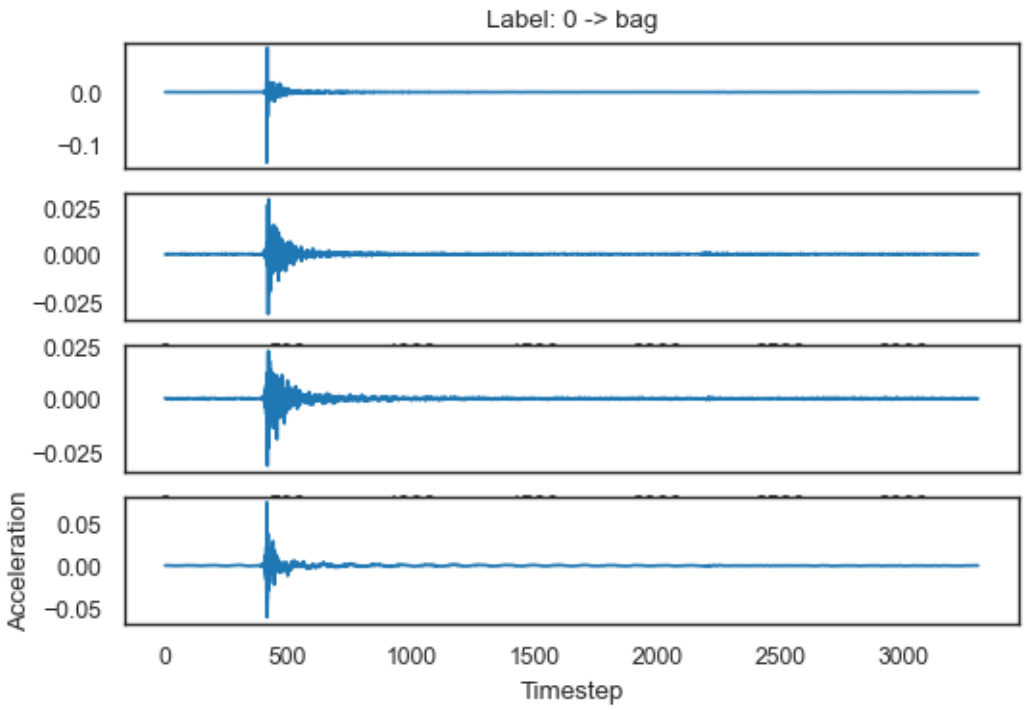
[77]:
```
for _ in range(5):
    i = np.random.randint(0, data['features'].shape[0])
```

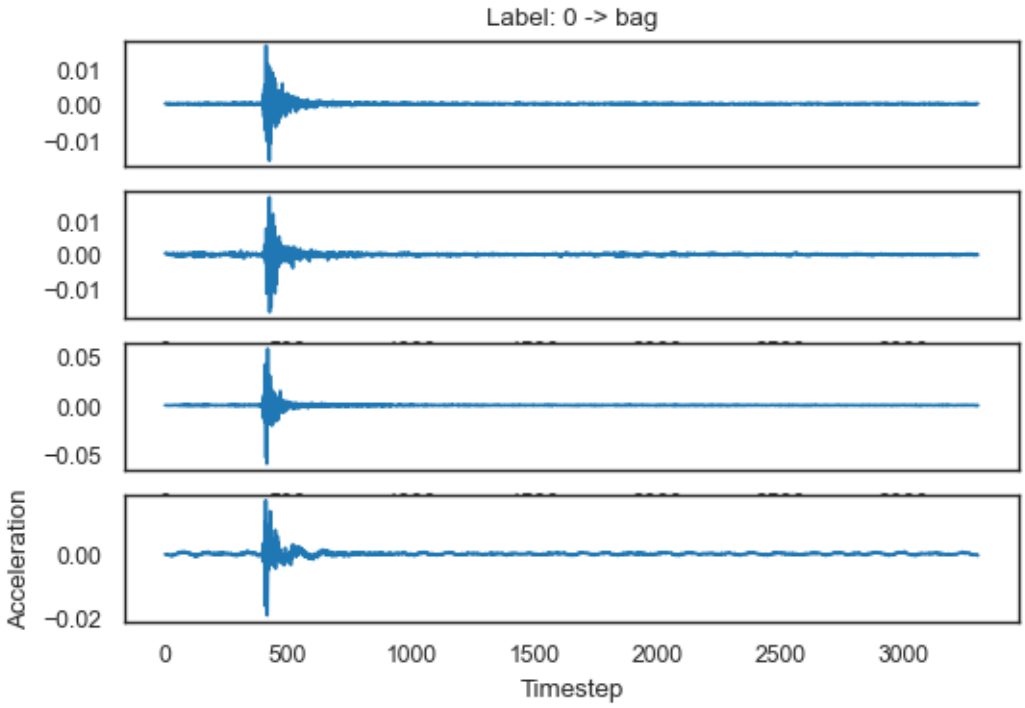
```
    fig, ax = plt.subplots(4, 1, dpi=100)
    for j in range(4):
        ax[j].plot(data['features'][i, j])
    ax[-1].set_xlabel('Timestep')
    ax[-1].set_ylabel('Acceleration')
    ax[0].set_title('Label: {0:d} -> {1:s}'.format(data['labels_1'][i],
                                            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




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





The array `labels_2` includes integers from 0 to 6 indicating the detailed label of the experiment.

The correspondence between integers and text labels is:

[78]:
```
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:

[79]:
```
# The features
X = data['features']
# The labels_1
y1 = data['labels_1']
# The labels_2
y2 = data['labels_2']
# The locations
y3 = data['loc_ids']
```

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

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

Answer:

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

[83]:
```
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
```

[84]:
```
# You can make the network like this:
net = Net(3)
```

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

[85]:
```
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.

[86]:
```
def train_cnn(X, y, net, n_batch, epochs, lr, test_size=0.33):
    """
    A function that trains a regression neural network using stochatic gradient
    descent and returns the trained network. The loss function being minimized␣
 ↪is
    `loss_func`.

    Parameters:

    X           -    The observed features
    y           -    The observed targets
```

```
    net         -    The network you want to fit
    n_batch     -    The batch size you want to use for stochastic optimization
    epochs      -    How many times do you want to pass over the training␣
↪dataset.
    lr          -    The learning rate for the stochastic optimization algorithm.
    test_size   -    What percentage of the data should be used for testing␣
↪(validation).
    """
    # Split the data
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33)

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

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

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

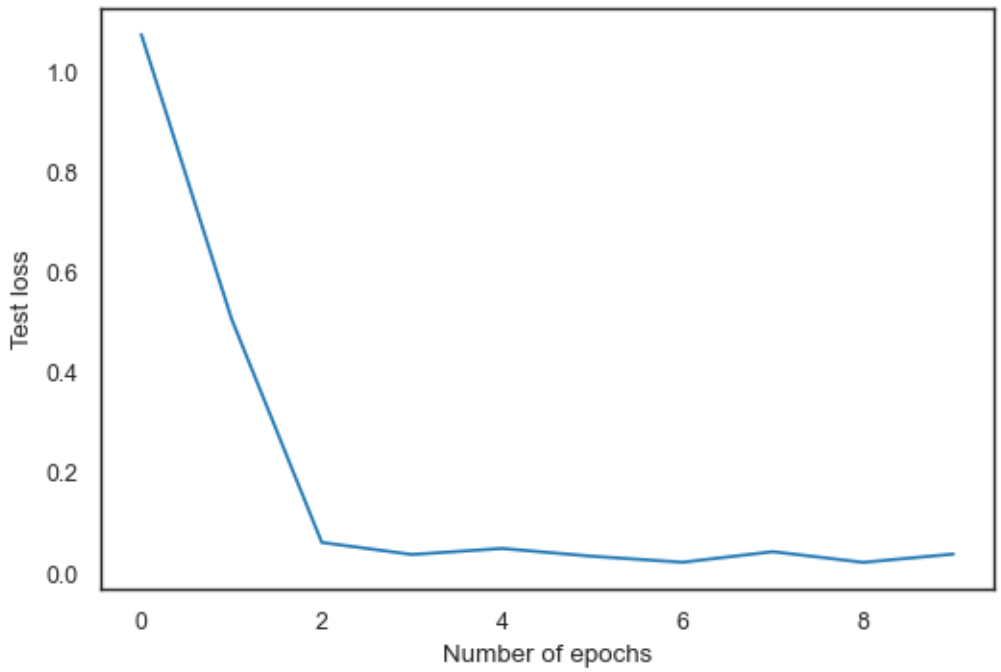
Now experiment with the epochs, the learning rate, and the batch size until this works.

[87]:
```
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 = 0.43190%
Epoch 2: accuracy = 0.70805%
Epoch 3: accuracy = 0.98345%
Epoch 4: accuracy = 0.98871%
Epoch 5: accuracy = 0.99097%
Epoch 6: accuracy = 0.99398%
Epoch 7: accuracy = 0.99624%
Epoch 8: accuracy = 0.99473%
Epoch 9: accuracy = 0.99774%
Epoch 10: accuracy = 0.99248%
```

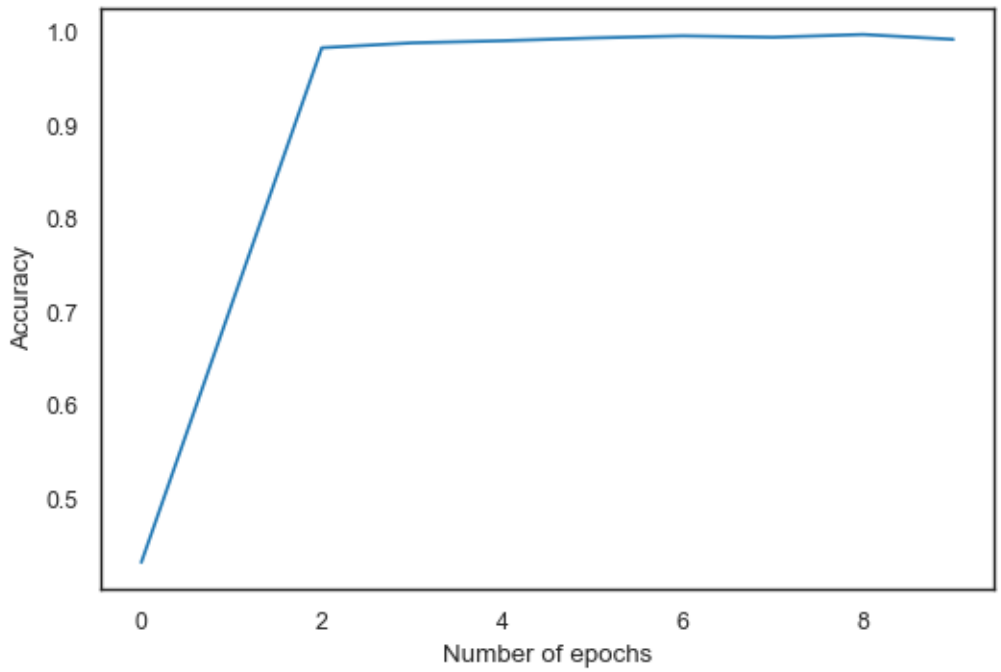
Plot the evolution of the test loss as a function of epochs.

[88]:
```
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.

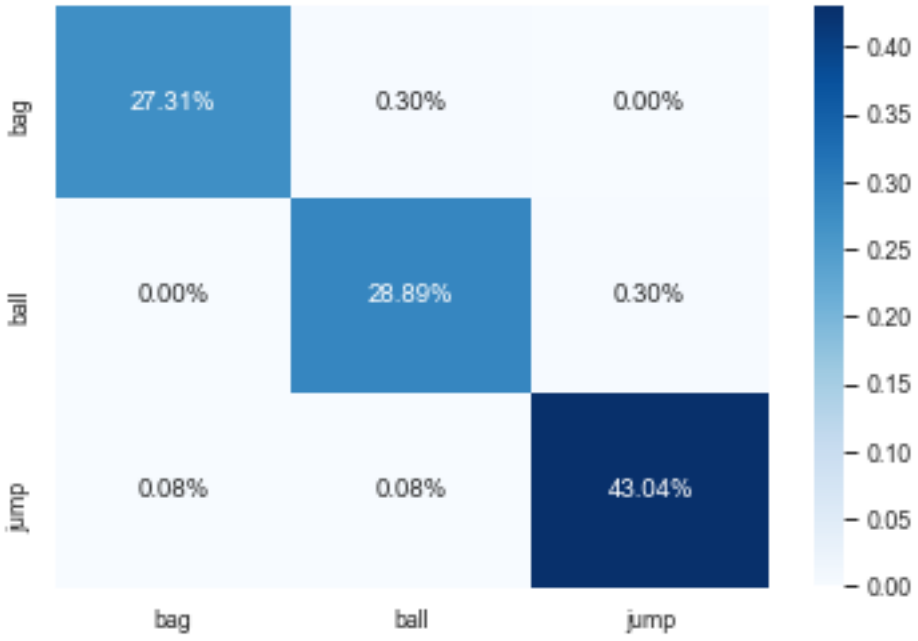
[89]:
```
fig, ax = plt.subplots(dpi=100)
ax.plot(accuracy)
ax.set_xlabel('Number of epochs')
ax.set_ylabel('Accuracy');
```



Plot the confusion matrix.

[90]:
```
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)
```

[91]:
```
sns.heatmap(cf_matrix/np.sum(cf_matrix), annot=True,
            fmt='.2%', cmap='Blues',
            xticklabels=LABELS_1_TO_TEXT.values(),
            yticklabels=LABELS_1_TO_TEXT.values());
```



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

Repeat what you did above for `y2`.

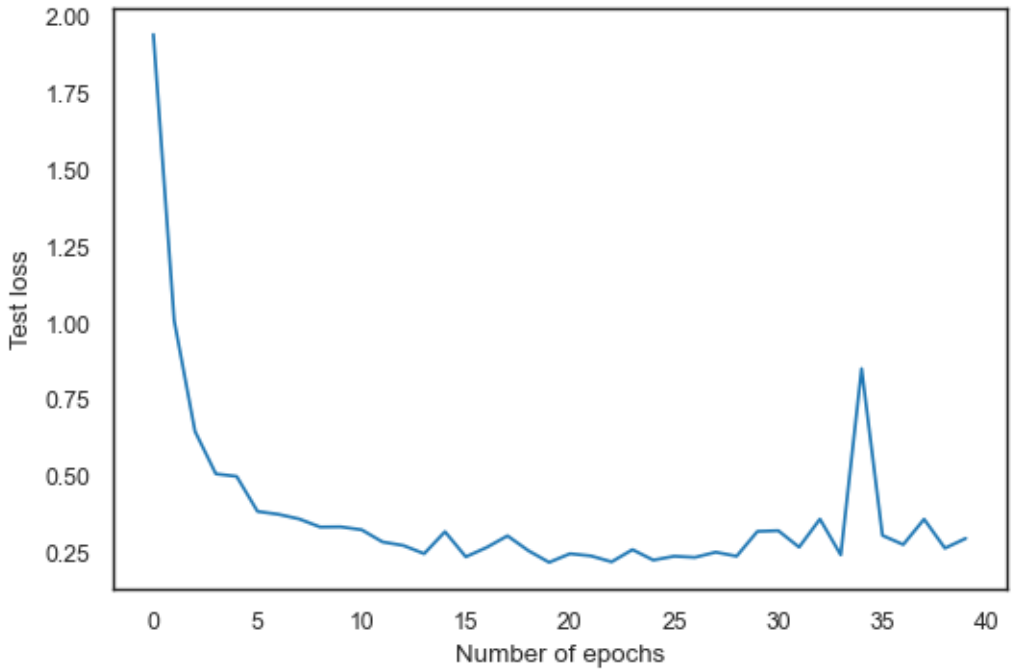
Answer:

[94]:
```
# You can make the network like this:
net = Net(7)

epochs = 40
lr = 0.01
n_batch = 50
trained_model, test_loss, accuracy, X_train, y_train, X_test, y_test =␣
 ↪train_cnn(X, y2, net, n_batch, epochs, lr)
```

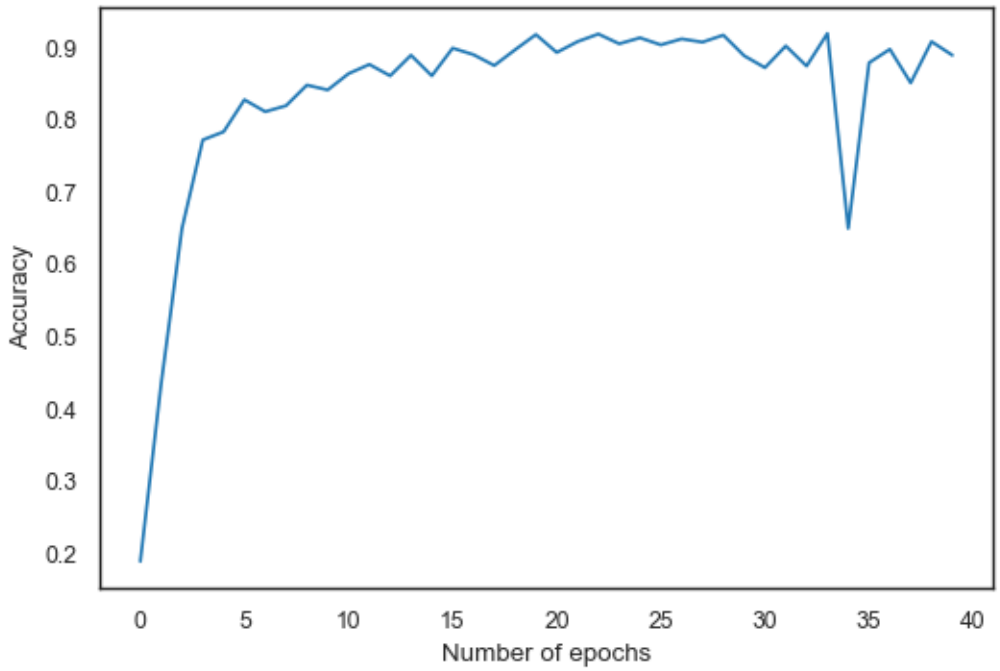
```
Epoch 1: accuracy = 0.18886%
Epoch 2: accuracy = 0.43491%
Epoch 3: accuracy = 0.65011%
Epoch 4: accuracy = 0.77276%
Epoch 5: accuracy = 0.78405%
Epoch 6: accuracy = 0.82844%
Epoch 7: accuracy = 0.81189%
Epoch 8: accuracy = 0.82017%
Epoch 9: accuracy = 0.84876%
Epoch 10: accuracy = 0.84199%
Epoch 11: accuracy = 0.86456%
```

```
Epoch 12: accuracy = 0.87735%
Epoch 13: accuracy = 0.86155%
Epoch 14: accuracy = 0.89014%
Epoch 15: accuracy = 0.86155%
Epoch 16: accuracy = 0.89992%
Epoch 17: accuracy = 0.89090%
Epoch 18: accuracy = 0.87585%
Epoch 19: accuracy = 0.89767%
Epoch 20: accuracy = 0.91874%
Epoch 21: accuracy = 0.89391%
Epoch 22: accuracy = 0.90895%
Epoch 23: accuracy = 0.91949%
Epoch 24: accuracy = 0.90594%
Epoch 25: accuracy = 0.91422%
Epoch 26: accuracy = 0.90444%
Epoch 27: accuracy = 0.91272%
Epoch 28: accuracy = 0.90820%
Epoch 29: accuracy = 0.91798%
Epoch 30: accuracy = 0.88939%
Epoch 31: accuracy = 0.87284%
Epoch 32: accuracy = 0.90293%
Epoch 33: accuracy = 0.87509%
Epoch 34: accuracy = 0.92024%
Epoch 35: accuracy = 0.65011%
Epoch 36: accuracy = 0.87961%
Epoch 37: accuracy = 0.89842%
Epoch 38: accuracy = 0.85177%
Epoch 39: accuracy = 0.90895%
Epoch 40: accuracy = 0.89014%
```

[95]:
```
fig, ax = plt.subplots(dpi=100)
ax.plot(test_loss)
ax.set_xlabel('Number of epochs')
ax.set_ylabel('Test loss');
```

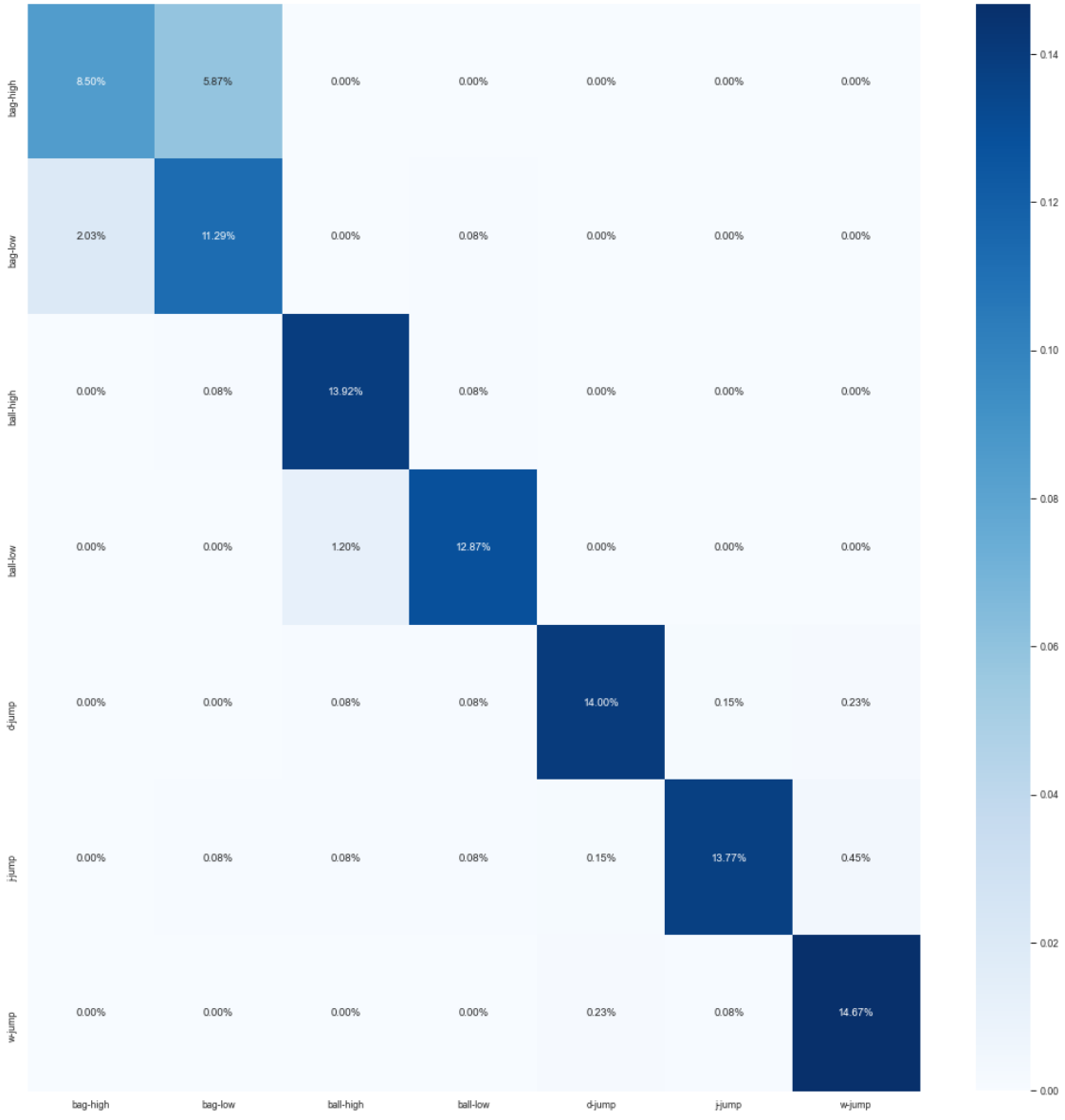


[96]:
```
fig, ax = plt.subplots(dpi=100)
ax.plot(accuracy)
ax.set_xlabel('Number of epochs')
ax.set_ylabel('Accuracy');
```



[97]:
```
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)
```

[98]:
```
fig, ax = plt.subplots(figsize=(18,18))
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
8.50%
5.87%
0.00%
0.00%
0.00%
0.00%
0.00%
2.03%
11.29%
0.00%
0.08%
0.00%
0.00%
0.00%
0.00%
0.08%
13.92%
0.08%
0.00%
0.00%
0.00%
0.00%
0.00%
1.20%
12.87%
0.00%
0.00%
0.00%
0.00%
0.00%
0.08%
0.08%
14.00%
0.15%
0.23%
0.00%
0.08%
0.08%
0.08%
0.15%
13.77%
0.45%
0.00%
0.00%
0.00%
0.00%
0.23%
0.08%
14.67%
bag-high
bag-low
ball-high
ball-low
d-jump
j-jump
w-jump
0.14
0.12
0.10
0.08
0.06
0.04
0.02
0.00

[ ]: