Homework 7

Contents

References

- Lectures 24-26 (inclusive).

Instructions

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

▶ Show code cell source

Student details

- **First Name:**

Skip to main content

- **Email:**

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/a
```

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],
       ...,
       [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:

```
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	Sound
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	

1503 rows × 6 columns

Skip to main content

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:

```
# Your code here
```

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:

```
# Your code here
```

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:

```
# Your code here
```

Skip to main content

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 = # your code here
```

```
# 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)
```

```
# 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.
```

Skip to main content

```
    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
    all parameters.
    """
    # Your code here
    raise NotImplementedError('Implement me and delete this line')
```

```
# 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_
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()))
```

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

Skip to main content

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=No
        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.
        """
```

Skip to main content

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

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

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

    Arguments:

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

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

Skip to main content

```
    # 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`
    raise NotImplementedError('Define the optimizer object! Delete me then!')
    optimizer = # your code here

    # 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
            # Your code here
            raise NotImplementedError('Write stochastic gradient step code! Delete me

        # 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
```

Skip to main content

```
# 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
)
```

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:

```
# Your code here
```

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

```
# Your code here
```

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

```
# Your code here
```

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:

Skip to main content

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

```
fig, ax = plt.subplots(dpi=100)
for tl, n_batch in zip(test_losses, batches):
    ax.plot(tl, label='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

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:

```
# your code here
```

Skip to main content

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:

```
# Your code here
```

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:

```
# your code here
```

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.

Skip to main content

```
best_model = # set this equal to your best model

def plot_sound_level_as_func_of_stream_vel(
    freq=2500,
    angle_of_attack=10,
    chord_length=0.1,
    suc_side_disp_thick=0.01,
    ax=None,
    label=None
):

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

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

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

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

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

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

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

Problem 2 - Classification with DNNs

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

Skip to main content

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)
```

Let's go over these one by one. First, the `features`. These are the accelertion sensor

Skip to main content

```
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'][
```

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

Skip to main content

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']
```

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
```

Skip to main content

```
        self.fc1 = nn.Linear(16 * 131, 200)
        self.fc2 = nn.Linear(200, 50)
        self.fc3 = nn.Linear(50, num_labels)

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

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

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

```
cnn_loss_func = # your code here
```

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.

```
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 (validatio
    """
```

Skip to main content

```
    # 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`
    raise NotImplementedError('Define the optimizer object! Delete me then!')
    optimizer = # your code here

    # 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
            raise NotImplementedError('Write stochastic gradient step code! Delete me

        # 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
```

Skip to main content

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

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

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

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

Part B - Train a CNN to predict the the low-level type

Skip to main content

of observation (bag-high, bag-low, etc.)

Repeat what you did above for `y2`.

Answer:

```
# your code here
```