HW Autoencoders

Overview

In this homework, you will get introduced to Autoencoders, a group of architectures used for encoding compact representations of model inputs and then reconstructing them. This has a variety of real-world use cases such as compression, pre-training encoder modules, and more. It is also closely related to the Variational Autoencoder models that we will see later and which can be used to generate new synthetic data.

More specifically, you will implement a vanilla and then a stacked autoencoder model. Then, you will train each on **Heart Failure Prediction** and compare the results.

17 EXERCISES

About Raw Data

Pneumonia is a lung disease characterized by inflammation of the airspaces in the lungs, most commonly due to an infection. In this section, you will train a CNN model to classify Pneumonia disease (Pneumonia/Normal) based on chest X-Ray images.

The chest X-ray images (anterior-posterior) were selected from retrospective cohorts of pediatric patients of one to five years old. All chest X-ray imaging was performed as part of patients' routine clinical care. You can refer to this link30154-5) for more information.

In [1]:

```python
### Import all the libraries used
import os
import csv
import pickle
import random
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import pandas as pd
import time

### Set random seed
seed = 24
random.seed(seed)
np.random.seed(seed)
torch.manual_seed(seed)
os.environ["PYTHONHASHSEED"] = str(seed)


# record start time
_START_RUNTIME = time.time()

# Define data and weight path
DATA_PATH = "../HW4_Autoencoder-lib/data"
```

In [2]:

```python
!ls ../HW4_Autoencoder-lib/data
```

1 Load and Visualize the Data [10 points]

The data is under `DATA_PATH`. In this part, you are required to load the data into the data loader, and calculate some statistics.

In [3]:

```python
#input
# folder: str, 'train', 'val', or 'test'
#output
# number_normal: number of normal samples in the given folder
# number_pneumonia: number of pneumonia samples in the given folder
def get_count_metrics(folder, data_path=DATA_PATH):

    '''
    TODO: Implement this function to return the number of normal and pneumonia samples
          Hint: !ls $DATA_PATH
    '''

    normal_ls = os.listdir(os.path.join(data_path, folder, 'NORMAL'))
    pneumonia_ls = os.listdir(os.path.join(data_path, folder, 'PNEUMONIA'))
    return len(normal_ls), len(pneumonia_ls)


#output
# train_loader: train data loader (type: torch.utils.data.DataLoader)
# val_loader: val data loader (type: torch.utils.data.DataLoader)
def load_data(data_path=DATA_PATH):

    '''
    TODO: Implement this function to return the data loader for
    train and validation dataset. Set batchsize to 32.

    You should add the following transforms (https://pytorch.org/docs/stable/torchvis
        1. transforms.RandomResizedCrop: the images should be cropped to 224 x 224
        2. transforms.RandomResizedCrop: the images should be compressed to 24 x 24
        3. transforms.ToTensor: just to convert data/labels to tensors
        4. flatten_transform: to flatten the images away from their 3 x 24 x 24 repres
    You should set the *shuffle* flag for *train_loader* to be True, and False for *va

    HINT: Consider using `torchvision.datasets.ImageFolder`.
    '''

    import torchvision
    import torchvision.datasets as datasets
    import torchvision.transforms as transforms

    flatten_transform = transforms.Lambda(lambda x: torch.flatten(x))
    transforms = transforms.Compose([
        transforms.RandomResizedCrop((224, 224)),
        transforms.RandomResizedCrop((24, 24)),
        transforms.ToTensor(),
        flatten_transform,
    ])

    train_dataset = torchvision.datasets.ImageFolder(root=os.path.join(data_path, 'tra
    val_dataset = torchvision.datasets.ImageFolder(root=os.path.join(data_path, 'val')

    BATCH_SIZE = 32

    train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=BATCH_SIZE, s
    val_loader = torch.utils.data.DataLoader(val_dataset, batch_size=BATCH_SIZE, shuf


    return train_loader, val_loader
```

In [4]:

```python
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''

assert type(get_count_metrics('train')) is tuple
assert type(get_count_metrics('val')) is tuple

assert get_count_metrics('train') == (335, 387)
assert get_count_metrics('val') == (64, 104)
```

In [5]:

```python
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''

train_loader, val_loader = load_data()

assert type(train_loader) is torch.utils.data.dataloader.DataLoader

assert len(train_loader) == 23
```

In [6]:

```python
# DO NOT MODIFY THIS PART


import torchvision
import matplotlib.pyplot as plt

def imshow(img, title):
    npimg = img.numpy()
    plt.figure(figsize=(15, 7))
    plt.axis('off')
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.title(title)
    plt.show()

def show_batch_images(dataloader, k=8):
    images, labels = next(iter(dataloader))
    images = images.reshape(-1, 3, 24, 24)
    images = images[:k]
    labels = labels[:k]
    img = torchvision.utils.make_grid(images, padding=3)
    imshow(img, title=["NORMAL" if x==0  else "PNEUMONIA" for x in labels])

train_loader, val_loader = load_data()
for i in range(2):
    show_batch_images(train_loader)
```

---

2 Build the Models [30 points]

In this section we will build four different variants of Autoencoder architectures

2.1 Vanilla Autoencoder [5 points]

The first thing we will do is build the simple autoencoder model. For each patient, the vanilla autoencoder model will take an input tensor of 1728-dim, and produce an output tensor of 1728-dim as well that is meant to closely mirror the original input. However, in between the model will compress those 1728 dimensions into just 16 such that it will build an intermediate representation which contains all of the information of the entire 1728 dimensions in just 16 numbers.

The detailed model architecture for you to follow is shown in the table below, but it will be broken down into the encoder half and decoder half.

Layers	Configuration	Activation Function	Output Dimension (batch, feature)
fully connected	input size 1728, output size 128	ReLU	(32, 128)
fully connected	input size 128, output size 16	ReLU	(32, 16)
fully connected	input size 16, output size 128	ReLU	(32, 128)
fully connected	input size 128, output size 1728	Sigmoid	(32, 1728)

In [7]:

```python
"""
TODO: Build the MLP shown above.
HINT: Consider using `nn.Linear`, `torch.relu`, and `torch.sigmoid`.
"""

class VanillaAutoencoder(nn.Module):
    def __init__(self):
        super(VanillaAutoencoder, self).__init__()

        # DO NOT change the names
        self.fc1 = None
        self.fc2 = None
        self.fc3 = None
        self.fc4 = None

        """
        TODO: Initialize the model layers as shown above.
        """

        self.fc1 = nn.Linear(1728, 128)
        self.fc2 = nn.Linear(128, 16)
        self.fc3 = nn.Linear(16, 128)
        self.fc4 = nn.Linear(128, 1728)

    def encode(self, x):
        """
        TODO: Perform encoding operation with fc1, fc2, and the corresponding activat
        """
        # your code here
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        return x

    def decode(self, x):
        x = torch.relu(self.fc3(x))
        x = torch.sigmoid(self.fc4(x))
        return x

    def forward(self, x):
        return self.decode(self.encode(x))

# initialize the NN
model = VanillaAutoencoder()
print(model)
```

In [8]:

```python
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''

assert model.fc1.in_features == 1728, f'First layer input size is wrong! Should be 172
assert model.fc1.out_features == 128, f'First layer output size is wrong! Should be 12
assert model.fc2.in_features == 128, f'Second layer input size is wrong! Should be 128
assert model.fc2.out_features == 16, f'Second layer output size is wrong! Should be 16
assert model.fc3.in_features == 16, f'Third layer input size is wrong! Should be 16!=
assert model.fc3.out_features == 128, f'Third layer output size is wrong! Should be 12
assert model.fc4.in_features == 128, f'Fourth layer input size is wrong! Should be 128
assert model.fc4.out_features == 1728, f'Fourth layer output size is wrong! Should be
```

In [9]:

```python
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''
```

2.2 Sparse Autoencoder [5 Points]

Next, we will be constructing a sparse autoencoder model. While the biggest difference between the Sparse Autoencoder and Vanilla Autoencoder will come later in our training function by adding regularization in the loss function, we will also use the sigmoid activation function for all of our hidden layers here as well.

The detailed model architecture for you to follow is shown in the table below, and it will also be broken down into the encoder half and decoder half.

Layers	Configuration	Activation Function	Output Dimension (batch, feature)
fully connected	input size 1728, output size 128	Sigmoid	(32, 128)
fully connected	input size 128, output size 16	Sigmoid	(32, 16)
fully connected	input size 16, output size 128	Sigmoid	(32, 128)
fully connected	input size 128, output size 1728	Sigmoid	(32, 1728)

In [10]:

```python
"""
TODO: Build the MLP shown above.
HINT: Consider using `nn.Linear` and `torch.sigmoid`.
"""

class SparseAutoencoder(nn.Module):
    def __init__(self):
        super(SparseAutoencoder, self).__init__()

        # DO NOT change the names
        self.fc1 = None
        self.fc2 = None
        self.fc3 = None
        self.fc4 = None

        """
        TODO: Initialize the model layers as shown above.
        """
        # your code here
        self.fc1 = nn.Linear(1728, 128)
        self.fc2 = nn.Linear(128, 16)
        self.fc3 = nn.Linear(16, 128)
        self.fc4 = nn.Linear(128, 1728)

        # used in training as sparsity regularization
        self.data_rho = 0

    def encode(self, x):
        """
        TODO: Perform encoding operation with fc1, fc2, and the corresponding activat
        """
        # your code here
        x = torch.sigmoid(self.fc1(x))
        x = torch.sigmoid(self.fc2(x))
        return x

    def decode(self, x):
        """
        TODO: Perform decoding operation with fc3, fc4, and the corresponding activat
        """
        # your code here
        x = torch.sigmoid(self.fc3(x))
        x = torch.sigmoid(self.fc4(x))
        return x

    def forward(self, x):
        x = self.encode(x)
        self.data_rho = x.mean(0)
        x = self.decode(x)
        return x


# initialize the NN
model = SparseAutoencoder()
print(model)
```

In [11]:

```python
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''

assert model.fc1.in_features == 1728, f'First layer input size is wrong! Should be 172
assert model.fc1.out_features == 128, f'First layer output size is wrong! Should be 12
assert model.fc2.in_features == 128, f'Second layer input size is wrong! Should be 128
assert model.fc2.out_features == 16, f'Second layer output size is wrong! Should be 16
assert model.fc3.in_features == 16, f'Third layer input size is wrong! Should be 16!=
assert model.fc3.out_features == 128, f'Third layer output size is wrong! Should be 12
assert model.fc4.in_features == 128, f'Fourth layer input size is wrong! Should be 128
assert model.fc4.out_features == 1728, f'Fourth layer output size is wrong! Should be
```

In [12]:

```python
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''
```

2.3 Denoising Autoencoder [10 Points]

Next, we will be constructing a denoising autoencoder model. This follows the Vanilla Autoencoder but adds noise to the input in order to train the model to be able to both handle noisy input as well as serve as regularization to prevent overfitting. While the input is now noisy, the model still attempts to reconstruct the original input.

The detailed model architecture for you to follow is the same as with the Vanilla Autoencoder and is shown in the table below, and it will also be broken down into the encoder half and decoder half.

Layers	Configuration	Activation Function	Output Dimension (batch, feature)
fully connected	input size 1728, output size 128	ReLU	(32, 128)
fully connected	input size 128, output size 16	ReLU	(32, 16)
fully connected	input size 16, output size 128	ReLU	(32, 128)
fully connected	input size 128, output size 1728	Sigmoid	(32, 1728)

In [13]:

```python
"""
TODO: Build the MLP shown above.
HINT: Consider using `nn.Linear`, `torch.relu`, and `torch.sigmoid`.
"""

class DenoisingAutoencoder(nn.Module):
    def __init__(self):
        super(DenoisingAutoencoder, self).__init__()

        # DO NOT change the names
        self.fc1 = None
        self.fc2 = None
        self.fc3 = None
        self.fc4 = None

        """
        TODO: Initialize the model layers as shown above.
        """
        # your code here
        self.fc1 = nn.Linear(1728, 128)
        self.fc2 = nn.Linear(128, 16)
        self.fc3 = nn.Linear(16, 128)
        self.fc4 = nn.Linear(128, 1728)

    def encode(self, x):
        """
        TODO: Perform encoding operation with fc1, fc2, and the corresponding activat
        """
        # your code here
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        return x

    def decode(self, x):
        x = torch.relu(self.fc3(x))
        x = torch.sigmoid(self.fc4(x))
        return x

    def forward(self, x):
        noise = None
        std = 0.1
        mean = 0
        """
        TODO: Generate the noise from the normal distribution with the above mean and

        Note that the size of the noise should be the same as x.

        Hint: Use torch.randn().
        """
        # your code here
        noise = torch.randn(x.size()) * std
        x = x + noise
        return self.decode(self.encode(x))

# initialize the NN
model = DenoisingAutoencoder()
print(model)
```

In [14]:

```python
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''

assert model.fc1.in_features == 1728, f'First layer input size is wrong! Should be 172
assert model.fc1.out_features == 128, f'First layer output size is wrong! Should be 12
assert model.fc2.in_features == 128, f'Second layer input size is wrong! Should be 128
assert model.fc2.out_features == 16, f'Second layer output size is wrong! Should be 16
assert model.fc3.in_features == 16, f'Third layer input size is wrong! Should be 16!=
assert model.fc3.out_features == 128, f'Third layer output size is wrong! Should be 12
assert model.fc4.in_features == 128, f'Fourth layer input size is wrong! Should be 128
assert model.fc4.out_features == 1728, f'Fourth layer output size is wrong! Should be
```

In [15]:

```python
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''
```

In [16]:

```python
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''
```

2.4 Stacked Autoencoder [10 Points]

Finally, we will be constructing a more complex and better performing stacked autoencoder model. For each patient, we will still take an input tensor of 1728-dim, and produce an output tensor of 1728-dim as well that is meant to closely mirror the original input. We will also still compress those 1728 dimensions into just 16. However, instead of performing such a compression just once, we will do it three times in a row using Vanilla Autoencoder models as subcomponents

In [17]:

```python
"""
TODO: Build the StackedAutoencoder using your VanillaAutoencoder architecture.
"""

class StackedAutoencoder(nn.Module):
    def __init__(self):
        super(StackedAutoencoder, self).__init__()

        # DO NOT change the names
        self.ae1 = None
        self.ae2 = None
        self.ae3 = None

        """
        TODO: Initialize three Vanilla Autoencoders and assign them to self.ae1, self
        """
        # your code here
        self.ae1 = VanillaAutoencoder()
        self.ae2 = VanillaAutoencoder()
        self.ae3 = VanillaAutoencoder()

    def forward(self, x):
        x = self.ae1(x)
        x = self.ae2(x)
        x = self.ae3(x)
        return x

    def encode(self, x):
        """
        TODO: While we didn't implement the forward() function of the
        StackedAutoencoder as using an encode() and decode() function,
        we may still be interested in the future of extracting the
        compressed representation. So, implement the encode function
        to return the compressed representation from the third
        VanillaAutoencoder component (note you will have to call its
        encode function).
        """
        # your code here
        x = self.ae1(x)
        x = self.ae2(x)
        x = self.ae3.encode(x)
        return x

# initialize the NN
model = StackedAutoencoder()
print(model)
```

In [18]:

```python
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''

assert isinstance(model.ae1, VanillaAutoencoder), f'First autoencoder should be a Van
assert isinstance(model.ae2, VanillaAutoencoder), f'Second autoencoder should be a Va
assert isinstance(model.ae3, VanillaAutoencoder), f'Third autoencoder should be a Van
```

In [19]:

```python
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''
```

In [20]:

```python
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''
```

3 Training the Networks [60 points]

In this step, you will train each of the three autoencoder architectures and compare the results

Unlike most of our past loss functions that go with the classification tasks we have see, here we will be using Mean Squared Error loss which is typically used in reconstruction settings such as ours and also in regression tasks (in which outputs are numeric values instead of probabilities and class labels)

In [21]:

```python
"""
TODO: Define the loss (MSELoss), assign it to `criterion`.

REFERENCE: https://pytorch.org/docs/stable/generated/torch.nn.MSELoss.html#torch.nn.MS
"""

criterion = None

# your code here
criterion = nn.MSELoss()
```

In [22]:

```python
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''
_loss = criterion(torch.Tensor([0., 1., 0.5]), torch.Tensor([1., 1., 1.]))
assert abs(_loss.tolist() - 0.4167) < 1e-3, "MSELoss is wrong"
```

In [23]:

```python
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''
```

Now let us train the NN model we previously created.

First, let us implement the `evaluate` function that will be called to evaluate the model performance when training.

Note: Our evaluation uses the same loss function that we use during training

In [24]:

```python
from sklearn.metrics import *

#input: Y_pred,Y_true
#output: mean squared error, mean absolute error
def classification_metrics(X_reconstructed, X_original):
    mse, mae = mean_squared_error(X_original, X_reconstructed), \
               mean_absolute_error(X_original, X_reconstructed)
    return mse, mae



#input: model, loader
def evaluate(model, loader):
    model.eval()
    all_X_original = torch.FloatTensor()
    all_X_reconstructed = torch.FloatTensor()
    for x, _ in loader:
        x_reconstructed = model(x)
        """
        TODO: Add the correct values to the lists in order to keep a
        running tab of all of the original and reconstructed inputs.

        Hint: use torch.cat().
        """
        # your code here
        all_X_original = torch.cat((all_X_original, x))
        all_X_reconstructed = torch.cat((all_X_reconstructed, x_reconstructed))

    mse, mae = classification_metrics(all_X_reconstructed.detach().numpy(), all_X_orig
    print(f"mse: {mse:.3f}, mae: {mae:.3f}")
    return mse, mae
```

In [25]:

```python
print("model perfomance before training:")
# initialized the model
model = VanillaAutoencoder()
mae_train_init = evaluate(model, train_loader)[1]
mae_val_init = evaluate(model, val_loader)[1]
```

In [26]:

```python
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''
assert mae_train_init > 0.1, "mae is less than 0.1! Please check this is random initia
```

This time we will be using a slightly more advanced optimizer option than the SGD optimizer that we have seen in the past. Instead, we will be using the Adam optimize which utilizes concepts such as momentum to offer a more refined and effective training. However, from your end it works almost exactly the same.

In [40]:

```python
"""
TODO: Define the optimizer (Adam) with learning rate 0.001, assign it to `optimizer`.

REFERENCE: https://pytorch.org/docs/stable/optim.html
"""
def get_optimizer(model):
    optimizer = None

    # your code here
    optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

    return optimizer
```

In [28]:

```python
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''
```

To train the model, you should follow the following step:

- Clear the gradients of all optimized variables
- Forward pass: compute predicted outputs by passing inputs to the model
- Calculate the loss (with an extra regularization term if the model is a SparseAutoencoder)
- Backward pass: compute gradient of the loss with respect to model parameters
- Perform a single optimization step (parameter update)
- Update average training loss

In [41]:

```python
def train_model(model):
    # number of epochs to train the model
    n_epochs = 10

    # get the correct type of optimizer for the model
    optimizer = get_optimizer(model)

    # prep model for training
    model.train()

    train_loss_arr = []
    for epoch in range(n_epochs):

        train_loss = 0
        for x, _ in train_loader:
            """ Step 1. clear gradients """
            optimizer.zero_grad()
            """
            TODO: Step 2. perform forward pass using `model`, save the output to x_rec
                  Step 3. calculate the loss using `criterion`, save the output to los
                      If the model is a SparseAutoencoder, the loss will have an addit
                      regularization penalty. This is calculated by:
                          average of (- rho * log(data_rho)  +  (1 - rho) * log(1 - da
                      where we will use rho of 0.1
            """

            x_reconstructed = model.forward(x)
            loss = criterion(x_reconstructed, x)
            # your code here

            if isinstance(model, SparseAutoencoder):
                penalty = None
                rho = 0.1
                data_rho = model.data_rho
                penalty = -(rho * torch.log(data_rho)  +  (1 - rho) * torch.log(1 - da
                loss = loss + (0.5 * penalty)
            """ Step 4. backward pass """
            loss.backward()
            """ Step 5. optimization """
            optimizer.step()
            """ Step 6. record loss """
            train_loss += loss.item()

        train_loss = train_loss / len(train_loader)
        if epoch % 2 == 0:
            train_loss_arr.append(np.mean(train_loss))
            print('Epoch: {} \tTraining Loss: {:.6f}'.format(epoch, train_loss))
            evaluate(model, val_loader)

    return model, train_loss_arr
```

In [42]:

```python
vanilla_model = VanillaAutoencoder()
vanilla_model, vanilla_train_loss_arr = train_model(vanilla_model)
```

In [31]:

```python
sparse_model = SparseAutoencoder()
sparse_model, sparse_train_loss_arr = train_model(sparse_model)
```

In [32]:

```python
denoising_model = DenoisingAutoencoder()
denoising_model, denoising_train_loss_arr = train_model(denoising_model)
```

In [33]:

```python
stacked_model = StackedAutoencoder()
stacked_model, stacked_train_loss_arr = train_model(stacked_model)
```

In [34]:

```python
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''

assert sorted(list(np.round(vanilla_train_loss_arr[:5], 2)), reverse=True) == list(np
```

In [35]:

```python
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''

assert np.mean(sparse_train_loss_arr) > np.mean(vanilla_train_loss_arr), f"Sparse tra
```

In [36]:

```python
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''
```

In [37]:

```python
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''
```

In [38]:

```python
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''
```

In [39]:

```python
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''
```

In [ ]:

In [ ]: