HW2_NN

October 3, 2025

1 HW2 Neural Network

1.1 Overview

In this homework, you will get introduced to PyTorch, a framework for building and training neural networks. PyTorch in a lot of ways behaves like the arrays you love from Numpy. These Numpy arrays, after all, are just tensors. PyTorch takes these tensors and makes it simple to move them to GPUs for the faster processing needed when training neural networks. It also provides a module that automatically calculates gradients (for backpropagation) and another module specifically for building neural networks.

More specifically, you will first learn some PyTorch basics. And then, you will train a simple neural network on **Heart Failure Prediction** (same as HW1 but with neural network).

---

[2]:
```
import os
import sys

DATA_PATH = "../HW2_NN-lib/data/"

sys.path.append('../HW2_NN-lib')
```

1.2 1 PyTorch Basics [30 points]

It turns out neural network computations are just a bunch of linear algebra operations on tensors, a generalization of matrices. A vector is a 1-dimensional tensor, a matrix is a 2-dimensional tensor, an array with three indices is a 3-dimensional tensor (RGB color images for example). The fundamental data structure for neural networks are tensors and PyTorch (as well as pretty much every other deep learning framework) is built around tensors.

With the basics covered, it is time to explore how we can use PyTorch to build a simple neural network.

[3]:
```
import random
import numpy as np
import torch
import torch.nn as nn
```

[4]:
```
# set seed
seed = 24
random.seed(seed)
np.random.seed(seed)
torch.manual_seed(seed)
os.environ["PYTHONHASHSEED"] = str(seed)
```

1.2.1 1.1 ReLU Implementation from scratch [7 points]

[5]:
```
#input
# x: torch.Tensor
#output
# relu(x): torch.Tensor
def relu(x):

    """
    TODO: Implement a ReLU activation function from **scratch**.

    REFERENCE: https://pytorch.org/docs/stable/generated/torch.nn.ReLU.
 ↪html#torch.nn.ReLU
    """

    # your code here
    return x * (x > 0).type_as(x)
```

[6]:
```
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''

# test if `activation` directly calls `torch.relu`
random_tensor = torch.randn((2, 2))
_relu = torch.relu
del torch.relu
try:
    relu(random_tensor)
    torch.relu = _relu
except:
    print("`relu` not implemented from scratch!")
    torch.relu = _relu
    raise

# test on some random input
random_tensor = torch.randn((1, 1))[0][0]
assert torch.allclose(relu(random_tensor), torch.relu(random_tensor)), "relu()␣
 ↪is wrong for {}!".format(random_tensor)
```

```
random_tensor = torch.randn((1, 1))[0][0]
assert torch.allclose(relu(random_tensor), torch.relu(random_tensor)), "relu()␣
 ↪is wrong for {}!".format(random_tensor)
random_tensor = torch.randn((2, 2))
assert torch.allclose(relu(random_tensor), torch.relu(random_tensor)), "relu()␣
 ↪is wrong!"
```

[7]:
```
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''
```

[7]: '\nAUTOGRADER CELL. DO NOT MODIFY THIS.\n'

1.2.2 1.2 Sigmoid Implementation from scratch [7 points]

[8]:
```
#input
# x: torch.Tensor
#output
# sigmoid(x): torch.Tensor
def sigmoid(x):

    """
    TODO: Implement a Sigmoid activation function from **scratch**.
    HINT: Consider `torch.exp()`.
    """

    # your code here
    return 1.0 / (1.0 + torch.exp(-x))
```

[9]:
```
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''

# test if `activation` directly calls `torch.sigmoid`
random_tensor = torch.randn((2, 2))
_sigmoid = torch.sigmoid
del torch.sigmoid
try:
    sigmoid(random_tensor)
    torch.sigmoid = _sigmoid
except:
    print("`activation` not implemented from scratch!")
    torch.sigmoid = _sigmoid
    raise
```

[10]:
```
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''
```

[10]:
```
'\nAUTOGRADER CELL. DO NOT MODIFY THIS.\n'
```

1.2.3 1.3 Softmax Implementation from scratch [8 points]

It should be noted that softmax degenerates to sigmoid when we have 2 classes.

[11]:
```
#input
# x: torch.Tensor, 2D matrix
#output
# softmax(x): torch.Tensor, 2D matrix with sum over rows is 1
def softmax(x):

    """
    TODO: Implement a Softmax activation function from **scratch**.
    HINT: Consider `torch.exp()`.
    """

    # your code here
    x_stable = x - x.max(dim=1, keepdim=True).values
    exps = torch.exp(x_stable)
    return exps / exps.sum(dim=1, keepdim=True)
```

[12]:
```
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''

# test if `activation` directly calls `torch.sigmoid`
random_tensor = torch.randn((2, 2))
_softmax = torch.nn.functional.softmax
del torch.nn.functional.softmax
try:
    softmax(random_tensor)
    torch.nn.functional.softmax = _softmax
except:
    print("`activation` not implemented from scratch!")
    torch.nn.functional.softmax = _softmax
    raise
```

[13]:
```
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''
```

[13]:
```
'\nAUTOGRADER CELL. DO NOT MODIFY THIS.\n'
```

1.2.4 1.4 Single layer network with sigmoid [8 points]

Now, let us try to use the `sigmoid` function to calculate the output for a simple single layer network.

[14]:
```
# Generate some data
# Features are 5 random normal variables
features = torch.randn((1, 5))
# weights for our data, random normal variables again
weights = torch.randn_like(features)
# and a bias term
bias = torch.randn((1, 1))
```

Above I generated data we can use to get the output of our simple network. This is all just random for now, going forward we will start using normal data.

`features = torch.randn((1, 5))` creates a tensor with shape (1, 5), one row and five columns, that contains values randomly distributed according to the normal distribution with a mean of zero and standard deviation of one.

`weights = torch.randn_like(features)` creates another tensor with the same shape as features, again containing values from a normal distribution.

`bias = torch.randn((1, 1))` creates a single value from a normal distribution.

Use the generated data to calculate the output of this simple single layer network. Input features are `features`, weights are `weights`, and bias are `bias`. Use `sigmoid` as the activation function.

[15]:
```
#input
# features: torch.Tensor
# weights: torch.Tensor
# bias: torch.Tensor
#output
# output of a sinlge layer network: torch.Tensor
def single_layer_network(features, weights, bias):

    """
    TODO: Calculate the output of this simple single layer network.
    HINT: Consider `torch.mm()` or `torch.matmul()`.
    """

    # your code here
    # linear = X W^T + b → shape (batch, 1)
    lin = torch.matmul(features, weights.T) + bias
    return sigmoid(lin)
```

[16]:
```
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''

# test if function `single_layer_network` calls function `sigmoid`
```

```
orig_sigmoid = sigmoid
del sigmoid
try:
    single_layer_network(features, weights, bias)
except NameError:
    sigmoid = orig_sigmoid
    pass
else:
    print('Function `sigmoid` is not used!')
    sigmoid = orig_sigmoid
    raise

# test on some random input
features = torch.Tensor([[0.1, 0.2, 0.3]])
weights = torch.Tensor([[0.1, 0.2, 0.3]])
bias = torch.Tensor([0])
assert torch.allclose(single_layer_network(features, weights, bias), torch.
 ↪Tensor([[0.5349]]), atol=1e-4), "single_layer_network() is wrong!"
features = torch.Tensor([[1, 0, 0]])
weights = torch.Tensor([[4, 0, 0]])
bias = torch.Tensor([-1])
assert torch.allclose(single_layer_network(features, weights, bias), torch.
 ↪Tensor([[0.9526]]), atol=1e-4), "single_layer_network() is wrong!"
features = torch.Tensor([[0.1, 0.2, 0.3]])
weights = torch.Tensor([[0.1, 0.2, 0.3]])
bias = torch.Tensor([-0.5])
assert torch.allclose(single_layer_network(features, weights, bias), torch.
 ↪Tensor([[0.4110]]), atol=1e-4), "single_layer_network() is wrong!"
```

[17]:
```
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''
```

[17]:
```
'\nAUTOGRADER CELL. DO NOT MODIFY THIS.\n'
```

That is how you can calculate the output for a sinlge layer. The real power of this algorithm happens when you start stacking these individual units into layers and stacks of layers, into a network of neurons. The output of one layer of neurons becomes the input for the next layer. We will explore this in the next problem.

1.3 2 NN with PyTorch [70 points]

Deep learning networks tend to be massive with dozens or hundreds of layers, that is where the term "deep" comes from. You can build one of these deep networks using only weight matrices as we did in the previous problem, but in general it is very cumbersome and difficult to implement. PyTorch has a nice module `nn` that provides a nice way to efficiently build large neural networks.

Previously, you have tried to perform Heart Failure Prediction using traditional machine learning

methods such as logistic regression, support vector machine, and decision tree. In this problem, you will train a neural network to do exactly the same task. The data is the same as HW1. We have processed the data for you. The data is saved in SVMLight format under `DATA_PATH`.

[18]:
```
!ls {DATA_PATH}
```

```
features_svmlight.train  features_svmlight.val
```

1.3.1 2.1 Load the Data

This part has been done for you.

[19]:
```
import utils


""" load SVMLight data """
# training data
X_train, Y_train = utils.get_data_from_svmlight(DATA_PATH + "features_svmlight.
 ↪train")
# validation data
X_val, Y_val = utils.get_data_from_svmlight(DATA_PATH + "features_svmlight.val")

""" convert to torch.tensor """
X_train = torch.from_numpy(X_train.toarray()).type(torch.float)
Y_train = torch.from_numpy(Y_train).type(torch.float)
X_val = torch.from_numpy(X_val.toarray()).type(torch.float)
Y_val = torch.from_numpy(Y_val).type(torch.float)

print("X_train shape:", X_train.shape)
print("Y_train shape:", Y_train.shape)
print("X_val shape:", X_val.shape)
print("Y_val shape:", Y_val.shape)
```

```
X_train shape: torch.Size([2485, 1473])
Y_train shape: torch.Size([2485])
X_val shape: torch.Size([604, 1473])
Y_val shape: torch.Size([604])
```

Now, we will create a `TensorDataset` to wrap those tensors. (https://pytorch.org/docs/stable/data.html#torch.utils.data.TensorDataset)

[20]:
```
from torch.utils.data import TensorDataset

train_dataset = TensorDataset(X_train, Y_train)
val_dataset = TensorDataset(X_val, Y_val)

print("Size of train_dataset:", len(train_dataset))
print("Size of val_dataset:", len(val_dataset))
```

```
#If you index train_dataset now, you will get a (data, label) tuple
print(train_dataset[0])
print([_t.shape for _t in train_dataset[0]])
```

```
Size of train_dataset: 2485
Size of val_dataset: 604
(tensor([0., 0., 0.,  …, 0., 0., 0.]), tensor(0.))
[torch.Size([1473]), torch.Size([])]
```

Next, we will load the dataset into a dataloader so that we can we can use it to loop through the dataset for training and validating. (https://pytorch.org/docs/stable/data.html#torch.utils.data.DataLoader)

[21]:
```
from torch.utils.data import DataLoader

# how many samples per batch to load
batch_size = 32

# prepare dataloaders
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=batch_size)

print("# of train batches:", len(train_loader))
print("# of val batchse:", len(val_loader))
```

```
# of train batches: 78
# of val batchse: 19
```

You will notice that the data loader is created with a batch size of 32, and `shuffle=True`.

The batch size is the number of images we get in one iteration from the data loader and pass through our network, often called a batch.

And `shuffle=True` tells it to shuffle the dataset every time we start going through the data loader again.

[22]:
```
train_iter = iter(train_loader)
x, y = next(train_iter)

print('Shape of a batch x:', x.shape)
print('Shape of a batch y:', y.shape)
```

```
Shape of a batch x: torch.Size([32, 1473])
Shape of a batch y: torch.Size([32])
```

1.3.2 2.2 Build the Model [30 points]

Now, let us build a real NN model. For each patient, the NN model will take an input tensor of 1473-dim, and produce an output tensor of 1-dim (0 for normal, 1 for heart failure). The detailed model architecture is shown in the table below.

Layers	Configuration	Activation Function	Output Dimension (batch, feature)
fully connected	input size 1473, output size 64	ReLU	(32, 64)
fully connected	input size 64, output size 32	ReLU	(32, 32)
dropout	probability 0.5	-	(32, 32)
fully connected	input size 32, output size 1	Sigmoid	(32, 1)

[23]:
```
"""
TODO: Build the MLP shown above.
HINT: Consider using `nn.Linear`, `nn.Dropout`, `torch.relu`, `torch.sigmoid`.
"""

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

        # DO NOT change the names
        self.fc1 = nn.Linear(1473, 64)
        self.fc2 = nn.Linear(64, 32)
        self.dropout = nn.Dropout(p=0.5)
        self.fc3 = nn.Linear(32, 1)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = self.dropout(x)
        x = torch.sigmoid(self.fc3(x))
        return x

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

```
Net(
  (fc1): Linear(in_features=1473, out_features=64, bias=True)
  (fc2): Linear(in_features=64, out_features=32, bias=True)
  (dropout): Dropout(p=0.5, inplace=False)
  (fc3): Linear(in_features=32, out_features=1, bias=True)
)
```

[24]:
```
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''

assert model.fc1.in_features == 1473, f'Input layer input size is wrong! Should␣
↪be 1473!={model.fc1.in_features}'
```

```
assert model.fc1.out_features == 64, f'First layer output size is wrong! Should␣
↪be 64!={model.fc1.out_features}'
assert model.fc2.in_features == 64, f'Second layer input size is wrong! Should␣
↪be 64!={model.fc2.in_features}'
assert model.fc2.out_features == 32, f'Second layer output size is wrong!␣
↪Should be 32!={model.fc2.out_features}'
assert model.fc3.in_features == 32, f'Third layer input size is wrong! Should␣
↪be 32!={model.fc3.in_features}'
assert model.fc3.out_features == 1, f'Final output size is wrong! Should be 1!
↪={model.fc3.out_features}'
```

[25]:
```
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''
```

[25]:
```
'\nAUTOGRADER CELL. DO NOT MODIFY THIS.\n'
```

[26]:
```
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''
```

[26]:
```
'\nAUTOGRADER CELL. DO NOT MODIFY THIS.\n'
```

Now that we have a network, let's see what happens when we pass in an image.

[27]:
```
# Grab some data
train_iter = iter(train_loader)
x, y = next(train_iter)

# Forward pass through the network
output = model.forward(x)

print('Input x shape:', x.shape)
print('Output shape: ', output.shape)
```

```
Input x shape: torch.Size([32, 1473])
Output shape:  torch.Size([32, 1])
```

1.3.3 2.3 Train the Network [40 points]

In this step, you will train the NN model.

Neural networks with non-linear activations work like universal function approximators. There is some function that maps your input to the output. The power of neural networks is that we can train them to approximate this function, and basically any function given enough data and compute time.

[28]:
```
model = Net()
```

Losses in PyTorch.

Let us start by seeing how we calculate the loss with PyTorch. Through the `nn.module`, PyTorch provides losses such as the binary cross-entropy loss (`nn.BCELoss`). You will usually see the loss assigned to `criterion`.

As noted in the last part, with a classification problem such as Heart Failure Prediction, we are using the Sigmoid function to predict heart failure probability. With a Sigmoid output, you want to use binary cross-entropy as the loss. To actually calculate the loss, you first define the criterion then pass in the output of your network and the correct labels.

[29]:
```
"""
TODO: Define the loss (BCELoss), assign it to `criterion`.

REFERENCE: https://pytorch.org/docs/stable/generated/torch.nn.BCELoss.
 ↪html#torch.nn.BCELoss
"""



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

[30]:
```
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''
_loss = criterion(torch.Tensor([0.1, 0.2, 0.9]), torch.Tensor([0., 1., 0.]))
assert abs(_loss.tolist() - 1.3391) < 1e-3, "BCELoss is wrong"
```

[31]:
```
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''
```

[31]:
```
'\nAUTOGRADER CELL. DO NOT MODIFY THIS.\n'
```

Optimizer in PyTorch.

Optimizer can update the weights with the gradients. We can get these from PyTorch's `optim` package. For example we can use stochastic gradient descent with `optim.SGD`.

[32]:
```
"""
TODO: Define the optimizer (SGD) with learning rate 0.01, assign it to␣
 ↪`optimizer`.

REFERENCE: https://pytorch.org/docs/stable/optim.html
"""



```

```
# your code here
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
```

[33]:
```
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''
```

[33]:
```
'\nAUTOGRADER CELL. DO NOT MODIFY THIS.\n'
```

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: For prediction, probability > 0.5 is considered class 1 otherwise class 0

[34]:
```
from sklearn.metrics import *

#input: Y_score,Y_pred,Y_true
#output: accuracy, auc, precision, recall, f1-score
def classification_metrics(Y_score, Y_pred, Y_true):
    acc, auc, precision, recall, f1score = accuracy_score(Y_true, Y_pred), \
                                           roc_auc_score(Y_true, Y_score), \
                                           precision_score(Y_true, Y_pred), \
                                           recall_score(Y_true, Y_pred), \
                                           f1_score(Y_true, Y_pred)
    return acc, auc, precision, recall, f1score



#input: model, loader
def evaluate(model, loader):
    model.eval()
    all_y_true = torch.LongTensor()
    all_y_pred = torch.LongTensor()
    all_y_score = torch.FloatTensor()
    for x, y in loader:
        with torch.no_grad():
            y_hat = model(x).view(-1)          # scores in [0,1]
            y_pred = (y_hat > 0.5).long()      # threshold to class
        all_y_true = torch.cat((all_y_true, y.long().cpu()), dim=0)
        all_y_pred = torch.cat((all_y_pred, y_pred.cpu()), dim=0)
        all_y_score = torch.cat((all_y_score, y_hat.cpu()), dim=0)

    acc, auc, precision, recall, f1 = classification_metrics(
        all_y_score.numpy(), all_y_pred.numpy(), all_y_true.numpy()
    )
    print(f"acc: {acc:.3f}, auc: {auc:.3f}, precision: {precision:.3f}, "
```

```
          f"recall: {recall:.3f}, f1: {f1:.3f}")
    return acc, auc, precision, recall, f1
```

[35]:
```
print("model perfomance before training:")
# initialized the model
# model = Net()
auc_train_init = evaluate(model, train_loader)[1]
auc_val_init = evaluate(model, val_loader)[1]
```

```
model perfomance before training:
acc: 0.479, auc: 0.482, precision: 0.567, recall: 0.453, f1: 0.504
acc: 0.450, auc: 0.474, precision: 0.583, recall: 0.335, f1: 0.426
```

[36]:
```
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''
assert auc_train_init < 0.6, "auc is greater than 0.60! Please check this is␣
 ↪random initialization and no training. So, accuracy sould be lower"
```

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 - Backward pass: compute gradient of the loss with respect to model parameters - Perform a single optimization step (parameter update) - Update average training loss

[37]:
```
n_epochs = 100
model.train()
train_loss_arr = []

for epoch in range(n_epochs):
    train_loss = 0.0
    for x, y in train_loader:
        optimizer.zero_grad()                  # Step 1
        y_hat = model(x)                       # Step 2 (shape: [B,1])
        loss = criterion(y_hat, y.view(-1,1)) # Step 3 (match shapes)
        loss.backward()                        # Step 4
        optimizer.step()                       # Step 5
        train_loss += loss.item()              # Step 6

    train_loss = train_loss / len(train_loader)
    if epoch % 20 == 0:
        train_loss_arr.append(np.mean(train_loss))
        print(f"Epoch: {epoch}\tTraining Loss: {train_loss:.6f}")
        evaluate(model, val_loader)
```

```
Epoch: 0        Training Loss: 0.690303
acc: 0.608, auc: 0.495, precision: 0.608, recall: 1.000, f1: 0.756
Epoch: 20       Training Loss: 0.672617
acc: 0.608, auc: 0.677, precision: 0.608, recall: 1.000, f1: 0.756
```

```
Epoch: 40       Training Loss: 0.627500
acc: 0.664, auc: 0.704, precision: 0.644, recall: 0.997, f1: 0.783
Epoch: 60       Training Loss: 0.535144
acc: 0.725, auc: 0.732, precision: 0.721, recall: 0.894, f1: 0.798
Epoch: 80       Training Loss: 0.466434
acc: 0.717, auc: 0.741, precision: 0.736, recall: 0.834, f1: 0.782
```

[38]:
```
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''

assert sorted(list(np.round(train_loss_arr[:5], 2)), reverse=True) == list(np.
 ↪round(train_loss_arr[:5], 2)), \
f"Training loss should decrease! Please check!"
```

[39]:
```
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''
```

[39]:
```
'\nAUTOGRADER CELL. DO NOT MODIFY THIS.\n'
```

[ ]: