CS184A/284A: AI in Biology and Medicine

HW3

Predicting TF Binding Sites

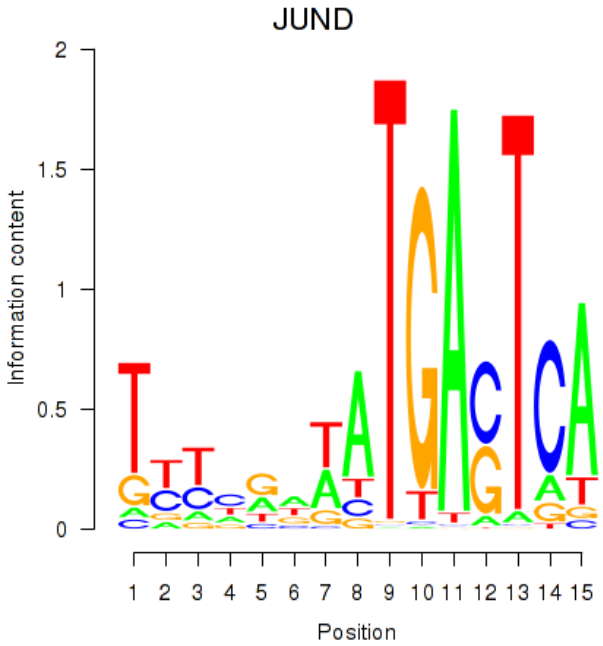
Transcription Factors (TFs) are proteins that bind to the DNA and help regulate gene transcription. The TFs have to recognize some "motif" on the DNA upstream from the gene, and DNA accessibility also plays a role.

In tis problem set, we will develop ML methods to predict which sequences can be bound by a transcription factor called JUND. The binding profile of JUND expressed in terms of a sequence logo is shown in the following picture.

More information on JUND can be found here:

- https://www.genecards.org/cgi-bin/carddisp.pl?gene=JUND

```
# get the image
from IPython import display
display.Image("https://www.ismara.unibas.ch/ISMARA/scratch/NHBE_SC2/ismara_report/log
```



MLP model

In this assignment you'll write an MLP model the predict whether a segments of the human chromosome 22 (Chr22) contain the binding sites for the JUND TF. You can modify the mlp notebook I shared with you to work on this problem. You need to have at least one hidden layer. You have to compute a weighted loss, and include accessibility information in your model, as described below.

Dataset

The data comprises 101 length segments from Chr22, with each position a one-hot vector denoting one of the four bases (A, C, G, T). Thus, each element of the input is 2d with size 101×4. Each such element has a target label 0 or 1, indicating whether the TF binds to that segment or not. The data also includes a weight per input element, since there are only a few binding sites (0.42%), so that you'd obtain an accuracy of 99.58% just by predicting there are no binding sites. This means you have to use the weights to discount the losses for label 0 and enhance the losses for label 1 items. Finally, there is an array of values, one per input element, that also indicates the chromosome accessibility for that segment.

Data Credit: Mohammed Zaki

Download data

- First, you need to download data file named "TF_data.zip" from Canvas. Unzip it and create the train, valid, test directories.
- If you use Google Colab, you can first upload the TF_data.zip file, and then run the follow command

```
!unzip TF_data.zip
```

- The data is split into training, validation and testing sets. Each set contains the following files:
  - shard-0-X.joblib: the set of 101 x 4 input elements
  - shard-0-y.joblib: the true labels: 0 or 1
  - shard-0-w.joblib: weight per input element
  - shard-0-a.joblib: accessibility value per input element
- After unzip the data file, you can read these files by using joblib.load function, which will populate a numpy array. For example

```
X = joblib.load('shard-0-X.joblib')
```

  will results in a numpy array X, which you can then convert to torch tensor, and so on.
- The roles of training, validation and testing sets:

- Use training set to tune the parameters of the model.
- Use validation set to select model structure and hyperparamters (e.g., number of epochs, learning rate, etc).
- Use test set for the final evaluation. You should never touch test set for either your model training or model selection.

```
# Uncomment the following command if you run this code in Google Colab
# !unzip TF_data.zip
```

```
import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
import joblib
from sklearn.metrics import precision_score, recall_score, confusion_matrix, f1_score
import itertools
```

```
if torch.cuda.is_available():
  device = torch.device("cuda")
else:
  device = torch.device("cpu")
```

Set up DataLoader for fetching training and testing data

Because we use mini-batch gradient descent for training. We need to set up dataloader that can provide us a minibatch of data samples.

```
from torch._C import dtype
from torch.utils.data import Dataset

class JUND_Dataset(Dataset):
    def __init__(self, data_dir):
        '''load X, y, w, a from data_dir'''
        super(JUND_Dataset, self).__init__()

        # load X, y, w, a from given data_dir
        # convert them into torch tensors
        self.X = torch.from_numpy(joblib.load(data_dir + '/shard-0-X.joblib')).float(
        self.y = torch.from_numpy(joblib.load(data_dir + '/shard-0-y.joblib')).float(
        self.w = torch.from_numpy(joblib.load(data_dir + '/shard-0-w.joblib')).float(
        self.a = torch.from_numpy(joblib.load(data_dir + '/shard-0-a.joblib')).float(
```

```
    def __len__(self):
        '''return len of dataset'''
        return self.X.shape[0]

    def __getitem__(self, idx):
        '''return X, y, w, and a values at index idx'''
        return self.X[idx],self.y[idx],self.w[idx], self.a[idx]


# get data
#
# You may need to change the directory if the data are not stored under current direc
#
train_dataset = JUND_Dataset('train_dataset')
test_dataset = JUND_Dataset('test_dataset')
```

Traing and test data

```
batch_size = 100

# Data loader
train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
                                           batch_size=batch_size,
                                           shuffle=True)

test_loader = torch.utils.data.DataLoader(dataset=test_dataset,
                                          batch_size=batch_size,
                                          shuffle=False)


print('Train data: ', len(train_dataset))
print('Test data: ', len(test_dataset))
```

```
Train data:  276216
Test data:  34528
```

Your result should look like:

```
Train data:  276216
Test data:  34528
```

Fetch a minibatch and check the size of the data

```
X,y,w,a = next(iter(train_loader))
```

```
# run the following code to check the size of data in each minibatch
X.shape, y.shape, w.shape, a.shape
```

```
(torch.Size([100, 101, 4]),
 torch.Size([100, 1]),
 torch.Size([100, 1]),
 torch.Size([100, 1]))
```

Your result should look like the following:

```
(torch.Size([100, 101, 4]),
 torch.Size([100, 1]),
 torch.Size([100, 1]),
 torch.Size([100, 1]))
```

Problem 1 - MLP

Define an MLP (multi-layer perceptron) with at least one hidden layer to predict the labels given inputs.

Please note the following:

- The label for each input is either 0 or 1, so this is essentially a binary classification problem.
- Input consist of both X and a:
  - X: represents the DNA sequence. Each position is a one-hot vector denoting one of the four bases (A, C, G, T). Thus, each element of the input is 2d with size 101×4. Since each input is treated as a vector in MLP, the 2d array needs to be flattened into a 404-dimensional vector.
  - a: represents the chromatin accessiblity of the input DNA sequence segment. You can think of "a" as an additional feature for each input. You can decide how to use it. For instance, if you are using hidden dimension of 128, then after concatenating the accessibility value, it will become a 129d vector, which should be fed to the final output layer of size 1, since we have a binary class/label.

An initial template code, representing a simple model, is provided. Your job is to change the definition of the model to improve the model's performance on the test dataset.

```
################################################################################
#  modify this code block
################################################################################
```

```
# MLP
class MyModel(nn.Module):
    def __init__(self, input_size=101*4, hidden_size=256):
        super(MyModel, self).__init__()
        self.layer1 = nn.Sequential(
            nn.Linear(input_size, hidden_size),
            nn.ReLU()
        )
        self.fc = nn.Linear(hidden_size + 1, 1)

    def forward(self, X, a):
        out = X.reshape(X.size(0),-1)
        out = self.layer1(out)
        out = torch.cat((out, a),1)
        out = self.fc(out)
        return out
# end of model definition
################################################################################
```

Have a test run of your model

```
model = MyModel().to(device)
# output = model(X,a)
```

Your model should run smoothly. The size of output should be 100 - the same as the minibatch size.

Training

Next you need to define a loss function and then run gradient descend to learn the weights of the neural net.

- There is a strong class imbalance problem in the training set (many more 0's than 1's).
- To handle the class imbalance problem, we treat each sample differently. Each data point is assigned a weight. Take a look at the variable named "w" in each data set.
- Define a loss function, in which the total loss is a weighted combination of losses coming from each sample. Use the weights specified in "w". Note that this definition is different from our typical loss, where each sample contributes equally to the final total loss.
- You should use binary_cross_entropy_with_logits with weight set to the weights per input element. Check out the documentation for details.

```
################################################################
```

```
# complete this code block
# define loss and optimizer
###########################################################
criterion = nn.BCEWithLogitsLoss()
###########################################################


# have a test run to check of your loss is defined properly
# loss = criterion(output, y)
```

You task:

You need to train the model on the training data, and use the **validation data** to select how many epochs you want to use and to choose the hidden dimension.

Use the **weighted prediction accuracy** as the evaluation metric. That is, sum of the weights of the correct predictions divided by the total weight across all the input elements.

Finally, report the **weighted accuracy on the test data.**

Run Training

```
# Choose hyper parameters and optimizer
num_epochs = 20
learning_rate = 0.005
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)


# Train the model
total_step = len(train_loader)
for epoch in range(num_epochs):
    for i, (X, y, w, a) in enumerate(train_loader):
        X = X.to(device)
        y = y.to(device)
        w = w.to(device)
        a = a.to(device)

        # Forward pass
        output = model(X, a)
        criterion.weight = w
        loss = criterion(output, y)

        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

```
        if (i+1) % 100 == 0:
            print ('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}'
                   .format(epoch+1, num_epochs, i+1, total_step, loss.item()))
```

```
Epoch [18/20], Step [2400/2763], Loss: 0.9758
Epoch [18/20], Step [2500/2763], Loss: 1.0153
Epoch [18/20], Step [2600/2763], Loss: 0.2318
Epoch [18/20], Step [2700/2763], Loss: 0.3048
Epoch [19/20], Step [100/2763], Loss: 0.1766
Epoch [19/20], Step [200/2763], Loss: 0.1661
Epoch [19/20], Step [300/2763], Loss: 0.2035
Epoch [19/20], Step [400/2763], Loss: 0.2298
Epoch [19/20], Step [500/2763], Loss: 0.2670
Epoch [19/20], Step [600/2763], Loss: 0.2014
Epoch [19/20], Step [700/2763], Loss: 0.2681
Epoch [19/20], Step [800/2763], Loss: 0.2029
Epoch [19/20], Step [900/2763], Loss: 0.2065
Epoch [19/20], Step [1000/2763], Loss: 0.3363
Epoch [19/20], Step [1100/2763], Loss: 1.0126
Epoch [19/20], Step [1200/2763], Loss: 0.7671
Epoch [19/20], Step [1300/2763], Loss: 0.2282
Epoch [19/20], Step [1400/2763], Loss: 0.2902
Epoch [19/20], Step [1500/2763], Loss: 1.1842
Epoch [19/20], Step [1600/2763], Loss: 0.4442
Epoch [19/20], Step [1700/2763], Loss: 0.2462
Epoch [19/20], Step [1800/2763], Loss: 1.0201
Epoch [19/20], Step [1900/2763], Loss: 0.1845
Epoch [19/20], Step [2000/2763], Loss: 1.1586
Epoch [19/20], Step [2100/2763], Loss: 0.1924
Epoch [19/20], Step [2200/2763], Loss: 1.2641
Epoch [19/20], Step [2300/2763], Loss: 0.2151
Epoch [19/20], Step [2400/2763], Loss: 0.2278
Epoch [19/20], Step [2500/2763], Loss: 0.1876
Epoch [19/20], Step [2600/2763], Loss: 0.2296
Epoch [19/20], Step [2700/2763], Loss: 0.2016
Epoch [20/20], Step [100/2763], Loss: 0.2309
Epoch [20/20], Step [200/2763], Loss: 0.2207
Epoch [20/20], Step [300/2763], Loss: 0.2005
Epoch [20/20], Step [400/2763], Loss: 0.2000
Epoch [20/20], Step [500/2763], Loss: 1.5933
Epoch [20/20], Step [600/2763], Loss: 0.8536
Epoch [20/20], Step [700/2763], Loss: 0.2310
Epoch [20/20], Step [800/2763], Loss: 0.2462
Epoch [20/20], Step [900/2763], Loss: 0.2089
Epoch [20/20], Step [1000/2763], Loss: 0.2178
Epoch [20/20], Step [1100/2763], Loss: 0.2683
Epoch [20/20], Step [1200/2763], Loss: 0.2033
Epoch [20/20], Step [1300/2763], Loss: 0.2702
Epoch [20/20], Step [1400/2763], Loss: 0.2741
Epoch [20/20], Step [1500/2763], Loss: 1.3213
Epoch [20/20], Step [1600/2763], Loss: 0.2127
Epoch [20/20], Step [1700/2763], Loss: 0.2655
Epoch [20/20], Step [1800/2763], Loss: 0.2381
Epoch [20/20], Step [1900/2763], Loss: 0.2230
Epoch [20/20], Step [2000/2763], Loss: 1.0873
Epoch [20/20], Step [2100/2763], Loss: 0.2114
```

```
Epoch [20/20], Step [2100/2763], Loss: 0.2114
Epoch [20/20], Step [2200/2763], Loss: 0.2126
Epoch [20/20], Step [2300/2763], Loss: 0.1741
Epoch [20/20], Step [2400/2763], Loss: 0.2265
Epoch [20/20], Step [2500/2763], Loss: 0.1973
Epoch [20/20], Step [2600/2763], Loss: 0.1733
Epoch [20/20], Step [2700/2763], Loss: 0.2193
```

Final Evaluation

Generate predictions on test set

```
model = model.to(torch.device("cpu"))
y_pred_list = []
y_target_list = []
weight_list = []

model.eval()
# Since we don't need model to back propagate the gradients in test set we use torch.
# reduces memory usage and speeds up computation
with torch.no_grad():
     for i, (X, y, w, a) in enumerate(test_loader):
        output = model(X, a)
        y_pred_tag = (output>0).int()
        y_pred_list.append(y_pred_tag.detach().numpy())
        y_target_list.append(y.detach().numpy())
        weight_list.append(w.detach().numpy())

#Takes arrays and makes them list of list for each batch
y_pred_list = [a.squeeze().tolist() for a in y_pred_list]
#flattens the lists in sequence
ytest_pred = list(itertools.chain.from_iterable(y_pred_list))


#Takes arrays and makes them list of list for each batch
y_target_list = [a.squeeze().tolist() for a in y_target_list]
#flattens the lists in sequence
ytest_target = list(itertools.chain.from_iterable(y_target_list))

weight_list = [a.squeeze().tolist() for a in weight_list]
test_weight = list(itertools.chain.from_iterable(weight_list))
```

Report

- Precision
- Recall

- F1 Score
- Confusion Matrix

```
conf_matrix = confusion_matrix(ytest_target ,ytest_pred)
print("Confusion Matrix of the Test Set")
print("-----------")
print(conf_matrix)
print("Precision of the MLP :\t"+str(precision_score(ytest_target,ytest_pred)))
print("Recall of the MLP    :\t"+str(recall_score(ytest_target,ytest_pred)))
print("F1 Score of the Model :\t"+str(f1_score(ytest_target,ytest_pred)))
```

```
Confusion Matrix of the Test Set
-----------
[[33180  1202]
 [  100    46]]
Precision of the MLP :  0.03685897435897436
Recall of the MLP    :  0.3150684931506849
F1 Score of the Model : 0.0659971305595409
```

Write code to **weighted prediction accuracy**

That is, sum of the weights of the correct predictions divided by the total weight across all the input elements.

Report the weighted accuracy on the test data.

```
############################################################
#
#  Complete the following function which calculates weight prediction accuracy
#
def weight_accuracy(predicted_y, true_y, weight):
  '''
    Inputs:
         predicted_y:  predicted labels
         true_y:  true labels
         weight:  weight of each sample
    return:
        sum of the weights of the correct predictions divided by the total weight acr
  '''
  difference = [predicted_y[i] == true_y[i] for i in range(len(predicted_y))]
  correct_weight_sum = sum([difference[i] * weight[i] for i in range(len(predicted_y)
  total_weight_sum = sum(weight)
  return correct_weight_sum / total_weight_sum

############################################################


# calculate weighted accuracy on the test data
weight_accuracy(ytest_pred, ytest_target, test_weight)
```

```
0.6400541623620446
```

Problem 2 - CNN

- Define a CNN model to solve the above problem.
- The CNN model receives the same inputs. Instead of using MLP, it uses convolution to extract features. *italicized text*
- Different from 2d images, the convolution will be 1d convolution
- Again, use validation set to design your model and select hyperparamters.
- Report weight accuracy on test set
- Comment on your observations.

CNN Details

- For the CNN model, you have to use the 1D convolution module. The in_channels will be 4, one per DNA base. You can decide what number of out_channels and kernel size you want to use.
- Check out torch Conv1d

```
torch.nn.Conv1d(in_channels, out_channels, kernel_size, stride=1, padding=0)
```

- Note the dimensions of the input required for the 1D convolution -- (N,C,L) a where N is a batch size, C denotes a number of channels, L is a length of signal sequence.
- That is the input sequence has to be 4×101 and not 101×4 as in the data. So you should use torch.swapaxes function to swap the last two axes (not the batch axis) in the forward function.

After the Conv1d, you can apply a relu activation, do dropout, and then try a maxpooling layer (1d). You can try more than one convolution layer too.

Finally, flatten out the last convolution layer and use as input to an MLP.

```
class CNN(nn.Module):
    def __init__(self, out_channels=8, conv_kernel_size=8, maxpooling_kernel_size=16,
        super(CNN, self).__init__()
        self.conv1 = nn.Conv1d(4, out_channels, conv_kernel_size, stride=1, padding=0
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(p=0.1)
        self.maxpooling = nn.MaxPool1d(maxpooling_kernel_size, stride=4)
```

```
            self.mlp = nn.Sequential(
                # nn.Linear(out_channels * (103 - conv_kernel_size - maxpooling_kernel_si
                nn.Linear(161, hidden_size),
                nn.ReLU()
            )
            self.fc = nn.Sequential(
                nn.BatchNorm1d(hidden_size),
                nn.Linear(hidden_size, 1)
            )


        def forward(self, X, a):
            out = torch.swapaxes(X, 1, 2)
            out = self.conv1(out)
            out = self.relu(out)
            out = self.dropout(out)
            out = self.maxpooling(out)
            out = out.reshape(out.size(0),-1)
            out = torch.cat((out, a),1)
            out = self.mlp(out)
            out = self.fc(out)
            return out
    cnn_model = CNN().to(device)
    criterion = nn.BCEWithLogitsLoss()
    num_epochs = 20
    learning_rate = 0.01
    optimizer = torch.optim.SGD(cnn_model.parameters(), lr=learning_rate)

    # Train the model
    total_step = len(train_loader)
    for epoch in range(num_epochs):
        total_loss = 0
        for i, (X, y, w, a) in enumerate(train_loader):
            X = X.to(device)
            y = y.to(device)
            w = w.to(device)
            a = a.to(device)

            # Forward pass
            output = cnn_model(X, a)
            criterion.weight = w
            loss = criterion(output, y)

            # Backward and optimize
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            total_loss += loss.item()

            if (i+1) % 100 == 0:
```

```
        print ('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}'
               .format(epoch+1, num_epochs, i+1, total_step, loss.item()))
    # print ('Epoch [{}/{}], Loss: {:.4f}'.format(epoch+1, num_epochs, total_loss/tot
```

```
Epoch [18/20], Step [2400/2763], Loss: 0.3330
Epoch [18/20], Step [2500/2763], Loss: 0.3286
Epoch [18/20], Step [2600/2763], Loss: 0.3216
Epoch [18/20], Step [2700/2763], Loss: 0.3301
Epoch [19/20], Step [100/2763], Loss: 0.9741
Epoch [19/20], Step [200/2763], Loss: 0.3266
Epoch [19/20], Step [300/2763], Loss: 0.3330
Epoch [19/20], Step [400/2763], Loss: 2.3320
Epoch [19/20], Step [500/2763], Loss: 0.3288
Epoch [19/20], Step [600/2763], Loss: 0.3261
Epoch [19/20], Step [700/2763], Loss: 1.3317
Epoch [19/20], Step [800/2763], Loss: 0.3269
Epoch [19/20], Step [900/2763], Loss: 0.3203
Epoch [19/20], Step [1000/2763], Loss: 0.3159
Epoch [19/20], Step [1100/2763], Loss: 0.3192
Epoch [19/20], Step [1200/2763], Loss: 0.3071
Epoch [19/20], Step [1300/2763], Loss: 0.3047
Epoch [19/20], Step [1400/2763], Loss: 0.8095
Epoch [19/20], Step [1500/2763], Loss: 0.3178
Epoch [19/20], Step [1600/2763], Loss: 0.2989
Epoch [19/20], Step [1700/2763], Loss: 2.3068
Epoch [19/20], Step [1800/2763], Loss: 0.3018
Epoch [19/20], Step [1900/2763], Loss: 0.3078
Epoch [19/20], Step [2000/2763], Loss: 2.2233
Epoch [19/20], Step [2100/2763], Loss: 0.3077
Epoch [19/20], Step [2200/2763], Loss: 0.7627
Epoch [19/20], Step [2300/2763], Loss: 1.9836
Epoch [19/20], Step [2400/2763], Loss: 1.8074
Epoch [19/20], Step [2500/2763], Loss: 2.2180
Epoch [19/20], Step [2600/2763], Loss: 0.6137
Epoch [19/20], Step [2700/2763], Loss: 0.3178
Epoch [20/20], Step [100/2763], Loss: 0.3147
Epoch [20/20], Step [200/2763], Loss: 0.2940
Epoch [20/20], Step [300/2763], Loss: 1.1526
Epoch [20/20], Step [400/2763], Loss: 0.5010
Epoch [20/20], Step [500/2763], Loss: 0.3398
Epoch [20/20], Step [600/2763], Loss: 0.3247
Epoch [20/20], Step [700/2763], Loss: 1.0581
Epoch [20/20], Step [800/2763], Loss: 0.3079
Epoch [20/20], Step [900/2763], Loss: 0.9007
Epoch [20/20], Step [1000/2763], Loss: 0.3017
Epoch [20/20], Step [1100/2763], Loss: 0.2869
Epoch [20/20], Step [1200/2763], Loss: 0.2905
Epoch [20/20], Step [1300/2763], Loss: 0.2810
Epoch [20/20], Step [1400/2763], Loss: 0.2964
Epoch [20/20], Step [1500/2763], Loss: 1.4168
Epoch [20/20], Step [1600/2763], Loss: 0.2935
Epoch [20/20], Step [1700/2763], Loss: 0.9218
Epoch [20/20], Step [1800/2763], Loss: 0.2966
Epoch [20/20], Step [1900/2763], Loss: 0.3213
Epoch [20/20], Step [2000/2763], Loss: 0.7501
Epoch [20/20], Step [2100/2763], Loss: 0.3135
Epoch [20/20], Step [2200/2763], Loss: 1.0474
```

```
Epoch [20/20], Step [2200/2763], Loss: 1.0474
Epoch [20/20], Step [2300/2763], Loss: 0.5496
Epoch [20/20], Step [2400/2763], Loss: 0.3350
Epoch [20/20], Step [2500/2763], Loss: 0.3521
Epoch [20/20], Step [2600/2763], Loss: 0.3307
Epoch [20/20], Step [2700/2763], Loss: 0.3240
```

```
y_pred_list = []
y_target_list = []
weight_list = []

cnn_model = cnn_model.to(torch.device("cpu"))
cnn_model.eval()
#Since we don't need model to back propagate the gradients in test set we use torch.n
# reduces memory usage and speeds up computation
with torch.no_grad():
     for i, (X, y, w, a) in enumerate(test_loader):
        output = cnn_model(X, a)
        y_pred_tag = (output>0).int()
        y_pred_list.append(y_pred_tag.detach().numpy())
        y_target_list.append(y.detach().numpy())
        weight_list.append(w.detach().numpy())

#Takes arrays and makes them list of list for each batch
y_pred_list = [a.squeeze().tolist() for a in y_pred_list]
#flattens the lists in sequence
ytest_pred = list(itertools.chain.from_iterable(y_pred_list))


#Takes arrays and makes them list of list for each batch
y_target_list = [a.squeeze().tolist() for a in y_target_list]
#flattens the lists in sequence
ytest_target = list(itertools.chain.from_iterable(y_target_list))

weight_list = [a.squeeze().tolist() for a in weight_list]
test_weight = list(itertools.chain.from_iterable(weight_list))


conf_matrix = confusion_matrix(ytest_target ,ytest_pred)
print("Confusion Matrix of the Test Set")
print("-----------")
print(conf_matrix)
print("Precision of the MLP :\t"+str(precision_score(ytest_target,ytest_pred)))
print("Recall of the MLP    :\t"+str(recall_score(ytest_target,ytest_pred)))
print("F1 Score of the Model :\t"+str(f1_score(ytest_target,ytest_pred)))
```

```
Confusion Matrix of the Test Set
-----------
[[27184  7198]
 [   76    70]]
Precision of the MLP :  0.009631260319207484
```

```
Recall of the MLP    :  0.4794520547945205
```

```
weight_accuracy(ytest_pred, ytest_target, test_weight)
```

```
0.6350491580385516
```

The CNN model outperforms the basic MLP model by scanning through the sequence with grouping and max pooling. It can better capture the key characteristics of targeting label.

Problem 3 - LSTM

- For the LSTM model, use the encoder followed by a two layer MLP approach. That is, pass the input sequence (batch) through the LSTM and use the last hidden layer as the representation or embedding vector for the sequence. You can choose the dimensionality of the hidden layer. Next, use this vector as input to a two fully connected MLP layers -- the first connects the input vector to the hidden layer (again you can choose the size of the hidden layer), and the second connects the hidden to the output neuron. Use dropout and relu as appropriate.
- Keep in mind that for the input to the LSTM module in pytorch use batch_first=True. This means that the batch dimension comes first, so the input is (N×101×4), which is how the input data is structured. Make note of the output of the LSTM layer so that you store the last hidden layer as the representation, to be used as input to the MLP layers.
- Also, before feeding the output of the hidden layer to the output layer, you must concatenate the accessibility value. So if you are using hidden dimension of 128, then after concatenating the accessibility value, it will become a 129d vector, which should be fed to the final output layer of size 1, since we have a binary class/label.
- You should use binary_cross_entropy_with_logits with weight set to the weights per input element.
- You need to train the model on the training data, and use the validation data to select how many epochs you want to use and to choose the hidden dimension. Use the weighted prediction accuracy as the evaluation metric. That is, sum of the weights of the correct predictions divided by the total weight across all the input elements. Finally, report the weighted accuracy on the test data.

```
class LSTM(nn.Module):
    def __init__(self, lstm_hidden_size=8, hidden_size1=4, hidden_size2=2):
        super(LSTM, self).__init__()
        self.lstm = nn.LSTM(input_size=4, hidden_size=lstm_hidden_size, num_layers=4,
        self.layer1 = nn.Sequential(
            nn.Linear(lstm_hidden_size + 1, hidden_size1),
```

```
            nn.ReLU(),
            nn.Dropout(p=0.2)
        )
        self.layer2 = nn.Sequential(
            nn.Linear(hidden_size1, hidden_size2),
            nn.ReLU(),
            nn.Dropout(p=0.2)
        )
        self.fc = nn.Linear(hidden_size2, 1)


    def forward(self, X, a):
        output, (h_n, c_n) = self.lstm(X)
        out = h_n[-1]
        out = torch.cat((out, a),1)

        out = self.layer1(out)
        out = self.layer2(out)
        out = self.fc(out)
        return out


LSTM_model = LSTM().to(device)
criterion = nn.BCEWithLogitsLoss()
# loss = criterion(output, y)
num_epochs = 15
learning_rate = 0.01
optimizer = torch.optim.SGD(LSTM_model.parameters(), lr=learning_rate)


# Train the model
total_step = len(train_loader)
for epoch in range(num_epochs):
    for i, (X, y, w, a) in enumerate(train_loader):
        X = X.to(device)
        y = y.to(device)
        w = w.to(device)
        a = a.to(device)

        # Forward pass
        output = LSTM_model(X, a)
        criterion.weight = w
        loss = criterion(output, y)

        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if (i+1) % 100 == 0:
```

```
            print ('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}'
                   .format(epoch+1, num_epochs, i+1, total_step, loss.item()))
```

```
Epoch [4/15], Step [1000/2763], Loss: 0.3487
Epoch [4/15], Step [1100/2763], Loss: 0.3435
Epoch [4/15], Step [1200/2763], Loss: 0.3683
Epoch [4/15], Step [1300/2763], Loss: 0.3570
Epoch [4/15], Step [1400/2763], Loss: 0.3479
Epoch [4/15], Step [1500/2763], Loss: 0.3635
Epoch [4/15], Step [1600/2763], Loss: 0.3615
Epoch [4/15], Step [1700/2763], Loss: 1.1566
Epoch [4/15], Step [1800/2763], Loss: 0.3526
Epoch [4/15], Step [1900/2763], Loss: 1.1320
Epoch [4/15], Step [2000/2763], Loss: 1.1479
Epoch [4/15], Step [2100/2763], Loss: 0.3452
Epoch [4/15], Step [2200/2763], Loss: 0.3602
Epoch [4/15], Step [2300/2763], Loss: 1.1525
Epoch [4/15], Step [2400/2763], Loss: 0.3434
Epoch [4/15], Step [2500/2763], Loss: 1.1790
Epoch [4/15], Step [2600/2763], Loss: 0.3401
Epoch [4/15], Step [2700/2763], Loss: 1.1682
Epoch [5/15], Step [100/2763], Loss: 0.3451
Epoch [5/15], Step [200/2763], Loss: 0.3356
Epoch [5/15], Step [300/2763], Loss: 1.1986
Epoch [5/15], Step [400/2763], Loss: 1.1780
Epoch [5/15], Step [500/2763], Loss: 1.1580
Epoch [5/15], Step [600/2763], Loss: 0.3574
Epoch [5/15], Step [700/2763], Loss: 1.9515
Epoch [5/15], Step [800/2763], Loss: 1.1704
Epoch [5/15], Step [900/2763], Loss: 0.3585
Epoch [5/15], Step [1000/2763], Loss: 0.3516
Epoch [5/15], Step [1100/2763], Loss: 0.3386
Epoch [5/15], Step [1200/2763], Loss: 0.3411
Epoch [5/15], Step [1300/2763], Loss: 0.3457
Epoch [5/15], Step [1400/2763], Loss: 0.3560
Epoch [5/15], Step [1500/2763], Loss: 1.1683
Epoch [5/15], Step [1600/2763], Loss: 0.3548
Epoch [5/15], Step [1700/2763], Loss: 0.3360
Epoch [5/15], Step [1800/2763], Loss: 0.3476
Epoch [5/15], Step [1900/2763], Loss: 1.9549
Epoch [5/15], Step [2000/2763], Loss: 1.1406
Epoch [5/15], Step [2100/2763], Loss: 0.3602
Epoch [5/15], Step [2200/2763], Loss: 0.3588
Epoch [5/15], Step [2300/2763], Loss: 1.1505
Epoch [5/15], Step [2400/2763], Loss: 0.3482
Epoch [5/15], Step [2500/2763], Loss: 0.3522
Epoch [5/15], Step [2600/2763], Loss: 0.3432
Epoch [5/15], Step [2700/2763], Loss: 0.3445
Epoch [6/15], Step [100/2763], Loss: 0.3511
Epoch [6/15], Step [200/2763], Loss: 1.1434
Epoch [6/15], Step [300/2763], Loss: 0.3631
Epoch [6/15], Step [400/2763], Loss: 0.3750
Epoch [6/15], Step [500/2763], Loss: 0.3840
Epoch [6/15], Step [600/2763], Loss: 0.3684
Epoch [6/15], Step [700/2763], Loss: 0.3515
Epoch [6/15], Step [800/2763], Loss: 1.1470
Epoch [6/15], Step [900/2763], Loss: 0.3557
```

```
Epoch [6/15], Step [1000/2763], Loss: 0.3487
Epoch [6/15], Step [1100/2763], Loss: 0.3406
Epoch [6/15], Step [1200/2763], Loss: 1.9792
Epoch [6/15], Step [1300/2763], Loss: 1.1516
```

```
y_pred_list = []
y_target_list = []
weight_list = []

LSTM_model = LSTM_model.to(torch.device("cpu"))
LSTM_model.eval()
#Since we don't need model to back propagate the gradients in test set we use torch.n
# reduces memory usage and speeds up computation
with torch.no_grad():
     for i, (X, y, w, a) in enumerate(test_loader):
        output = LSTM_model(X, a)
        y_pred_tag = (output>0).int()
        y_pred_list.append(y_pred_tag.detach().numpy())
        y_target_list.append(y.detach().numpy())
        weight_list.append(w.detach().numpy())

#Takes arrays and makes them list of list for each batch
y_pred_list = [a.squeeze().tolist() for a in y_pred_list]
#flattens the lists in sequence
ytest_pred = list(itertools.chain.from_iterable(y_pred_list))


#Takes arrays and makes them list of list for each batch
y_target_list = [a.squeeze().tolist() for a in y_target_list]
#flattens the lists in sequence
ytest_target = list(itertools.chain.from_iterable(y_target_list))

weight_list = [a.squeeze().tolist() for a in weight_list]
test_weight = list(itertools.chain.from_iterable(weight_list))


conf_matrix = confusion_matrix(ytest_target ,ytest_pred)
print("Confusion Matrix of the Test Set")
print("-----------")
print(conf_matrix)
print("Precision of the MLP :\t"+str(precision_score(ytest_target,ytest_pred)))
print("Recall of the MLP    :\t"+str(recall_score(ytest_target,ytest_pred)))
print("F1 Score of the Model :\t"+str(f1_score(ytest_target,ytest_pred)))
```

```
weight_accuracy(ytest_pred, ytest_target, test_weight)
```

The long short-term memory model combines both recent and older seen data, so it is especially to process sequenced input such as gene. While it gives the implementer freedom with various hyper parameters, it is also challenging to choose the optimal ones to produce a good result. I need a better understanding of the working mechanism behind the model to give a better implementation.

Statement of Collaboration

It is mandatory to include a Statement of Collaboration in each submission, with respect to the guidelines below. Include the names of everyone involved in the discussions (especially in-person ones), and what was discussed.

All students are required to follow the academic honesty guidelines posted on the course website. For programming assignments, in particular, I encourage the students to organize (perhaps using Campuswire) to discuss the task descriptions, requirements, bugs in my code, and the relevant technical content before they start working on it.

However, you should not discuss the specific solutions, and, as a guiding principle, you are not allowed to take anything written or drawn away from these discussions (i.e. no photographs of the blackboard, written notes, referring to Campuswire, etc.).

Especially after you have started working on the assignment, try to restrict the discussion to Campuswire as much as possible, so that there is no doubt as to the extent of your collaboration.

Complete your statement of collaboration here:

No collaboration

What to submit

- Export a notebook as PDF
    - Go to Main menu | File and select Print . pdf.

- Upload your jupyter notebook PDF on gradescope
- The notebook must have output values for the final test accuracy.
- Do not submit the data file or directories.

Created in Deepnote