ECG Data Classification with MINA

Overview

In this section, you will implement an advanced CNN+RNN model with attention mechanism to classify ECG recordings. Specifically, we face a binary classification problem, and the goal is to distinguish atrial fibrillation (AF), an alternative rhythm, from the normal sinus rhythm.

22 EXERCISES

In [1]:

```python
import os
import random
import numpy as np
import pandas as pd
import torch
import torch.nn as nn
import torch.nn.functional as F

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

# define data path
DATA_PATH = "../HW4_MINA-lib/data/"
```

1 ECG Data Data [10 points]

We will be using a fraction of the data in the public Physionet 2017 Challenge. More details can be found in the link.

ECG recordings were sampled at 300Hz, and for the purpose of this task, the data we use is separated into 10-second-segments.

1.1 Preprocessing

Because the preprocessing of the data requires a tremendous amount of memory and time, for the sake of this homework, the data has already been preprocessed.

Specifically, for each raw data (an ECG recording sampled at 300Hz), we did the following:

1. split the dataset into training/validation/test sets with a ratio of [placeholder]
2. for each recording, we normalize the data to have a mean of 0 and a standard deviation of 1
3. slide and cut the recording into overlapping 10-second-segments (stride = $\frac{5}{3}$ second for class 0, and $\frac{5}{30}$ second for class 1 to oversample).
4. use FIR bandpass filter to transform the data from 1 channel to 4 channels.

The last step of the data preprocessing is computing the knowledge. As we can see below, the AF signals exhibit different patterns at different levels. We computed knoledge features at different levels to guide the attention mechanism. More details are in Section 2. Beat/Rhythm/Frequency

1.2 Load the Data [10 points]

Due to the resource constraints, the data and knowledge features have already been computed. Let's load them below.

In [2]:

```python
train_dict = pd.read_pickle(os.path.join(DATA_PATH, 'train.pkl'))
test_dict = pd.read_pickle(os.path.join(DATA_PATH, 'test.pkl'))

print(f"There are {len(train_dict['Y'])} training data, {len(test_dict['Y'])} test dat
print(f"Shape of X: {train_dict['X'][:, 0,:].shape} = (#channels, n)")
print(f"Shape of beat feature: {train_dict['K_beat'][:, 0, :].shape} = (#channels, n)'
print(f"Shape of rhythm feature: {train_dict['K_rhythm'][:, 0, :].shape} = (#channels,
print(f"Shape of frequency feature: {train_dict['K_freq'][:, 0, :].shape} = (#channels
```

In [3]:

```python
train_dict = pd.read_pickle(os.path.join(DATA_PATH, 'train.pkl'))
test_dict = pd.read_pickle(os.path.join(DATA_PATH, 'test.pkl'))

print(f"There are {len(train_dict['Y'])} training samples \n \
ie: len(train_dict['Y'] = {len(train_dict['Y'])} training samples \n \
There are {len(test_dict['Y'])} test samples \n \
ie: len(test_dict['Y'] = {len(test_dict['Y'])}")
print(f"Shape of X, \n \
ie: train_dict['X'][:, 0,:]: {train_dict['X'][:, 0,:].shape} = (#channels, n)")
print(f"Shape of beat feature, \n \
ie: train_dict['K_beat'][:, 0, :]: {train_dict['K_beat'][:, 0, :].shape} = (#channels,
print(f"Shape of rhythm feature, \n \
ie: train_dict['K_rhythm'][:, 0, :]: {train_dict['K_rhythm'][:, 0, :].shape} = (#chann
print(f"Shape of frequency feature: \n \
ie: train_dict['K_freq'][:, 0, :]: {train_dict['K_freq'][:, 0, :].shape} = (#channels,
```

You will need to define a ECGDataset class, and then define the DataLoader as well.

In [4]:

```python
# [x**2 for x in range(10)] # [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
[(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
```

In [5]:

```python
a = torch.ones(1,0,1)
b = torch.ones(1,1,1)
c = torch.cat((a,b), dim=1)
print("c.shape: ", c.shape)
```

In [6]:

```python
from torch.utils.data import Dataset
class ECGDataset(Dataset):
    def __init__(self, data_dict):
        """
        TODO: init the Dataset instance.
        """
        #  code

        self.dataset = data_dict

        # END

    def __len__(self):
        """
        TODO: Denotes the total number of samples
        """
        # CODE

        return len(self.dataset['Y'])
        #return len(self.y)

        # END

    def __getitem__(self, i):
        """
        TODO: Generates one sample of data as?
            return the ((X, K_beat, K_rhythm, K_freq), Y) for the i-th data.
            Do not return ((X, K_beat, K_rhythm, K_freq), Y)
            Be careful which dimension you are indeXing.
        """
        # CODE

        return ((torch.tensor(self.dataset['X'][:, i, :], dtype = torch.float32),
        torch.tensor(self.dataset['K_beat'][:, i, :], dtype = torch.float32),
        torch.tensor(self.dataset['K_rhythm'][:, i, :], dtype = torch.float32),
        torch.tensor(self.dataset['K_freq'][:, i, :], dtype = torch.float32)),
        torch.tensor(self.dataset['Y'][i], dtype=torch.long))

        # END
### -----END CLASS DEF------ ###

from torch.utils.data import DataLoader
def load_data(dataset, batch_size=128):
    """
    Return a DataLoader object with a Dataset object and and batch size.
    Note that since the data has already been shuffled, we set shuffle=False
    """

    def my_collate(batch):

        """
        param: batch: this is essentially [dataset[i] for i in [...]]
        batch[i] should be ((Xi, Ki_beat, Ki_rhythm, Ki_freq), Yi)

        TODO: write a collate function such that it outputs ((X, K_beat, K_rhythm, K_
            each output variable is a batched version of what's in the input *batch*,
            [dataset[i] for i in [...]]
            each output variable should be either float tensor, eXcept Y is long tens
            If applicable, channel dim precedes batch dim
            e.g. the shape of each Xi is (# channels, n). In the output, X should be
        """
        # CODE

        # collect X1, X2, ... , X(LEN(BATCH), each w size = (# channels, n) = (4, 300

        # batch is a (128 X 2) list,
        # each rows is a two item tuple like (Xi, Ki_beat, Ki_rhythm, Ki_freq), Yi)
#

#           (list_X, list_K_beat, list_K_rhythm, list_K_freq), list_Y = zip(*batch)

#           (list_X, list_K_beat, list_K_rhythm, list_K_freq), list_Y = ([], [], [], [],
        # create zero tensors
        X = torch.zeros(4, 0, 3000)
        K_beat = torch.zeros(4, 0, 3000)
        K_rhythm = torch.zeros(4, 0, 60)
        K_freq = torch.zeros(4, 0, 1)
        Y = torch.zeros(0)

        for i, element in enumerate(batch):
            X = torch.cat((X, torch.tensor(element[0][0]).unsqueeze(dim=1)), dim = 1)
            K_beat = torch.cat((K_beat, torch.tensor(element[0][1]).unsqueeze(dim=1)),
            K_rhythm = torch.cat((K_rhythm, torch.tensor(element[0][2]).unsqueeze(dim=
            K_freq = torch.cat((K_freq, torch.tensor(element[0][3]).unsqueeze(dim=1)),

            Y = torch.cat((Y, torch.tensor(element[1]).unsqueeze(dim=0)), dim=0)

        # END

        return (X.float(), K_beat.float(), K_rhythm.float(), K_freq.float()), Y

        # my_collate takes batch = [dataset[i] for i in [...]]
        # and returns batch[i], batch[1], batch[2], ... batch[batch_size?],
        # batch[i] = ((Xi, Ki_beat, Ki_rhythm, Ki_freq), Yi)

    # load_data takes dataset oject and batch size and returns a dataloader object.
    return torch.utils.data.DataLoader(dataset, batch_size=batch_size, shuffle=False,


train_loader = load_data(ECGDataset(train_dict))
test_loader = load_data(ECGDataset(test_dict))
```

Why is there no test on test_loader. Why can I pass the test provided when test_loader cannot do what it is meant to do? next(iter(test_loader)) returns an error.

Use this to run one iteration and get one batch.
`z = next(iter(test_loader))`

In [7]:

```python
print(np.asarray(list(zip(next(iter(test_loader))))).shape)
```

next(iter(test_loader))

In [8]:

```python
assert iter(train_loader).next()[0][0].shape == (4, 128, 3000)
assert iter(train_loader).next()[0][1].shape == (4, 128, 3000)
assert iter(train_loader).next()[0][2].shape == (4, 128, 60)
assert iter(train_loader).next()[1].shape == (128,)
```

In [9]:

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

assert len(train_loader.dataset) == 1696, "Length of training data incorrect."
assert len(train_loader) == 14, "Length of the training dataloader incorrect - maybe
assert [x.shape for x in train_loader.dataset[0][0]] == [(4,3000), (4,3000), (4,60),
```

In [ ]:

2 Model Defintions [75 points]

Now, let us implement a model that involves RNN, CNN and attention mechanism. More specifically, we will implement MINA: Multilevel Knowledge-Guided Attention for Modeling Electrocardiography Signals.

2.1 Knowledge-guided attention [15 points]

Knowledge-guided attention is an attention mechanism that introduces prior knowledge (such as features proposed by human experts) in the features used by the attention mechanism. We will first define the general KnowledgeAttn module, and use it at different levels later.

There are three steps:

- 1. concatenate the input (X) and knowledge (K).
- 2. pass it through a linear layer, a tanh, another linear layer, and softmax: $attn = softmax(V^\top \tanh(W^\top \begin{bmatrix} X \\ K \end{bmatrix}))$
- 3. use attention values to sum X: $output = \sum_{i=1}^{D} attn_i x_i$ where $attn_i$ is a scalar and x_i is a vector.

In [10]:

```python
import torch.nn.functional as F

class KnowledgeAttn(nn.Module):
    def __init__(self, input_features, attn_dim):
        """
        This is the general knowledge-guided attention module.

        It will:
        1. transform the input and knowledge with 2 linear layers
        2. compute attention
        3. aggregate.

        :param input_features: the number of features for each
        :param attn_dim: the number of hidden nodes in the attention mechanism

        TODO:
            define the following 2 linear layers WITHOUT bias (with the names provided
                att_W: a Linear layer of shape (input_features + n_knowledge, attn_dim
                att_v: a Linear layer of shape (attn_dim, 1)
            init the weights using self.init() (already given)
        """

        super(KnowledgeAttn, self).__init__()

        self.input_features = input_features
        self.attn_dim = attn_dim
        self.n_knowledge = 1

        # CODE

        # self.att_W = nn.Linear(input_features + n_knowledge, attn_dim)
        # self.att_v = nn.Linear(attn_dim, 1)

        # n_knowledge not defined

        self.att_W = nn.Linear(self.input_features + self.n_knowledge, self.attn_dim,
        self.att_v = nn.Linear(self.attn_dim, 1, bias = False)

        # END

        self.init() # init the weights

    def init(self):

        nn.init.normal_(self.att_W.weight)
        nn.init.normal_(self.att_v.weight)

    @classmethod
    def attention_sum(cls, x, attn):
        """

        :param x: of shape (-1, D, nfeatures)
        :param attn: of shape (-1, D, 1)

        TODO: return the weighted sum of x along the middle axis with weights even in
        output should be (-1, nfeatures)
        """

        # CODE

        # attn[1], i = 1, 2, ... D is a scalar,
        # x[i], i = 1, 2, ..., D is a vector
        # x[-1, 1, 1], vector multiplied elementwise by attn[-1, 1, 1], scalar
        # x[-1, 2, 1], vector multiplied elementwise by attn[-1, 2, 1], scalar,
        # plane is repeated allong dim=2 by broadcasting
        ...
        # in the equation above is attn[i]x[i] elementwise or dot product??
        # x[-1, D, 1], " " " "   attn[-1, D, 1]


        # output should be (-1, nfeatures)

        return torch.sum(attn * x, dim = 1)

        # END


    def forward(self, x, k):
        """
        :param x: shape of (-1, D, input_features)
        :param k: shape of (-1, D, 1)
        :return:
            out: shape of (-1, input_features), the aggregated x
            attn: shape of (-1, D, 1)
        TODO:
            concatenate the input x and knowledge k together (on the last dimension)
            pass the concatenated output through the learnable Linear transforms
                first att_W, then tanh, then att_v
                the output shape should be (-1, D, 1)
            to get attention values, apply softmax on the output of linear layer
                You could use F.softmax(). Be careful which dimension you apply softma
            aggregate x using the attention values via self.attention_sum, and return
        """
        # CODE

        # concatenate the input x and knowledge k together (on the last dimension)

        xk = torch.cat((x, k), dim = -1)

        # pass the concatenated output through the learnable Linear transforms

        xk = self.att_W(xk)
        # xk = self.tanh(xK) # 'KnowledgeAttn' object has no attribute 'tanh'
        xk = torch.tanh(xk)
        xk = self.att_v(xk) # out_features = 1  # xk.shape = (-1, D, 1) ??

        # apply softmax on the output of linear layer to get attention values,
        # You could use F.softmax(). Be careful which dimension you apply softmax ove

        # attn: shape of (-1, D, 1)

        attn =  F.softmax(xk, dim = 1)

        # aggregate x using the attention values via self.attention_sum and return.
        # Q: Aggregate x or xk?
        # A: x because the concatenation and transformations are only to get the attn
        # then attn values are simply applied to x
        # out: shape of (-1, input_features), the aggregated x

        out = self.attention_sum(x, attn)

        # END

        return out, attn
```

In [11]:

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

def float_tensor_equal(a, b, eps=1e-3):
    return torch.norm(a-b).abs().max().tolist() < eps

def testKnowledgeAttn():
    m = KnowledgeAttn(2, 2)
    m.att_W.weight.data = torch.tensor([[0.3298,  0.7045, -0.1067],
                                        [0.9656,  0.3090,  1.2627]], requires_grad=Tru
    m.att_v.weight.data = torch.tensor([[-0.2368,  0.5824]], requires_grad=True)

    x = torch.tensor([[[-0.6898, -0.9098], [0.0230,  0.2879], [-0.2534, -0.3190]],
                      [[ 0.5412, -0.3434], [0.0289, -0.2837], [-0.4120, -0.7858]]])
    k = torch.tensor([[ 0.5469,  0.3948, -1.1430], [0.7815, -1.4787, -0.2929]]).unsque
    out, attn = m(x, k)

    tout = torch.tensor([[-0.2817, -0.2531], [0.2144, -0.4387]])
    tattn = torch.tensor([[[0.3482], [0.4475], [0.2043]],
                          [[0.5696], [0.1894], [0.2410]]])
    assert float_tensor_equal(attn, tattn), "The attention values are wrong"
    assert float_tensor_equal(out, tout), "output of the attention module is wrong"

testKnowledgeAttn()
```

In [ ]:

2.2 MINA [60 points]

We will now use the knowledge-guided attention mechanism to construct MINA. The overall structure is show below. From "Input" to "Sliding Window Segmentation" has already been done in the data preprocessing part, and in this section we will need to define things above "Segment" MINAstructure

Here, CNN (`BeatNet`) is used to capture beat information, Bi-LSTM (`RhythmNet`) is used to capture rhythm level information, and the from $c^{(i)}$ to p is aggregating frequency levle infomration (`FreqNet`). Note that although the input has 4 channels, we actually need to handle each channel separately because they have different meanings after we did the FIR. Thus, we will need 4 `BeatNet` s, 4 `RhythmNet` s, and 1 `FreqNet` .

We will need:

- 4 CNN (BeatNet) n
- 4 Bi-LSTM (RhythmNet)
- one `FreqNet` , ie from c(i) to p above.

`BeatNet` and `RhythmNet` act on each of four channels separately.

MINA has three different knowledge guided attention mechanisms:

- Beat Level K_{beat}: extract beat knowledge which is represented by the first-order difference and a convolutional operation $Conv_\alpha$ for each segment
- Rhythm Level K_{rhythn}: extract rhythm features represented by the standard deviation on each segment
- Frequency Level K_{freq}: frequency features are represented by the power spectral density (PSD), which is a popular measure of energy in signal processing.

2.2.1 BeatNet [20 points]

For BeatNet, the attention α is computed by the following:

$$\alpha = softmax(V_\alpha^\top \tanh(W_\alpha^\top \begin{bmatrix} \mathbf{L} \\ \mathbf{K}_{beat} \end{bmatrix}))$$

Here, L is output by the convolutional layers, and K_{beat} is the computed beat level knowledge features.

Attention for the CNN step/ beat level/local information:

In [12]:

```python
class BeatNet(nn.Module):

    def __init__(self, n=3000, T=50,
                 conv_out_channels=64):
        """
        :param n: size of each 10-second-data
        :param T: size of each smaller segment used to capture local information in th
        :param conv_out_channels: also called number of filters/kernels

        TODO: We will define a network that does two things. Specifically:
            1. use one 1-D convolutional layer to capture local informatoin, on x and
            the computed beat level knowledge features, (see forward())
                conv:
                 -The kernel size should be set to 32
                 -the number of filters should be set to *conv_out_channels*.
                 -Stride should be *conv_stride*
                conv_k: same as conv, except that it has only 1 filter instead of *con
            2. an attention mechanism to aggregate the convolution outputs. Specifical
                attn: KnowledgeAttn with
                -input_features equaling conv_out_channels,
                -attn_dim = att_cnn_dim
        """
        super(BeatNet, self).__init__() # multiple inheritance
        self.n, self.M, self.T = n, int(n/T), T
        self.conv_out_channels = conv_out_channels
        self.conv_kernel_size = 32
        self.conv_stride = 2

        # Define conv and conv_k, the two Conv1d modules

        # CODE

        self.conv = nn.Conv1d(in_channels = 1,
                              out_channels = self.conv_out_channels,
                              kernel_size = 32,
                              stride = self.conv_stride
                             )

        self.conv_k = nn.Conv1d(in_channels = 1,
                              out_channels = 1,
                              kernel_size = 32,
                              stride = self.conv_stride
                             )

        # END

        self.att_cnn_dim = 8

        #Define attn, the KnowledgeAttn module

        # CODE

        self.attn = KnowledgeAttn(input_features = conv_out_channels,
                                  attn_dim = self.att_cnn_dim
                                   )
        self.rlu = nn.ReLU()

        # END

    def forward(self, x, k_beat):
        """
        :param x: shape (batch, n)
        :param k_beat: shape (batch, n)
        :return:
            out: shape (batch, M, self.conv_out_channels)
            alpha: shape (batch * M, N, 1) where N is a result of convolution
        TODO:
            [Given] reshape the data - convert x/k_beat of shape (batch, n) to (batch
                Use torch.Tensor.view() for all reshapes in this HW
            apply convolution on x and k_beat
                pass the reshaped x through self.conv, and then ReLU
                pass the reshaped k_beat through self.conv_k, and then ReLU

            (at this step, you might need to swap axix 1 & 2 to align the dimensions

            pass the conv'd x and conv'd knowledge through self.attn to get the output
            [Given] reshape the output *out* to be of shape (batch, M, self.conv_out_
        """

        #### Prior to given reshaping ####

        # batch = 37, n = 408

        # convert x/k_beat of shape (batch, n) to (batch * M, 1, T), where n = MT
        # M = 12
        # T = 34
        x = x.view(-1, self.T).unsqueeze(1)
        k_beat = k_beat.view(-1, self.T).unsqueeze(1)

        # CODE

        # x.shape:  torch.Size([444, 1, 34])

        # k_beat.shape:  torch.Size([444, 1, 34])

        x = self.conv(x)
        x = self.rlu(x)
        x = torch.permute(x, (0, 2, 1))

        k_beat = self.conv_k(k_beat)
        k_beat = self.rlu(k_beat)
        k_beat = torch.permute(k_beat, (0, 2, 1))


        # swap axis 1 & 2 so that self.attn runs.
        # KnowledgeAttn uses torch.cat((x,k), dim = -1)
        # this requries all but the last dimension to match

        out, alpha = self.attn(x, k_beat)

        # END

        out = out.view(-1, self.M, self.conv_out_channels)

        return out, alpha
```

In [13]:

```python
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''
_testm = BeatNet(12 * 34, 34, 56)
assert isinstance(_testm.conv, torch.nn.Conv1d) and isinstance(_testm.conv_k, torch.nn
assert _testm.conv.bias.shape == torch.Size([56]) and _testm.conv.weight.shape == torc
assert _testm.conv_k.bias.shape == torch.Size([1]) and _testm.conv_k.weight.shape == t
assert isinstance(_testm.attn, KnowledgeAttn), "Should use one KnowledgeAttn Module"

_out, _alpha =_testm(torch.randn(37, 12*34), torch.randn(37, 12*34))
assert _alpha.shape == torch.Size([444,2,1]), "The attention's dimension is incorrect"
assert _out.shape==torch.Size([37, 12,56]), "The output's dimension is incorrect"
del _testm, _out, _alpha
```

In [ ]:

2.2.2 RhythmNet [20 points]

For Rhythm, the attention β is computed by the following:

$$\beta = softmax(V_\beta^\top \tanh(W_\beta^\top \begin{bmatrix} \mathbf{H} \\ \mathbf{K}_{rhythm} \end{bmatrix}))$$

Here, $\mathbf{H}$ is output by the Bi-LSTMs, and K_{rhythm} is the computed rhythm level knowledge features.

TODO: We will define a network that does two things to handle rhythms. Specifically:

```
1. use a bi-directional LSTM to process the learned local representations from
the CNN part
2. an attention mechanism to aggregate the convolution outputs. Specifically:
3. output layers
```

torch.nn.LSTM()
See:

- parameters
- inputs: input, (h_0, c_0)
- output: output, (h_n, c_n)
- examples:

```
rnn = nn.LSTM(10, 20, 2)
input = torch.randn(5, 3, 10)
h0 = torch.randn(2, 3, 20)
c0 = torch.randn(2, 3, 20)
output, (hn, cn) = rnn(input, (h0, c0))
```

torch.nn.Linear()

Applies linear trans to input, $O_{ut} = I_n W_{eights}^T + b$
Takes in_f, out_f = last dim of in and out matrices. All other dims must match. weights.shape = (out_f, in_f)

In [14]:

```python
class RhythmNet(nn.Module):
    def __init__(self, n=3000, T=50, input_size=64, rhythm_out_size=8):
        """
        :param n: size of each 10-second-data
        :param T: size of each smaller segment used to capture local information in th
        :param input_size: This is the same as the # of filters/kernels in the CNN pa
        :param rhythm_out_size: output size of this netowrk

        """

        #input_size is the cnn_out_channels
        super(RhythmNet, self).__init__()
        self.n, self.M, self.T = n, int(n/T), T
        self.input_size = input_size

        self.rnn_hidden_size = 32

        ### define lstm: LSTM Input is of shape (batch size, M, input_size)

        # CODE

        # lstm: bidirectional, 1 layer, batch_first, and hidden_size should be set to
        # size = features


        self.lstm = nn.LSTM(input_size = self.input_size,
                            hidden_size = self.rnn_hidden_size,
                            bidirectional = True,
                            batch_first = True
                           )
        # END

        ### Attention mechanism: define attn to be a KnowledgeAttn

        self.att_rnn_dim = 8

        # CODE

        # attn: KnowledgeAttn with input_features equaling lstm output ???, and attn_

        # the LSTM output size is the hidden_size argument, except in bidir is 2 time
        # When you have a bidirectional LSTM, it concatenates the forward and reverse
        # So, your input size to attn will be double that of the hidden_size which in

        self.attn = KnowledgeAttn(input_features = 2 * self.rnn_hidden_size, attn_dim

        # END

        ### Define the Dropout and fully connecte layers (fc and do)

        self.out_size = rhythm_out_size

        # CODE

        # fc: a Linear layer making the output of shape (..., self.out_size)
        # weights.shape = (self.out_size, 2 * self.rnn_hidden_size)
        self.fc = nn.Linear(in_features = 2 * self.rnn_hidden_size, out_features = sel

        # do: a Dropout layer with p=0.5, default
        self.do = nn.Dropout()

        self.rl = nn.ReLU()

        # END

    def forward(self, x, k_rhythm):
        """
        :param x: shape (batch, M, self.input_size)
        :param k_rhythm: shape (batch, M)
        :return:
            out: shape (batch, self.out_size)
            beta: shape (batch, M, 1)
        """

        # CODE


        # reshape the k_rhythm->(batch, M, 1) (HINT: use k_rhythm.unsqueeze())
        k_rhythm = k_rhythm.unsqueeze(dim = 2)

        # pass the RESHAPED x through lstm
        output, (h_n, c_n) = self.lstm(x)
        # output, (hn, cn) = self.lstm(input, (h0, c0))

        # denote the final output as *out*, and the attention output as *beta*
        # pass the lstm output and knowledge through attn
        # assign the results to out, beta

        out, beta = self.attn(output, k_rhythm) # beta = out, pass the aggregated x va

        # pass the output results, not the attn, through fully connected layer - ReLU

        out = self.fc(out.expand(-1, 2 * self.rnn_hidden_size))
        out = self.rl(out)
        out = self.do(out)

        # END

        return out, beta
```

In [15]:

```python
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''
_B, _M, _T = 17, 23, 31
_testm = RhythmNet(_M * _T, _T, 37)
assert isinstance(_testm.lstm, torch.nn.LSTM), "Should use nn.LSTM"
assert _testm.lstm.bidirectional, "LSTM should be bidirectional"
assert isinstance(_testm.attn, KnowledgeAttn), "Should use one KnowledgeAttn Module"
assert isinstance(_testm.fc, nn.Linear) and _testm.fc.weight.shape == torch.Size([8,6
assert isinstance(_testm.do, nn.Dropout), "Dropout layer is not defined correctly"

_out, _beta = _testm(torch.randn(_B, _M, 37), torch.randn(_B, _M))
assert _beta.shape == torch.Size([_B,_M,1]), "The attention's dimension is incorrect"
assert _out.shape==torch.Size([_B, 8]), "The output's dimension is incorrect"
del _testm, _out, _beta,  _B, _M, _T
```

In [ ]:

2.2.3 FreqNet [20 points]

The attention γ is computed by the following:

$$\gamma = softmax(V_\gamma^\top \tanh(W_\gamma^\top \begin{bmatrix} \mathbf{Q} \\ \mathbf{K}_{freq} \end{bmatrix}))$$

Here, $\mathbf{Q}$ is output of the RhythmNets, and K_{freq} is the computed frequency level knowledge features.

This is the main network that orchestrates the previously defined attention modules:
A. definition ie `init_` :

1. define n_channels many BeatNet and RhythmNet modules. e
2. define frequency (channel) level knowledge-guided attention module
3. output layer: a Linear layer for 2 classes output

Execution: ie `forward` method: Use the attention submodules to process data from each channel separately, and then pass the output through an attention on frequency for the final output

nn.ModuleList()
Holds submodules in a list.

ModuleList can be indexed like a regular Python list, but modules it contains are properly registered, and will be visible by all Module methods.

Example:

```
class MyModule(nn.Module):
    def __init__(self):
        super().__init__()
        self.linears = nn.ModuleList([nn.Linear(10, 10) for i in range(10)])

    def forward(self, x):
        # ModuleList can act as an iterable, or be indexed using ints
        for i, l in enumerate(self.linears):
            x = self.linears[i // 2](x) + l(x)
        return x
```

append() Appends a given module of nn.Module type to the end of the list.

In [16]:

```python
from IPython.core.debugger import set_trace

class FreqNet(nn.Module):
    def __init__(self, n_channels=4, n=3000, T=50):
        """
        This is the main network that orchestrates the previously defined attention mo
        """
        super(FreqNet, self).__init__()

        self.n, self.M, self.T = n, int(n / T), T
        # :param n_channels: number of channels (F in the paper). Define #channels # 
        # :param n: size of each 10-second-data
        # :param T: size of each smaller segment used to capture local information in

        self.n_class = 2
        self.n_channels = n_channels

        self.conv_out_channels = 64
        self.rhythm_out_size = 8
        self.beat_nets = nn.ModuleList()
        self.rhythm_nets = nn.ModuleList()

        #use self.beat_nets.append() and self.rhythm_nets.append() to append 4 BeatNe

        # CODE

        # 1. define n_channels many BeatNet and RhythmNet modules. (Hint: use nn.Modu
            # beat_nets: for each, pass parameter conv_out_channel into the init()
            # rhythm_nets: for each, pass conv_out_channel as input_size,
            #                     and self.rhythm_out_size as the output size

        for i in range(self.n_channels):
            self.beat_nets.append(BeatNet(n = self.n,
                                          T = self.T,
                                          conv_out_channels=self.conv_out_channels
                                         )
                                 )

            self.rhythm_nets.append(RhythmNet(n = self.n,
                                              T = self.T,
                                              input_size = self.conv_out_channels,
                                              rhythm_out_size = self.rhythm_out_size
                                             )
                                   )
        # END


        self.att_channel_dim = 2

        # 2. define frequency (channel) level knowledge-guided attention module
          # attn: KnowledgeAttn with input_features equaling rhythm_out_size, and att

        # CODE

        self.attn = KnowledgeAttn(input_features = self.rhythm_out_size, \
                                  attn_dim = self.att_channel_dim
                                 )
        # END

        ### Create the fully-connected output layer (fc)

        # 3. output layer: a Linear layer for 2 classes output

        # CODE

        self.fc = nn.Linear(in_features = self.rhythm_out_size,
                            out_features = self.n_class) ### ???

        # END


        """
        :param x: shape (n_channels, batch, n)
        :param k_beats: (n_channels, batch, n)
        :param k_rhythms: (n_channels, batch, M)
        :param k_freq: (n_channels, batch, 1)

        :return:
```

```python
        out: softmax output for each data point, shpae (batch, n_class)
        gama: the attention value on channels

    TODO:
        1. [Given] pass each channel of x through the corresponding beat_net, then
           We will discard the attention (alpha and beta) outputs for now
           Using ModuleList for self.beat_nets/rhythm_nets is necessary for the
        2. [Given] stack the output from 1 together into a tensor of shape (batch,
        3. pass result from 2 and k_freq through attention module, to get the agg
           You might need to do use k_freq.permute() to tweak the shape of k_fre
        4. pass aggregated result from 3 through the final fully connected layer.
        5. Apply Softmax to normalize output to a probability distribution (over
    """
    def forward(self, x, k_beats, k_rhythms, k_freq):

        new_x = [None for _ in range(self.n_channels)]

        # 1. pass each channel of x through the corresponding beat_net, then rhythm_n
            # Do not collect the attention outputs
            # We used ModuleList so that the gradient would propagate

        for i in range(self.n_channels):
            tx, _ = self.beat_nets[i](x[i], k_beats[i])
            new_x[i], _ = self.rhythm_nets[i](tx, k_rhythms[i])


        # 2. stack the output into a tensor with shape (batch, n_channels, rhythm_out
        # output shape: [128,8]
#         stack along dim = 1 to get [128,4,8]
        x = torch.stack(new_x, 1)

        # CODE

        # 3. pass result from 2 and k_freq through attention module, to get the aggre
        # k_freq.shape:  torch.Size([4, 17, 1]) => 17, 4, 1
        # out: shape of (-1, input_features)

        out, gama = self.attn(x, torch.permute(k_freq, (1, 0, 2)))

        # 4. pass aggregated result from 3 through the final fully connected layer.
        out = self.fc(out)

        # 5. Apply Softmax to normalize output to a probability distribution (over 2
        out = torch.softmax(out, dim = 1)

        # EMD

        return out, gama
```

In [17]:

```python
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''
_B, _M, _T = 17, 59, 109
_testm = FreqNet(n=_M * _T, T=_T)
assert isinstance(_testm.attn, KnowledgeAttn), "Should use one KnowledgeAttn Module"
assert isinstance(_testm.fc, nn.Linear) and _testm.fc.weight.shape == torch.Size([2,8
assert isinstance(_testm.beat_nets, nn.ModuleList), "beat_nets has to be a ModuleList

_out, _gamma = _testm(torch.randn(4, _B, _M * _T), torch.randn(4, _B, _M * _T), torch
assert _gamma.shape == torch.Size([_B, 4, 1]), "The attention's dimension is incorrec
assert _out.shape==torch.Size([_B, 2]), "The output's dimension is incorrect"
del _testm, _out, _gamma,  _B, _M, _T
```

In [ ]:

3 Training and Evaluation [15 points]

In this part we will define the training procedures, train the model, and evaluate the model on the test set.

train_model parameters:

- model: The instance of FreqNet that we are training
- train_dataloader: the DataLoader of the training data
- n_epoch: number of epochs to train
- lr: learning rate
- device: cpu or gpu/cuda

return/output:

- _model): trained model
- _loss*history*: recorded training loss history - should be just a list of float

train_model tasks:
I. Specify the optimizer (*optimizer*) to be optim.Adam
II. Specify the loss function (*loss_func*) to be CrossEntropyLoss
III. Within the loop, do the normal training procedures:

- A. pass the input through the model
- B. pass the output through loss_func to compute the loss
- C. zero out currently accumulated gradient, use loss.basckward to backprop the gradients, then call optimizer.step

eval_model tasks:

returns:

> **pred_all:** prediction of model on the dataloder.
>
> > *Should be an 2D numpy float array where the second dimension has length 2.*
>
> **Y_test:** truth labels. Should be an numpy array of ints

tasks:

1. evaluate the model using on the data in the dataloder.
2. Add all the prediction and truth to the corresponding list
3. Convert pred_all and Y_test to numpy arrays (of shape (n_data_points, 2))

ADAM
params (iterable) – iterable of parameters to optimize or dicts defining parameter groups

```
torch.optim.Adam(
                params,
                lr=0.001,
                betas=(0.9, 0.999),
                eps=1e-08,
                weight_decay=0,
                amsgrad=False, *,
                foreach=None,
                maximize=False,
                capturable=False,
                differentiable=False,
                fused=None
                )
```

CrossEntropyLoss

```
torch.nn.CrossEntropyLoss(
                          weight=None,
                          ignore_index=- 100,
                          reduction='mean',
                          label_smoothing=0.0
                          )
```

loss between input logits and target
input: unnormalized logits for each class.
input has to be a Tensor of size (C), for unbached input (minibatch, C) or (minibatch, c, d1, d2, ... dk) for d = dim.
target:
Class indices in the range (0, C), C = number of classes (*preferred*)
or
Probabilities for each class

In [18]:

```python
def train_model(model, train_dataloader, n_epoch=5, lr=0.003, device=None):
    import torch.optim as optim

    device = device or torch.device('cpu')
    model.train()

    loss_history = []

    # CODE

    # I. Specify the optimizer (*optimizer*) to be optim.Adam

    optimizer = optim.Adam(model.parameters(), lr=lr)
    # II. Specify the loss function (loss_func) to be CrossEntropyLoss
    loss_func = nn.CrossEntropyLoss()

    # END

    for epoch in range(n_epoch):
        curr_epoch_loss = []
        for (X, K_beat, K_rhythm, K_freq), Y in train_dataloader:
#             print("########## X.shape: ############", X.shape)
            # CODE

            # III. Within the loop, do the normal training procedures:
            pred = model(X, K_beat, K_rhythm, K_freq)[0] # Y = model(X)   ## ???  # A.
            loss = loss_func(pred, Y.long())# B. pass the output through loss_func to

            # C.
            optimizer.zero_grad() # i. zero out currently accumulated gradient,
            loss.backward() # ii. use loss.basckward to backprop the gradient
            optimizer.step() # iii. call optimizer.step

            # END

            curr_epoch_loss.append(loss.cpu().data.numpy())
        print(f"epoch{epoch}: curr_epoch_loss={np.mean(curr_epoch_loss)}")
        loss_history += curr_epoch_loss
    return model, loss_history

def eval_model(model, dataloader, device=None):

    device = device or torch.device('cpu')
    model.eval()
    pred_all = []
    Y_test = []
    for (X, K_beat, K_rhythm, K_freq), Y in dataloader:

        # CODE

        # 1. evaluate the model using on the data in the dataloder.
        # Should be an 2D numpy float array where the second dimension has length 2.
        # 2. Add all the prediction and truth to the corresponding list
        # 3. Convert pred_all and Y_test to numpy arrays (of shape (n_data_points, 2)
        pred_all.append(model(X, K_beat, K_rhythm, K_freq)[0].detach().numpy().astype
        Y_test.append(Y.numpy())

        # END

    pred_all = np.concatenate(pred_all, axis=0)
    Y_test = np.concatenate(Y_test, axis=0)

    return pred_all, Y_test
```

In [ ]:

In [19]:

```python
device = torch.device('cpu')
n_epoch = 4
lr = 0.003
n_channel = 4
n_dim=3000
T=50

model = FreqNet(n_channel, n_dim, T)
model = model.to(device)

model, loss_history = train_model(model, train_loader, n_epoch=n_epoch, lr=lr, device=
pred, truth = eval_model(model, test_loader, device=device)
#pd.to_pickle((pred, truth), "./deliverable.pkl")
```

In [20]:

```python
pred, truth = eval_model(model, test_loader, device=device)
```

In [24]:

```python
pred.shape, truth.shape
```

In [21]:

```python
%debug
```

`f1_score(y_true, y_pred)`

In [26]:

In [27]:

```python
def evaluate_predictions(truth, pred):
    """
    TODO: Evaluate the performance of the predictoin via AUROC, and F1 score
    each prediction in pred is a vector representing [p_0, p_1].
    When defining the scores we are interesed in detecting class 1 only, ie 0, 1
    (Hint: use roc_auc_score and f1_score from sklearn.metrics, be sure to read their
    return: auroc, f1
    """
    from sklearn.metrics import roc_auc_score, f1_score

    auroc = roc_auc_score(truth, pred[:, 1])
    pred = np.argmax(pred, axis=1)
    f1=f1_score(truth, pred)

    return auroc, f1
```

In [28]:

```python
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''
pred, truth = eval_model(model, test_loader, device=device)
auroc, f1 = evaluate_predictions(truth, pred)
print(f"AUROC={auroc} and F1={f1}")

assert auroc > 0.8 and f1 > 0.7, "Performance is too low {}. Something's probably off
```

In [ ]:

In [ ]: