HW4 RETAIN

Overview

Previously, you tried heart failure predictioin with classical machine learning models, neural network (NN), and recurrent neural network (RNN).

In this question, you will try a different approach. You will implement RETAIN, a RNN model with attention mechanism, proposed by Choi et al. in the paper RETAIN: An Interpretable Predictive Model for Healthcare using Reverse Time Attention Mechanism.

---

10 EXERCISES

In [1]:

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

In [2]:

```python
# 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_RETAIN-lib/data/"
```

About Raw Data

We will perform heart failure prediction using the diagnosis codes. We will use the same dataset from HW3 RNN, which is synthesized from MIMIC-III.

The data has been preprocessed for you. Let us load them and take a look.

In [3]:

```python
pids = pickle.load(open(os.path.join(DATA_PATH,'train/pids.pkl'), 'rb'))
vids = pickle.load(open(os.path.join(DATA_PATH,'train/vids.pkl'), 'rb'))
hfs = pickle.load(open(os.path.join(DATA_PATH,'train/hfs.pkl'), 'rb'))
seqs = pickle.load(open(os.path.join(DATA_PATH,'train/seqs.pkl'), 'rb'))
types = pickle.load(open(os.path.join(DATA_PATH,'train/types.pkl'), 'rb'))
rtypes = pickle.load(open(os.path.join(DATA_PATH,'train/rtypes.pkl'), 'rb'))

assert len(pids) == len(vids) == len(hfs) == len(seqs) == 1000
assert len(types) == 619
```

where

- `pids` : contains the patient ids
- `vids` : contains a list of visit ids for each patient
- `hfs` : contains the heart failure label (0: normal, 1: heart failure) for each patient
- `seqs` : contains a list of visit (in ICD9 codes) for each patient
- `types` : contains the map from ICD9 codes to ICD-9 labels
- `rtypes` : contains the map from ICD9 labels to ICD9 codes

Let us take a patient as an example.

In [4]:

```python
# take the 3rd patient as an example

print("Patient ID:", pids[3])
print("Heart Failure:", hfs[3])
print("# of visits:", len(vids[3]))
for visit in range(len(vids[3])):
    print(f"\t{visit}-th visit id:", vids[3][visit])
    print(f"\t{visit}-th visit diagnosis labels:", seqs[3][visit])
    print(f"\t{visit}-th visit diagnosis codes:", [rtypes[label] for label in seqs[3]
```

Note that `seqs` is a list of list of list. That is, `seqs[i][j][k]` gives you the k-th diagnosis codes for the j-th visit for the i-th patient.

And you can look up the meaning of the ICD9 code online. For example, `DIAG_276` represetns *disorders of fluid electrolyte and acid-base balance*.

Further, let see number of heart failure patients.

In [5]:

```python
print("number of heart failure patients:", sum(hfs))
print("ratio of heart failure patients: %.2f" % (sum(hfs) / len(hfs)))
```

1 Build the dataset [15 points]

1.1 CustomDataset [5 points]

This is the same as HW3 RNN.

First, let us implement a custom dataset using PyTorch class `Dataset`, which will characterize the key features of the dataset we want to generate.

We will use the sequences of diagnosis codes `seqs` as input and heart failure `hfs` as output.

In [6]:

```python
from torch.utils.data import Dataset


class CustomDataset(Dataset):

    def __init__(self, seqs, hfs):
        self.x = seqs
        self.y = hfs

    def __len__(self):

        """
        TODO: Return the number of samples (i.e. patients).
        """

        # your code here
        return len(self.x)

    def __getitem__(self, index):

        """
        TODO: Generates one sample of data.

        Note that you DO NOT need to covert them to tensor as we will do this later.
        """

        # your code here
        return self.x[index], self.y[index]


dataset = CustomDataset(seqs, hfs)
```

In [7]:

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

dataset = CustomDataset(seqs, hfs)

assert len(dataset) == 1000
```

1.2 Collate Function [5 points]

This is the same as HW3 RNN.

As you note that, we do not convert the data to tensor in the built `CustomDataset`. Instead, we will do this using a collate function `collate_fn()`.

This collate function `collate_fn()` will be called by `DataLoader` after fetching a list of samples using the indices from `CustomDataset` to collate the list of samples into batches.

For example, assume the `DataLoader` gets a list of two samples.

```
[ [ [0, 1, 2], [8, 0] ],
  [ [12, 13, 6, 7], [12], [23, 11] ] ]
```

where the first sample has two visits `[0, 1, 2]` and `[8, 0]` and the second sample has three visits `[12, 13, 6, 7]`, `[12]`, and `[23, 11]`.

The collate function `collate_fn()` is supposed to pad them into the same shape (3, 4), where 3 is the maximum number of visits and 4 is the maximum number of diagnosis codes.

```
[ [ [0, 1, 2, *0*], [8, 0, *0*, *0*], [*0*, *0*, *0*, *0*]  ],
  [ [12, 13, 6, 7], [12, *0*, *0*, *0*], [23, 11, *0*, *0*] ] ]
```

Further, the padding information will be stored in a mask with the same shape, where 1 indicates that the diagnosis code at this position is from the original input, and 0 indicates that the diagnosis code at this position is the padded value.

```
[ [ [1, 1, 1, 0], [1, 1, 0, 0], [0, 0, 0, 0] ],
  [ [1, 1, 1, 1], [1, 0, 0, 0], [1, 1, 0, 0] ] ]
```

Lastly, we will have another diagnosis sequence in reversed time. This will be used in our RNN model for masking. Note that we only flip the true visits.

```
[ [ [8, 0, *0*, *0*], [0, 1, 2, *0*], [*0*, *0*, *0*, *0*]  ],
  [ [23, 11, *0*, *0*], [12, *0*, *0*, *0*], [12, 13, 6, 7] ] ]
```

And a reversed mask as well.

```
[ [ [1, 1, 0, 0], [1, 1, 1, 0], [0, 0, 0, 0] ],
  [ [1, 1, 0, 0], [1, 0, 0, 0], [1, 1, 1, 1], ] ]
```

We need to pad the sequences into the same length so that we can do batch training on GPU. And we also need this mask so that when training, we can ignored the padded value as they actually do not contain any information.

In [8]:

```python
def collate_fn(data):
    """
    TODO: Collate the the list of samples into batches. For each patient, you need to
        sequences to the sample shape (max # visits, max # diagnosis codes). The padd
        is stored in `mask`.

    Arguments:
        data: a list of samples fetched from `CustomDataset`

    Outputs:
        x: a tensor of shape (# patiens, max # visits, max # diagnosis codes) of type
        masks: a tensor of shape (# patiens, max # visits, max # diagnosis codes) of
        rev_x: same as x but in reversed time. This will be used in our RNN model for
        rev_masks: same as mask but in reversed time. This will be used in our RNN mod
        y: a tensor of shape (# patiens) of type torch.float

    Note that you can obtains the list of diagnosis codes and the list of hf labels
        using: `sequences, labels = zip(*data)`
    """

    sequences, labels = zip(*data)

    y = torch.tensor(labels, dtype=torch.float)

    num_patients = len(sequences)
    num_visits = [len(patient) for patient in sequences]
    num_codes = [len(visit) for patient in sequences for visit in patient]

    max_num_visits = max(num_visits)
    max_num_codes = max(num_codes)

    x = torch.zeros((num_patients, max_num_visits, max_num_codes), dtype=torch.long)
    rev_x = torch.zeros((num_patients, max_num_visits, max_num_codes), dtype=torch.lon
    masks = torch.zeros((num_patients, max_num_visits, max_num_codes), dtype=torch.boo
    rev_masks = torch.zeros((num_patients, max_num_visits, max_num_codes), dtype=torch
    for i_patient, patient in enumerate(sequences):
        for j_visit, visit in enumerate(patient):
            """
            TODO: update `x`, `rev_x`, `masks`, and `rev_masks`
            """
            # your code here
            x[i_patient, j_visit, :len(visit)] = torch.tensor(visit, dtype=torch.long)
            masks[i_patient, j_visit, :len(visit)] = 1
            rev_x[i_patient, len(patient) - 1 - j_visit, :len(visit)] = torch.tensor(v
            rev_masks[i_patient, len(patient) - 1 - j_visit, :len(visit)] = 1

    return x, masks, rev_x, rev_masks, y
```

In [9]:

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

from torch.utils.data import DataLoader

loader = DataLoader(dataset, batch_size=10, collate_fn=collate_fn)
loader_iter = iter(loader)
x, masks, rev_x, rev_masks, y = next(loader_iter)

assert x.dtype == rev_x.dtype == torch.long
assert y.dtype == torch.float
assert masks.dtype == rev_masks.dtype == torch.bool

assert x.shape == rev_x.shape == masks.shape == rev_masks.shape == (10, 3, 24)
assert y.shape == (10,)
```

Now we have `CustomDataset` and `collate_fn()`. Let us split the dataset into training and validation sets.

In [10]:

```python
from torch.utils.data.dataset import random_split

split = int(len(dataset)*0.8)

lengths = [split, len(dataset) - split]
train_dataset, val_dataset = random_split(dataset, lengths)

print("Length of train dataset:", len(train_dataset))
print("Length of val dataset:", len(val_dataset))
```

1.3 DataLoader [5 points]

This is the same as HW3 RNN.

Now, we can load the dataset into the data loader.

In [11]:

```python
from torch.utils.data import DataLoader

def load_data(train_dataset, val_dataset, collate_fn):

    '''
    TODO: Implement this function to return the data loader for  train and validation
    Set batchsize to 32. Set `shuffle=True` only for train dataloader.

    Arguments:
        train dataset: train dataset of type `CustomDataset`
        val dataset: validation dataset of type `CustomDataset`
        collate_fn: collate function

    Outputs:
        train_loader, val_loader: train and validation dataloaders

    Note that you need to pass the collate function to the data loader `collate_fn()`
    '''

    batch_size = 32
    # your code here
    train_loader = DataLoader(train_dataset, collate_fn=collate_fn, batch_size=batch_s
    val_loader = DataLoader(val_dataset, collate_fn=collate_fn, batch_size=batch_size)
    return train_loader, val_loader


train_loader, val_loader = load_data(train_dataset, val_dataset, collate_fn)
```

In [12]:

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

train_loader, val_loader = load_data(train_dataset, val_dataset, collate_fn)

assert len(train_loader) == 25, "Length of train_loader should be 25, instead we got 
```

2 RETAIN [70 points]

RETAIN is essentially a RNN model with attention mechanism.

The idea of attention is quite simple: it boils down to weighted averaging. Let us consider machine translation in class as an example. When generating a translation of a source text, we first pass the source text through an encoder (an LSTM or an equivalent model) to obtain a sequence of encoder hidden states $\boldsymbol{h}_1, \dots, \boldsymbol{h}_T$. Then, at each step of generating a translation (decoding), we selectively attend to these encoder hidden states, that is, we construct a context vector $\boldsymbol{c}_i$ that is a weighted average of encoder hidden states.

$$\boldsymbol{c}_i = \sum_j a_{ij} \boldsymbol{h}_j$$

We choose the weights a_{ij} based both on encoder hidden states $\boldsymbol{h}_1, \dots, \boldsymbol{h}_T$ and decoder hidden states $\boldsymbol{s}_1, \dots, \boldsymbol{s}_T$ and normalize them so that they encode a categorical probability distribution $p(\boldsymbol{h}_j | \boldsymbol{s}_i)$.

$$\boldsymbol{a}_i = \text{Softmax}\left(a(\boldsymbol{s}_i, \boldsymbol{h}_j)\right)$$

RETAIN has two different attention mechanisms.

- One is to help figure out what are the important visits. This attention α_i, which is scalar for the i-th visit, tells you the importance of the i-th visit.
- Then we have another similar attention mechanism. But in this case, this attention ways β_i is a vector. That gives us a more detailed view of underlying cause of the input. That is, which are the important features within a visit.

Unfolded view of RETAIN's architecture: Given input sequence $\mathbf{x}_1, \dots, \mathbf{x}_i$, we predict the label $\mathbf{y}_i$.

- Step 1: Embedding,
- Step 2: generating α values using RNN-α,
- Step 3: generating β values using RNN-β,
- Step 4: Generating the context vector using attention and representation vectors,
- Step 5: Making prediction.

Note that in Steps 2 and 3 we use RNN in the reversed time.

Let us first implement RETAIN step-by-step.

2.1 Step 2: AlphaAttention [20 points]

Implement the alpha attention in the second equation of step 2.

In [13]:

```python
class AlphaAttention(torch.nn.Module):

    def __init__(self, embedding_dim):
        super().__init__()
        """
        Define the linear layer `self.a_att` for alpha-attention using `nn.Linear()`;

        Arguments:
            embedding_dim: the embedding dimension
        """

        self.a_att = nn.Linear(embedding_dim, 1)

    def forward(self, g, rev_masks):
        """
        TODO: Implement the alpha attention.

        Arguments:
            g: the output tensor from RNN-alpha of shape (batch_size, # visits, embedd
            rev_masks: the padding masks in reversed time of shape (batch_size, # vis

        Outputs:
            alpha: the corresponding attention weights of shape (batch_size, # visits,

        HINT:
            1. Calculate the attention score using `self.a_att`
            2. Mask out the padded visits in the attention score with -1e9.
            3. Perform softmax on the attention score to get the attention value.
        """

        # your code here
        x = self.a_att(g)
        vmask, _ = torch.max(rev_masks, dim=-1)
        vmask = vmask.unsqueeze(dim=-1)
        alpha = (~vmask) * (-1e9) + vmask * x
#         print(F.softmax(alpha, dim=1))
        return F.softmax(alpha, dim=1)
```

In [14]:

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

2.2 Step 3: BetaAttention [20 points]

Implement the beta attention in the second equation of step 3.

In [15]:

```python
class BetaAttention(torch.nn.Module):

    def __init__(self, embedding_dim):
        super().__init__()
        """
        Define the linear layer `self.b_att` for beta-attention using `nn.Linear()`;

        Arguments:
        """

        self.b_att = nn.Linear(embedding_dim, embedding_dim)


    def forward(self, h):
        """
        TODO: Implement the beta attention.

        Arguments:
            h: the output tensor from RNN-beta of shape (batch_size, # visits, embeddi

        Outputs:
            beta: the corresponding attention weights of shape (batch_size, # visitsse

        HINT: consider `torch.tanh`
        """

        # your code here
        return torch.tanh(self.b_att(h))
```

In [16]:

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

2.3 Attention Sum [30 points]

Implement the sum of attention in step 4.

In [17]:

```python
def attention_sum(alpha, beta, rev_v, rev_masks):
    """
    TODO: mask select the hidden states for true visits (not padding visits) and then
        sum the them up.

    Arguments:
        alpha: the alpha attention weights of shape (batch_size, # visits, 1)
        beta: the beta attention weights of shape (batch_size, # visits, embedding_dim
        rev_v: the visit embeddings in reversed time of shape (batch_size, # visits, e
        rev_masks: the padding masks in reversed time of shape (batch_size, # visits, 

    Outputs:
        c: the context vector of shape (batch_size, embedding_dim)

    NOTE: Do NOT use for loop.
    """

    # your code here
    vmask, _ = torch.max(rev_masks, dim=-1)
    vmask = vmask.unsqueeze(dim=-1)
    a = (vmask * alpha)
    b = (rev_v * beta)
    c = (a * b).sum(dim=1)
    return c
```

In [18]:

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

2.4 Build RETAIN

Now, we can build the RETAIN model.

In [19]:

```python
def sum_embeddings_with_mask(x, masks):
    """
    Mask select the embeddings for true visits (not padding visits) and then sum the e

    Arguments:
        x: the embeddings of diagnosis sequence of shape (batch_size, # visits, # diag
        masks: the padding masks of shape (batch_size, # visits, # diagnosis codes)

    Outputs:
        sum_embeddings: the sum of embeddings of shape (batch_size, # visits, embeddin
    """

    x = x * masks.unsqueeze(-1)
    x = torch.sum(x, dim = -2)
    return x
```

In [20]:

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

    def __init__(self, num_codes, embedding_dim=128):
        super().__init__()
        # Define the embedding layer using `nn.Embedding`. Set `embDimSize` to 128.
        self.embedding = nn.Embedding(num_codes, embedding_dim)
        # Define the RNN-alpha using `nn.GRU()`; Set `hidden_size` to 128. Set `batch_
        self.rnn_a = nn.GRU(embedding_dim, embedding_dim, batch_first=True)
        # Define the RNN-beta using `nn.GRU()`; Set `hidden_size` to 128. Set `batch_
        self.rnn_b = nn.GRU(embedding_dim, embedding_dim, batch_first=True)
        # Define the alpha-attention using `AlphaAttention()`;
        self.att_a = AlphaAttention(embedding_dim)
        # Define the beta-attention using `BetaAttention()`;
        self.att_b = BetaAttention(embedding_dim)
        # Define the linear layers using `nn.Linear()`;
        self.fc = nn.Linear(embedding_dim, 1)
        # Define the final activation layer using `nn.Sigmoid().
        self.sigmoid = nn.Sigmoid()

    def forward(self, x, masks, rev_x, rev_masks):
        """
        Arguments:
            rev_x: the diagnosis sequence in reversed time of shape (# visits, batch_s
            rev_masks: the padding masks in reversed time of shape (# visits, batch_s

        Outputs:
            probs: probabilities of shape (batch_size)
        """
        # 1. Pass the reversed sequence through the embedding layer;
        rev_x = self.embedding(rev_x)
        # 2. Sum the reversed embeddings for each diagnosis code up for a visit of a p
        rev_x = sum_embeddings_with_mask(rev_x, rev_masks)
        # 3. Pass the reversed embegginds through the RNN-alpha and RNN-beta layer sep
        g, _ = self.rnn_a(rev_x)
        h, _ = self.rnn_b(rev_x)
        # 4. Obtain the alpha and beta attentions using `AlphaAttention()` and `BetaA
        alpha = self.att_a(g, rev_masks)
        beta = self.att_b(h)
        # 5. Sum the attention up using `attention_sum()`;
        c = attention_sum(alpha, beta, rev_x, rev_masks)
        # 6. Pass the context vector through the linear and activation layers.
        logits = self.fc(c)
        probs = self.sigmoid(logits)
        return probs.squeeze(dim=-1)


# load the model here
retain = RETAIN(num_codes = len(types))
retain
```

In [21]:

```python
assert retain.att_a.a_att.in_features == 128, "alpha attention input features is wrong
assert retain.att_a.a_att.out_features == 1, "alpha attention output features is wrong
assert retain.att_b.b_att.in_features == 128, "beta attention input features is wrong"
assert retain.att_b.b_att.out_features == 128, "beta attention output features is wro
```

3 Training and Inferencing [10 points]

Then, let us implement the `eval()` function first.

In [22]:

```python
from sklearn.metrics import precision_recall_fscore_support, roc_auc_score


def eval(model, val_loader):

    """
    Evaluate the model.

    Arguments:
        model: the RNN model
        val_loader: validation dataloader

    Outputs:
        precision: overall precision score
        recall: overall recall score
        f1: overall f1 score
        roc_auc: overall roc_auc score

    REFERENCE: checkout https://scikit-learn.org/stable/modules/classes.html#module-sk
    """

    model.eval()
    y_pred = torch.LongTensor()
    y_score = torch.Tensor()
    y_true = torch.LongTensor()
    model.eval()
    for x, masks, rev_x, rev_masks, y in val_loader:
        y_logit = model(x, masks, rev_x, rev_masks)
        """
        TODO: obtain the predicted class (0, 1) by comparing y_logit against 0.5,
              assign the predicted class to y_hat.
        """
        y_hat = None
        # your code here
        y_hat = y_logit >= 0.5
        y_score = torch.cat((y_score,  y_logit.detach().to('cpu')), dim=0)
        y_pred = torch.cat((y_pred,  y_hat.detach().to('cpu')), dim=0)
        y_true = torch.cat((y_true, y.detach().to('cpu')), dim=0)

    p, r, f, _ = precision_recall_fscore_support(y_true, y_pred, average='binary')
    roc_auc = roc_auc_score(y_true, y_score)
    return p, r, f, roc_auc
```

Now let us implement the `train()` function. Note that `train()` should call `eval()` at the end of each training epoch to see the results on the validation dataset.

In [23]:

```python
def train(model, train_loader, val_loader, n_epochs):
    """
    Train the model.

    Arguments:
        model: the RNN model
        train_loader: training dataloder
        val_loader: validation dataloader
        n_epochs: total number of epochs
    """

    for epoch in range(n_epochs):
        model.train()
        train_loss = 0
        for x, masks, rev_x, rev_masks, y in train_loader:
            optimizer.zero_grad()
            y_hat = model(x, masks, rev_x, rev_masks)
            """
            TODO: calculate the loss using `criterion`, save the output to loss.
            """
            loss = None
            # your code here
            loss = criterion(y_hat, y)
            loss.backward()
            optimizer.step()
            train_loss += loss.item()
        train_loss = train_loss / len(train_loader)
        print('Epoch: {} \t Training Loss: {:.6f}'.format(epoch+1, train_loss))
        p, r, f, roc_auc = eval(model, val_loader)
        print('Epoch: {} \t Validation p: {:.2f}, r:{:.2f}, f: {:.2f}, roc_auc: {:.2f
    return round(roc_auc, 2)
```

In [24]:

```python
# load the model
retain = RETAIN(num_codes = len(types))

# load the loss function
criterion = nn.BCELoss()
# load the optimizer
optimizer = torch.optim.Adam(retain.parameters(), lr=1e-3)

n_epochs = 5
train(retain, train_loader, val_loader, n_epochs)
```

In [25]:

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

4 Sensitivity analysis [5 points]

We will train the same model but with different hyperparameters. We will be using 1 and 0.001 for learning rate, and 4, 128 for embedding dimensions. It shows how model performance varies with different values of learning rate and embedding dimensions.

In [26]:

```python
lr_hyperparameter = [1, 1e-3]
embedding_dim_hyperparameter = [4, 128]
n_epochs = 5
results = {}

for lr in lr_hyperparameter:
    for embedding_dim in embedding_dim_hyperparameter:
        print ('='*50)
        print ({'learning rate': lr, "embedding_dim": embedding_dim})
        print ('-'*50)
        """
        TODO:
            1. Load the model by specifying `embedding_dim` as input to RETAIN. It wil
            2. Load the loss function `nn.BCELoss`
            3. Load the optimizer `torch.optim.Adam` with learning rate using `lr` va
        """
        # your code here
        retain = RETAIN(num_codes = len(types), embedding_dim=embedding_dim)
        criterion = nn.BCELoss()
        optimizer = torch.optim.Adam(retain.parameters(), lr=lr)

        train(retain, train_loader, val_loader, n_epochs)
        roc_auc = train(retain, train_loader, val_loader, n_epochs)
        results['lr:{},emb:{}'.format(str(lr), str(embedding_dim))] =  roc_auc
```

In [27]:

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

assert results['lr:1,emb:128'] < 0.7, "auc roc should be below 0.7! Since higher lear
```

In [28]:

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

In [ ]: