HW4_CAML

March 27, 2022

1 HW4 CAML

1.1 Overview

In this question, we will implement Convolutional Attention for Multi-Label classification (CAML) proposed by Mullenbach et al. in the paper "Explainable Prediction of Medical Codes from Clinical Text".

Clinical notes are text documents that are created by clinicians for each patient encounter. They are typically accompanied by medical codes, which describe the diagnosis and treatment. Annotating these codes is labor intensive and error prone; furthermore, the connection between the codes and the text is not annotated, obscuring the reasons and details behind specific diagnoses and treatments. Thus, let us implement CAML, an attentional convolutional network to predict medical codes from clinical text.

Image courtsey: link

---

[1]:
```
import os
import csv
import pickle
import random
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import pandas as pd
```

[2]:
```
# 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_CAML-lib/data/"
```

1.2 Dataset

For this question, we will be using the Indiana University Chest X-Ray dataset. The goal is to predict diseases using chest x-ray reports.

Navigate to the data folder `DATA_PATH`, there are several files:

- `train_df.csv`, `test_df.csv`: these two files contains the data used for training and testing.
  - `Report ID` refers to a unique chest x-ray report.
  - `Text` refers to the clinical report text.
  - `Label` refers to the diseases.
- `vocab.csv`: this file contains the vocabularies used in the clinical text.

[3]:
```
!ls {DATA_PATH}
```

```
test_df.csv  train_df.csv  vocab.csv
```

For example, the first chest x-ray report in `train_df.csv` has: - `Report ID`: 1 - `Text`: the cardiac silhouette and mediastinum size are within normal limits . there is no pulmonary edema . there is no focal consolidation . there are no xxxx of a pleural effusion . there is no evidence of pneumothorax . normal chest xxxxx . - `Label`: [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

where label is a multi-hot vector representing the following diseases:

```
normal
cardiomegaly
scoliosis / degenerative
fractures bone
pleural effusion
thickening
pneumothorax
hernia hiatal
calcinosis
emphysema / pulmonary emphysema
pneumonia / infiltrate / consolidation
pulmonary edema
pulmonary atelectasis
cicatrix
opacity
nodule / mass
airspace disease
hypoinflation / hyperdistention
catheters indwelling / surgical instruments / tube inserted / medical device
other
```

So this report 1 is labeled as "normal".

1.3 1 Prepare the Dataset [30 points]

1.3.1 1.1 Helper Functions [10 points]

To begin, weith, let us first implement some helper functions we will use later.

[4]:
```python
def to_index(sequence, token2idx):
    """
    TODO: convert the sequnce of tokens to indices.
    If the word in unknown, then map it to '<unk>'.

    INPUT:
        sequence (type: list of str): a sequence of tokens
        token2idx (type: dict): a dictionary mapping token to the corresponding␣
↪index

    OUTPUT:
        indices (type: list of int): a sequence of indicies

    EXAMPLE:
        >>> sequence = ['hello', 'world', 'unknown_word']
        >>> token2idx = {'hello': 0, 'world': 1, '<unk>': 2}
        >>> to_index(sequence, token2idx)
        [0, 1, 2]
    """
    # your code here
    # raise NotImplementedError
    unk = '<unk>'
    return [(token2idx[k] if k in token2idx else token2idx[unk]) for k in␣
↪sequence]
```

[5]:
```python
# sequence = ['hello', 'world', 'unknown_word']
# token2idx = {'hello': 0, 'world': 1, '<unk>': 2}
# unk = '<unk>'
# k = "unknown_word"
# [(token2idx[k] if k in token2idx else token2idx[unk]) for k in sequence]
```

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

sequence = ['hello', 'world', 'unknown_word']
token2idx = {'hello': 0, 'world': 1, '<unk>': 2}
assert to_index(sequence, token2idx) == [0, 1, 2], "to_index() is wrong!"
```

1.3.2 1.2 CustomDataset [10 points]

Now, 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 clinical text as input and medical codes as output.

[7]:

```
from torch.utils.data import Dataset

NUM_WORDS = 1253
NUM_CLASSES = 20


class CustomDataset(Dataset):

    def __init__(self, filename):
        # read in the data files
        self.data = pd.read_csv(filename)
        # load word lookup
        self.idx2word, self.word2idx = self.load_lookup(f'{DATA_PATH}/vocab.
 ↪csv', padding=True)
        assert len(self.idx2word) == len(self.word2idx) == NUM_WORDS

    def load_lookup(self, filename, padding=False):
        """ load lookup for word """
        idx2token = {}
        with open(filename, 'r') as f:
            for i, line in enumerate(f):
                line = line.strip()
                idx2token[i] = line
        token2idx = {w:i for i,w in idx2token.items()}
        return idx2token, token2idx

    def __len__(self):

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

        # your code here
        # raise NotImplementedError
        # return len(self.idx2word)
        return len(self.data['Report ID'])

    def __getitem__(self, index):

        """
        TODO: Generate one sample of data.

        Hint: convert text to indices using to_index();
        """
        data = self.data.iloc[index]
        text = data['Text'].split(' ')
        label = data['Label']
```

```
        # convert label string to list
        label = [int(l) for l in label.strip('[]').split(', ')]
        assert len(label) == NUM_CLASSES
        # your code here
        # raise NotImplementedError
        text = to_index(text, self.word2idx)

        # return text as long tensor, labels as float tensor;
        return torch.tensor(text, dtype=torch.long), torch.tensor(label,␣
↪dtype=torch.float)
```

[8]:
```
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''

dataset = CustomDataset(f'{DATA_PATH}/train_df.csv')
assert len(dataset) == 3141, "__len__() is wrong!"

text, labels = dataset[1]

assert type(text) is torch.Tensor, "__getitem__(): text is not tensor!"
assert type(labels) is torch.Tensor, "__getitem__(): labels is not tensor!"
assert text.dtype is torch.int64, "__getitem__(): text is not of type long!"
assert labels.dtype is torch.float32, "__getitem__(): labels is not of type␣
↪float!"
```

[ ]:
```

```

1.3.3 1.3 Collate Function [10 points]

The 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.

```
[ [3,  1,  2, 8, 5],
  [12, 13, 6, 7, 12, 23, 11] ]
```

where the first sample has text `[3, 1, 2, 8, 5]` the second sample has text `[12, 13, 6, 7, 12, 23, 11]`.

The collate function `collate_fn()` is supposed to pad them into the same shape (7), where 7 is the maximum number of tokens.

```
[ [3,  1,  2, 8, 5, *0*, *0*],
  [12, 13, 6, 7, 12, 23,  11 ]
```

where `*0*` indicates the padding token.

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.

[9]:
```python
from torch.nn.utils.rnn import pad_sequence

def collate_fn(data):
    """
    TODO: implement the collate function.

    STEP: 1. pad the text using pad_sequence(). Set `batch_first=True`.
          2. stack the labels using torch.stack().

    OUTPUT:
        text: the padded text, shape: (batch size, max length)
        labels: the stacked labels, shape: (batch size, num classes)
    """
    text, labels = zip(*data)

    # your code here
    # raise NotImplementedError
    text = pad_sequence(text, batch_first=True, padding_value=0)
    labels = pad_sequence(labels, batch_first=True, padding_value=0)

    return text, labels
```

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

from torch.utils.data import DataLoader

dataset = CustomDataset(f'{DATA_PATH}/train_df.csv')
loader = DataLoader(dataset, batch_size=10, collate_fn=collate_fn)
loader_iter = iter(loader)
text, labels = next(loader_iter)

assert text.shape == (10, 104), "collate_fn(): text has incorrect shape!"
assert labels.shape == (10, 20), "collate_fn(): labels has incorrect shape!"
```

[ ]:
```python

```

All done, now let us load the dataset and data loader.

[11]:
```python
train_set = CustomDataset(f'{DATA_PATH}/train_df.csv')
test_set = CustomDataset(f'{DATA_PATH}/test_df.csv')
train_loader = DataLoader(train_set, batch_size=32, collate_fn=collate_fn,␣
 ↪shuffle=True)
test_loader = DataLoader(test_set, batch_size=32, collate_fn=collate_fn)
```

1.4 2 Model [50 points]

Next, we will implement the CAML model.

CAML is a convolutional neural network (CNN)-based model. It employs a per-label attention mechanism, which allows the model to learn distinct document representations for each label.

[12]:
```
from math import floor
from torch.nn.init import xavier_uniform_


class CAML(nn.Module):

    def __init__(self, kernel_size=10, num_filter_maps=16, embed_size=100,␣
↪dropout=0.5):
        super(CAML, self).__init__()

        # embedding layer
        self.embed = nn.Embedding(NUM_WORDS, embed_size, padding_idx=0)
        self.embed_drop = nn.Dropout(p=dropout)

        # initialize conv layer as in section 2.1
        self.conv = nn.Conv1d(embed_size, num_filter_maps,␣
↪kernel_size=kernel_size, padding=int(floor(kernel_size/2)))
        xavier_uniform_(self.conv.weight)

        # context vectors for computing attention as in section 2.2
        self.U = nn.Linear(num_filter_maps, 20)
        xavier_uniform_(self.U.weight)

        # final layer: create a matrix to use for the NUM_CLASSES binary␣
↪classifiers as in section 2.3
        self.final = nn.Linear(num_filter_maps, NUM_CLASSES)
        xavier_uniform_(self.final.weight)

    def forward_embed(self, text):
        """
        TODO: Feed text through the embedding (self.embed) and dropout layer␣
↪(self.embed_drop).

        INPUT:
            text: (batch size, seq_len)

        OURPUT:
            text: (batch size, seq_len, embed_size)
        """
        # your code here
        # raise NotImplementedError
```

```python
        t = self.embed(text)
        k = self.embed_drop(t)
        # print(f"forward_embed# {k.size()}")
        return k

    def forward_conv(self, text):
        """
        TODO: Feed text through the convolution layer (self.conv) and tanh␣
↪activation function (torch.tanh)
        in eq (1) in the paper.

        INTPUT:
            text: (batch size, embed_size, seq_len)

        OUTPUT:
            text: (batch size, num_filter_maps, seq_len)
        """
        # your code here
        # raise NotImplementedError
        c = self.conv(text)
        t_out = torch.tanh(c)
        # print(f"forward_conv# {t_out.size()}")
        return t_out

    def forward_calc_atten(self, text):
        """
        TODO: calculate the attention weights in eq (2) in the paper. Be sure␣
↪to read the documentation for
        F.softmax()

        INPUT:
            text: (batch size, seq_len, num_filter_maps)

        OUTPUT:
            alpha: (batch size, num_class, seq_len), the attention weights

        STEP: 1. multiply `self.U.weight` with `text` using torch.matmul();
              2. apply softmax using `F.softmax()`.
        """
        # (batch size, seq_len, num_filter_maps) -> (batch size,␣
↪num_filter_mapsseq_len)
        # print(f"calc_attn text# {text.size()}")
        text = text.transpose(1,2)
        # print(f"calc_attn text_trasp# {text.size()}")

        w = torch.unsqueeze(self.U.weight, dim=0)
        w = w.repeat(text.shape[0], 1, 1)
```

```python
        # print(f"calc_attn w# {w.size()}")

        # print(f"calc_attn self.U.weight# {self.U.weight.size()}")

        # your code here
        # raise NotImplementedError
        step1 = torch.matmul(w, text)
        # print(f"calc_attn step1# {step1.size()}")

        step2 = F.softmax(step1, dim=2)
        # print(f"calc_attn step2# {step2.size()}")
        return step2

    def forward_aply_atten(self, alpha, text):
        """
        TODO: apply the attention in eq (3) in the paper.

        INPUT:
            text: (batch size, seq_len, num_filter_maps)
            alpha: (batch size, num_class, seq_len), the attention weights

        OUTPUT:
            v: (batch size, num_class, num_filter_maps), vector representations␣
↪for each label

        STEP: multiply `alpha` with `text` using torch.matmul().
        """
        # your code here
        # raise NotImplementedError
        r = torch.matmul(alpha, text)
        # print(f"forward_aply_atten# {r.size()}")
        return r

    def forward_linear(self, v):
        """
        TODO: apply the final linear classification in eq (5) in the paper.

        INPUT:
            v: (batch size, num_class, num_filter_maps), vector representations␣
↪for each label

        OUTPUT:
            y_hat: (batch size, num_class), label probability

        STEP: 1. multiply `self.final.weight` v `text` element-wise using torch.
↪mul();
              2. sum the result over dim 2 (i.e. num_filter_maps);
```

```
                3. add the result with `self.final.bias`;
                4. apply sigmoid with torch.sigmoid().
        """
        # your code here
        # raise NotImplementedError
        # step1 = torch.mul(self.final.weight, v)
        # print("----")
        # print(step1)

        step1 = torch.mul(v, self.final.weight)
        # print("----")
        # print(step1)

        step2 = torch.sum(step1, dim=2)
        step3 = step2 + self.final.bias
        step4 = torch.sigmoid(step3)
        # print(f"forward_linear step4# {step4.size()}")
        return step4

    def forward(self, text):
        """ 1. get embeddings and apply dropout """
        text = self.forward_embed(text)
        # (batch size, seq_len, embed_size) -> (batch size, embed_size,␣
↪seq_len);
        text = text.transpose(1, 2)

        """ 2. apply convolution and nonlinearity (tanh) """
        text = self.forward_conv(text)
        # (batch size, num_filter_maps, seq_len) -> (batch size, seq_len,␣
↪num_filter_maps);
        text = text.transpose(1,2)

        """ 3. calculate attention """
        alpha = self.forward_calc_atten(text)

        """ 3. apply attention """
        v = self.forward_aply_atten(alpha, text)

        """ 4. final layer classification """
        y_hat = self.forward_linear(v)

        return y_hat


model = CAML()
```

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

model = CAML()
model.eval()
```

```
[13]: CAML(
  (embed): Embedding(1253, 100, padding_idx=0)
  (embed_drop): Dropout(p=0.5, inplace=False)
  (conv): Conv1d(100, 16, kernel_size=(10,), stride=(1,), padding=(5,))
  (U): Linear(in_features=16, out_features=20, bias=True)
  (final): Linear(in_features=16, out_features=20, bias=True)
)
```

1.5 3 Training and Inferencing [20 points]

```python
[14]: model = CAML()
```

```python
[15]: import torch.optim as optim

optimizer = optim.Adam(model.parameters(), lr=0.001)
criterion = nn.BCELoss()
```

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

```python
[16]: from sklearn.metrics import precision_recall_fscore_support


def eval(model, test_loader):

    """
    INPUT:
        model: the CAML model
        test_loader: dataloader

    OUTPUT:
        precision: overall micro precision score
        recall: overall micro recall score
        f1: overall micro f1 score

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

    model.eval()
```

```
    y_pred = torch.LongTensor()
    y_true = torch.LongTensor()
    model.eval()
    for sequences, labels in test_loader:
        """
        TODO: 1. preform forward pass
              2. obtain the predicted class (0, 1) by comparing forward pass␣
↪output against 0.5,
                 assign the predicted class to y_hat.
        """
        # your code here
        # raise NotImplementedError
        ### Begin - my code
        # y_hat = model(sequences)
        m = model(sequences)
        y_hat = torch.zeros_like(m)
        # print(f"eval m# {m.size()}")
        y_hat[m > 0.5] = 1
        # y_true = labels
        ### End - my code

        y_pred = torch.cat((y_pred,  y_hat.detach().to('cpu')), dim=0)
        y_true = torch.cat((y_true, labels.detach().to('cpu')), dim=0)

    p, r, f, _ = precision_recall_fscore_support(y_true, y_pred,␣
↪average='micro')
    return p, r, f
```

[17]:

```
def train(model, train_loader, test_loader, n_epochs):
    """
    INPUT:
        model: the CAML model
        train_loader: dataloder
        val_loader: dataloader
        n_epochs: total number of epochs
    """
    for epoch in range(n_epochs):
        model.train()
        train_loss = 0
        for sequences, labels in train_loader:
            optimizer.zero_grad()
            """
            TODO: 1. perform forward pass using `model`, save the output to␣
↪y_hat;
                  2. calculate the loss using `criterion`, save the output to␣
↪loss.
            """
```

```
            y_hat, loss = None, None
            # your code here
            # raise NotImplementedError
            ### Begin - My code
            y_hat = model(sequences)
            loss = criterion(y_hat, labels)
            ### End - My code

            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 = eval(model, test_loader)
        print('Epoch: {} \t Validation p: {:.2f}, r:{:.2f}, f: {:.2f}'.
 ↪format(epoch+1, p, r, f))


# number of epochs to train the model
n_epochs = 40

train(model, train_loader, test_loader, n_epochs)
```

```
Epoch: 1         Training Loss: 0.475205
Epoch: 1         Validation p: 0.00, r:0.00, f: 0.00
Epoch: 2         Training Loss: 0.284034
Epoch: 2         Validation p: 0.00, r:0.00, f: 0.00
Epoch: 3         Training Loss: 0.238292
Epoch: 3         Validation p: 1.00, r:0.00, f: 0.00
Epoch: 4         Training Loss: 0.217297
Epoch: 4         Validation p: 0.88, r:0.14, f: 0.24
Epoch: 5         Training Loss: 0.202708
Epoch: 5         Validation p: 0.83, r:0.22, f: 0.35
Epoch: 6         Training Loss: 0.193146
Epoch: 6         Validation p: 0.84, r:0.23, f: 0.36
Epoch: 7         Training Loss: 0.185287
Epoch: 7         Validation p: 0.85, r:0.24, f: 0.37
Epoch: 8         Training Loss: 0.178618
Epoch: 8         Validation p: 0.87, r:0.25, f: 0.39
Epoch: 9         Training Loss: 0.171000
Epoch: 9         Validation p: 0.89, r:0.32, f: 0.48
Epoch: 10        Training Loss: 0.162060
Epoch: 10        Validation p: 0.90, r:0.36, f: 0.52
Epoch: 11        Training Loss: 0.153955
Epoch: 11        Validation p: 0.91, r:0.44, f: 0.59
Epoch: 12        Training Loss: 0.146145
Epoch: 12        Validation p: 0.92, r:0.47, f: 0.62
```

```
Epoch: 13       Training Loss: 0.138957
Epoch: 13       Validation p: 0.93, r:0.49, f: 0.65
Epoch: 14       Training Loss: 0.133500
Epoch: 14       Validation p: 0.92, r:0.52, f: 0.66
Epoch: 15       Training Loss: 0.127677
Epoch: 15       Validation p: 0.92, r:0.54, f: 0.68
Epoch: 16       Training Loss: 0.123398
Epoch: 16       Validation p: 0.92, r:0.56, f: 0.69
Epoch: 17       Training Loss: 0.119230
Epoch: 17       Validation p: 0.91, r:0.58, f: 0.71
Epoch: 18       Training Loss: 0.115446
Epoch: 18       Validation p: 0.93, r:0.59, f: 0.72
Epoch: 19       Training Loss: 0.112091
Epoch: 19       Validation p: 0.92, r:0.60, f: 0.73
Epoch: 20       Training Loss: 0.109252
Epoch: 20       Validation p: 0.93, r:0.62, f: 0.74
Epoch: 21       Training Loss: 0.106028
Epoch: 21       Validation p: 0.93, r:0.62, f: 0.74
Epoch: 22       Training Loss: 0.103364
Epoch: 22       Validation p: 0.92, r:0.64, f: 0.76
Epoch: 23       Training Loss: 0.100763
Epoch: 23       Validation p: 0.92, r:0.66, f: 0.77
Epoch: 24       Training Loss: 0.098222
Epoch: 24       Validation p: 0.92, r:0.68, f: 0.78
Epoch: 25       Training Loss: 0.096119
Epoch: 25       Validation p: 0.92, r:0.68, f: 0.78
Epoch: 26       Training Loss: 0.094269
Epoch: 26       Validation p: 0.92, r:0.69, f: 0.79
Epoch: 27       Training Loss: 0.092843
Epoch: 27       Validation p: 0.92, r:0.70, f: 0.79
Epoch: 28       Training Loss: 0.090868
Epoch: 28       Validation p: 0.91, r:0.71, f: 0.80
Epoch: 29       Training Loss: 0.088362
Epoch: 29       Validation p: 0.92, r:0.71, f: 0.80
Epoch: 30       Training Loss: 0.086917
Epoch: 30       Validation p: 0.92, r:0.72, f: 0.81
Epoch: 31       Training Loss: 0.084275
Epoch: 31       Validation p: 0.91, r:0.73, f: 0.81
Epoch: 32       Training Loss: 0.081971
Epoch: 32       Validation p: 0.91, r:0.74, f: 0.81
Epoch: 33       Training Loss: 0.083603
Epoch: 33       Validation p: 0.91, r:0.74, f: 0.82
Epoch: 34       Training Loss: 0.080553
Epoch: 34       Validation p: 0.91, r:0.74, f: 0.81
Epoch: 35       Training Loss: 0.080310
Epoch: 35       Validation p: 0.92, r:0.74, f: 0.82
Epoch: 36       Training Loss: 0.077791
Epoch: 36       Validation p: 0.91, r:0.75, f: 0.82
```

```
Epoch: 37       Training Loss: 0.076524
Epoch: 37       Validation p: 0.90, r:0.75, f: 0.82
Epoch: 38       Training Loss: 0.077179
Epoch: 38       Validation p: 0.90, r:0.75, f: 0.82
Epoch: 39       Training Loss: 0.076030
Epoch: 39       Validation p: 0.91, r:0.75, f: 0.82
Epoch: 40       Training Loss: 0.074001
Epoch: 40       Validation p: 0.90, r:0.76, f: 0.82
```

[18]:

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

p, r, f = eval(model, test_loader)
assert f > 0.70, "f1 below 0.70!"
```

[ ]:

```
```