Lab Journal 04.04.2023

Previous: Lab Journal 01.04.2023.

1) Creating one-hot embedding for reference

One-hot embedding can be created using the pytorch method, or the sklearn one. Finally, there is a builtin one-hot encoder in pandas.
First we create a dictionary for one-hot encoding with padding:

```
from collections import Counter # Counter inherits from dict. It creates
a dictionary of counts of unique elements.
from itertools import chain # chain transforms a starred list of lists
into a list
counts = Counter(chain(*data['junction_aa'].apply(list)))
vocabulary = set([k for k, v in counts.items()])
aa_to_id = {'<pad>':0, **{k:i for k, i in zip(vocabulary, range(1,
len(vocabulary)))}} # A dictionary with padding
```

Ok, it turns out, there are two potential ways to understand one-hot embedding in the present context. The one is when the embedding is 20-dimensional: each amino acid is encoded by a unit 20-dimensional vector, and the whole sequence embedding is just a normalized sum of all constituent amino acids. Such an embedding is obviously very imprecise, and there is not much sense in creating this, when we have a trainable `nn.Embedding` layer. The second option is to create $20 \times n_{max}$ matrix, where n is the maximal number of amino acids in the sequences. The authors of the original work used this approach: here is their code for embedding:

```
def encode_onehot_padded(aa_seqs):
'''
one-hot encoding of a list of amino acid sequences with padding
parameters:

- aa_seqs : list with CDR3 sequences returns:
- enc_aa_seq : list of np.ndarrays containing padded, encoded amino acid
sequences
'''
### Create an Amino Acid Dictionary
aa_list = sorted(['A', 'C', 'D', 'E','F','G', 'H','I', 'K', 'L','M','N',
\
'P', 'Q', 'R', 'S', 'T', 'V', 'W', 'Y', '-'])

aa_dict = {char : l for char, l in zip(aa_list, np.eye(len(aa_list),
k=0))}
```

```
#####pad the longer sequences with '-' sign
#1) identify the max length
max_seq_len = max([len(x) for x in aa_seqs])
#2) pad the shorter sequences with '-'
aa_seqs = [seq + (max_seq_len - len(seq))*'-' for i, seq in
enumerate(aa_seqs)]
# encode sequences:
sequences=[]

for seq in aa_seqs:

        e_seq=np.zeros((len(seq),len(aa_list)))
        count=0

        for aa in seq:
                if aa in aa_list:
                        e_seq[count]=aa_dict[aa]
                        count+=1
                else:
                        print ("Unknown amino acid in peptides: "+ aa +",
encoding aborted!\n")

                sequences.append(e_seq)

        enc_aa_seq = np.asarray(sequences)

return enc_aa_seq
```

Our attempt to create a dataloader for one-hot encodded sequences looks like this:

```
from torch.utils.data import Dataset
import torch.nn.functional as F
from torch.nn.utils.rnn import pad_sequence

class OneHotAminoDataset(Dataset):

    def __init__(self, dataset_X, dataset_y, aa_to_id, DEVICE):
        self.dataset_X = dataset_X.reset_index(drop=True)
        self.dataset_y = dataset_y.reset_index(drop=True)
        self.aa_to_id = aa_to_id
        self.length = dataset_y.shape[0]
        self.device = DEVICE

    def __len__(self):
        return self.length

    def __getitem__(self, idx):
        tokens = list(self.dataset_X.iloc[idx])
        ids = torch.LongTensor([self.aa_to_id[token] for token in tokens
```

```
if token in self.aa_to_id])
        ids = F.one_hot(ids, num_classes=20)
        y = self.dataset_y[idx]

        return ids, y

    def collate_fn(self, batch):
        ids, y = list(zip(*batch))
        padded_ids = pad_sequence(ids, batch_first=True).to(self.device)
        y = torch.LongTensor(y)


        return padded_ids, y
```

But first we've tested the general pipeline on a simple example:

```
from torch.utils.data import Dataset
import torch.nn.functional as F
from torch.nn.utils.rnn import pad_sequence

class AminoDataset(Dataset):

    def __init__(self, dataset_X, dataset_y, aa_to_id, DEVICE):
        self.dataset_X = dataset_X.reset_index(drop=True)
        self.dataset_y = dataset_y.reset_index(drop=True)
        self.aa_to_id = aa_to_id
        self.length = dataset_y.shape[0]
        self.device = DEVICE

    def __len__(self):
        return self.length

    def __getitem__(self, idx):
        tokens = list(self.dataset_X.iloc[idx])
        ids = torch.LongTensor([self.aa_to_id[token] for token in tokens
if token in self.aa_to_id])
        y = self.dataset_y[idx]

        return ids, y

    def collate_fn(self, batch):
        ids, y = list(zip(*batch))
        padded_ids = pad_sequence(ids, batch_first=True).to(self.device)
        y = torch.LongTensor(y)


        return padded_ids, y
```

Use the standard sklearn.model_selection `train_test_split`, to split our initial train dataset

```
from sklearn.model_selection import train_test_split

train_X, test_X, train_y, test_y = train_test_split(data['junction_aa'],
data['Label'], test_size=0.1)

train_dataset = OneHotAminoDataset(train_X, train_y, aa_to_id, device)
test_dataset = OneHotAminoDataset(test_X, test_y, aa_to_id, device)
```

Create samplers:

```
from torch.utils.data import RandomSampler, SequentialSampler

train_sampler = RandomSampler(train_dataset)
test_sampler = SequentialSampler(test_dataset)
```

Create DataLoaders:

```
from torch.utils.data import DataLoader
train_iterator = DataLoader(train_dataset, batch_size=64,
sampler=train_sampler, collate_fn=train_dataset.collate_fn)
test_iterator = DataLoader(test_dataset, batch_size=64,
sampler=test_sampler, collate_fn=test_dataset.collate_fn)
```

Create a simple NN with self-learning embedding:

```
from torch import nn

class Linear_with_learned_embedding(nn.Module):

    def __init__(self, vocab_size, embedding_dim):

        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.embedding2hidden = nn.Linear(embedding_dim, 10)
        self.act1 = nn.ReLU()
        self.dropout = nn.Dropout(p=0.5)
        self.hidden2out = nn.Linear(10, 1)
        self.act2 = nn.Sigmoid()

    def forward(self, text):
        embedded = self.embedding(text)
        mean_emb = torch.mean(embedded, dim=1)
        hidden = self.embedding2hidden(mean_emb)
        hidden = self.act1(hidden)
        hidden = self.dropout(hidden)
```

```
        out = self.hidden2out(hidden)
        proba = self.act2(out)
        return proba
```

Create the test model:

```
model_1 = Linear_with_learned_embedding(vocab_size=len(vocabulary)+1,
embedding_dim=20)
```

~~Create~~ copypaste the training loop:

```
def train(model, iterator, optimizer, criterion):
    print('Training...')
    epoch_loss = 0
    model.train()

    for i, (texts, ys) in enumerate(iterator):
        optimizer.zero_grad()
        preds_proba = model(texts).flatten()
        loss = criterion(preds_proba, ys.float())
        loss.backward()
        optimizer.step()
        epoch_loss += loss.item()

        if not (i + 1) % 20:
            print(f'Train loss: {epoch_loss/i}')

    return epoch_loss / len(iterator)

def validate(model, iterator, criterion):
    print("\nValidating...")
    epoch_loss = 0
    model.eval()
    with torch.no_grad():
        for i, (texts, ys) in enumerate(iterator):
            predictions = model(texts).flatten()
            loss = criterion(predictions, ys.float())
            epoch_loss += loss.item()
            if not (i + 1) % 5:
              print(f'Val loss: {epoch_loss/i}')

    return epoch_loss / len(iterator)
```

Create optimizer

```
import torch.optim as optim
optimizer = optim.Adam(model_1.parameters(), lr=0.01)
criterion = nn.BCELoss()
```

```
model_1 = model_1.to(device)
criterion = criterion.to(device)
```

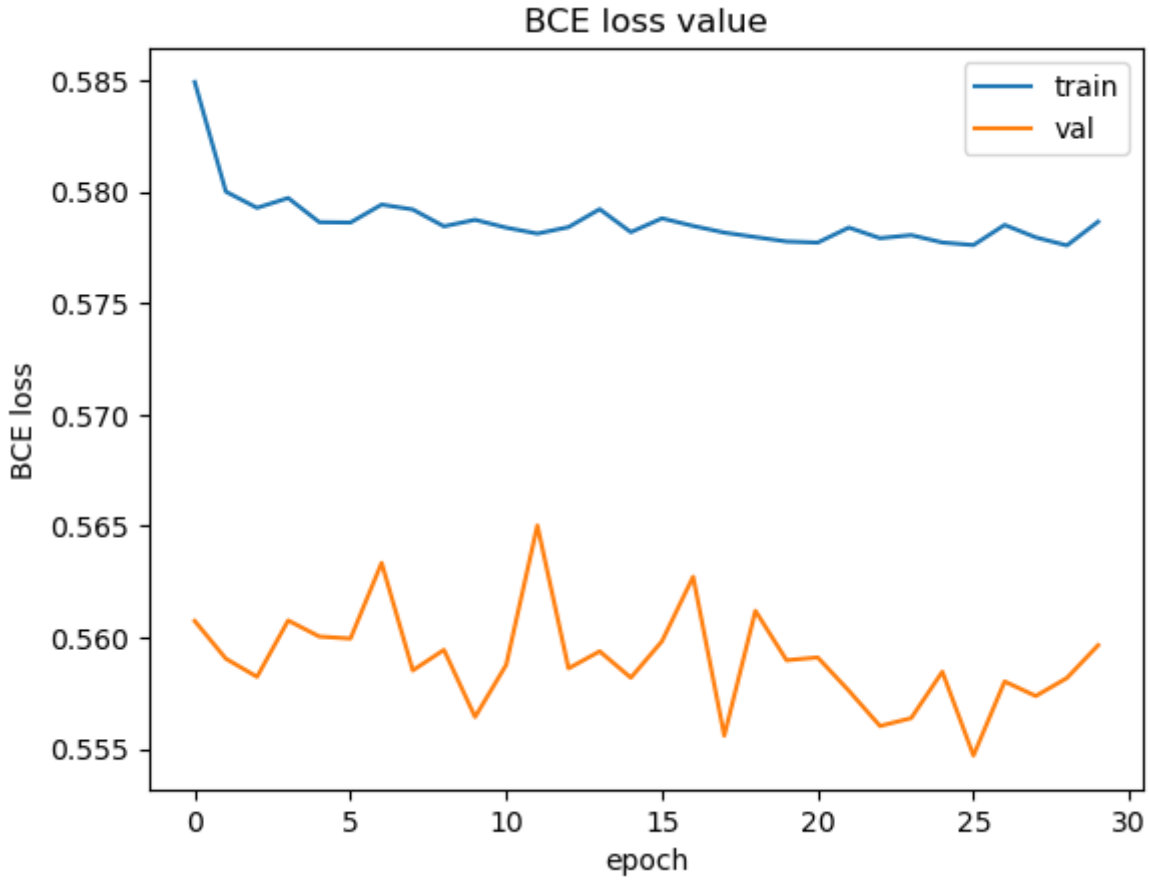
Train and evaluate the model:

```
losses_train = []
losses_validate = []

for i in range(30):
    print(f'\nstarting Epoch {i}')
    epoch_loss_on_train = train(model_1, train_iterator, optimizer,
criterion)
    losses_train.append(epoch_loss_on_train)

    epoch_loss_on_validation = validate(model_1, test_iterator,
criterion)
    losses_validate.append(epoch_loss_on_validation)
```

Graph the results:

```
import matplotlib.pyplot as plt
plt.plot(losses_train)
plt.plot(losses_validate)
plt.title('BCE loss value')
plt.ylabel('BCE loss')
plt.xlabel('epoch')
plt.legend(['train', 'val'], loc='upper right')
plt.show()
```

They are puzzling:



The next steps:

1. Finish the one-hot encoding DataLoader
2. Create a 2dCNN compatible with this one-hot DataLoader, with a number of parameters roughly equal to the one in the RNN model
3. Train the model and test accuracy metrices against the model from the paper

Previous: Lab Journal 01.04.2023
Next: