customer-sentiment-analysis

April 2, 2024

[1]:
```
# all the necessary imports
import torch
from torch.utils.data import Dataset, DataLoader
import torch.nn as nn
from torch import optim
import torchtext
# from torchtext.data import Field, LabelField
# from torchtext.utils.data import TabularDataset
# from torchtext.utils.data import Iterator, BucketIterator
from torchtext.legacy.data import Field, TabularDataset, BucketIterator,␣
↪Iterator
```

[2]:
```
# set the seed
manual_seed = 572
torch.manual_seed(manual_seed)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

CNN's for text classification

Convolutional Neural Networks (CNNs) are essentially a special case of a normal feed forward network, instead of being "densely" connected (note you may see people refer to Feed Forward networks as "dense layers"), nodes in CNNs connect to a smaller set of nodes, defined by the "filters size" of the network. CNNs thus have a smaller local windows to look at data, but make up for it by generally using many additional filters, which might be able to learn different aspects of the data. These networks turn out to be extremely useful for processing images, audio, and anything with some sort of spatial properties to the data.

It turns out they can also be used for any sentence classification task. Words in a sentence it turns out have a sort of 1D spatial ordering, which means some classification tasks can benefit from this CNNs ability to operate over the length of the sequence. In addition, because of the sparsity of the connections, you end up being able to make much smaller networks that retain a great deal of power.

Pytorch 1D CNNs and max-pooling Here's a quick example of how these networks function:

[3]:
```
x = torch.rand((2,5,10))   # batch size 2 with length 10 and 5 dim embedding.

in_dim = 5
filters = 4
```

```python
cnn1d = nn.Conv1d(in_dim,filters,kernel_size=3,padding=1)
max_pool = nn.MaxPool1d(kernel_size=3, padding=1)
activation = nn.ReLU()
x = cnn1d(x)
x = activation(x)
print(x)
print("Take the highest value in each window using max pool")
print(max_pool(x))
```

```
tensor([[[0.0115, 0.1652, 0.0000, 0.0210, 0.1613, 0.3903, 0.0000, 0.1766,
          0.1741, 0.2820],
         [0.0000, 0.0000, 0.0083, 0.1430, 0.0000, 0.0000, 0.2834, 0.0585,
          0.2041, 0.0000],
         [0.0456, 0.4565, 0.1949, 0.3690, 0.0888, 0.2262, 0.0732, 0.5068,
          0.5604, 0.3049],
         [0.0000, 0.0000, 0.0170, 0.1959, 0.0000, 0.0990, 0.0000, 0.0000,
          0.0000, 0.0000]],

        [[0.2387, 0.2856, 0.0000, 0.0000, 0.2042, 0.3228, 0.0630, 0.0000,
          0.0000, 0.2929],
         [0.0556, 0.2714, 0.0000, 0.0000, 0.0000, 0.1017, 0.1376, 0.1463,
          0.0000, 0.0000],
         [0.1951, 0.6214, 0.0341, 0.6379, 0.3405, 0.5166, 0.4090, 0.2046,
          0.3178, 0.5490],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.2466]]], grad_fn=<ReluBackward0>)
Take the highest value in each window using max pool
tensor([[[0.1652, 0.1613, 0.3903, 0.2820],
         [0.0000, 0.1430, 0.2834, 0.2041],
         [0.4565, 0.3690, 0.5068, 0.5604],
         [0.0000, 0.1959, 0.0990, 0.0000]],

        [[0.2856, 0.2042, 0.3228, 0.2929],
         [0.2714, 0.0000, 0.1463, 0.0000],
         [0.6214, 0.6379, 0.5166, 0.5490],
         [0.0000, 0.0000, 0.0000, 0.2466]]], grad_fn=<SqueezeBackward1>)
```

Here we've used a 1D CNN and max pooling to "summarize" our data, boiling a length 10 series to only 4 items.

[4]:
```python
cnn1d.weight.size()
```

[4]:
```
torch.Size([4, 5, 3])
```

[5]:
```python
param_count = 0
for layer in cnn1d.parameters():
    param_count += layer.numel()
```

```
print(param_count)
```

```
64
```

At the first layer there's an embedding of length 5 which is passed through a CNN layer with kernel size of 3 and there are 4 such filters. Thus we get 5 x 4 x3 = 60 parameters. Finally there are 4 bias values, one for each filter. Hence the total number of parameters are 64.

Size through CNNs CNNs can be a little tricky because as the data passes through them the dimensions might change based on number of filters, padding, and two other things ,stride and dilation. MaxPooling also quickly can decrease the size of the data. This is useful especially as a way to "feature extract" or "dimensionality reduction" but it's important to make sure we get the right output dimensions.

[6]:
```
## Test your answer by making a random tensor of the appropriate size,
## passing it through your proposed CNN+Maxpool and printing out the final size.

x = torch.rand((10,25,30))    # batch size 10 with length 30 and 25 dim␣
 ↪embedding.

in_dim = 25
filters = 5

ks = 3

cnn1d = nn.Conv1d(in_dim,filters,kernel_size=ks,padding=ks//2)
max_pool = nn.MaxPool1d(kernel_size=3, padding=1)
activation = nn.ReLU()
x = cnn1d(x)
x = activation(x)
```

[7]:
```
x.shape
```

[7]:
```
torch.Size([10, 5, 30])
```

[8]:
```
y = max_pool(x)
y.shape
```

[8]:
```
torch.Size([10, 5, 10])
```

1D CNN Model for Sentiment Analysis

[9]:
```
import torchtext

# define the white space tokenizer to get tokens
def tokenize_en(tweet):
    """
    Tokenizes English tweet from a string into a list of strings (tokens)
```

```
    """
    return tweet.strip().split()

# define the TorchText's fields
TEXT = Field(sequential=True, tokenize=tokenize_en, lower=True)
LABEL = Field(sequential=False, unk_token = None)


train, val, test = TabularDataset.splits(
    path="./data/sentiment-twitter-2016-task4/", # the root directory where the␣
 ↪data lies
    train='train.tsv', validation="dev.tsv", test="test.tsv", # file names
    format='tsv',
    skip_header=False, # if your tsv file has a header, make sure to pass this␣
 ↪to ensure it doesn't get proceesed as data!
    fields=[('tweet', TEXT), ('label', LABEL)])

TEXT.build_vocab(train, min_freq=3) # builds vocabulary based on all the words␣
 ↪that occur at least twice in the training set
LABEL.build_vocab(train)

train_iter, val_iter, test_iter = BucketIterator.splits(
 (train, val, test), # we pass in the datasets we want the iterator to draw␣
 ↪data from
 batch_sizes=(64,64,64),
 sort_key=lambda x: len(x.tweet),
 sort=True,
# A key to use for sorting examples in order to batch together examples with␣
 ↪similar lengths and minimize padding.
 sort_within_batch=True
)

VOCAB_SIZE = len(TEXT.vocab.stoi)
LABEL_SIZE = len(LABEL.vocab.stoi)

WORD_VEC_SIZE=300
# Note, the parameters to Embedding class below are:
# num_embeddings (int): size of the dictionary of embeddings
# embedding_dim (int): the size of each embedding vector
# For more details on Embedding class, see: https://github.com/pytorch/pytorch/
 ↪blob/master/torch/nn/modules/sparse.py

class ConvNet(nn.Module):

    def __init__(self, layer_num, filtersize, filters,nonlin, output_size,␣
 ↪VOCAB_SIZE,  WORD_VEC_SIZE):  #feel free to add additional parameters
```

```
        super(ConvNet, self).__init__()
        self.embedding = nn.Embedding(VOCAB_SIZE, WORD_VEC_SIZE, sparse=True)
        self.embedding.weight.data.normal_(0.0,0.05)
        self.layers = nn.ModuleList()
        self.nonlin = nonlin
        for i in range(layer_num):
            if i == 0:
            # YOUR CODE HERE    (FIRST LAYER CNN CODE)
                self.layers.append(nn.Conv1d(WORD_VEC_SIZE, filters,␣
↪kernel_size=filtersize, padding=filtersize//2))
            else:
                # YOUR CODE HERE    (LATER LAYER CNN CODE)
                self.layers.append(nn.Conv1d(filters,␣
↪filters,kernel_size=filtersize,padding=filtersize//2))
            self.layers.append(self.nonlin)

        self.max_layer = nn.AdaptiveMaxPool1d(1)
        self.output = nn.Linear(filters, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, x):
        # YOUR CODE HERE
        # PASS x THROUGH EMBEDDING (CHECK DIMENSIONS!!!):
        x = self.embedding(x) # Ouput: (L, N, C)
        # ENSURE DIMs CORRECT FOR CNN:
        x = x.permute(1, 2, 0) # CNN needs input (N,C,L)
        for layer in self.layers:
            x = layer(x)
        x = self.max_layer(x).squeeze(dim=-1)
        x =self.softmax(self.output(x))
        return x
```

[10]:
```
from sklearn.metrics import accuracy_score

def train(loader,model,criterion,optimizer,device):
    total_loss = 0.0
    # iterate throught the data loader
    num_sample = 0
    for batch in loader:
        # load the current batch
        batch_input = batch.tweet
        batch_output = batch.label

        batch_input = batch_input.to(device)
        batch_output = batch_output.to(device)
        # forward propagation
        # pass the data through the model
```

```
        model_outputs = model(batch_input)
        # compute the loss
        cur_loss = criterion(model_outputs, batch_output)
        total_loss += cur_loss.item()

        # backward propagation (compute the gradients and update the model)
        # clear the buffer
        optimizer.zero_grad()
        # compute the gradients
        cur_loss.backward()
        # update the weights
        optimizer.step()

        num_sample += batch_output.shape[0]
    return total_loss/num_sample

# evaluation logic based on classification accuracy
def evaluate(loader,model,criterion,device):
    all_pred=[]
    all_label = []
    with torch.no_grad(): # impacts the autograd engine and deactivate it.␣
 ↪reduces memory usage and speeds up computation
        for batch in loader:
             # load the current batch
            batch_input = batch.tweet
            batch_output = batch.label

            batch_input = batch_input.to(device)
            # forward propagation
            # pass the data through the model
            model_outputs = model(batch_input)
            # identify the predicted class for each example in the batch
            probabilities, predicted = torch.max(model_outputs.cpu().data, 1)
            # put all the true labels and predictions to two lists
            all_pred.extend(predicted)
            all_label.extend(batch_output)

    accuracy = accuracy_score(all_label, all_pred)
    return accuracy
```

Example run through:

[11]:
```
model = ConvNet(4, 3, 10, nn.ReLU(), LABEL_SIZE, VOCAB_SIZE, WORD_VEC_SIZE)
model = model.to(device)
model
```

[11]:
```
ConvNet(
  (embedding): Embedding(3330, 300, sparse=True)
  (layers): ModuleList(
    (0): Conv1d(300, 10, kernel_size=(3,), stride=(1,), padding=(1,))
    (1): ReLU()
    (2): Conv1d(10, 10, kernel_size=(3,), stride=(1,), padding=(1,))
    (3): ReLU()
    (4): Conv1d(10, 10, kernel_size=(3,), stride=(1,), padding=(1,))
    (5): ReLU()
    (6): Conv1d(10, 10, kernel_size=(3,), stride=(1,), padding=(1,))
    (7): ReLU()
  )
  (nonlin): ReLU()
  (max_layer): AdaptiveMaxPool1d(output_size=1)
  (output): Linear(in_features=10, out_features=3, bias=True)
  (softmax): LogSoftmax(dim=1)
)
```

[12]:
```
# Getting a batch of data

for batch in train_iter:
    tweets = batch.tweet
    labels = batch.label
    break  #we use first batch as an example.
```

[13]:
```
tweets.shape
```

[13]:
```
torch.Size([8, 64])
```

[14]:
```
# Passing input to GPU
tweets = tweets.to(device)
labels = labels.to(device)

out = model(tweets)
out
```

[14]:
```
tensor([[-1.3135, -0.8572, -1.1816],
        [-1.3131, -0.8567, -1.1827],
        [-1.3132, -0.8567, -1.1826],
        [-1.3135, -0.8567, -1.1823],
        [-1.3136, -0.8566, -1.1825],
        [-1.3135, -0.8565, -1.1827],
        [-1.3136, -0.8569, -1.1820],
        [-1.3133, -0.8575, -1.1814],
        [-1.3135, -0.8572, -1.1816],
        [-1.3139, -0.8575, -1.1809],
        [-1.3134, -0.8571, -1.1818],
```

```
[-1.3137, -0.8569, -1.1819],
[-1.3133, -0.8571, -1.1819],
[-1.3137, -0.8576, -1.1809],
[-1.3135, -0.8568, -1.1822],
[-1.3137, -0.8573, -1.1814],
[-1.3135, -0.8571, -1.1817],
[-1.3137, -0.8573, -1.1814],
[-1.3137, -0.8572, -1.1814],
[-1.3135, -0.8566, -1.1824],
[-1.3129, -0.8571, -1.1823],
[-1.3136, -0.8573, -1.1815],
[-1.3137, -0.8573, -1.1813],
[-1.3135, -0.8569, -1.1820],
[-1.3134, -0.8572, -1.1817],
[-1.3136, -0.8573, -1.1815],
[-1.3136, -0.8567, -1.1822],
[-1.3137, -0.8566, -1.1823],
[-1.3135, -0.8570, -1.1820],
[-1.3136, -0.8568, -1.1821],
[-1.3135, -0.8568, -1.1822],
[-1.3144, -0.8575, -1.1804],
[-1.3131, -0.8572, -1.1821],
[-1.3137, -0.8570, -1.1817],
[-1.3138, -0.8575, -1.1810],
[-1.3140, -0.8566, -1.1821],
[-1.3133, -0.8574, -1.1815],
[-1.3129, -0.8573, -1.1820],
[-1.3134, -0.8564, -1.1829],
[-1.3139, -0.8570, -1.1816],
[-1.3134, -0.8565, -1.1827],
[-1.3136, -0.8570, -1.1819],
[-1.3134, -0.8573, -1.1816],
[-1.3132, -0.8576, -1.1813],
[-1.3137, -0.8568, -1.1821],
[-1.3137, -0.8575, -1.1810],
[-1.3140, -0.8566, -1.1821],
[-1.3136, -0.8573, -1.1813],
[-1.3134, -0.8576, -1.1812],
[-1.3136, -0.8572, -1.1816],
[-1.3135, -0.8572, -1.1816],
[-1.3135, -0.8572, -1.1816],
[-1.3139, -0.8570, -1.1816],
[-1.3132, -0.8579, -1.1809],
[-1.3132, -0.8577, -1.1812],
[-1.3132, -0.8581, -1.1806],
[-1.3134, -0.8578, -1.1810],
[-1.3135, -0.8576, -1.1811],
```

```
        [-1.3134, -0.8568, -1.1822],
        [-1.3134, -0.8568, -1.1822],
        [-1.3133, -0.8575, -1.1813],
        [-1.3137, -0.8572, -1.1815],
        [-1.3133, -0.8577, -1.1811],
        [-1.3134, -0.8575, -1.1814]], grad_fn=<LogSoftmaxBackward0>)
```

[15]:
```
LEARNING_RATE = 0.01
criterion = nn.NLLLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=LEARNING_RATE)
```

[16]:
```
loss = criterion(out, labels)
loss
```

[16]:
```
tensor(1.1290, grad_fn=<NllLossBackward0>)
```

[17]:
```
# Sanity code to check if all works
train(train_iter, model, criterion, optimizer, device)
```

[17]:
```
0.01713936573266983
```

[18]:
```
evaluate(train_iter, model, criterion, device)
```

[18]:
```
0.3405
```

[19]:
```
evaluate(val_iter, model, criterion, device)
```

[19]:
```
0.3826913456728364
```

[20]:
```
evaluate(test_iter, model, criterion, device)
```

[20]:
```
0.5012601783637068
```

1D CNN Performance Based on our initial network we'd like to compare how depth matters vs number of filters in a given layer.

[21]:
```
import scipy.stats

LEARNING_RATE=.1
MAX_EPOCHS=10

def random_search(num_iter):
    results = []
    for i in range(num_iter):
        config = {
            #define hyperparameters here
            "layers": scipy.stats.randint.rvs(1,3),
```

```python
            "filters": scipy.stats.randint.rvs(10,200)
        }

        print("new config")
        print(config)
        model = ConvNet(config["layers"],3,config["filters"],nn.
↪ReLU(),output_size=3, VOCAB_SIZE=VOCAB_SIZE, WORD_VEC_SIZE=WORD_VEC_SIZE)
        model.to(device)
        criterion = nn.NLLLoss()
        optimizer = torch.optim.SGD(model.parameters(), lr=LEARNING_RATE)


        max_val = 0
        best_epoch = 0
        for epoch in range(MAX_EPOCHS):
        # train the model for one pass over the data
            train_loss = train(train_iter,model,criterion,optimizer,device)
        # compute the training accuracy
            train_acc = evaluate(train_iter,model,criterion,device)
        # compute the validation accuracy
            val_acc = evaluate(val_iter,model,criterion,device)
            if val_acc > max_val:
                max_val = val_acc
                best_epoch = epoch+1
        # print the loss for every epoch
            print('Epoch [{}/{}], Loss: {:.4f}, Training Accuracy: {:.4f},␣
↪Validation Accuracy: {:.4f}'.format(epoch+1, MAX_EPOCHS, train_loss,␣
↪train_acc, val_acc))
        results.append((max_val,best_epoch,config))
    return results
```

[22]:

```python
random_search(20)
```

```
new config
{'layers': 1, 'filters': 148}
Epoch [1/10], Loss: 0.0157, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [2/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [3/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [4/10], Loss: 0.0153, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [5/10], Loss: 0.0153, Training Accuracy: 0.5155, Validation Accuracy:
0.4222
Epoch [6/10], Loss: 0.0152, Training Accuracy: 0.5168, Validation Accuracy:
0.4302
```

```
Epoch [7/10], Loss: 0.0150, Training Accuracy: 0.5258, Validation Accuracy:
0.4452
Epoch [8/10], Loss: 0.0149, Training Accuracy: 0.5445, Validation Accuracy:
0.4622
Epoch [9/10], Loss: 0.0148, Training Accuracy: 0.5615, Validation Accuracy:
0.4637
Epoch [10/10], Loss: 0.0146, Training Accuracy: 0.5727, Validation Accuracy:
0.4642
new config
{'layers': 1, 'filters': 184}
Epoch [1/10], Loss: 0.0157, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [2/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [3/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [4/10], Loss: 0.0153, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [5/10], Loss: 0.0153, Training Accuracy: 0.5155, Validation Accuracy:
0.4212
Epoch [6/10], Loss: 0.0152, Training Accuracy: 0.5167, Validation Accuracy:
0.4292
Epoch [7/10], Loss: 0.0151, Training Accuracy: 0.5263, Validation Accuracy:
0.4442
Epoch [8/10], Loss: 0.0149, Training Accuracy: 0.5430, Validation Accuracy:
0.4622
Epoch [9/10], Loss: 0.0148, Training Accuracy: 0.5600, Validation Accuracy:
0.4682
Epoch [10/10], Loss: 0.0146, Training Accuracy: 0.5747, Validation Accuracy:
0.4652
new config
{'layers': 1, 'filters': 144}
Epoch [1/10], Loss: 0.0157, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [2/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [3/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [4/10], Loss: 0.0153, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [5/10], Loss: 0.0153, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [6/10], Loss: 0.0152, Training Accuracy: 0.5157, Validation Accuracy:
0.4222
Epoch [7/10], Loss: 0.0151, Training Accuracy: 0.5180, Validation Accuracy:
0.4277
Epoch [8/10], Loss: 0.0150, Training Accuracy: 0.5285, Validation Accuracy:
0.4437
```

```
Epoch [9/10], Loss: 0.0148, Training Accuracy: 0.5522, Validation Accuracy:
0.4587
Epoch [10/10], Loss: 0.0146, Training Accuracy: 0.5743, Validation Accuracy:
0.4667
new config
{'layers': 1, 'filters': 89}
Epoch [1/10], Loss: 0.0157, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [2/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [3/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [4/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [5/10], Loss: 0.0153, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [6/10], Loss: 0.0152, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [7/10], Loss: 0.0151, Training Accuracy: 0.5157, Validation Accuracy:
0.4232
Epoch [8/10], Loss: 0.0150, Training Accuracy: 0.5220, Validation Accuracy:
0.4362
Epoch [9/10], Loss: 0.0149, Training Accuracy: 0.5403, Validation Accuracy:
0.4477
Epoch [10/10], Loss: 0.0147, Training Accuracy: 0.5598, Validation Accuracy:
0.4632
new config
{'layers': 1, 'filters': 75}
Epoch [1/10], Loss: 0.0158, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [2/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [3/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [4/10], Loss: 0.0153, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [5/10], Loss: 0.0153, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [6/10], Loss: 0.0152, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [7/10], Loss: 0.0151, Training Accuracy: 0.5178, Validation Accuracy:
0.4287
Epoch [8/10], Loss: 0.0150, Training Accuracy: 0.5247, Validation Accuracy:
0.4432
Epoch [9/10], Loss: 0.0148, Training Accuracy: 0.5430, Validation Accuracy:
0.4567
Epoch [10/10], Loss: 0.0147, Training Accuracy: 0.5625, Validation Accuracy:
0.4677
```

```
new config
{'layers': 2, 'filters': 173}
Epoch [1/10], Loss: 0.0158, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [2/10], Loss: 0.0155, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [3/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [4/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [5/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [6/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [7/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [8/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [9/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [10/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
new config
{'layers': 1, 'filters': 73}
Epoch [1/10], Loss: 0.0157, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [2/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [3/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [4/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [5/10], Loss: 0.0153, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [6/10], Loss: 0.0153, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [7/10], Loss: 0.0152, Training Accuracy: 0.5157, Validation Accuracy:
0.4222
Epoch [8/10], Loss: 0.0151, Training Accuracy: 0.5183, Validation Accuracy:
0.4302
Epoch [9/10], Loss: 0.0150, Training Accuracy: 0.5375, Validation Accuracy:
0.4432
Epoch [10/10], Loss: 0.0148, Training Accuracy: 0.5600, Validation Accuracy:
0.4502
new config
{'layers': 1, 'filters': 188}
Epoch [1/10], Loss: 0.0157, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
```

```
Epoch [2/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [3/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [4/10], Loss: 0.0153, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [5/10], Loss: 0.0152, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [6/10], Loss: 0.0152, Training Accuracy: 0.5155, Validation Accuracy:
0.4237
Epoch [7/10], Loss: 0.0150, Training Accuracy: 0.5218, Validation Accuracy:
0.4387
Epoch [8/10], Loss: 0.0149, Training Accuracy: 0.5363, Validation Accuracy:
0.4557
Epoch [9/10], Loss: 0.0148, Training Accuracy: 0.5593, Validation Accuracy:
0.4682
Epoch [10/10], Loss: 0.0146, Training Accuracy: 0.5780, Validation Accuracy:
0.4727
new config
{'layers': 1, 'filters': 197}
Epoch [1/10], Loss: 0.0157, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [2/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [3/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [4/10], Loss: 0.0153, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [5/10], Loss: 0.0153, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [6/10], Loss: 0.0152, Training Accuracy: 0.5155, Validation Accuracy:
0.4227
Epoch [7/10], Loss: 0.0151, Training Accuracy: 0.5180, Validation Accuracy:
0.4317
Epoch [8/10], Loss: 0.0149, Training Accuracy: 0.5310, Validation Accuracy:
0.4467
Epoch [9/10], Loss: 0.0148, Training Accuracy: 0.5537, Validation Accuracy:
0.4652
Epoch [10/10], Loss: 0.0146, Training Accuracy: 0.5750, Validation Accuracy:
0.4697
new config
{'layers': 2, 'filters': 175}
Epoch [1/10], Loss: 0.0158, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [2/10], Loss: 0.0155, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [3/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
```

```
Epoch [4/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [5/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [6/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [7/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [8/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [9/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [10/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
new config
{'layers': 1, 'filters': 165}
Epoch [1/10], Loss: 0.0157, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [2/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [3/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [4/10], Loss: 0.0153, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [5/10], Loss: 0.0153, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [6/10], Loss: 0.0152, Training Accuracy: 0.5155, Validation Accuracy:
0.4222
Epoch [7/10], Loss: 0.0151, Training Accuracy: 0.5195, Validation Accuracy:
0.4297
Epoch [8/10], Loss: 0.0149, Training Accuracy: 0.5315, Validation Accuracy:
0.4557
Epoch [9/10], Loss: 0.0148, Training Accuracy: 0.5548, Validation Accuracy:
0.4717
Epoch [10/10], Loss: 0.0146, Training Accuracy: 0.5757, Validation Accuracy:
0.4822
new config
{'layers': 1, 'filters': 110}
Epoch [1/10], Loss: 0.0158, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [2/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [3/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [4/10], Loss: 0.0153, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [5/10], Loss: 0.0153, Training Accuracy: 0.5157, Validation Accuracy:
0.4222
```

```
Epoch [6/10], Loss: 0.0152, Training Accuracy: 0.5167, Validation Accuracy:
0.4282
Epoch [7/10], Loss: 0.0150, Training Accuracy: 0.5255, Validation Accuracy:
0.4492
Epoch [8/10], Loss: 0.0149, Training Accuracy: 0.5453, Validation Accuracy:
0.4562
Epoch [9/10], Loss: 0.0148, Training Accuracy: 0.5615, Validation Accuracy:
0.4512
Epoch [10/10], Loss: 0.0146, Training Accuracy: 0.5788, Validation Accuracy:
0.4647
new config
{'layers': 2, 'filters': 97}
Epoch [1/10], Loss: 0.0158, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [2/10], Loss: 0.0155, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [3/10], Loss: 0.0155, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [4/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [5/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [6/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [7/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [8/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [9/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [10/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
new config
{'layers': 1, 'filters': 99}
Epoch [1/10], Loss: 0.0157, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [2/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [3/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [4/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [5/10], Loss: 0.0153, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [6/10], Loss: 0.0152, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [7/10], Loss: 0.0151, Training Accuracy: 0.5160, Validation Accuracy:
0.4292
```

```
Epoch [8/10], Loss: 0.0150, Training Accuracy: 0.5238, Validation Accuracy:
0.4367
Epoch [9/10], Loss: 0.0149, Training Accuracy: 0.5380, Validation Accuracy:
0.4597
Epoch [10/10], Loss: 0.0147, Training Accuracy: 0.5560, Validation Accuracy:
0.4617
new config
{'layers': 1, 'filters': 104}
Epoch [1/10], Loss: 0.0157, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [2/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [3/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [4/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [5/10], Loss: 0.0153, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [6/10], Loss: 0.0152, Training Accuracy: 0.5155, Validation Accuracy:
0.4217
Epoch [7/10], Loss: 0.0151, Training Accuracy: 0.5180, Validation Accuracy:
0.4247
Epoch [8/10], Loss: 0.0150, Training Accuracy: 0.5280, Validation Accuracy:
0.4397
Epoch [9/10], Loss: 0.0149, Training Accuracy: 0.5475, Validation Accuracy:
0.4597
Epoch [10/10], Loss: 0.0147, Training Accuracy: 0.5647, Validation Accuracy:
0.4682
new config
{'layers': 1, 'filters': 24}
Epoch [1/10], Loss: 0.0157, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [2/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [3/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [4/10], Loss: 0.0153, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [5/10], Loss: 0.0153, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [6/10], Loss: 0.0152, Training Accuracy: 0.5157, Validation Accuracy:
0.4227
Epoch [7/10], Loss: 0.0151, Training Accuracy: 0.5190, Validation Accuracy:
0.4257
Epoch [8/10], Loss: 0.0150, Training Accuracy: 0.5273, Validation Accuracy:
0.4412
Epoch [9/10], Loss: 0.0149, Training Accuracy: 0.5427, Validation Accuracy:
0.4482
```

```
Epoch [10/10], Loss: 0.0148, Training Accuracy: 0.5547, Validation Accuracy:
0.4557
new config
{'layers': 1, 'filters': 129}
Epoch [1/10], Loss: 0.0157, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [2/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [3/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [4/10], Loss: 0.0153, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [5/10], Loss: 0.0153, Training Accuracy: 0.5155, Validation Accuracy:
0.4222
Epoch [6/10], Loss: 0.0152, Training Accuracy: 0.5178, Validation Accuracy:
0.4282
Epoch [7/10], Loss: 0.0151, Training Accuracy: 0.5260, Validation Accuracy:
0.4422
Epoch [8/10], Loss: 0.0149, Training Accuracy: 0.5412, Validation Accuracy:
0.4602
Epoch [9/10], Loss: 0.0148, Training Accuracy: 0.5570, Validation Accuracy:
0.4622
Epoch [10/10], Loss: 0.0146, Training Accuracy: 0.5720, Validation Accuracy:
0.4607
new config
{'layers': 2, 'filters': 105}
Epoch [1/10], Loss: 0.0158, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [2/10], Loss: 0.0155, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [3/10], Loss: 0.0155, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [4/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [5/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [6/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [7/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [8/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [9/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [10/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
new config
{'layers': 1, 'filters': 59}
```

```
Epoch [1/10], Loss: 0.0159, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [2/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [3/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [4/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [5/10], Loss: 0.0153, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [6/10], Loss: 0.0152, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [7/10], Loss: 0.0151, Training Accuracy: 0.5168, Validation Accuracy:
0.4262
Epoch [8/10], Loss: 0.0150, Training Accuracy: 0.5295, Validation Accuracy:
0.4502
Epoch [9/10], Loss: 0.0148, Training Accuracy: 0.5568, Validation Accuracy:
0.4582
Epoch [10/10], Loss: 0.0147, Training Accuracy: 0.5732, Validation Accuracy:
0.4707
new config
{'layers': 2, 'filters': 147}
Epoch [1/10], Loss: 0.0158, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [2/10], Loss: 0.0155, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [3/10], Loss: 0.0155, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [4/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [5/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [6/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [7/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [8/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [9/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
Epoch [10/10], Loss: 0.0154, Training Accuracy: 0.5157, Validation Accuracy:
0.4217
```

```
[22]: [(0.464232116058029, 10, {'layers': 1, 'filters': 148}),
       (0.46823411705852924, 9, {'layers': 1, 'filters': 184}),
       (0.46673336668334164, 10, {'layers': 1, 'filters': 144}),
       (0.46323161580790395, 10, {'layers': 1, 'filters': 89}),
```

```
(0.46773386693346675, 10, {'layers': 1, 'filters': 75}),
(0.42171085542771386, 1, {'layers': 2, 'filters': 173}),
(0.4502251125562781, 10, {'layers': 1, 'filters': 73}),
(0.47273636818409204, 10, {'layers': 1, 'filters': 188}),
(0.46973486743371684, 10, {'layers': 1, 'filters': 197}),
(0.42171085542771386, 1, {'layers': 2, 'filters': 175}),
(0.4822411205602801, 10, {'layers': 1, 'filters': 165}),
(0.46473236618309155, 10, {'layers': 1, 'filters': 110}),
(0.42171085542771386, 1, {'layers': 2, 'filters': 97}),
(0.46173086543271635, 10, {'layers': 1, 'filters': 99}),
(0.46823411705852924, 10, {'layers': 1, 'filters': 104}),
(0.45572786393196596, 10, {'layers': 1, 'filters': 24}),
(0.4622311155577789, 9, {'layers': 1, 'filters': 129}),
(0.42171085542771386, 1, {'layers': 2, 'filters': 105}),
(0.47073536768384194, 10, {'layers': 1, 'filters': 59}),
(0.42171085542771386, 1, {'layers': 2, 'filters': 147})]
```

1D CNN summary My best score is 48.47% with 2 layers and 73 filters. Based on the results having two layers gave a better score and having filters in the range of 50-80 gave the best results, anything higher reduced the scores possibly due to ovrefitting.

0.1 Building a Recurrent Neural Network

we use a corpus from the CL-Aff shared task. HappyDB is a dataset of about 100,000 `happy moments` crowd-sourced via Amazon's Mechanical Turk where each worker was asked to describe in a complete sentence `what made them happy in the past 24 hours`. Each user was asked to describe three such moments.

We have already preprocessed (tokenization, removing URLs, mentions, hashtags and so on) the tweets and placed it under `data/happy_db` folder in three files as `train.tsv`, `dev.tsv` and `test.tsv`.

Whitespace tokenizer

[40]:
```
from nltk.tokenize import WhitespaceTokenizer

def whitespace_tokenize(text):

    # your code goes here
    tokens = WhitespaceTokenizer().tokenize(text)
    return tokens
```

TorchText's Fields

[41]:
```
# your code goes here
TEXT = Field(sequential=True, tokenize=whitespace_tokenize, lower=False)
LABEL = Field(sequential=False, unk_token = None)
```

TabularDataset class and Fields

```
[42]: # your code goes here
train,val,test = TabularDataset.splits(path= "./data/happy_db/",train='train.
 ↪tsv', validation="dev.tsv", test="test.tsv", # file names
    format='tsv',
    skip_header=True, # if your tsv file has a header, make sure to pass this␣
 ↪to ensure it doesn't get proceesed as data!
    fields=[('tweet', TEXT), ('label', LABEL)])
```

Building vocab

```
[43]: # your code goes here
TEXT.build_vocab(train, max_size=5000) # builds vocabulary based on all the␣
 ↪words that occur at least twice in the training set
LABEL.build_vocab(train, max_size = 5000)
```

```
[44]: # your code goes here
print(len(TEXT.vocab.stoi))
print(len(LABEL.vocab.stoi))
```

```
5002
2
```

Constructing Iterators

```
[45]: # your code goes here
#from torchtext.data import Iterator, BucketIterator

train_iter, val_iter, test_iter = BucketIterator.splits(
 (train, val, test), # we pass in the datasets we want the iterator to draw␣
 ↪data from
 batch_sizes=(32,32,32),
 sort_key=lambda x: len(x.tweet),
 sort=True,
# A key to use for sorting examples in order to batch together examples with␣
 ↪similar lengths and minimize padding.
 sort_within_batch=True
)
```

Model creation

```
[46]: import torch.nn as nn
class LSTMmodel(nn.Module):

  def __init__(self, embedding_size, vocab_size, output_size, hidden_size,␣
 ↪num_layers):
    # In the constructor we define the layers for our model
    super(LSTMmodel, self).__init__()
```

```
        # word embedding lookup table
        self.embedding = nn.Embedding(num_embeddings=vocab_size,␣
↪embedding_dim=embedding_size, sparse=True)
        # core LSTM module
        self.LSTM_layer = nn.LSTM(input_size=300, hidden_size=500, num_layers=2)
        # activation function
        self.activation_fn = nn.Tanh()
        # classification related modules
        self.linear_layer = nn.Linear(hidden_size, output_size)
        self.softmax_layer = nn.LogSoftmax(dim=1)
        self.debug = False

    def forward(self, x):
        # In the forward function we define the forward propagation logic
        if self.debug:
            print("input word indices shape = ", x.size())
        out = self.embedding(x)
        if self.debug:
            print("word embeddings shape = ", out.size())
        out, _ = self.LSTM_layer(out) # since we are not feeding h_0 explicitly,␣
↪h_0 will be initialized to zeros by default
        if self.debug:
            print("RNN output (features from last layer of RNN for all timesteps)␣
↪shape = ", out.size())
        # classify based on the hidden representation after RNN processes the last␣
↪token
        out = out[-1]
        if self.debug:
            print("Tweet embeddings or RNN output (features from last layer of RNN␣
↪for the last timestep only) shape = ", out.size())
        out = self.activation_fn(out)
        if self.debug:
            print("ReLU output shape = ", out.size())
        out = self.linear_layer(out)
        if self.debug:
            print("linear layer output shape = ", out.size())
        out = self.softmax_layer(out) # accepts 2D or more dimensional inputs
        if self.debug:
            print("softmax layer output shape = ", out.size())
        return out
```

Instantiating the model

[47]:
```
# your code goes here
HIDDEN_SIZE = 500 # no. of units in the hidden layer
NUM_LAYERS = 2
EMBEDDING_SIZE = 300
```

```python
VOCAB_SIZE = len(TEXT.vocab.stoi)
NUM_CLASSES = len(LABEL.vocab.stoi)
```

[48]:
```python
model = LSTMmodel(EMBEDDING_SIZE, VOCAB_SIZE, NUM_CLASSES, HIDDEN_SIZE,
 ↪NUM_LAYERS)
print(model)
#model.to(device)
```

```
LSTMmodel(
  (embedding): Embedding(5002, 300, sparse=True)
  (LSTM_layer): LSTM(300, 500, num_layers=2)
  (activation_fn): Tanh()
  (linear_layer): Linear(in_features=500, out_features=2, bias=True)
  (softmax_layer): LogSoftmax(dim=1)
)
```

Create an optimizer for training

[49]:
```python
LEARNING_RATE = 0.1
criterion = nn.NLLLoss()
# create an instance of SGD with required hyperparameters
optimizer = torch.optim.SGD(model.parameters(), lr=LEARNING_RATE)
```

[50]:
```python
# your code goes here
# your code goes here
total_parameters = 0
for variable in model.parameters():
    # shape is an array of tf.Dimension
    shape = variable.shape
    variable_parameters = 1
    for dim in shape:
        variable_parameters *= dim
    total_parameters += variable_parameters
print("Total Parameters:",total_parameters)
```

```
Total Parameters: 5109602
```

[51]:
```python
points = 5110103
bits_per_point = 32
number_of_mega_byte= (((points*bits_per_point)/8)/10**6)
print("Model Mega Byte memory:",number_of_mega_byte)
```

```
Model Mega Byte memory: 20.440412
```

training and evaluation

[52]:
```python
import numpy as np
from sklearn.metrics import accuracy_score
from sklearn.metrics import f1_score
```

```
def train(loader):
    total_loss = 0.0
    # iterate throught the data loader
    num_sample = 0
    for batch in loader:
        # load the current batch
        batch_input = batch.tweet
        batch_output = batch.label

        batch_input = batch_input.to(device)
        batch_output = batch_output.to(device)
        # forward propagation
        # pass the data through the model
        model_outputs = model(batch_input)
        # compute the loss
        cur_loss = criterion(model_outputs, batch_output)
        total_loss += cur_loss.item()

        # backward propagation (compute the gradients and update the model)
        # clear the buffer
        optimizer.zero_grad()
        # compute the gradients
        cur_loss.backward()
        # update the weights
        optimizer.step()

        num_sample += batch_output.shape[0]
    return total_loss/num_sample

# evaluation logic based on classification accuracy
def evaluate(loader):
    all_pred=[]
    all_label = []
    with torch.no_grad(): # impacts the autograd engine and deactivate it.
 ↪reduces memory usage and speeds up computation
        for batch in loader:
             # load the current batch
            batch_input = batch.tweet
            batch_output = batch.label

            batch_input = batch_input.to(device)
            # forward propagation
            # pass the data through the model
            model_outputs = model(batch_input)
            # identify the predicted class for each example in the batch
            probabilities, predicted = torch.max(model_outputs.cpu().data, 1)
```

```
            # put all the true labels and predictions to two lists
            all_pred.extend(predicted)
            all_label.extend(batch_output)

    accuracy = accuracy_score(all_label, all_pred)
    f1score = f1_score(all_label, all_pred, average='macro')
    return accuracy,f1score
```

Saving the model

[53]:
```
# your code goes here
# Print model's state_dict
import os
if not os.path.exists("./ckpt"): # check if the directory doesn't exist already
    os.mkdir("./ckpt")
```

[54]:
```
# start the training
MAX_EPOCHS = 10
for epoch in range(MAX_EPOCHS):
    # train the model for one pass over the data
    train_loss = train(train_iter)
    # compute the training accuracy
    train_acc = evaluate(train_iter)
    # compute the validation accuracy
    val_acc = evaluate(val_iter)

    # print the loss for every epoch
    print('epoch ',epoch+1,'loss ', train_loss,'Train Accuracy &␣
 ↪F1',train_acc,'Validation Accuracy & F1 ', val_acc)

    # save model, optimizer, and number of epoch to a dictionary
    model_save = {
            'epoch': epoch,  # number of epoch
            'model_state_dict': model.state_dict(), # model parameters
            'optimizer_state_dict': optimizer.state_dict(), # save optimizer
            'loss': train_loss # training loss
            }

    # use torch.save to store
    torch.save(model_save, "./ckpt/model_{}.pt".format(epoch))
```

```
epoch  1 loss  0.019151628931109426 Train Accuracy & F1 (0.7309422348484849,
0.7137388123133535) Validation Accuracy & F1  (0.6732954545454546,
0.6316692464243143)
epoch  2 loss  0.013246820703374617 Train Accuracy & F1 (0.8325047348484849,
0.8324948186057661) Validation Accuracy & F1  (0.7926136363636364,
0.7924571141394334)
```

```
epoch  3 loss  0.011463872385458231 Train Accuracy & F1 (0.7906013257575758,
0.7885851163107714) Validation Accuracy & F1  (0.7679924242424242,
0.7676421588658725)
epoch  4 loss  0.010660828917576564 Train Accuracy & F1 (0.8290719696969697,
0.8284218987839711) Validation Accuracy & F1  (0.7897727272727273,
0.7894396482038348)
epoch  5 loss  0.009733532781176495 Train Accuracy & F1 (0.8325047348484849,
0.8315165046642996) Validation Accuracy & F1  (0.7878787878787878,
0.7867151404983717)
epoch  6 loss  0.008951072622122329 Train Accuracy & F1 (0.8135653409090909,
0.8113326949609299) Validation Accuracy & F1  (0.7471590909090909,
0.7425695386005999)
epoch  7 loss  0.008422675708337038 Train Accuracy & F1 (0.8693181818181818,
0.8691640062979097) Validation Accuracy & F1  (0.8210227272727273,
0.8210225667744264)
epoch  8 loss  0.007800204066425618 Train Accuracy & F1 (0.8938210227272727,
0.8937999504810787) Validation Accuracy & F1  (0.8248106060606061,
0.8246783840903891)
epoch  9 loss  0.0071551693174776365 Train Accuracy & F1 (0.8894412878787878,
0.889384151593454) Validation Accuracy & F1  (0.8172348484848485,
0.816931303428496)
epoch  10 loss  0.0065436911794346415 Train Accuracy & F1 (0.8776041666666666,
0.8752106220183131) Validation Accuracy & F1  (0.8418560606060606,
0.8366226150275844)
```

[55]:
```python
# define a new model

model2 = LSTMmodel(EMBEDDING_SIZE, VOCAB_SIZE, NUM_CLASSES, HIDDEN_SIZE,␣
 ↪NUM_LAYERS)

# load checkpoint

checkpoint = torch.load("./ckpt/model_9.pt")

# assign the parameters of checkpoint to this new model

model2.load_state_dict(checkpoint['model_state_dict'])
model2.to(device)

print(model2) # can be used for inference or for further training
```

```
LSTMmodel(
  (embedding): Embedding(5002, 300, sparse=True)
  (LSTM_layer): LSTM(300, 500, num_layers=2)
  (activation_fn): Tanh()
  (linear_layer): Linear(in_features=500, out_features=2, bias=True)
  (softmax_layer): LogSoftmax(dim=1)
)
```

Model Evaluation

[56]:
```
# your code goes here
accuracy, f1score = evaluate(test_iter)
print("Test Accuracy : ", accuracy)
print("Test F1 : ", f1score)
```

```
Test Accuracy :  0.8314393939393939
Test F1 :  0.8265100807419248
```