Project Documentation (Shuhong Zheng)

This is the documentation of my project “Sentiment Analysis for Reviews”. It aims at performing sentiment analysis (mainly positive or negative) for reviews from the IMDb website for movies.

1) An overview of the function of the code

The function of this code is to use the training data in IMDb dataset to train a model, and then use the trained model to perform sentiment analysis.

2) Implementation details

The code is made of mainly two parts: (1) The main training code; (2) The model definition. The main training code is in the “main.py”. I have detailed comments in the code but I also introduce it here. The first part is parsing the arguments in the command line, including the model architecture we use and the number of epochs we are going to train:

```
# Parse the argument to determine the model architecture we are using and our training epochs
parser = argparse.ArgumentParser()
parser.add_argument("--model_arch", type=str, default='cnn',
                    help='specify the model architecture we are using for sentiment analysis')
parser.add_argument("--n_epochs", type=int, default=10,
                    help='specify the number of epochs to train the model')
args = parser.parse_args()
```

Then it is a part loading the training and testing data of IMDb and then tokenize them. We are using the “basic_english” tokenizer provided in torchtext:

```
# Load training and testing data and tokenize them
train_data, test_data = datasets.load_dataset('imdb', split=['train', 'test'])
tokenizer = torchtext.data.utils.get_tokenizer('basic_english')
```

After defining the tokenizer, we are going to convert the training and testing data to tokens:

```
def tokenize_data(example, tokenizer):
    tokens = {'tokens': tokenizer(example['text'])}
    return tokens

train_data = train_data.map(tokenize_data, fn_kwargs={'tokenizer': tokenizer})
test_data = test_data.map(tokenize_data, fn_kwargs={'tokenizer': tokenizer})
```

Afterwards, the next step is to split the whole training set into training and validation data, where the validation data can be used during training to evaluate the performance of the model. Thus, it can help us pick the best model during training:

```
# Split the training and validation data
train_valid_data = train_data.train_test_split(test_size=0.25)
train_data = train_valid_data['train']
valid_data = train_valid_data['test']

min_freq = 3
special_tokens = ['<unk>', '<pad>']
```

We then build the vocabulary list from the dataset, using the tool provided by the torchtext package:

```
# Build the vocabulary
vocab = torchtext.vocab.build_vocab_from_iterator(train_data['tokens'],
                                                  min_freq=min_freq,
                                                  specials=special_tokens)

unk_index = vocab['<unk>']
pad_index = vocab['<pad>']

vocab.set_default_index(unk_index)
```

Then we determine the model architecture according to the parameters defined in the command line:

```
# Determine the model architecture according to the parameter
if args.model_arch == "cnn":
    vocab_size = len(vocab)
    embedding_dim = 300
    n_filters = 100
    filter_sizes = [3,5,7]
    output_dim = len(train_data.unique('label'))
    dropout_rate = 0.25
    model = CNN(vocab_size, embedding_dim, n_filters, filter_sizes, output_dim, dropout_rate, pad_index)
elif args.model_arch == "nbow":
    model = NBoW(vocab_size, embedding_dim, output_dim, pad_index)
else:
    raise NotImplementedError
```

Afterwards, we initialize the settings of the training process, including the loss function, the optimizer, etc.:

```
# Initialize the embedding mapping, the optimizer, and the loss function
vectors = torchtext.vocab.GloVe()
optimizer = optim.Adam(model.parameters())
criterion = nn.CrossEntropyLoss()
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

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

Then, we specify the dataloaders:

```
# Specify dataloaders
train_dataloader = torch.utils.data.DataLoader(train_data, batch_size=batch_size, collate_fn=collate)
valid_dataloader = torch.utils.data.DataLoader(valid_data, batch_size=batch_size, collate_fn=collate)
test_dataloader = torch.utils.data.DataLoader(test_data, batch_size=batch_size, collate_fn=collate)
```

This is the training function:

```
# Training function
def train(dataloader, model, criterion, optimizer, device):

    model.train()
    epoch_loss = 0
    epoch_accuracy = 0

    for batch in dataloader:
        tokens = batch['ids'].to(device)
        labels = batch['labels'].to(device)
        if args.model_arch == "lstm":
            predictions = model(tokens, batch['length'].to(device))
        else:
            predictions = model(tokens)
        loss = criterion(predictions, labels)
        accuracy = get_accuracy(predictions, labels)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        epoch_loss += loss.item()
        epoch_accuracy += accuracy.item()

    return epoch_loss / len(dataloader), epoch_accuracy / len(dataloader)
```

This is the evaluation function:

```
# Evaluation function
def evaluate(dataloader, model, criterion, device):

    model.eval()
    epoch_loss = 0
    epoch_accuracy = 0

    with torch.no_grad():
        for batch in dataloader:
            tokens = batch['ids'].to(device)
            labels = batch['labels'].to(device)
            predictions = model(tokens)
            loss = criterion(predictions, labels)
            accuracy = get_accuracy(predictions, labels)
            epoch_loss += loss.item()
            epoch_accuracy += accuracy.item()

    return epoch_loss / len(dataloader), epoch_accuracy / len(dataloader)
```

And this is the function to calculate the accuracy:

```python
# The function that calculates the accuracy
def get_accuracy(predictions, labels):
    batch_size = predictions.shape[0]
    predicted_classes = predictions.argmax(1, keepdim=True)
    correct_predictions = predicted_classes.eq(labels.view_as(predicted_classes)).sum()
    accuracy = correct_predictions.float() / batch_size
    return accuracy
```

Finally, this is the training “for-loop”:

```python
n_epochs = args.n_epochs

for epoch in range(n_epochs):

    train_loss, train_acc = train(train_dataloader, model, criterion, optimizer, device)
    valid_loss, valid_acc = evaluate(valid_dataloader, model, criterion, device)

    print(f'epoch: {epoch+1}')
    print(f'train_loss: {train_loss:.3f}, train_acc: {train_acc:.3f}')
    print(f'valid_loss: {valid_loss:.3f}, valid_acc: {valid_acc:.3f}')
```

As for the model definitions, they are in the “models” directory. There exist two options: N-Bag-of-Words and CNN. For N-Bag-of-Words, the model will pool all the representations into one vector, and then use a linear layer to map it into the classification value:

```python
import torch.nn as nn

class NBoW(nn.Module):
    def __init__(self, vocab_size, embedding_dim, output_dim, pad_index):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim, padding_idx=pad_index)
        self.fc = nn.Linear(embedding_dim, output_dim)

    def forward(self, text):
        embedded = self.embedding(text)
        pooled = embedded.mean(dim=1)
        prediction = self.fc(pooled)
        return prediction
```

For the CNN architecture, it is more complicated compared to the N-Bag-of-Words, as they would have a CNN layer to first calculate the adjacent features from the nearby text embeddings, and then map into the classification value via a linear layer:

```python
import torch.nn as nn
import torch
class CNN(nn.Module):
    def __init__(self, vocab_size, embedding_dim, n_filters, filter_sizes, output_dim, dropout_rate,
                 pad_index):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim, padding_idx=pad_index)
        self.convs = nn.ModuleList([nn.Conv1d(embedding_dim,
                                              n_filters,
                                              filter_size)
                                    for filter_size in filter_sizes])
        self.fc = nn.Linear(len(filter_sizes) * n_filters, output_dim)
        self.dropout = nn.Dropout(dropout_rate)

    def forward(self, ids):
        embedded = self.dropout(self.embedding(ids))
        embedded = embedded.permute(0,2,1)
        conved = [torch.relu(conv(embedded)) for conv in self.convs]
        pooled = [conv.max(dim=-1).values for conv in conved]
        cated = self.dropout(torch.cat(pooled, dim=-1))
        prediction = self.fc(cated)
        return prediction
```

3) Software usage

The first step to run this code is to install the environment. It is a special notice that the code will require torchtext version to be 0.10.0 for the function of torchtext.vocab.build_vocab_from_iterator to have the correct behavior. Thus, the torch version should be at least 1.9.0 to satisfy the requirement.

After finishing the installing the environment, we can run the code by

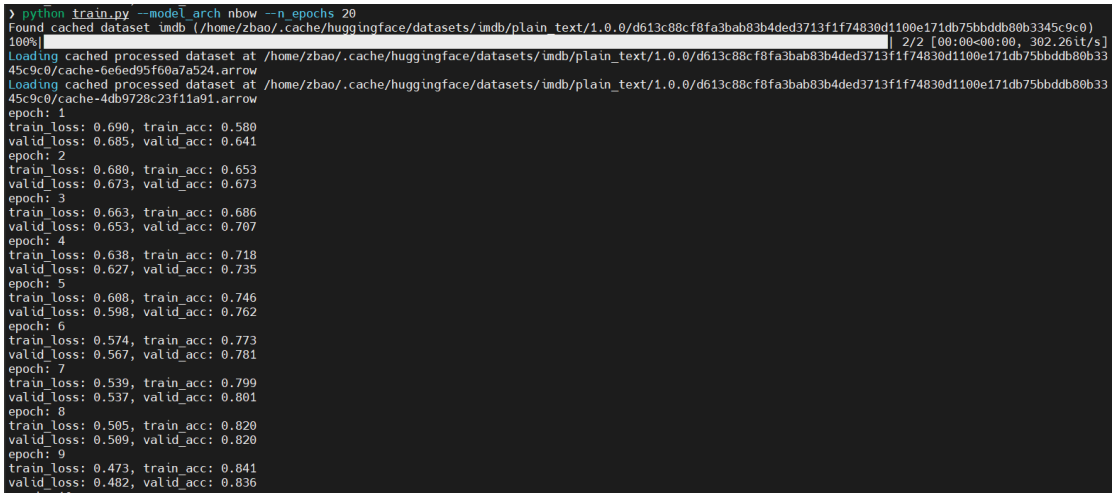
```
python train.py --model_arch nbow --n_epochs 20
```

to use the N-Bag-of-Words framework. If using the CNN framework, we can run

```
python train.py --model_arch cnn --n_epochs 20
```

Or we can specify different number of training epochs to ablate this setting.

After typing in the command in the command line, we could have the data loaded, the pretrained tokenizer / model loaded, and then starts the training process, and it will print out the results during the training process:



4) Contribution of each team member

Since I am the only team member in my project team, all the works of this project are done solely by myself.