Assignment 2 - Recurrent Neural Networks

Programming (Full points: 100)

In this assignment, our goal is to use PyTorch to implement Recurrent Neural Networks (RNN) for sentiment analysis task. Sentiment analysis is to classify sentences (input) into certain sentiments (output labels), which includes positive, negative and neutral.

We will use a benckmark dataset, SST, for this assignment.

- we download the SST dataset from torchtext package, and do some preprocessing to build vocabulary and split the dataset into training/validation/test sets. You don't need to modify the code in this step.

```
!pip install torchtext==0.6.0.
```

```
Requirement already satisfied: torchtext==0.6.0. in /usr/local/lib/python3.10/dist-packages (0.6.0)
Requirement already satisfied: tqdm in /usr/local/lib/python3.10/dist-packages (from torchtext==0.6.0.) (4.66.1)
Requirement already satisfied: requests in /usr/local/lib/python3.10/dist-packages (from torchtext==0.6.0.) (2.31.0)
Requirement already satisfied: torch in /usr/local/lib/python3.10/dist-packages (from torchtext==0.6.0.) (2.0.1+cu118)
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages (from torchtext==0.6.0.) (1.23.5)
Requirement already satisfied: six in /usr/local/lib/python3.10/dist-packages (from torchtext==0.6.0.) (1.16.0)
Requirement already satisfied: sentencepiece in /usr/local/lib/python3.10/dist-packages (from torchtext==0.6.0.) (0.1.99)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.10/dist-packages (from requests->torchtext==0.6.0.) (3
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-packages (from requests->torchtext==0.6.0.) (3.4)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10/dist-packages (from requests->torchtext==0.6.0.) (2.0.6)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/dist-packages (from requests->torchtext==0.6.0.) (2023.7.
Requirement already satisfied: filelock in /usr/local/lib/python3.10/dist-packages (from torch->torchtext==0.6.0.) (3.12.4)
Requirement already satisfied: typing-extensions in /usr/local/lib/python3.10/dist-packages (from torch->torchtext==0.6.0.) (4.5.0)
Requirement already satisfied: sympy in /usr/local/lib/python3.10/dist-packages (from torch->torchtext==0.6.0.) (1.12)
Requirement already satisfied: networkx in /usr/local/lib/python3.10/dist-packages (from torch->torchtext==0.6.0.) (3.1)
Requirement already satisfied: jinja2 in /usr/local/lib/python3.10/dist-packages (from torch->torchtext==0.6.0.) (3.1.2)
Requirement already satisfied: triton==2.0.0 in /usr/local/lib/python3.10/dist-packages (from torch->torchtext==0.6.0.) (2.0.0)
Requirement already satisfied: cmake in /usr/local/lib/python3.10/dist-packages (from triton==2.0.0->torch->torchtext==0.6.0.) (3.27.6)
Requirement already satisfied: lit in /usr/local/lib/python3.10/dist-packages (from triton==2.0.0->torch->torchtext==0.6.0.) (17.0.2)
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.10/dist-packages (from jinja2->torch->torchtext==0.6.0.) (2.1.3
Requirement already satisfied: mpmath>=0.19 in /usr/local/lib/python3.10/dist-packages (from sympy->torch->torchtext==0.6.0.) (1.3.0)
```

```
import copy
import torch
from torch import nn
from torch import optim
import torchtext
from torchtext import data
from torchtext import datasets

TEXT = data.Field(sequential=True, batch_first=True, lower=True)
LABEL = data.LabelField()

# load data splits
train_data, val_data, test_data = datasets.SST.splits(TEXT, LABEL)

# build dictionary
TEXT.build_vocab(train_data)
LABEL.build_vocab(train_data)

# hyperparameters
vocab_size = len(TEXT.vocab)
label_size = len(LABEL.vocab)
padding_idx = TEXT.vocab.stoi['<pad>']
embedding_dim = 128
hidden_dim = 128

# build iterators
train_iter, val_iter, test_iter = data.BucketIterator.splits(
    (train_data, val_data, test_data),
    batch_size=128)
```

- define the training and evaluation function in the cell below.

▾ (25 points)

```
import torch
import torch.nn as nn
import torch.optim as optim

#Train Model Method
def train(model, dataloader, optimizer, criterion, device):
    model.train()
    total_loss = 0.0
    correct = 0
    total = 0

    for batch in dataloader:
        inputs, labels = batch.text.to(device), batch.label.to(device)
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        total_loss += loss.item()
        _, predicted = torch.max(outputs, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

    accuracy = 100 * correct / total
    return total_loss / len(dataloader), accuracy

#Evaluate Model Method
def evaluate(model, dataloader, criterion, device):
    model.eval()
    total_loss = 0.0
    correct = 0
    total = 0

    with torch.no_grad():
        for batch in dataloader:
            inputs, labels = batch.text.to(device), batch.label.to(device)
            outputs = model(inputs)
            loss = criterion(outputs, labels)

            total_loss += loss.item()
            _, predicted = torch.max(outputs, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    accuracy = 100 * correct / total
    return total_loss / len(dataloader), accuracy
```

- build a RNN model for sentiment analysis in the cell below. We have provided several hyperparameters we needed for building the model, including vocabulary size (vocab_size), the word embedding dimension (embedding_dim), the hidden layer dimension (hidden_dim), the number of layers (num_layers) and the number of sentence labels (label_size). Please fill in the missing codes, and implement a RNN model.

▾ (40 points)

```
import torch
import torch.nn as nn

class RNNClassifier(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, label_size, padding_idx,num_layers):
        super(RNNClassifier, self).__init__()
        self.vocab_size = vocab_size
        self.embedding_dim = embedding_dim
        self.hidden_dim = hidden_dim
        self.label_size = label_size
        self.num_layers = num_layers
```

```
        # Embedding Layers
        self.embedding = nn.Embedding(self.vocab_size, self.embedding_dim, padding_idx=padding_idx)

        # Using LSTM
        self.lstm = nn.LSTM(
            input_size=self.embedding_dim,
            hidden_size=self.hidden_dim,
            num_layers=self.num_layers,
            batch_first=True  # Use LSTM, set batch_first=True
        )

        # Adding a linear layer for classification.
        self.fc = nn.Linear(self.hidden_dim, self.label_size)

    def zero_state(self, batch_size):
        #initial hidden state.
        return None

    def forward(self, text):
        #forward function of the model.

        # Embedding layer
        embedded = self.embedding(text)

        # LSTM layer
        lstm_out, _ = self.lstm(embedded)

        # Get the output of the last time step via LSTM
        output = lstm_out[:, -1, :]

        # Linear layer for classification
        output = self.fc(output)

        return output
```

- train the model and compute the accuracy in the cell below.

▾ (20 points)

```
# hyperparameters
vocab_size = len(TEXT.vocab)
label_size = len(LABEL.vocab)
padding_idx = TEXT.vocab.stoi['<pad>']
embedding_dim = 128
hidden_dim = 128
label_size = 3 # Number of output labels
padding_idx = TEXT.vocab.stoi['<pad>']
num_layers = 1
learning_rate = 0.001
num_epochs = 20
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

# Instantiate the model
model = RNNClassifier(vocab_size, embedding_dim, hidden_dim, label_size, padding_idx,num_layers)
model.to(device)  # Move the model to GPU if available

# Define the optimizer and loss function
optimizer = optim.Adam(model.parameters(), lr=learning_rate)
criterion = nn.CrossEntropyLoss()

# Training loop
for epoch in range(num_epochs):
    model.train()  # Set the model to training mode
    total_train_loss = 0.0
    total_train_accuracy = 0.0

    for batch in train_iter:
        inputs, labels = batch.text.to(device), batch.label.to(device)
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
```

```
            total_train_loss += loss.item()
            _, predicted = torch.max(outputs, 1)
            total_train_accuracy += (predicted == labels).sum().item()

        # Calculate average training loss and accuracy for the epoch
        average_train_loss = total_train_loss / len(train_iter)
        average_train_accuracy = 100 * total_train_accuracy / len(train_data)

        # Validation
        model.eval()
        total_val_loss = 0.0
        total_val_accuracy = 0.0

        with torch.no_grad():
            for batch in val_iter:
                inputs, labels = batch.text.to(device), batch.label.to(device)
                outputs = model(inputs)
                val_loss = criterion(outputs, labels)

                total_val_loss += val_loss.item()
                _, predicted = torch.max(outputs, 1)
                total_val_accuracy += (predicted == labels).sum().item()

        # Calculate average validation loss and accuracy for the epoch
        average_val_loss = total_val_loss / len(val_iter)
        average_val_accuracy = 100 * total_val_accuracy / len(val_data)

        # Print training and validation results for the epoch
        print(f"Epoch {epoch + 1}/{num_epochs}")
        print(f"Train Loss: {average_train_loss:.4f}, Train Accuracy: {average_train_accuracy:.2f}%")
        print(f"Validation Loss: {average_val_loss:.4f}, Validation Accuracy: {average_val_accuracy:.2f}%")

    #evaluate on the testing set
    model.eval()  # Set the model to evaluation mode
    total_test_accuracy = 0.0

    with torch.no_grad():
        for batch in test_iter:
            inputs, labels = batch.text.to(device), batch.label.to(device)
            outputs = model(inputs)
            _, predicted = torch.max(outputs, 1)
            total_test_accuracy += (predicted == labels).sum().item()

    # Calculate and print the test accuracy
    test_accuracy = 100 * total_test_accuracy / len(test_data)
    print("")
    print(f"Test Accuracy: {test_accuracy:.2f}%")
```

```
     Train Loss: 1.0462, Train Accuracy: 42.03%
     Validation Loss: 1.0671, Validation Accuracy: 40.15%
     Epoch 3/20
     Train Loss: 1.0448, Train Accuracy: 41.65%
     Validation Loss: 1.1326, Validation Accuracy: 40.69%
     Epoch 4/20
     Train Loss: 1.0347, Train Accuracy: 43.83%
     Validation Loss: 1.0779, Validation Accuracy: 46.32%
     Epoch 5/20
     Train Loss: 1.0088, Train Accuracy: 49.18%
     Validation Loss: 1.0773, Validation Accuracy: 46.87%
     Epoch 6/20
     Train Loss: 0.9508, Train Accuracy: 56.04%
     Validation Loss: 1.0442, Validation Accuracy: 50.32%
     Epoch 7/20
     Train Loss: 0.8715, Train Accuracy: 62.61%
     Validation Loss: 1.0181, Validation Accuracy: 53.59%
     Epoch 8/20
     Train Loss: 0.7761, Train Accuracy: 67.60%
     Validation Loss: 1.0137, Validation Accuracy: 54.86%
     Epoch 9/20
     Train Loss: 0.7006, Train Accuracy: 71.36%
     Validation Loss: 1.0267, Validation Accuracy: 55.86%
     Epoch 10/20
     Train Loss: 0.6219, Train Accuracy: 74.63%
     Validation Loss: 1.0454, Validation Accuracy: 55.40%
     Epoch 11/20
```

```
Validation Loss: 1.1135, Validation Accuracy: 53.59%
Epoch 13/20
Train Loss: 0.4679, Train Accuracy: 83.40%
Validation Loss: 1.1061, Validation Accuracy: 53.50%
Epoch 14/20
Train Loss: 0.4290, Train Accuracy: 85.30%
Validation Loss: 1.1513, Validation Accuracy: 54.41%
Epoch 15/20
Train Loss: 0.3883, Train Accuracy: 87.18%
Validation Loss: 1.1404, Validation Accuracy: 55.68%
Epoch 16/20
Train Loss: 0.3527, Train Accuracy: 89.07%
Validation Loss: 1.1737, Validation Accuracy: 55.95%
Epoch 17/20
Train Loss: 0.3214, Train Accuracy: 90.17%
Validation Loss: 1.1774, Validation Accuracy: 55.40%
Epoch 18/20
Train Loss: 0.2874, Train Accuracy: 91.83%
Validation Loss: 1.2249, Validation Accuracy: 55.86%
Epoch 19/20
Train Loss: 0.2572, Train Accuracy: 92.85%
Validation Loss: 1.2629, Validation Accuracy: 55.77%
Epoch 20/20
Train Loss: 0.2491, Train Accuracy: 93.07%
Validation Loss: 1.3092, Validation Accuracy: 53.68%

Test Accuracy: 57.92%
```

Note: with the Initial RNN and parameters the model is overfitting and we need to increase the Accuracy on Test and Validation sets, The Model is Overfitting

- try to train a model with better accuracy in the cell below. For example, you can use different optimizers such as SGD and Adam. You can also compare different hyperparameters and model size.

▾ (15 points), to obtain FULL point in this problem, the accuracy needs to be higher than 70%

To Increase the accuracy and performance on Test and Validation Dataset. We have made the following changes

1. Remove Stop Words using NLTK Package
2. Used pretrained embeddings like Glove
3. Increased the batch size
4. Used Bidirectional RNN
5. Used a scheduler with Optimization Function
6. Used Dropout and LL1 Regularization

```
# Load the GloVe vectors
TEXT.build_vocab(train_data, vectors="glove.6B.300d", unk_init=torch.Tensor.normal_)
LABEL.build_vocab(train_data)

import copy
import torch
from torch import nn
from torch import optim
import torchtext
from torchtext import data
from torchtext import datasets
import nltk
from nltk.corpus import stopwords

# Download NLTK's stopwords data
nltk.download('stopwords')

TEXT = data.Field(sequential=True, batch_first=True, lower=True)
LABEL = data.LabelField()

# Load data splits
train_data, val_data, test_data = datasets.SST.splits(TEXT, LABEL)

# Build dictionary
TEXT.build_vocab(train_data, vectors="glove.6B.300d", unk_init=torch.Tensor.normal_)
LABEL.build_vocab(train_data)

# Get the English stop words from NLTK
```

```
stop_words = set(stopwords.words('english'))

# Define a function to remove stop words from a list of tokens
def remove_stopwords(tokens):
    return [word for word in tokens if word not in stop_words]

# Apply the stop word removal function during tokenization
TEXT.preprocessing = data.Pipeline(remove_stopwords)

# Build iterators
train_iter, val_iter, test_iter = data.BucketIterator.splits(
    (train_data, val_data, test_data),
    batch_size=128)
```

```
 [nltk_data] Downloading package stopwords to /root/nltk_data...
 [nltk_data]   Package stopwords is already up-to-date!
```

```
import torch
import torch.nn as nn

class BetterRNN(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, label_size, num_layers, dropout_prob):
        super(BetterRNN, self).__init__()  # Note the parentheses after super

        # Embedding Layer
        self.embedding = nn.Embedding(vocab_size, embedding_dim)

        # LSTM Layer
        self.lstm = nn.LSTM(
            input_size=embedding_dim,
            hidden_size=hidden_dim,
            num_layers=num_layers,
            bidirectional=True,
            batch_first=True,
            dropout=dropout_prob           # Dropout for regularization
        )

        # Fully Connected Layer
        self.fc = nn.Linear(2 * num_layers * hidden_dim, label_size)  # Output layer

    def forward(self, input_sentences):
        # Input Embedding
        embedded = self.embedding(input_sentences)

        # LSTM Layer
        output, _ = self.lstm(embedded)

        # Extract the final hidden states from both directions
        h_n = torch.cat((output[:, -1, :hidden_dim], output[:, 0, hidden_dim:]), dim=1)

        # Fully Connected Layer
        logits = self.fc(h_n)

        return logits



#hyperparameters
vocab_size = len(TEXT.vocab)
embedding_dim = 300
hidden_dim = 128
label_size = 3
padding_idx = TEXT.vocab.stoi['<pad>']
num_layers = 8
dropout_prob = 0.4
l1_reg_weight = 0.01
learning_rate = 0.001
num_epochs = 4
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

model = BetterRNN(vocab_size, embedding_dim, hidden_dim, label_size, padding_idx, dropout_prob)
model.to(device)  # Move the model to GPU if available

# Define the optimizer and loss function
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

from torch.optim.lr_scheduler import StepLR
```

```
# Create the StepLR scheduler
step_size = 2
gamma = 0.08
scheduler = StepLR(optimizer, step_size=step_size, gamma=gamma)

criterion = nn.CrossEntropyLoss()

# Training loop
for epoch in range(num_epochs):
    model.train()  # Set the model to training mode
    total_train_loss = 0.0
    total_train_accuracy = 0.0

    for batch in train_iter:
        inputs, labels = batch.text.to(device), batch.label.to(device)
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        total_train_loss += loss.item()
        _, predicted = torch.max(outputs, 1)
        total_train_accuracy += (predicted == labels).sum().item()

    # Calculate average training loss and accuracy for the epoch
    average_train_loss = total_train_loss / len(train_iter)
    average_train_accuracy = 100 * total_train_accuracy / len(train_data)

    # Validation
    model.eval()  # Set the model to evaluation mode
    total_val_loss = 0.0
    total_val_accuracy = 0.0

    with torch.no_grad():
        for batch in val_iter:
            inputs, labels = batch.text.to(device), batch.label.to(device)
            outputs = model(inputs)
            val_loss = criterion(outputs, labels)

            total_val_loss += val_loss.item()
            _, predicted = torch.max(outputs, 1)
            total_val_accuracy += (predicted == labels).sum().item()

    # Calculate average validation loss and accuracy for the epoch
    average_val_loss = total_val_loss / len(val_iter)
    average_val_accuracy = 100 * total_val_accuracy / len(val_data)

    # Print training and validation results for the epoch
    print(f"Epoch {epoch + 1}/{num_epochs}")
    print(f"Train Loss: {average_train_loss:.4f}, Train Accuracy: {average_train_accuracy:.2f}%")
    print(f"Validation Loss: {average_val_loss:.4f}, Validation Accuracy: {average_val_accuracy:.2f}%")

# After training, you cana also evaluate on the test set if needed
model.eval()  # Set the model to evaluation mode
total_test_accuracy = 0.0

with torch.no_grad():
    for batch in test_iter:
        inputs, labels = batch.text.to(device), batch.label.to(device)
        outputs = model(inputs)
        _, predicted = torch.max(outputs, 1)
        total_test_accuracy += (predicted == labels).sum().item()

# Calculate and print the test accuracy
test_accuracy = 100 * total_test_accuracy / len(test_data)
print("")
print(f"Test Accuracy: {test_accuracy:.2f}%")
```

```
/usr/local/lib/python3.10/dist-packages/torch/nn/modules/rnn.py:71: UserWarning: dropout option adds dropout after all but last recurrer
  warnings.warn("dropout option adds dropout after all but last "
Epoch 1/4
Train Loss: 1.0395, Train Accuracy: 45.73%
Validation Loss: 1.0190, Validation Accuracy: 48.05%
Epoch 2/4
Train Loss: 0.9174, Train Accuracy: 58.24%
Validation Loss: 0.9372, Validation Accuracy: 58.04%
```

```
Epoch 3/4
Train Loss: 0.7196, Train Accuracy: 69.85%
Validation Loss: 0.9870, Validation Accuracy: 58.13%
Epoch 4/4
Train Loss: 0.5269, Train Accuracy: 78.87%
Validation Loss: 1.0595, Validation Accuracy: 58.49%

Test Accuracy: 62.40%
```

As we can see the Validation and Test Accuracy got increased, Training Set Accuracy Got Decreased as we are using Dropout and L1 Regularization during Training.