Manually Graded Assignment #2

Revised description with hints

Introduction

In this assignment, you will apply your knowledge of Recurrent Neural Networks (RNN) and PyTorch to build a sentiment analysis model. Sentiment analysis is a common Natural Language Processing (NLP) task where the objective is to classify sentences into different sentiment categories, such as positive, negative, or neutral. This task is widely used in various applications, including social media monitoring, customer feedback analysis, and market research.

To achieve this, you will use the Stanford Sentiment Treebank (SST) dataset, a benchmark dataset in sentiment analysis. The assignment will guide you through downloading and preprocessing the SST dataset using the `torchtext` *library, building a vocabulary, and splitting the dataset into training, validation, and test sets.*

You will then construct an RNN model to perform sentiment analysis. The model will be built using PyTorch, and you will be provided with several key hyperparameters, such as vocabulary size, embedding dimension, and hidden layer dimension. Your task is to complete the implementation of the RNN model, including the embedding layer, the recurrent layer, and the fully connected layer.

After building the model, you will train it on the SST dataset and evaluate its performance on the validation and test sets. You are encouraged to experiment with different optimizers, such as SGD and Adam, and to fine-tune hyperparameters to improve the model's accuracy.

By the end of this assignment, you will have a solid understanding of how to implement RNNs for sentiment analysis and how to optimize their performance using PyTorch. This hands-on experience will be valuable in applying deep learning techniques to various NLP tasks in your future projects.

Step 1: Load and Preprocess the Data

In this step, we will load and preprocess the Stanford Sentiment Treebank (SST) dataset using the `torchtext` *library. This involves several key tasks:*

1. **Import Required Libraries**: *We start by importing necessary libraries, including* `torch` *for deep learning,* `torchtext` *for handling text data, and* `copy` *for handling data copying.*
2. **Define Fields**: *We define two fields:* `TEXT` *and* `LABEL`. *The* `TEXT` *field handles the sentence input, specifying that the data is sequential, should be handled in batches, and should be converted to lowercase. The* `LABEL` *field handles the sentiment labels.*

3. **Load Data Splits**: *Using the* `torchtext.datasets` *module, we load the SST dataset and split it into training, validation, and test sets. This is done using the* `datasets.SST.splits` *method, which takes the* `TEXT` *and* `LABEL` *fields as parameters.*

4. **Build Vocabulary**: *We build the vocabulary for the* `TEXT` *and* `LABEL` *fields using the training data. This step involves creating a mapping of each unique word and label to a corresponding integer index. The* `build_vocab` *method of the* `Field` *class is used for this purpose.*

5. **Define Hyperparameters**: *We define several hyperparameters that will be used in building the RNN model:*

   - `vocab_size`: *The size of the vocabulary, i.e., the number of unique words in the dataset.*
   - `label_size`: *The number of unique sentiment labels.*
   - `padding_idx`: *The index used for padding short sentences.*
   - `embedding_dim`: *The dimension of the word embeddings.*
   - `hidden_dim`: *The dimension of the hidden layer in the RNN.*

6. **Build Iterators**: *We create iterators for the training, validation, and test sets using the* `data.BucketIterator.splits` *method. These iterators will yield batches of data during training and evaluation. The* `batch_size` *parameter specifies the number of samples in each batch.*

By the end of this step, we will have preprocessed the SST dataset, built the necessary vocabulary, and created data iterators to facilitate batch processing during model training and evaluation. This setup is essential for efficiently handling and processing the text data in subsequent steps.

```python
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=32)
```

Step 2: Build an RNN Model for Sentiment Analysis

In this step, we will design and implement a Recurrent Neural Network (RNN) model to classify sentences into sentiment categories such as positive, negative, or neutral. RNNs are particularly well-suited for tasks involving sequential data, such as text, because they can capture temporal dependencies and contextual information within the sequences.

To build our RNN model, we will use the following hyperparameters:

- **vocabulary size (`vocab_size`)**: *The total number of unique words in our dataset.*
- **embedding dimension (`embedding_dim`)**: *The size of the dense vector representations for each word. This allows the model to capture semantic information about the words.*
- **hidden layer dimension (`hidden_dim`)**: *The number of units in the hidden layer of the RNN. This determines the capacity of the model to capture dependencies in the data.*
- **number of layers (`num_layers`)**: *The number of stacked recurrent layers in the model. Multiple layers can help capture more complex patterns in the data.*
- **number of sentence labels (`label_size`)**: *The number of unique sentiment labels in the dataset, which is the output size of the model.*

The key components of the RNN model will include:

1. **Embedding Layer**: *Converts input words into dense vectors of fixed size (`embedding_dim`). This layer helps in capturing semantic information about the words and reduces the dimensionality of the input data.*
2. **Recurrent Layer**: *Processes the embedded word sequences to capture the temporal dependencies and contextual relationships within the sentences. This will be implemented using an RNN, LSTM, or GRU.*
3. **Fully Connected Layer**: *Maps the output from the recurrent layer to the sentiment labels. This layer helps in making the final classification decision.*

You will need to implement the following parts of the model:

- **Initialization of Layers***: Define and initialize the embedding, recurrent, and fully connected layers.*
- **Forward Pass***: Implement the forward function, which specifies how the input data passes through each layer of the model to produce the output.*

By completing this step, you will have a functional RNN model designed for sentiment analysis. This model will then be trained and evaluated on the SST dataset in subsequent steps. The performance of the model can be optimized by experimenting with different hyperparameters and training techniques.

Below is the code provided for this step. You need to complete this code.

```
class RNNClassifier(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, label_size,
padding_idx):
        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 = 1

        # add the layers required for sentiment analysis.
        self.embedding = nn.Embedding(self.vocab_size, self.embedding_dim,
padding_idx=padding_idx)

    def zero_state(self, batch_size):
        # implement the function, which returns an initial hidden state.
        return None

    def forward(self, text):
        # implement the forward function of the model.
        embedding = self.embedding(text)
        return None
```

Step 3: Train the RNN Model

In this step, we will train the RNN model using the Stanford Sentiment Treebank (SST) dataset. Training the model involves feeding the data into the network, computing the loss, performing backpropagation to calculate gradients, and updating the model's weights to minimize the loss. Here's what needs to be done:

1. **Initialize the Model, Optimizer, and Loss Function***:*
    - *Instantiate the* `RNNClassifier` *model with the defined hyperparameters.*
    - *Choose an optimizer to update the model's weights, such as SGD or Adam.*

  - *Define the loss function to measure how well the model's predictions match the actual labels. For a classification task,* `CrossEntropyLoss` *is typically used.*

2. **Training Loop***:*

   - *Iterate over multiple epochs to allow the model to learn from the entire dataset multiple times.*
   - *For each batch of data, perform the following:*
     - *Zero the gradients to prevent accumulation from previous batches.*
     - *Perform a forward pass through the model to obtain predictions.*
     - *Calculate the loss between the predictions and the actual labels.*
     - *Perform a backward pass to compute the gradients.*
     - *Update the model's parameters using the optimizer.*
   - *Track the training loss and accuracy to monitor the model's performance over time.*

3. **Validation***:*

   - *After each epoch, evaluate the model on the validation set to check its performance on unseen data.*
   - *Track the validation loss and accuracy to ensure the model is not overfitting.*

By the end of this step, you should have a trained RNN model that has learned to classify sentences based on

sentiment. The performance of the model will be measured in terms of accuracy and loss on both the training and validation sets.

There is no code provided for this step. You need to write the code.

Step 4: Optimize Hyperparameters

In this step, we will focus on optimizing the hyperparameters of the RNN model to achieve better accuracy. Hyperparameters significantly impact the performance and efficiency of the model, so tuning them is essential. Here's what needs to be done:

1. **Experiment with Different Optimizers***:*

   - *Compare different optimization algorithms such as SGD (Stochastic Gradient Descent) and Adam (Adaptive Moment Estimation).*
   - *Assess the impact of each optimizer on the convergence speed and final accuracy of the model.*

2. **Adjust Learning Rate***:*

   - *Experiment with different learning rates for the chosen optimizer.*

   - *A too-high learning rate might cause the model to converge quickly to a suboptimal solution, while a too-low learning rate might make the training process unnecessarily slow.*

3. **Vary Batch Size***:*

   - *Try different batch sizes to see how they affect the training dynamics and model performance.*

   - *Larger batch sizes can lead to more stable gradient estimates but require more memory.*

4. **Modify Model Architecture***:*

   - *Experiment with different numbers of hidden units in the RNN layer to find the right balance between model capacity and computational efficiency.*

   - *Try stacking multiple RNN layers to capture more complex patterns in the data.*

5. **Incorporate Regularization Techniques***:*

   - *Use dropout layers to prevent overfitting by randomly setting a fraction of the input units to zero at each update during training.*

   - *Adjust the dropout rate to find the optimal value that reduces overfitting without significantly hindering training.*

6. **LR Scheduler***:*

   - *Implement a learning rate scheduler to gradually decrease the learning rate with increasing epochs, which can help achieve better convergence. More details can be found in the PyTorch documentation.*

7. **Saving the Best Model***:*

   - *Write code to save the model at the epoch with the highest validation accuracy to ensure that you retain the best-performing model.*

8. **Trying New Models***:*

   - *Explore different models, such as LSTM or GRU, to replace the RNN model. You can find details on recurrent layers in the PyTorch documentation.*

By systematically experimenting with and tuning these hyperparameters, you can optimize the model's performance, resulting in higher accuracy and better generalization to new, unseen data.