Hands-on Coding Session

Minor in AI - IIT ROPAR

1st April, 2025

Introduction

In this document, we will break down a deep learning code used for Natural Language Processing (NLP). The code primarily focuses on building a Recurrent Neural Network (RNN), including LSTMs and GRUs, to predict the next word in a sequence, generate Word2Vec embeddings.

Deep RNN for Next-Word Prediction

We start by defining a basic Deep Recurrent Neural Network (RNN) model for predicting the next word in a sequence. The core of the model consists of three main components:

- **Embedding Layer:** Converts word indices into dense vector representations (embeddings).
- **RNN Layer:** Processes the word embeddings sequentially and generates hidden states for each time step.
- **Fully Connected Layer:** Maps the final hidden state to the vocabulary space, predicting the next word.

Defining the Deep RNN Class

```python
class DeepRnn(nn.Module):
    def __init__(self, vocab_size, embed_dim, hidden_dim, num_layers):
        super(DeepRnn, self).__init__()
        self.embeddings = nn.Embedding(vocab_size, embed_dim)  # Embedding layer
        self.rnn = nn.RNN(embed_dim, hidden_dim, num_layers, batch_first=True)  # RNN
            layer
        self.fc = nn.Linear(hidden_dim, vocab_size)  # Fully connected layer for
            prediction
```

The `__init__` method initializes the model and defines the following layers:

- `nn.Embedding(vocab_size, embed_dim)`: This embedding layer transforms word indices (from the vocabulary) into dense vectors (word embeddings) of size `embed_dim`.
- `nn.RNN(embed_dim, hidden_dim, num_layers, batch_first=True)`: The RNN layer processes the input sequence of word embeddings, capturing temporal dependencies. It outputs hidden states at each time step. The number of layers (`num_layers`) and the size of the hidden state (`hidden_dim`) are configurable.
- `nn.Linear(hidden_dim, vocab_size)`: A fully connected layer that projects the hidden state from the RNN into a vector of size `vocab_size`, representing the logits for each word in the vocabulary. This layer is used for prediction.

Forward Pass

```python
def forward(self, x):
    embeddings = self.embeddings(x)  # Embedding layer
    output, hidden = self.rnn(embeddings)  # RNN layer
    output = self.fc(output[:, -1, :])  # Fully connected layer
    return output
```

- `self.embeddings(x)`: The input word indices are passed through the embedding layer, transforming them into dense word vectors (embeddings).
- `self.rnn(embeddings)`: The word embeddings are processed through the RNN layer, generating hidden states for each time step in the sequence.
- `self.fc(output[:, -1, :])`: The last hidden state of the RNN (after processing the entire sequence) is passed through the fully connected layer to make a prediction. We use only the last hidden state (`output[:, -1, :]`) as it contains information about the entire input sequence.

Training Setup

Hyperparameters

```
EMBED_DIM = 64
HIDDEN_DIM = 128
NUM_LAYERS = 3
VOCAB_SIZE = len(vocab)
```

- **'EMBED_DIM'** sets the dimension of each word embedding.
- **'HIDDEN_DIM'** specifies the size of the hidden state in the RNN.
- **'NUM_LAYERS'** defines how many RNN layers are stacked on top of each other.
- **'VOCAB_SIZE'** is the total number of unique words in the vocabulary.

Device Setup

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = DeepRnn(VOCAB_SIZE, EMBED_DIM, HIDDEN_DIM, NUM_LAYERS).to(device)
```

This code checks if a GPU is available, otherwise it falls back to the CPU.

DataLoader Setup

```
from torch.utils.data import TensorDataset, DataLoader
train_dataset = TensorDataset(torch.tensor(inputs), torch.tensor(targets))
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
```

- **'TensorDataset'** is used to pair the inputs and targets into a dataset object.
- **'DataLoader'** wraps the dataset and enables mini-batch processing with shuffling for better generalization.

Loss Function and Optimizer

```
criterion = nn.CrossEntropyLoss()  # For multi-class classification
optimizer = optim.Adam(model.parameters(), lr=0.0001)  # Adam optimizer
```

- **'CrossEntropyLoss'** is commonly used for multi-class classification problems, such as predicting the next word in the vocabulary.
- **'Adam'** is an adaptive optimizer that uses learning rate adjustments to efficiently update model parameters.

Training Loop

The following function defines how the model is trained over multiple epochs:

```python
def train(model, train_loader, criterion, optimizer, epochs):
    model.train()
    loss_values = []
    for epoch in range(epochs):
        total_loss = 0
        for batch_inputs, batch_targets in train_loader:
            batch_inputs, batch_targets = batch_inputs.to(device), batch_targets.to(
                device)
            optimizer.zero_grad()  # Zero gradients from previous step
            outputs = model(batch_inputs)  # Forward pass
            loss = criterion(outputs, batch_targets)  # Compute loss
            loss.backward()  # Backpropagate gradients
            optimizer.step()  # Update parameters
            total_loss += loss.item()  # Track total loss
        avg_loss = total_loss / len(train_loader)
        loss_values.append(avg_loss)
        print(f"Epoch␣[{epoch␣+␣1}/{epochs}],␣Loss:␣{avg_loss:.4f}")
    return loss_values
```

This function trains a model using a given dataset, loss function, and optimizer over multiple epochs. Here's what it does step-by-step:

- **Model in Training Mode**: `model.train()` sets the model to training mode.
- **Epoch Loop**: For each epoch:
  - It iterates through batches of data (`train_loader`).
  - **Data to Device**: Moves the input data and target labels to the correct device (CPU/GPU).
  - **Zero Gradients**: Clears gradients from the previous step (`optimizer.zero_grad()`).
  - **Forward Pass**: Passes input through the model to get predictions.
  - **Loss Calculation**: Compares predictions with actual labels and computes the loss.
  - **Backpropagation**: Computes gradients (`loss.backward()`).
  - **Optimizer Step**: Updates model parameters using gradients (`optimizer.step()`).
  - Tracks and accumulates the total loss.
- **Average Loss**: At the end of each epoch, calculates and prints the average loss for that epoch.
- **Return Loss**: Returns the list of average loss values over all epochs.

Next-Word Prediction Function

Once the model is trained, we can use it for word prediction.

```python
def predict_next_word(model, input_text, seq_length):
    model.eval()  # Set model to evaluation mode
    words = pre_process_data(input_text)  # Preprocess input text
    tokens = [word_to_index[word] if word in word_to_index else 0 for word in words]
    tokens = tokens[-seq_length:]  # Use only the last seq_length tokens
    tokens = torch.tensor(tokens).unsqueeze(0).to(device)

    with torch.no_grad():  # Turn off gradient tracking
        output = model(tokens)  # Get model prediction
        predicted_index = torch.argmax(output, dim=1).item()  # Get predicted word index

    return index_to_word[predicted_index]  # Return predicted word
```

The `predict_next_word` function predicts the next word in a sequence given some input text. Here's what it does step-by-step:

- **Set Model to Evaluation Mode**: `model.eval()` sets the model to evaluation mode, disabling layers like dropout that are only used during training.

- **Preprocess Input**: `pre_process_data(input_text)` is used to preprocess the input text (e.g., tokenizing, cleaning).
- **Convert Words to Indices**: Each word in the preprocessed text is converted to its corresponding index in the vocabulary. If a word is not found in the vocabulary, it is replaced with the index for the `<unk>` (unknown) token.
- **Use Last `seq_length` Tokens**: Only the last `seq_length` tokens are retained from the input sequence.
- **Move to Device**: The tokens are converted to a PyTorch tensor and moved to the correct device (CPU/GPU).
- **Turn Off Gradient Tracking**: `with torch.no_grad()` is used to disable gradient tracking, as we are only performing inference and don't need gradients.
- **Forward Pass**: The model generates a prediction for the next word based on the input sequence.
- **Get Predicted Word Index**: The predicted word index is found by applying `torch.argmax(output, dim=1)` to the model's output, which gives the index of the word with the highest probability.
- **Return Predicted Word**: The index of the predicted word is mapped back to the word using `index_to_word`, and the predicted word is returned.

Advanced Models: LSTM, GRU, Bi-directional RNN

We can extend our basic RNN model to include **LSTMs**, **GRUs**, and **Bi-directional RNNs**.

```python
class Text_prediction(nn.Module):
    def __init__(self, vocab_size, embed_dim, hidden_dim, num_layers, model_type="RNNs",
         bidirectional=False):
        super(Text_prediction, self).__init__()
        self.embeddings = nn.Embedding(vocab_size, embed_dim)

        if model_type == "RNNs":
            self.rnn = nn.RNN(embed_dim, hidden_dim, num_layers, batch_first=True,
                bidirectional=bidirectional)
        elif model_type == "LSTMs":
            self.rnn = nn.LSTM(embed_dim, hidden_dim, num_layers, batch_first=True,
                bidirectional=bidirectional)
        elif model_type == "GRUs":
            self.rnn = nn.GRU(embed_dim, hidden_dim, num_layers, batch_first=True,
                bidirectional=bidirectional)

        if bidirectional:
            self.fc = nn.Linear(hidden_dim * 2, vocab_size)  # Double hidden_dim for
                bidirectional
        else:
            self.fc = nn.Linear(hidden_dim, vocab_size)
```

The `Text_prediction` class defines a model for text prediction that can use different types of RNN architectures (RNN, LSTM, or GRU). It includes an embedding layer, an RNN (or its variants), and a fully connected layer to predict the next word in the sequence. Here's a breakdown of the class:

- **Initialization**:
  - `self.embeddings = nn.Embedding(vocab_size, embed_dim)`: This embedding layer transforms word indices into dense vectors of size `embed_dim`.
  - **RNN Type Selection**: The model can use either an `RNN`, `LSTM`, or `GRU` layer. This is controlled by the `model_type` parameter.
  - `self.rnn`: The RNN layer is defined based on the selected `model_type`:
    - If `model_type` is `RNNs`, an RNN layer is used.
    - If `model_type` is `LSTMs`, an LSTM layer is used.
    - If `model_type` is `GRUs`, a GRU layer is used.

  - `bidirectional`: If set to `True`, the RNN processes the sequence in both forward and backward directions, doubling the size of the hidden state.
  - `self.fc`: This fully connected layer takes the output of the RNN and projects it to the vocabulary size (`vocab_size`).
    * If `bidirectional=True`, the size of the hidden state is doubled, so the input to the fully connected layer is `hidden_dim * 2`.
    * If `bidirectional=False`, the size of the hidden state is `hidden_dim`.

Word2Vec Model for Embedding Learning

```python
class Word2Vec(nn.Module):
    def __init__(self, vocab_size, embed_dim):
        super(Word2Vec, self).__init__()
        self.embeddings = nn.Embedding(vocab_size, embed_dim)  # Word embeddings
        self.output_layer = nn.Linear(embed_dim, vocab_size)  # Output layer for context
             prediction

    def forward(self, x):
        x = self.embeddings(x)  # Convert word indices to embeddings
        x = self.output_layer(x)  # Predict context words
        return x
```

The `Word2Vec` class is a simple implementation of a word embedding model, commonly used for learning word representations in natural language processing tasks. The model is built using PyTorch and has two main layers: an embedding layer and a linear output layer. Here's a breakdown of the class:

- **Initialization**:
  - `self.embeddings = nn.Embedding(vocab_size, embed_dim)`: This is the embedding layer, which converts word indices into dense vectors (word embeddings). The size of the embedding vectors is defined by `embed_dim`.
  - `self.output_layer = nn.Linear(embed_dim, vocab_size)`: This is the output layer, which predicts context words for each input word. It maps the embedding vector to a vector of size `vocab_size`, representing the likelihood of each word in the vocabulary being a context word for the input word.
- **Forward Pass**:
  - `x = self.embeddings(x)`: The input word indices are passed through the embedding layer, converting them into word embeddings.
  - `x = self.output_layer(x)`: The resulting embeddings are passed through the output layer, which predicts the context words (or context word probabilities) for each input word.
  - `return x`: The model returns the predicted context word probabilities (logits).

This model learns to predict context words based on an input word, and the word embeddings are learned during training by minimizing a suitable loss function (typically cross-entropy).

Sentiment Analysis using PyTorch

In this section, we demonstrate how to perform sentiment analysis on a dataset of cyberbullying-related tweets using deep learning techniques in PyTorch. This includes:

- Text preprocessing.
- Vocabulary creation.
- Data loading.
- Model creation using RNN, LSTM, or GRU.
- Training and evaluation.

Loading and Preprocessing Data

The first step is to load the dataset, which is assumed to be in CSV format. In this case, we have a dataset of cyberbullying tweets stored in the file `cyberbullying_tweets.csv`.

```
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import random
import matplotlib.pyplot as plt
import re
from collections import Counter
import pandas as pd

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")
file_path = "cyberbullying_tweets.csv"
df = pd.read_csv(file_path)
df.head()
print(df["cyberbullying_type"].unique())
```

This code loads the CSV file into a Pandas DataFrame and prints the first few rows of the data. It also prints the unique values from the `cyberbullying_type` column, which represents the different classes (labels) in our dataset.

Cleaning Tweets

We need to preprocess the text data before passing it to our model. This includes:

- Converting the text to lowercase.
- Removing URLs and special characters.
- Tokenizing the text into words.

The following function performs these tasks:

```
def clean_tweets(text):
    text = text.lower()  # Convert to lowercase
    text = re.sub(r"http\S+|www\S+|https\S+","",text,flags = re.MULTILINE)  # Remove
        URLs
    text = re.sub(r"\@\w+|\#","",text)  # Remove mentions and hashtags
    words = text.split()  # Tokenize by splitting on spaces
    return words

df["tweet_text"] = df["tweet_text"].apply(clean_tweets)
df.head()
```

Building Vocabulary

Once the tweets are cleaned, we build the vocabulary by counting the occurrences of each word in the entire dataset. We use the `Counter` class from the `collections` module for this task:

```
words_counts = Counter(word for sentence in df["tweet_text"] for word in sentence)
vocab = {word:i+2  for i,word in enumerate(words_counts.keys())}  # Assign indices to
    each word
vocab["<pad>"] = 0  # Padding token
vocab["<unk>"] = 1  # Unknown token
```

Here, we create a vocabulary dictionary, where each word is mapped to a unique index. We add special tokens `<pad>` and `<unk>` to handle padding and unknown words.

Converting Words to Indices

Next, we convert the words in each tweet to their corresponding indices from the vocabulary:

```
def text_to_indices(text):
    return [vocab.get(word,vocab["<unk>"]) for word in text]  # Replace words with their
         indices

df["text_indices"] = df["tweet_text"].apply(text_to_indices)
```

Label Encoding

The target labels (cyberbullying categories) need to be converted into numerical format for classification. We create a mapping from the label to a numeric value:

```
label_mapping = {label:i for i,label in enumerate(df["cyberbullying_type"].unique())}
df["label"] = df["cyberbullying_type"].map(label_mapping)
```

Train-Test Split

We split the dataset into training and testing sets, with 80% of the data used for training and 20% for testing:

```
train_size = int(0.8*len(df))
train_df = df[:train_size]
test_df = df[train_size:]
```

Creating the Dataset and DataLoader

In PyTorch, we use the `Dataset` and `DataLoader` classes to handle batching and shuffling of the data.

```
from torch.utils.data import Dataset, DataLoader

class TextDataset(Dataset):
    def __init__(self, texts, labels, seq_length=100):
        self.texts = texts
        self.labels = labels
        self.seq_length = seq_length

    def __len__(self):
        return len(self.texts)

    def __getitem__(self, index):
        text = self.texts[index]
        label = self.labels[index]

        # Pad or truncate the sequence to the fixed length
        if len(text) > self.seq_length:
            text = text[:self.seq_length]
        else:
            text += [vocab["<pad>"]] * (self.seq_length - len(text))

        return torch.tensor(text, dtype=torch.long), torch.tensor(label, dtype=torch.
            long)

train_dataset = TextDataset(train_df["text_indices"].values, train_df["label"].values)
test_dataset = TextDataset(test_df["text_indices"].values, test_df["label"].values)

train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)
```

This code creates a custom dataset and data loader for handling text data. Here's a breakdown of each part:

- **'TextDataset' Class**: This class is a custom dataset that processes the input text and labels.
  - **'__init__(self, texts, labels, seq_length=100)'**: The constructor initializes the dataset with the text data, labels, and sequence length.
  - **'self.texts = texts'**: This stores the input text data (in tokenized form) in the dataset object.
  - **'self.labels = labels'**: This stores the labels for the corresponding text data.
  - **'self.seq_length = seq_length'**: This defines the fixed length of the input sequences. If sequences are shorter than this length, they will be padded; if they are longer, they will be truncated.
- **'__len__(self)' Method**: This method returns the number of samples (texts) in the dataset.
  - **'return len(self.texts)'**: It simply returns the length of the `texts` list, which tells how many samples are in the dataset.
- **'__getitem__(self, index)' Method**: This method retrieves a specific sample (text and label) by index.
  - **'text = self.texts[index]'**: It retrieves the text (word indices) for the sample at the specified index.
  - **'label = self.labels[index]'**: It retrieves the corresponding label for that sample.
  - **Padding or Truncating**:
    - If the sequence is longer than the defined `seq_length`, it is truncated to the `seq_length`.
    - If the sequence is shorter, it is padded with the padding token `<pad>` until the sequence reaches `seq_length`.
  - **'return torch.tensor(text, dtype=torch.long), torch.tensor(label, dtype=torch.long)'**: The text (word indices) and the label are returned as PyTorch tensors. The tensor type is `torch.long`, as the indices are integers.
- **'train_dataset' and 'test_dataset'**: These are instances of the `TextDataset` class, created from the training and testing data.
  - **'train_dataset = TextDataset(train_df["text_indices"].values, train_df["label_values"])'**: This creates a dataset for the training data, using the tokenized text indices and the corresponding labels from the `train_df` DataFrame.
  - **'test_dataset = TextDataset(test_df["text_indices"].values, test_df["label_values"])'**: Similarly, this creates a dataset for the testing data from the `test_df` DataFrame.
- **'DataLoader'**: This class is used to load data in batches. It can also shuffle the data and load it in parallel.
  - **'train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)'**: This creates a data loader for the training dataset. It will load the data in batches of 32 and shuffle the data for better generalization during training.
  - **'test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)'**: This creates a data loader for the test dataset. The test data is not shuffled because we just need to evaluate the model on the entire dataset.

Model Creation

The model for this task is a simple RNN, LSTM, or GRU-based classifier. It takes in sequences of word indices and classifies them into one of the cyberbullying categories. Here's the model architecture:

```python
class Text_classifier(nn.Module):
    def __init__(self, vocab_size, embed_dim, hidden_dim, num_layers, number_classes,
        model_type="RNNs", bidirectional=False):
        super(Text_classifier, self).__init__()
        self.embeddings = nn.Embedding(vocab_size, embed_dim)

        if model_type == "RNNs":
            self.rnn = nn.RNN(embed_dim, hidden_dim, num_layers, batch_first=True,
                bidirectional=bidirectional)
        elif model_type == "LSTMs":
            self.rnn = nn.LSTM(embed_dim, hidden_dim, num_layers, batch_first=True,
                bidirectional=bidirectional)
        elif model_type == "GRUs":
            self.rnn = nn.GRU(embed_dim, hidden_dim, num_layers, batch_first=True,
                bidirectional=bidirectional)

        if bidirectional:
            self.fc = nn.Linear(hidden_dim * 2, number_classes)
        else:
            self.fc = nn.Linear(hidden_dim, number_classes)

    def forward(self, x):
        embeddings = self.embeddings(x)
        output, hidden = self.rnn(embeddings)
        output = self.fc(output[:, -1, :])  # Take the last hidden state for
            classification
        return output
```

The model consists of:

- An embedding layer.
- An RNN, LSTM, or GRU layer (depending on the model type).
- A fully connected layer that produces the final class predictions.

Training the Model

The training loop is defined as follows:

```python
def train(model, train_loader, criterion, optimizer, epochs):
    model.train()
    loss_values = []

    for epoch in range(epochs):
        total_loss = 0
        for batch_inputs, batch_targets in train_loader:
            batch_inputs, batch_targets = batch_inputs.to(device), batch_targets.to(
                device)

            optimizer.zero_grad()
            outputs = model(batch_inputs)

            loss = criterion(outputs, batch_targets)
            loss.backward()
            optimizer.step()

            total_loss += loss.item()

        avg_loss = total_loss / len(train_loader)
        loss_values.append(avg_loss)
        print(f"Epoch [{epoch+1}/{epochs}], Loss: {avg_loss:.4f}")

    return loss_values
```

The `train` function trains a model for a specified number of epochs. Here's a brief breakdown of its steps:

- **Model in Training Mode**: `model.train()` sets the model to training mode.
- **Epoch Loop**: For each epoch, the function:
  - Iterates through batches of data from `train_loader`.
  - Moves the data and labels to the correct device (CPU/GPU).
  - Computes predictions using the model.
  - Calculates the loss using the `criterion`.
  - Performs backpropagation and updates model parameters using `optimizer`.
- **Average Loss**: After each epoch, the average loss is computed and printed.
- **Return Loss**: The function returns the list of average losses for each epoch.

Key Takeaways

The session delves into the construction and training of deep learning models, particularly in the domain of natural language processing (NLP). The following points summarize the essential concepts and methods discussed:

- **Deep Recurrent Neural Network (RNN) for Next-Word Prediction**: The foundation of the approach is a Deep Recurrent Neural Network (RNN) designed for next-word prediction. The model consists of three primary components: an embedding layer, a recurrent layer (which can be an RNN, LSTM, or GRU), and a fully connected layer that produces the final prediction.
- **Forward Pass of the Model**: The forward pass of the model entails passing the input data through the embedding layer, followed by the recurrent layer, and finally through the fully connected layer to generate the predicted output.
- **Hyperparameter Tuning**: The model's performance is influenced by hyperparameters such as the embedding dimension, hidden state size, and the number of layers within the recurrent network. These parameters are crucial for determining the model's complexity and capacity to learn temporal dependencies.
- **Device Management**: A key aspect of model deployment involves ensuring compatibility with available hardware, whether CPU or GPU. This is facilitated by PyTorch's device management capabilities, which optimize the computational process.
- **Text Preprocessing**: Text preprocessing is an essential step in NLP tasks. This includes cleaning the data, tokenizing it into words, and padding sequences to ensure uniform length, enabling batch processing.
- **Data Handling with DataLoader**: Efficient data handling is achieved using PyTorch's DataLoader, which allows for automatic batching, shuffling, and parallel data loading, ensuring that the model can be trained efficiently on large datasets.
- **Optimizer and Loss Function**: The Adam optimizer is employed in conjunction with cross-entropy loss, both of which are standard for classification tasks. These components facilitate the optimization of model parameters through backpropagation.
- **Training the Model**: Training is conducted over multiple epochs, with performance being monitored through the calculation of average loss, which provides insight into the model's learning progress.
- **Word2Vec for Embedding Learning**: Word2Vec is introduced as a method for learning word embeddings, where each word is represented by a dense vector. This model predicts context words based on a given target word, facilitating the capture of semantic relationships between words.
- **Application to Text Classification**: The techniques covered are applicable to various text classification tasks, such as sentiment analysis, where the model's architecture can be adapted to classify text into predefined categories based on learned representations.