Inspiring Excellence

Course Title: Neural Networks
Course Code: CSE425
Assignment: Implementing a Vanilla RNN and Analyzing its Limitations

Submitted by-
Name: Tasnia Juha
ID: 20101486
Section: 02

Implementing a Vanilla RNN and Analyzing its Limitations

Introduction: The output labels of feedforward models, including Multilayer Perceptron Neural Networks (MLP) and Convolutional Networks (CNN), are mapped from fixed-size input data (vectors of fixed dimensions). They are quite effective and have been applied to many different activities. Both have shown themselves to be especially effective at solving classification or regression prediction issues, and CNNs have effectively taken over as the preferred technique for solving any prediction problem that uses input picture data. However, a large amount of data is available as **sequences** rather than fixed-size vectors. RNNs (Recurrent Neural Networks) were created to deal with **sequential data**. The network is trained to store relevant previous inputs, which feed into future predictions, using a type of internal memory.

The main goal of this assignment was to use Vanilla Recurrent Neural Networks to build a language model that would output text that was coherent and relevant to its settings based on the input content. Because it naturally analyzes sequential input, we've decided to implement the model. We want to train our RNN on a large and diverse dataset to give it the ability to understand complex patterns and structures present in human language. This will allow our model to generate content that not only flows logically but also closely matches the semantic context.

Data Preparation: The dataset chosen for this project is "Reddit Comments scores - NLP" sourced from Kaggle, presents a substantial and diverse collection of over 4 million Reddit comments. Within this corpus, a balanced distribution of 2 million highly downvoted comments and 2 million highly upvoted comments provides a unique opportunity for training a robust language model capable of contextually relevant text generation.
The selection of such a large corpus is motivated by the inherent advantage of capturing an extensive range of linguistic patterns, expressions, and contexts present in real-world communication. In the realm of language model training and text generation, a larger dataset enhances the model's ability to learn intricate dependencies and nuanced relationships within language. Moreover, the diverse array of upvoted and downvoted comments introduces a rich spectrum of writing styles, sentiment, and content, further contributing to the model's capacity to generate coherent and contextually appropriate text.

By utilizing this extensive dataset, the resultant language model is poised to encapsulate the intricacies of language usage, effectively adapting to various prompts and generating meaningful text responses. The expansive nature of the dataset ensures that the model attains a comprehensive grasp of linguistic diversity, enabling it to craft contextually relevant and coherent text outputs, making it well-suited for the objectives of our project.

Preprocessing the Dataset: The dataset includes a 'text' and a 'parent_text' column, which are the key focus of this work. The dataset was examined for dimensions and preprocessed by checking for and deleting null values if any were found by importing relevant modules and libraries such as numpy, pandas, nltk, re etc. More crucially, the dataset(df) was tokenized into fewer characters to minimize complexity and improve model efficiency. The duplicate rows were also removed before tokenizing.

Mounting from drive:

```
from google.colab import drive
drive.mount('/content/drive')
```

Importing necessary libraries:

```
import nltk
nltk.download('punkt')
nltk.download('omw-1.4')
nltk.download('punkt')
nltk.download("stopwords")
nltk.download('wordnet')
import pandas as pd
import numpy as np
!pip install beautifulsoup4
!pip install lxml
from bs4 import BeautifulSoup
import unicodedata
import re
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer
```

Loading Dataset:

```
df= pd.read_csv('/content/drive/MyDrive/425/comments_negative.csv')
```

Removing Duplicate Rows:

```
duplicate_rows = df[df.duplicated()]
duplicate_counts = duplicate_rows.value_counts()
print("Duplicate Rows:")
print(duplicate_counts)
df = df.drop_duplicates('text')
df = df.drop_duplicates('parent_text')
```

Preprocessing and Tokenization:

```
!pip install beautifulsoup4
!pip install lxml
from bs4 import BeautifulSoup
import unicodedata
import re
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer

# Define a function to perform text normalization
def normalize_text(a):
    a = str(a).lower()
    a=removeTags(a)
    a = BeautifulSoup(a, "html.parser").text

    # Remove non-alphabetic characters
    a = re.sub(r'[^a-zA-Z]', ' ', a)
    a=re.sub(r"\s+[a-zA-Z]\s+", ' ', a)
    a= re.sub(r'\s+', ' ', a)

    # Tokenize text into words
    words = nltk.word_tokenize(a)

    # Remove stop words
    stop_words = set(stopwords.words('english'))
    words = [word for word in words if word not in stop_words]

    # Lemmatize words
    lemmatizer = WordNetLemmatizer()
    words = [lemmatizer.lemmatize(word) for word in words]
```

```
    # Join words back into a normalized sentence
    normalized_text = ' '.join(words)

    return normalized_text
tag = re.compile(r'<[*%>]+>')

def removeTags(b):
 return tag.sub('', b)

# Apply the normalization function to the 'text' column
df['text'] = df['text'].apply(normalize_text)
df['parent_text'] = df['parent_text'].apply(normalize_text)
```

Here I used libraries like BeautifulSoup, regular expressions, and NLTK to normalize text data. It converts text to lowercase, removes HTML tags, non-alphabetic characters, and extra whitespace. Tokenization breaks text into words, stop words are removed, and lemmatization reduces words to their root forms. Finally, the processed words are rejoined into normalized sentences, enhancing the quality and consistency of the text data in the ‘text’ and ‘parent_text’ columns of the DataFrame 'df'.

Separately tokenizing:

```
# Tokenization function
def tokenize(text):
    return text.split()

# Apply tokenization to the 'text' and 'parent_text' columns
df['text'] = df['text'].apply(tokenize)
df['parent_text'] = df['parent_text'].apply(tokenize)
```

Here, a tokenization function named tokenize is defined that splits a given input text into individual words. This function is applied to the 'text' and 'parent_text' columns of the DataFrame 'df'. By tokenizing the text data, each comment is broken down into a sequence of words, enabling further processing and analysis at the word level. This step is crucial for preparing the text data for subsequent natural language processing tasks, such as building a vocabulary, converting text to sequences of integers, and training language models.

Now for the splitting part, a train-validation split of 3:7, 4:6, and 2:8 was performed, and an average was calculated afterwards.

```
# Split ratios
split_ratios = [0.3, 0.4, 0.2]

# Calculate the sizes of each split
num_examples = len(df)
num_train_samples = int(num_examples * split_ratios[0])
num_val_samples = int(num_examples * split_ratios[1])
num_test_samples = num_examples - num_train_samples - num_val_samples

# Split the dataset
train_dataset = df[:num_train_samples]
val_dataset = df[num_train_samples:num_train_samples + num_val_samples]
test_dataset = df[num_train_samples + num_val_samples:]

# Calculate average train-validation split ratio
average_train_val_ratio = (num_train_samples / (num_train_samples +
num_val_samples))

# Print average train-validation split ratio
print(f"Average Train-Validation Split: {average_train_val_ratio:.2f}")
```

The output came: Average Train-Validation Split: 0.43

Building a Vocabulary:

```
# Building a Vocabulary
# Combine 'text' and 'parent_text' columns to build the vocabulary
combined_text = df['text'].tolist() + df['parent_text'].tolist()

# Create a Tokenizer and fit on the combined text
tokenizer = tokenize()
tokenizer.fit_on_texts(combined_text)

# Vocabulary size
vocab_size = len(tokenizer.word_index) + 1
```

```
# Print vocabulary size
print(f"Vocabulary size: {vocab_size}")
```

The code merges 'text' and 'parent_text', creating a combined text for vocabulary building. It uses a Tokenizer to process the combined text and create a vocabulary. The vocabulary's size is determined, and its count is printed, signifying the number of unique words in the dataset.

Convert Text to Sequences of Integers:

```
# Convert Text to Sequences of Integers
text_sequences = tokenizer.texts_to_sequences(df['text'])
parent_text_sequences = tokenizer.texts_to_sequences(df['parent_text'])
```

Pad Sequences to Make them Uniform in Length:

```
# Pad Sequences to Make them Uniform in Length
max_sequence_length = 100  # Define the maximum sequence length
text_sequences_padded = pad_sequences(text_sequences,
maxlen=max_sequence_length, padding='post', truncating='post')
parent_text_sequences_padded = pad_sequences(parent_text_sequences,
maxlen=max_sequence_length, padding='post', truncating='post')
```

Performing One-hot Encoding:

```
import tensorflow as tf

# Perform One-hot Encoding
onehot_text = tf.keras.utils.to_categorical(text_sequences_padded,
num_classes=vocab_size)
onehot_parent_text =
tf.keras.utils.to_categorical(parent_text_sequences_padded,
num_classes=vocab_size)

# Print information
print(f"Example one-hot encoded text: {onehot_text[0]}")
```

Here, it’s continued from the previous steps and perform the following:

- Building a Vocabulary: Combining the 'text' and 'parent_text' columns to build a vocabulary using the Tokenizer class from Keras.
- Convert Text to Sequences of Integers: Then convert the tokenized text to sequences of integers using the Tokenizer's texts_to_sequences method.
- Pad Sequences to Make them Uniform in Length: Padding the sequences to ensure they have the same length using the pad_sequences function from Keras.
- Perform One-hot Encoding: Performing one-hot encoding on the padded sequences using Keras' to_categorical function.

Implementing a Vanilla RNN:

In order to implement the model, here is a VanillaRNN class that inherits from tf.keras. model is used. The class includes an embedding layer to convert integer-encoded tokens to dense vectors, a Vanilla RNN layer to process sequences, and a dense (fully connected) layer with a softmax activation for output. The call method defines the forward pass of the model.

```
import tensorflow as tf

class VanillaRNN(tf.keras.Model):
    def __init__(self, vocab_size, embedding_dim, hidden_units):
        super(VanillaRNN, self).__init__()
        self.embedding = tf.keras.layers.Embedding(vocab_size,
embedding_dim)
        self.rnn = tf.keras.layers.SimpleRNN(hidden_units,
return_sequences=True)
        self.dense = tf.keras.layers.Dense(vocab_size,
activation='softmax')

    def call(self, inputs):
        embedded = self.embedding(inputs)
        rnn_output = self.rnn(embedded)
        logits = self.dense(rnn_output)
        return logits

# Parameters
vocab_size = vocab_size = len(tokenizer.word_index) + 1
# Vocabulary size from the previous steps
embedding_dim = 128
hidden_units = 256
```

```
# Initialize the model
model = VanillaRNN(vocab_size, embedding_dim, hidden_units)
```

In this step, I defined the training process for the Vanilla RNN model. I have defined the loss function (Sparse Categorical Crossentropy) and the optimizer (Adam). Then, loop is run through the dataset in mini-batches, calculates gradients using backpropagation, and updates the model's parameters using the optimizer. Then it prints the loss for each batch during training.

```
# Define loss and optimizer
loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
optimizer = tf.keras.optimizers.Adam()

# Training loop
num_epochs = 10
batch_size = 64

for epoch in range(num_epochs):
   for batch_start in range(0, len(onehot_text), batch_size):
       batch_end = batch_start + batch_size
       batch_input = onehot_text[batch_start:batch_end]
       batch_target = text_sequences_padded[batch_start:batch_end]

       with tf.GradientTape() as tape:
           logits = model(batch_input)
           loss = loss_fn(batch_target, logits)

       gradients = tape.gradient(loss, model.trainable_variables)
       optimizer.apply_gradients(zip(gradients,
model.trainable_variables))

       print(f"Epoch [{epoch+1}/{num_epochs}], Batch
[{batch_start//batch_size+1}/{len(onehot_text)//batch_size}], Loss:
{loss:.4f}")
```

```
# Define a different learning rate
optimizer = tf.keras.optimizers.Adam(learning_rate=0.001)
```

```
hidden_units = 128  # Change the number of hidden units
model = VanillaRNN(vocab_size, embedding_dim, hidden_units)
```

Here, the training progress is already being monitored and printed in the previous block as part of the training loop. You'll see the loss for each batch during each epoch.

Text Generation:

```
# Choose a random starting seed text
seed_text = "I love"
seed_text = normalize_text(seed_text)
seed_sequence = tokenizer.texts_to_sequences([seed_text])
seed_padded = pad_sequences(seed_sequence, maxlen=max_sequence_length,
padding='post', truncating='post')

# Generate text based on the seed
generated_text = []
num_words_to_generate = 50

for _ in range(num_words_to_generate):
   prediction = model(seed_padded)
   predicted_word_index = tf.argmax(prediction, axis=-1).numpy()[0][-1]

   # Convert predicted index to word
   predicted_word = tokenizer.index_word[predicted_word_index]
   generated_text.append(predicted_word)

   # Update seed with new word
   seed_padded = pad_sequences([[predicted_word_index]],
maxlen=max_sequence_length, padding='post', truncating='post')

# Join generated words into a sentence
generated_sentence = ' '.join(generated_text)

# Print generated text
print("Generated Text:")
print(generated_sentence)
```

In this code, I have generated text using the trained Vanilla RNN model. Starting with a seed text (e.g., "I love"), normalizing it, converting it to sequences, and padding it. Then, it iteratively predicts the next word using the model and update the seed with the new word. The loop continues for a specified number of words to generate (num_words_to_generate).

The generated words are stored in the generated_text list and then joined into a sentence. This process allows us to generate text based on different input prompts. We can experiment with various seed texts and observe how the generated text changes.

After generating text, we can analyze the quality and coherence of the generated sentences. Comparing them with the training dataset to assess if the Vanilla RNN has learned meaningful patterns and is able to produce contextually relevant text.

Limitations of Vanilla RNN: Vanilla Recurrent Neural Networks (RNNs) struggle to capture long-term dependencies inside sequences, which is generally owing to the vanishing gradient problem. This issue, shown by gradients that decline dramatically as they backpropagate, limits the model's ability to hold data across remote time steps. This disadvantage reduces the efficiency of language models because Vanilla RNNs have difficulty delivering cohesive and contextually suitable text for long sequences. Furthermore, concerns with gradients vanishing and exploding during training hamper the optimization process, breaking convergence and eventually reducing model performance. To address these issues, more complex models such as Long Short-Term Memory (LSTM) networks and Gated Recurrent Units (GRUs) have emerged. Memory cells and gating mechanisms are used in these models to successfully address dependencies over long time periods. Additionally, the attention-based Transformer design, which is commonly used in contemporary language models, has demonstrated outstanding performance in capturing distant associations and successfully addresses the drawbacks of Vanilla RNNs.

LSTM’s and GRU’s can be found in speech recognition, speech synthesis, and text generation (as we’ll see shortly). You can even use them to generate captions for videos. Neither LSTMs or GRUs have fundamentally different architectures from the vanilla RNNs. The control flow remains similar: it processes data by passing on information as it propagates forward. The differences lie in how the hidden state is computed and the operations within the LSTM or GRU cells.

Conclusion: The main goal of the task was to build a language model using a Vanilla Recurrent Neural Network that would produce text that was coherent and acceptable for the given context. The Reddit Comments scores - NLP dataset was used, and before it was put into the Vanilla RNN model for training, it underwent cleaning and preprocessing. The vanishing gradient problem is frequently cited as the reason why Vanilla Recurrent Neural Networks (RNNs) are unable to capture long-term dependencies inside sequences. This problem prevents the model from preserving information over long time horizons since gradients degrade dramatically during backpropagation. More advanced models, like as Long Short-Term Memory (LSTM) networks and Gated Recurrent Units (GRUs), have emerged in response to these issues. These models combine memory cells and gating techniques to effectively handle dependencies over long time spans.

References:

https://www.kaggle.com/code/danofer/reddit-comments-scores-nlp

https://medium.com/@annikabrundyn1/the-beginners-guide-to-recurrent-neural-networks-and-text-generation-44a70c34067f

https://medium.com/@VersuS_/coding-a-recurrent-neural-network-rnn-from-scratch-using-pytorch-a6c9fc8ed4a7