CSE-425
Assignment - 01

Submitted By:
Mohammad Shafkat Hasan
Section: 04
ID: 19101077

Date of Submission:
9th August 2023

1. Data Preparation:

a) Select a Dataset:
We can pick a large text corpus for training a language model. This might be a compilation of books, journal papers, Wikipedia pages, or other text sources. The dataset should be as big and diverse as possible to help the language model understand patterns and produce text that makes sense.

b) Preprocess the Dataset:
We must preprocess the data before feeding it to the RNN. The steps consist of:

- Convert the text to lowercase to ensure case insensitivity.
- Remove any special characters, numbers, or irrelevant symbols.
- Tokenize the text into words or characters to create a vocabulary.

c) Split the Dataset:
Divide the dataset into training and validation sets in order to develop and test the language model. Most splits fall into one of two categories: 70% training and 30% validation or 80% training and 20% validation.

2. Implementing a Vanilla RNN:

a) Design and Implement a Vanilla RNN:
Choose a deep learning library like TensorFlow, PyTorch, or Keras to implement the Vanilla RNN. Here's a high-level structure description of the Vanilla RNN:

1. Import the TensorFlow library.
2. Define a class called VanillaRNN that inherits from tf.keras.Model.
3. Initialize the class with three parameters: vocab_size, embedding_dim, and hidden_units.
4. Inside the constructor (**init**), create three layers for the model:
   a. An embedding layer to convert input tokens into dense vectors.
   b. A SimpleRNN layer with the specified number of hidden_units, returning sequences.
   c. A Dense layer with a softmax activation function to predict the next token.
5. Define the call method for the class to define the forward pass of the model:
   a. Pass the input through the embedding layer.
   b. Feed the output of the embedding layer into the SimpleRNN layer.
   c. Pass the output of the SimpleRNN through the Dense layer to get the final predictions.
6. Outside the class, create an instance of the VanillaRNN model using the specified vocab_size, embedding_dim, and hidden_units.
7. The model is now ready to be used for language modeling or other sequence generation tasks.

Three layers make up the Vanilla RNN: the input layer, the hidden layer, and the output layer. The network can keep information across successive steps thanks to a loop created by the hidden layer's output being fed back into it.

b) Train the Vanilla RNN:
Prepare the training loop using the backpropagation through time (BPTT) algorithm:

1. Prepare training and validation datasets by tokenizing and preprocessing the data.
2. Define the loss function as Sparse Categorical Crossentropy and the optimizer as Adam.
3. Define a loss function that calculates the mean loss, accounting for padding in the target sequences.
4. Define a training step function that computes gradients and applies them to update the model's trainable variables.
5. Set the number of epochs for training (10 in this case).
6. Perform the training loop, iterating over the specified number of epochs.
7. For each epoch, calculate the total loss by iterating over batches in the training dataset and running the train_step function.
8. Print the average loss at the end of each epoch to monitor the training progress.

Apply the backpropagation through time (BPTT) technique to the training dataset to train the Vanilla RNN. The backpropagation extension known as BPTT takes the temporal aspect of the RNN into account. It enables the model to incorporate lessons from earlier time steps and change the weights accordingly.

c) Experiment with Different Hyperparameters:
To enhance the performance of the model, play around with various hyperparameters like the learning rate, quantity of hidden units, and length of the sequence. The RNN's ability to learn complicated patterns depends on both the number of hidden units and the learning rate, which regulates the step size during gradient descent.

d) Monitor the Training Process:
Keep an eye on the loss and other pertinent metrics (like perplexity) on the validation set during training. We can assess the model's performance and identify problems like over- or underfitting using this.
It's important to keep in mind that the example given above is a simplified version, and the syntax and particulars may change based on the deep learning library we select. The procedures listed here should give we a broad notion of how to go about putting a Vanilla RNN for text creation into practice.

3. Text Generation:

a) After training the Vanilla RNN:
Use the Vanilla RNN model to generate text depending on an input prompt after training it. The trained RNN is fed the initial input prompt to produce text, and the result of the model is used as

the input for the subsequent time step. To create a series of words or characters, repeat this method.

b) Experiment with different inputs:
Examine how the generated text changes by experimenting with various input prompts. To examine how the language model reacts and produces contextually relevant content, we can give the language model a variety of starting phrases or sentences.

c) Analyze the quality and coherence:
Comparing the generated text to the training dataset will allow you to evaluate its consistency and quality. The fluency, grammar, and adherence to the input prompt of the output text should all be evaluated. You should be aware that the Vanilla RNN may not always create flawless text and may occasionally produce incomprehensible or repetitious sequences.

4. Limitations of Vanilla RNN:

a) Research and discuss the limitations:
The shortcomings in Vanilla RNNs' ability to recognize long-term dependencies in sequences. The vanishing gradient problem, which affects vanilla RNNs, makes it difficult for the model to grasp long-range dependencies in sequences. The gradients can get very small as the sequence length rises, making it challenging to learn from information from the distant past.

b) Describe the vanishing and exploding gradient:
The gradient issues that can arise during training, such as vanishing and exploding. When the gradients of the loss function with respect to the parameters of the model backpropagate across time, they tend to exponentially decrease, this is known as the vanishing gradient problem. Effective weight updating by the RNN is hampered by this problem.

Conversely, the exploding gradient problem happens when the gradients grow exponentially during backpropagation. This can lead to unstable training and cause the model's weights to become too large, making it challenging to converge on a good solution.

c) Explain limitations:
The vanishing gradient issue can make it difficult for the Vanilla RNN to identify long-range dependencies in the text. The model can therefore struggle to comprehend context that depends on far-off words or letters.

Finding the model's ideal set of weights might be challenging due to instability brought on by the exploding gradient problem during training. As a result of this instability, training may experience sluggish convergence or even divergence.

d) Discuss potential solutions or alternative models:
Advanced RNN architectures like the Gated Recurrent Unit (GRU) and Long Short-Term Memory (LSTM) were created to solve the vanishing gradient issue. These systems better capture long-term interdependence by controlling the flow of information and gradients through the use of gating techniques.

Gradient clipping can be used to address the issue of bursting gradients. Gradient clipping is a technique that restricts the gradients' size during training to stop them from expanding too much and leading to instability.

5. Report and Presentation:

a) Implementation Summary:
In this project, we developed a language model based on a Vanilla Recurrent Neural Network (RNN) to generate coherent and contextually relevant text. Here's an overview of our implementation:
Dataset: We selected a diverse dataset consisting of a collection of articles from various domains. The dataset was preprocessed by tokenizing the text into words, removing punctuation, and lowercasing the text.
Model Architecture: We designed and implemented a Vanilla RNN using TensorFlow. The architecture consisted of an input layer, a hidden layer, and an output layer. The hidden layer's output was looped back into itself to capture sequential dependencies.
Training Process: The Vanilla RNN was trained on the preprocessed dataset using backpropagation through time (BPTT). We experimented with different hyperparameters, including learning rate and the number of hidden units, to optimize the model's performance. The training process was monitored by tracking the loss and validation metrics.
Text Generation: After training, we used the trained Vanilla RNN to generate text based on various input prompts. We observed how the generated text changed with different prompts and evaluated the quality and coherence of the generated text.

b) Limitations of Vanilla RNNs: While the Vanilla RNN showed promise in generating text, we identified several limitations that affected its performance:
1. Vanishing Gradient Problem: The Vanilla RNN struggled to capture long-term dependencies in sequences. As the sequence length increased, the gradients during backpropagation became very small, making it challenging for the model to learn from a distant context. This limitation impacted the model's ability to generate coherent and contextually relevant text, especially for longer sequences.
2. Exploding Gradient Problem: We also encountered the exploding gradient problem during training. In some cases, gradients grew exponentially, leading to training instability and slow convergence. This issue hindered the training process and affected the model's overall performance.

c) Presentation:

Slide 1: Introduction

- Brief overview of the project's goal and importance.

Slide 2: Data Preparation

- Details about the dataset selection, preprocessing steps and dataset split ratio.

Slide 3: Model Architecture

- Explanation of the Vanilla RNN architecture, including input, hidden, and output layers.
- Diagram illustrating the flow of information through the RNN.

Slide 4: Training Process

- Overview of the training process using BPTT.
- Graph showing the training and validation loss over epochs.
- Discussion of hyperparameter tuning and its impact on model performance.

Slide 5: Text Generation

- Explanation of how text generation works using the trained Vanilla RNN.
- Examples of generated text based on different input prompts.

Slide 6: Limitations of Vanilla RNNs

- Discussion of the vanishing gradient problem and its impact on long-term dependencies.
- Explanation of the exploding gradient problem and its effect on training stability.

Slide 7: Potential Solutions and Alternatives

- Introduction to LSTM (Long Short-Term Memory) and GRU (Gated Recurrent Unit) as alternative RNN architectures.
- Description of how these architectures address the limitations of Vanilla RNNs.

Slide 8: Findings and Analysis

- Summary of the project's outcomes, highlighting both successes and limitations.
- Comparison of generated text quality between different input prompts and sequence lengths.

Slide 9: Conclusion

- Recap of the project's objectives and key takeaways.
- Emphasis on the importance of addressing the limitations for more robust language modeling.

Slide 10: Future Improvements

- Suggestions for future work include implementing more advanced RNN architectures (LSTM, GRU) or exploring techniques like gradient clipping.
- Encouragement for further research and experimentation.

By effectively summarizing our implementation, analyzing the limitations of Vanilla RNNs, and suggesting potential improvements, we can convey the depth of our work and the insights gained from this language modeling project.