

CSE-425

Assignment - 01

*Submitted By:*

Mohammad Shafkat Hasan

Section: 04

ID: 19101077

Date of Submission:

3rd 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.