(Individual Assignment I - MGSC-695-075 – Adv Topics in Mgmt Science)

**Text Generation With RNNs**

*Name: Arham Anwar*

*ID: 261137773*

**Section 1 – Introduction:**

Text generation using Recurrent Neural Networks (RNNs) is a powerful technique in the field of natural language processing. This approach leverages the ability of RNNs to understand and generate sequential data, making them suitable for tasks such as text generation, language translation, and more. In this assignment, we explore the process of building and training RNN-based models to generate text in the style of Shakespeare. By setting up a controlled experiment with different model configurations and hyperparameters, we aim to understand the impact of these variations on the quality and creativity of the generated text.

**Section 2: Seed For Repeatability**

A function named set\_seed had been defined to ensure reproducibility across different runs of the same program. The random seed had been set to a fixed value, which in this case was 42. The random seed had been set to a fixed value, which in this case was 42. The purpose of setting this random seed had been to initialize the random number generators in Python's random module, NumPy (np), and PyTorch (torch), making the output of operations involving randomness deterministic.

**Section 3: Data Preparation**

**3.1. Loading:**

This part of the code had been designed for downloading the "Tiny Shakespeare" dataset from a specified URL. By the time the script was executed, the urllib.request.urlretrieve function had fetched the text file containing Shakespeare's works from an online repository and had saved it locally as 'shakespeare.txt'. The process had concluded with a print statement that confirmed the successful download and the name of the saved file.

**3.2. Lower Casing the Data:**

Converting text to lowercase standardizes the dataset by treating variations of the same word equally, simplifying searching, matching, and reducing the number of unique tokens for more efficient processing in machine learning tasks.

**3.3. Using 700K characters only:**

The model was trained only on seven hundred thousand characters between character 0.1M and 0.8M. This is done due to the compute and processing constraints. I did not remove the punctuations in one of the two final models. We will see results of removing and not removing punctuations later.

**3.4. Character dictionary:**

This code segment identifies all unique characters in the text, sorts them, and then creates two dictionaries: char\_to\_index maps each character to its corresponding unique index, and index\_to\_char does the reverse, mapping each index back to its character. This facilitates easy conversion between characters and their indices, which is essential for many text processing and machine learning applications.

**3.5. Sequence Configuration:**

The code had been designed to prepare training data for a sequence prediction model, where the aim is to predict the next character in a sequence of text. It sets a sequence length (`SEQ\_LENGTH`) of 40 characters and steps through the text in increments of 3 characters (`STEP\_SIZE`). For each step, it had collected a sequence of 40 characters as an input (`sentences`) and the subsequent character as the target (`next\_characters`). To transform the text data into a format suitable for machine learning, the script had initialized two numpy arrays, `X` and `y`, with dimensions based on the number of sequences extracted. It had then converted each character in the sequences into its corresponding index using the `char\_to\_index` mapping. This index conversion facilitates the use of categorical data in machine learning models, particularly those used in natural language processing, by representing textual information as numerical data.

**3.6. Final Data Model Onboarding:**

This code section defines a custom dataset class, `ShakespeareDataset`, tailored for handling sequence data for training machine learning models with PyTorch. This class takes input arrays `X` and target array `y`, converting them into PyTorch tensors of type `long`, which is suitable for categorical data often used in classification tasks. The class provides methods to determine the number of items in the dataset (`\_\_len\_\_`) and to retrieve a specific item by its index (`\_\_getitem\_\_`). This setup is crucial for iterating through the data during model training.

Additionally, the code sets a batch size of 256, meaning that each batch processed by the neural network during training will contain 256 examples. The dataset is then split into training and validation sets, with 80% of the data used for training and 20% for validation. The `DataLoader` objects for both the training and validation sets are created, with shuffling enabled for the training set to ensure that the data order varies. This helps improve model generalization by preventing the model from learning the order of the training data. The validation set data order is not shuffled to maintain consistency during evaluation.

**Section 4: Model building**

I have taken three approaches in model building –

* Model with Keras
* Model with Pytorch lightning (Punctuations Tokenized)
* Model with Pytorch lightning (Punctuations removed)

In this document, we will only be describing the third model, but the code for other models is also attached which have equally promising results.

The ShakespeareModel, implemented in PyTorch Lightning, predicts the next character in Shakespeare sequences. Key parameters include the number of unique characters (n\_chars), hidden layer size (hidden\_size), number of LSTM layers (num\_layers), learning rate (lr), and dropout rate (dropout). The LSTM processes the input sequence and initial states, while a fully connected layer predicts the next character.

The forward method converts input sequences to one-hot vectors and initializes LSTM states to zeros. The LSTM output feeds into the fully connected layer for prediction. The training\_step and validation\_step methods compute and log losses using cross-entropy. Validation\_epoch\_end aggregates and logs average validation losses. The Adam optimizer, chosen for its efficiency, is set up in configure\_optimizers.

The generate\_text method creates new sequences from a seed input, using a temperature parameter to add randomness. The model is instantiated with recommended hyperparameters, including increased hidden size and layers, reduced learning rate, and dropout for regularization. ModelCheckpoint and EarlyStopping callbacks save the best model and stop training early if validation loss stagnates. TensorBoardLogger logs training progress. The Trainer trains the model with the defined dataloaders, epochs, and GPU availability, preparing it for evaluation and text generation.

**Section 5: Model hyperparameters**

* Number of Unique Characters (`n\_chars`) = `len(char\_to\_index)`: This value is optimal as it ensures the model can process and generate any character present in the dataset, making the model versatile for text generation. Defining the size of the input and output layers based on the actual number of characters guarantees that the model captures the full character set, enabling precise and accurate text prediction.
* Hidden Size (`hidden\_size`) = `128`: This hidden size is ideal as it provides the model with ample capacity to learn and represent intricate sequences in the Shakespeare dataset, significantly improving the quality of generated text. A hidden size of 128 strikes the perfect balance between capturing complex patterns and maintaining computational efficiency, ensuring robust and coherent text generation.
* Number of Layers (`num\_layers`) = `3`: Three layers are the perfect choice, balancing model depth and complexity. This allows the model to learn hierarchical representations of the text effectively while mitigating the risk of overfitting. With three layers, the LSTM can understand and generate long-term dependencies in the text, enhancing the quality and coherence of the generated output without incurring excessive computational cost.
* Learning Rate (`lr`) = `0.001`: This learning rate is optimal as it enables the Adam optimizer to make gradual and stable updates to the model weights, ensuring reliable convergence and superior overall performance. A rate of 0.001 facilitates fine-tuning by making small, precise adjustments, preventing drastic changes that could destabilize training. This value ensures the model efficiently reaches the local minimum.
* Dropout Rate (`dropout`) = `0.3`: A dropout rate of 0.3 is the best choice as it introduces effective regularization, significantly reducing the likelihood of overfitting and enhancing the model's generalization capability. This dropout rate ensures the network does not rely too heavily on specific neurons by randomly setting a fraction of input units to zero during training. This leads to the development of robust features, improving the model's performance on unseen data and increasing its generalization ability.

**Section 6: Model Checkpoints & Best Weights**

The ModelCheckpoint and EarlyStopping callbacks work together to optimize the training process. ModelCheckpoint ensures that the best model weights are saved based on validation performance, while EarlyStopping prevents unnecessary training by stopping the process once the validation loss ceases to improve. This combination helps in efficiently training a robust model with optimal performance.

**6.1. Model Checkpoints and Weight Saving:**

In the provided code, the ModelCheckpoint callback is used to save the model checkpoints and weights during training. This callback monitors the validation loss (val\_loss) and saves the model's weights at the checkpoint directory specified (checkpoints/). The filename for each saved checkpoint includes the epoch number and the validation loss at that epoch (shakespeare-{epoch:02d}-{val\_loss:.2f}), making it easy to identify the best-performing model.

The ModelCheckpoint callback is configured with the following parameters:

* monitor='val\_loss': This specifies that the callback should monitor the validation loss.
* dirpath='checkpoints/': This is the directory where the checkpoints will be saved.
* filename='shakespeare-{epoch:02d}-{val\_loss:.2f}': This defines the format of the checkpoint filenames.
* save\_top\_k=1: This ensures that only the best model (with the lowest validation loss) is saved.
* mode='min': This specifies that the model with the minimum validation loss should be saved.
* save\_weights\_only=True: This indicates that only the model weights (not the entire model) should be saved.
* verbose=True: This enables verbose output, so information about checkpoint saving is printed during training.

**6.2. Early Stopping:**

The EarlyStopping callback is used to stop the training process early if the validation loss does not improve for a specified number of epochs (patience=3). This helps prevent overfitting and saves computational resources by terminating training when further training is unlikely to yield better performance.

The EarlyStopping callback is configured with the following parameters:

* monitor='val\_loss': This specifies that the callback should monitor the validation loss.
* patience=3: This sets the number of epochs with no improvement after which training will be stopped.
* mode='min': This specifies that early stopping should occur when the validation loss stops decreasing.
* verbose=True: This enables verbose output, so information about early stopping is printed during training.

**Section 7: Text Generation**

The text generation process in the ShakespeareModel involves encoding a seed text, using the model to predict subsequent characters, and iteratively generating new characters until the desired text length is achieved. The temperature parameter plays a crucial role in controlling the randomness of the predictions, allowing for a balance between creative diversity and coherent text generation. This method ensures that the generated text closely mimics the style and structure of the training data, producing sequences that resemble the original Shakespearean language.

This code uses the previously trained `ShakespeareModel` use checkpointed weights to generate text. The text generation function `generate\_text` is invoked four times with the same seed text but different `temperature` values (0.8, 0.2, 0.4, and 0.6). These temperature values adjust the randomness of the predictions:

1. Temperature 0.5:

A screenshot of a computer

Description automatically generated

1. Temperature 0.2: Lower temperature leads to more deterministic and repetitive text. It sharpens the probability distribution, making the model more confident in its next character prediction, which can lead to less diverse output.

A screenshot of a computer program

Description automatically generated

1. Temperature 0.4: This setting offers a balance between diversity and accuracy, slightly favoring predictable text while allowing for some variation.

A screenshot of a computer

Description automatically generated

1. Temperature 0.6: This is closer to a neutral setting where the output is neither too deterministic nor too random, providing a reasonable mix of predictability and creativity in the generated text.

A screenshot of a computer program

Description automatically generated

1. Temperature = 0.8

A screen shot of a computer program

Description automatically generated

1. Temperature = 1.0

A screenshot of a computer program

Description automatically generated

Each generated text instance is then printed, showcasing how varying the temperature affects the diversity and unpredictability of the text generated by the neural model. This feature is essential for tuning the output for different applications, whether more creative or more accurate text is desired.

**Section 8: Appendix**

**8.1. Model Outputs for “Punctuations Not Removed” Model:**

* + 1. Temperature 0.2: Lower temperature leads to more deterministic and repetitive text. It sharpens the probability distribution, making the model more confident in its next character prediction, which can lead to less diverse output.

A screenshot of a computer program

Description automatically generated

* + 1. Temperature 0.4: This setting offers a balance between diversity and accuracy, slightly favoring predictable text while allowing for some variation.

A screenshot of a computer program

Description automatically generated

* + 1. Temperature = 0.8

A screenshot of a computer program

Description automatically generated

8.2. Model Outpus for Keras Model

**A screen shot of a computer code

Description automatically generated**

**Thank You**