MGSC-695-075 - Adv Topics in Mgmt Science- Assignment -2

Text Generation Using Recurrent Neural Networks

Introduction:

The project explores the use of Recurrent Neural Networks (RNNs) for text generation using a dataset of Shakespeare texts. The code consists of preparation of the text data, building a dataset, and then utilize it to train a model and evaluate the model. Different architectures and models (RNN, LSTM and GRU) are tested to find the optimal model. Text generation is a common application in the field of natural language processing (NLP), where the objective is to generate text that mimics a particular style or corpus, in this case Shakespeare's plays.

Text Data Preparation:

The first section of the code focuses on preparing the text data for modeling. It starts with the doc2words function which cleans the data and splits it into lines, strips off any unwanted quotation marks, and then splits these lines into words. This is a preliminary step in text preprocessing, converting raw text data into a more manageable form. We then use a function to process these words to remove any punctuation. This step is used to ensure that the model learns from only text characters, improving its ability to generate meaningful text. Following this, a function is used to build a vocabulary from the processed words. It utilizes the Counter class to count word frequencies and sorts them. This vocabulary is essential for encoding words into integers, a form that can be processed by neural networks. The vocab_map function maps the vocabulary to integers and vice versa. This dual mapping is beneficial for both feeding data into the model (words to integers) and interpreting the output from the model.

Dataset Construction for RNN:

After the text is preprocessed and a vocabulary created, the next portion of the code is used to construct a dataset suitable for training an RNN. The TextDataset class initializes with the words, mapping of vocabulary to integers, sequence size, and batch size. In its preprocess_data method, it converts words into their integer representations. It then organizes these integers into sequences (defined by seq_size) that the RNN will use as inputs. The target for each sequence (used during training) is the sequence shifted by one position, aiming to predict the next word given the previous words.

Model Definition and Training

RNN Module Architecture

The RNNModule class defines a neural network for text generation that allows for the utilization of three different types of recurrent units: RNN, LSTM, and GRU (model_type). The constructor of the RNNModule initializes an embedding layer that maps each integer-encoded vocabulary word into a high-dimensional space, helping the model capture semantic relationships. Depending on the chosen model_type, the appropriate recurrent layer is initialized with specified parameters for dropout (to prevent overfitting), the number of layers, and whether the input and output tensors are provided as batches first. The output from the recurrent layers is passed through a fully connected layer, which outputs the probability distribution across the vocabulary for the next word in the sequence.

The forward method accepts input sequences and an initial state for the recurrent layer, computes the embeddings, passes these through the recurrent layer, and then through the dense layer to generate logits. For LSTM, it manages these states appropriately, ensuring they are passed through the network during sequential processing. The zero_state method initializes the hidden states (and cell states for LSTM) with zeros. This is particularly useful at the start of processing a new sequence or batch, ensuring no leftover state from previous batches influences the current computation.

Training and Evaluation Functions

The train_model function is trained for a number of epochs specified by the epochs parameter until early stopping is triggered if the validation loss does not improve for a given number of epochs (patience). The model processes batches of data, computes loss using CrossEntropyLoss, and performs backpropagation. Gradient clipping is used to prevent the exploding gradient problem, common in training RNNs. After each epoch, the model's performance is evaluated on the validation set, and the best model parameters are saved if the validation loss improves.

The evaluate_model function supports the training loop by evaluating the model on a validation set. It computes both the loss and the accuracy, providing insights into how well the model predicts the next word in sequences during validation. This function helps monitor the model's generalization capabilities and prevent overfitting.

Perplexity Computation

The compute_perplexity function is a specialized evaluation metric used commonly in language models. Perplexity measures how well a probability model predicts a sample. A lower perplexity indicates a better predictive performance. In the context of this text generation model, perplexity is computed as the exponential of the average loss per word over a dataset, providing a normalized measure of the model's uncertainty in its

predictions. The perplexity score is used to find the best model architecture for the text generation task.

Overview of Experiment Process

The code experiments with different configurations of recurrent neural network architectures to generate text based on a dataset resembling Shakespeare's style. The objective is to identify the model configuration that achieves the lowest perplexity, a metric that gauges how well a probability distribution or probability model predicts a sample.

Model Architectures

Three types of recurrent neural network architectures are tested: LSTM (Long Short-Term Memory), GRU (Gated Recurrent Units), and standard RNN (Recurrent Neural Network). These architectures are selected because they are fundamentally designed to handle sequences of data.

LSTM: Known for its ability to capture long-term dependencies in sequence data by handling the vanishing gradient problem, which is common in traditional RNNs.

GRU: Similar to LSTM in that it can also capture dependencies over long sequences but with a simpler structure, leading to faster training times.

RNN: The simplest form of a recurrent network, useful as a baseline to compare how much improvement the more complex structures provide.

Parameters : The script iterates over different combinations hyperparameters:

batch_size: It is the size of data batches fed into the network, affecting the speed and stability of the training process.

seq_size: is the length of the word sequences used as input to the network, impacting how much context the model sees during training.

embedding_size: is the dimensionality of the embeddings used to convert words into continuous vectors, influencing the model's ability to capture semantic nuances.

lstm_size: is the number of units in the hidden layers of the RNN, dictating the capacity of the network to learn from data.

num_layers: is the number of recurrent layers stacked in the network, which can enhance learning at the cost of increased computational complexity.

These parameters are varied to explore how different configurations influence model performance, specifically looking at their effect on perplexity.

Loss Function and Optimizer

CrossEntropyLoss: Used here as the loss function, is standard for classification tasks where the output is a probability distribution across classes—in this case, the vocabulary. It measures the discrepancy between the predicted probabilities and the target.

Adam Optimizer: Chosen for its efficiency in handling sparse gradients and adaptive learning rate capabilities, making it well-suited for training deep neural networks on large datasets.

Training and Evaluation

For each configuration, the code performs the following steps:

- Data Preparation: The text data is split into training and validation sets. A DataLoader handles batching and shuffling of the data.
- Model Initialization: An RNN model is instantiated with the specified parameters and moved to the appropriate computational device.
- Training: The model is trained for a set number of epochs or until early stopping is triggered if the validation loss does not improve after a specified number of epochs. Training involves forward and backward passes where the model learns by adjusting weights to minimize loss.
- Evaluation: After training, the model's perplexity on the validation set is calculated to evaluate its performance.

Selection of Best Model

The model that achieves the lowest perplexity across all configurations is considered the best model. This model is expected to predict the next word in sequences most effectively, indicating a better understanding and replication of the style and structure of the input text data.

Model Type	Batch Size	Seq Size	Embedding Size	LSTM Size	Num Layers	Perplexity
LSTM	16	32	64	128	1	628.5574
LSTM	16	32	64	128	2	620.4424
LSTM	16	32	64	128	3	673.5579
LSTM	16	32	64	256	1	621.4990
LSTM	16	32	64	256	2	620.3228
LSTM	16	32	64	256	3	681.3264
LSTM	16	32	128	128	1	648.6124
LSTM	16	32	128	128	2	603.7138

LSTM	16	32	128	128	3	674.8520
LSTM	16	32	128	256	1	603.0788
LSTM	16	32	128	256	2	615.3657
LSTM	16	32	128	256	3	696.5352
GRU	16	32	64	128	1	539.8554
GRU	16	32	64	128	2	582.9596
GRU	16	32	64	128	3	764.6699
GRU	16	32	64	256	1	533.6784
GRU	16	32	64	256	2	666.1320
GRU	16	32	64	256	3	1533.4836
GRU	16	32	128	128	1	531.7212
GRU	16	32	128	128	2	591.9791
GRU	16	32	128	128	3	1491.8560
GRU	16	32	128	256	1	531.9664
GRU	16	32	128	256	2	631.9314
GRU	16	32	128	256	3	933.1531
RNN	16	32	64	128	1	1763.6457
RNN	16	32	64	128	2	1921.2284
RNN	16	32	64	128	3	1875.6389
RNN	16	32	64	256	1	2834.3634
RNN	16	32	64	256	2	3527.2209
RNN	16	32	64	256	3	1622.9734
RNN	16	32	128	128	1	1664.4179
RNN	16	32	128	128	2	1639.4450
RNN	16	32	128	128	3	2108.2034
RNN	16	32	128	256	1	5358.1673
RNN	16	32	128	256	2	1811.8924
RNN	16	32	128	256	3	6192.5312

Resulting Insights :

When evaluating the performance of different recurrent neural network architectures on text generation tasks, Gated Recurrent Units (GRUs) was selected as the most effective model type. GRUs had the lowest perplexity scores, which is indicative of their capability in handling sequential data. The best performing GRU model achieved a perplexity score of 531.7212 with a configuration of batch_size=16, seq_size=32, embedding_size=128, lstm_size=128, and one layer.

Long Short-Term Memory (LSTM) models also demonstrated good performance, outperforming the basic Recurrent Neural Network (RNN) models, because of their ability to manage long-term dependencies more effectively. The best LSTM model recorded a perplexity of 603.0788 under similar configuration parameters to the top GRU model.

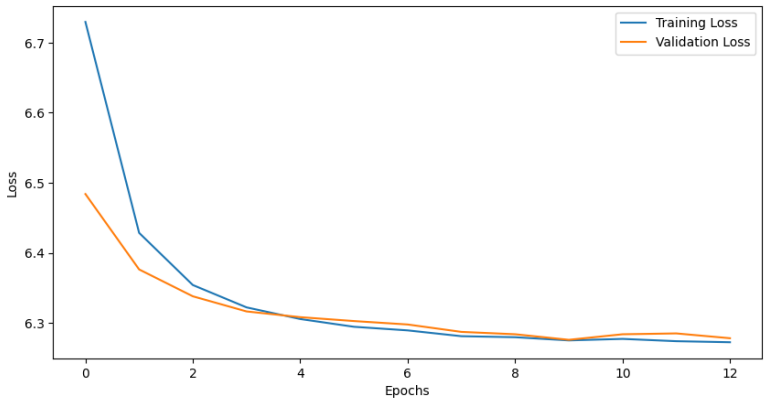
On the other hand, vanilla RNN models exhibited the highest perplexity scores, signaling a relatively poorer performance. This indicates their limitations in modeling more complex sequential patterns compared to GRU and LSTM models. The performance of RNN models did not significantly improve even with increased embedding and LSTM sizes, highlighting their inherent shortcomings.

GRU models with fewer layers (one or two) tend to yield better results, with additional layers potentially leading to overfitting or increased training difficulties. This was similarly noted in LSTM models, where configurations with fewer layers also tended to perform better. Both models showed that larger embedding and LSTM sizes could beneficially impact performance, suggesting a capacity to capture more complex patterns in data.

For both model types, employing fewer layers could mitigate overfitting and simplify the training process. Experimenting with larger embedding and LSTM sizes is advised to potentially enhance model performance. Additionally, further tuning of hyperparameters such as learning rate, weight decay, and dropout is done, utilizing packages like Optuna to optimize these parameters effectively.

Visualization

Finally, the training and validation losses of the best-performing model are plotted against epochs. This visualization helps in understanding the learning dynamics and stability of the model over time, providing insights into its convergence behavior and overfitting tendencies.



Text Generation Function

The generate_text function is designed to produce text using a trained RNN model based on a given initial seed of words. This function uses the model to generate coherent and stylistically similar text as might be seen in Shakespeare's works. The function processes

each word in the provided initial seed (words). For each word, it converts the word to its integer representation using the vocab_to_int mapping and wraps this integer in a tensor, which is also sent to the device. This tensor is then fed into the model along with the current state. The output from the model and the new state are captured, but only the state is updated iteratively as the seed words are processed. Once the initial words have been processed, the function generates new words. The last output tensor is used to determine the next word. Using torch.topk, it identifies the top k probabilities (where k is specified by top_k) and selects one at random to introduce variability in the generated text, making each generation potentially unique even with the same seed. The chosen word (an integer index) is then converted back to its string representation using the int_to_vocab dictionary and appended to the list of words. This process repeats for a specified number of iterations (100 in this case), each time feeding the previously chosen word back into the model to generate the next word. Finally, the function joins all the words in the list (including both the seed and the newly generated words) into a single string and prints it. This resulting string represents the generated text sequence, which ideally follows the linguistic style and context established by the initial seed.

Output Text Generated:

Prompt: To be

```
"To be so much to you my liege And all your highness pleasure and you are
not to my heart To see this ring and I do accuse your grace I have my lord
to be a servant to my heart To give me leave to you with me And in your
power and you must not have no power of your love And give you love for me
For your own life I am no more than yours To have you both to hear you I
do not so far To make a gentle daughter and your highness And to the gods"
```