Character-Level Language Model with Recurrent Neural Networks

Pouya Pourakbarian Niaz
Department of Mechanical Engineering
Robotics and Mechatronics Laboratory
KUİŞ AI Center
Koç University
pniaz20@ku.edu.tr

Abstract

This project aims to help us understand and use Recurrent Neural Network (RNN) models for modeling sequential data such as language. We will use an RNN model to do two different sequence modeling tasks. In part 1, we will use an RNN model to perform character prediction as in consonant-vowel classification, and in part 2, we will build a character-level language model based on an RNN which, given a context as a character string, predicts the next character at each position.

1 Part 1: Consonant-Vowel Prediction

In this part, we will use the *text8* dataset, which includes excerpts from Wikipedia to classify vowels and consonants. In this part, the dataset comes in preprocessed, such that the entire dataset only contains characters a to z, and the space character, nothing else. The string sequences meant to be used for training and testing are also separated with the new-line character, meaning the sequence length is actually given for us; it is 20 characters.

We use a single-layer LSTM network followed by a Dense network for this task. The input of the Dense (fully connected) network is the output of the final time step of the LSTM layer. This final output is meant to summarize the entire sequence while also containing useful information and features from the sequence to make predictions.

Given a character string as a context, the network will predict whether or not the next occurring character is a vowel (a, e, i, o, u) or consonant. That is, we have a binary classification problem.

1.1 Network Architecture

The network architecture is a standard RNN model. It includes an embedding layer, an RNN layer, and a fully-connected network after it, followed by a sigmoid activation at the end, with a single output neuron.

1.1.1 Embedding

The embedding layer is a layer that transforms the raw input, which in this case is the index of the character (0 for ‘a’, 1 for ‘b’, ..., and 26 for the space character) into a meaningful numerical representation on which the model can work more efficiently. Embedding typically works on an embedding matrix, which is randomly initialized and learned through the training process. Since there are only 27 possibilities for every timestep of the input (meaning vocabulary size is 27), and since the embedding dimension is typically smaller than the vocabulary size, an embedding dimension of 16 or 8 is sufficient for character-level language tasks. The input to the embedding layer typically has size

Preprint. Under review.

$N \times L$, where N is the minibatch size or number of data points, and L is the sequence length, in this case, 20 characters. The output of the embedding layer will be a tensor with size $N \times L \times H_{\text{emb}}$ where H_{emb} is the embedding dimension.

1.1.2 Recurrent Neural Network Layer

The embedded sequence can be fed directly to the RNN layer, which can be a Simple RNN layer, a GRU layer, or an LSTM layer. In this assignment, we tried all three and compared their performances. The LSTM will give an output for every input time step, but we will only use the last one, the output of the final LSTM cell. This output represents a helpful summary of the entire sequence. In other words, the output of the LSTM will be tensor with shape $N \times (D * H_{\text{LSTM}})$ where D is 1 if the LSTM is unidirectional, and 2 if it is bidirectional, and H_{RNN} is the hidden size of the LSTM. Bidirectional LSTM (sometimes dubbed "BiLSTM") has been seen to deliver better performance occasionally in language modeling tasks, which is why we also tried it in this assignment. Multiple LSTM layers can be stacked on top of each other, in which case the final output of the final LSTM layer would be used.

1.1.3 Fully-Connected Network

Following the LSTM layer(s), a fully-connected (Dense) network needs to process the information that is summarized and present in the final output of the LSTM to make a decision on the output, as in whether or not the next character in the sequence will be a consonant (0) or vowel (1). This fully-connected network is typically a simple multi-layer perceptron with one or more hidden layers. We used the ReLU activation function for the hidden layers and a Sigmoid activation function for the output layer. The output of the Dense network will have shape $N \times 1$.

1.2 Training

For the training process, an indexer is provided that gets a character and returns its index as an integer. When parsing the dataset, this indexer can be used to make a tensor of indices as the input of the embedding layer. The Binary Cross-Entropy loss function is used because we have a binary classification task at hand. After trying Adam, SGD, and RMSProp, Adam was finally chosen as the optimization algorithm for our training scenario. The default parameters of the Adam optimizer were retained, and only different values were tried empirically for the learning rate α. The learning rate also has an exponential decay factor γ so that as the learning progresses, the learning rate is attenuated gradually to make convergence easier and smoother. The minibatch size was chosen empirically. L_2 regularization and Dropout were used to prevent overfitting, batch normalization was used for smoothening the landscape and making the training faster and less sensitive to random initialization, as well as minimizing covariate shift.

1.3 Evaluation

The provided devset text is used as both a validation set and a test set to evaluate the data both per-epoch and after the training. This dataset is much smaller than the training dataset and contains words and phrases never seen in the training set. Validation (test set) error and accuracy are the metrics that we are curious about.

1.4 Hyperparameter Optimization

Different hyperparameters were chosen randomly and manually to determine which sets of hyperparameters eventuate in better devset performances. The following sets of hyperparameters were optimized in this study:

- **Embedding dimension:** 8 and 16 were tried as embedding dimensions. Both achieved comparable results, but a lower embedding dimension would obviously lead to less computational cost.
- **RNN type:** The RNN layer type can be Simple RNN, GRU, or LSTM. LSTM turned out to have the best performance in our trials.

- **RNN hidden size:** The hidden size of the RNN layer. 16, 32, 64, and 128 were tried in this study.
- **Bidirectional:** Both unidirectional and bidirectional RNNs were tried in this study.
- **RNN depth:** Number of stacked RNN layers. 1 and 2 were tried in this study.
- **Dense width:** Width of the hidden layer(s) of the fully-connected network, if any. Values of 16, 32, 64, and 128 were tried in this work.
- **Dense depth:** Number of hidden layers of the fully-connected network. Values of 0 (RNN output directly to output layer), 1, 2, and 3 were tried in this assignment.
- **Dropout:** The dropout probability of the fully-connected network. This is a probability between 0 and 1. Values of 0 (no Dropout), 0.1, 0.2, 0.3, 0.4, and 0.5 were tried in our efforts. The Dropout layer comes after every hidden layer's activation function (in our case, ReLU).
- **Batch normalization:** The batch-normalization layer is typically fit between every hidden layer and its activation function. Both cases *with* and *without* batch normalization were tried in our work.
- L_2 **regularization:** The L_2 weight regularization parameter λ was drawn geometrically from the range of 0 (No L_2 regularization), 0.00001 to 0.1.
- **Minibatch size:** The minibatch size can affect training time and training outcome though it is an unwanted effect. Small minibatch sizes induce more stochasticity (Adam gets closer to SGD) and can sometimes prevent overfitting or create underfitting, and vice versa. Minibatch sizes 8, 64, 256, and 512 were tried in this work.
- **Learning rate:** The learning rate α of the optimization algorithm can have significant effects on how the training proceeds across the loss landscape. The learning rate was chosen geometrically from the interval of 0.0001 to 1.0 while not touching the other default hyperparameters of the optimizer.
- **Learning rate decay factor:** This factor γ defines how the training slows down as it proceeds across the loss landscape. Values between 0.8 and 0.99 were chosen in this study.

1.5 Results

After some hyperparameter optimization, the parameters shown in Table 1 were used in this part of the assignment. Using these hyperparameters, a **test set accuracy of 78.3%** was obtained. Changing the hyperparameters sometimes drastically weakened the results, sometimes leading to overfitting or underfitting. On the other hand, some other hyperparameters were relatively ineffectual in the outcome of the training. Embedding dimensions 8 and 16 both had a good performance. GRU and Simple RNN had much weaker performance than LSTM. The hidden size of the LSTM had minimal effects on the performance as long as it was between 32 and 64. Bidirectional LSTM had slightly better performance than unidirectional LSTM. RNN depth of 2 did not improve the results in any of the trials. Dense width and depth were not very prominent in determining the model's performance. Dropout also did not improve model accuracy much, as there is not much overfitting. Using batch normalization slightly improved the results. L_2 regularization slightly prevented overfitting as long as it was smaller in order of magnitude than 0.001. Larger values quickly resulted in underfitting. Minibatch size also did not significantly affect model training as long as it was within the interval of 64 and 128. Learning rates larger than 0.1 in order of magnitude resulted in a jagged training procedure and low training performance, and values smaller than 0.1 resulted in slow training. The learning rate decay factor was not very influential as long as it was between 0.95 and 0.98.

2 Part 2: Character-Level Language Modeling

In this part of the assignment, an RNN model is used for constructing a language model operating at the character level. In other words, given a context as a character string, the model would predict the next character at each time step and the character to come immediately following the context.

In this kind of model, it is not sufficient for the model to only get a context, process it, and predict the next character. The model also needs to predict the next character at every time step. Therefore, the

Table 1: Hyperparameters used in the consonant-vowel classification problem

Hyperparameter	**Value**
Embedding dimension	8
RNN Type	LSTM
RNN hidden size	32
RNN Bidirectional	Yes
RNN Depth	1
Dense width	32
Dense depth	1
Dropout	0
Batch normalization	Yes
L_2 regularization	0.0001
Minibatch size	128
Epochs	60
Learning rate	0.1
Learning rate decay	0.8
Optimizer	Adam

output at every time step of the RNN must be connected to a fully-connected network, predicting the next time step in the sequence. Therefore, unlike the previous part, the outputs of all time steps (cells) of the RNN are needed for the fully-connected network rather than just the final time step.

When training this kind of language model, the RNN cell at a time step would predict the next time step, and in the next time step, assuming that the prediction was correct, the ground truth character embedding would be fed to the RNN cell. Therefore, there would be a one-time-step shift between the input sequence of the RNN and the target output sequence being predicted by all time steps of the RNN.

Unlike the previous section, in a language model such as this, the output of all time steps of the (final) RNN layer would be used as input to the fully-connected network. In other words, the $N \times L \times D * H_{\text{RNN}}$ output of the RNN would be fed to the fully-connected network, with each layer only changing the last dimension without touching the first two. The output of the fully-connected network would be a tensor with shape $N \times L \times K$ where $K = 27$ is the number of classes, of which there are 27 in this study (vocab size is 27). The output layer would have a SoftMax activation (or LogSoftMax if log-probabilities were desired) which would operate on the last dimension of the above tensor. Once the index of the maximum (log-)probability in the last dimension is selected (i.e., argmax) as the class prediction (prediction for what the next character will be), we will end up with an $N \times L$ tensor of indices denoting the predicted characters for all time steps of all datapoint. In this tensor, index $(i,\ j)$ represents the prediction for the $j + 1$'th character in the context of the i'th datapoint in the batch.

2.1 Procedure

The general procedure is similar to the previous case. Embeddings, training, evaluation, and hyperparameter tuning steps are the same as the first part of the assignment. There are minor differences, however.

Firstly, the data used here is the entire corpus of 100K characters of the *text8* dataset, which means the sequence length is not known anymore. This gives us the freedom to choose a sequence length for our approach. For extracting sequences, a predefined number of words was used to extract groups of many consecutive words from the training corpus. The maximum character length among these groups of consecutive words was chosen as the sequence length. Then, all the groups were post-padded with the space character until they all had the same character length. The space character was added to all of these groups to denote the space character as the start-of-sequence (<SOS>) and also the end-of-sequence (<EOS>) token. All of these groups held an equal number of consecutive words from the corpus (we used a running window to extract consecutive words), but they were padded to have the same character length. The sequences were concatenated to generate the training and dev set data. The provided Indexer was then used for extracting indices, and the rest of the procedure was the same as the last part.

Secondly, the Categorical Cross-Entropy loss function was used because it is no longer a binary classification problem but a multi-class classification problem. As mentioned before, in accordance with the loss function, the (Log)SoftMax activation function is often chosen for the output layer of the network.

The evaluation of this language model is not only done via the accuracy metric but also the perplexity of the model when evaluated on the entire dev set/test set. Perplexity is a measure of entropy and evaluates how reliably the model fits the test corpus overall. The lower the perplexity, the better. As per the assignment, a uniform sampler would achieve a perplexity of 27 in this task because that is the vocabulary size. The assignment requests a perplexity smaller than 7.

Table 2: Hyperparameters used in the character-level language modeling task

Hyperparameter	**Value**
Sequence length	10 words
Embedding dimension	8
RNN Type	LSTM
RNN hidden size	64
RNN Bidirectional	No
RNN Depth	1
Dense width	Inapplicable
Dense depth	0 (RNN output directly to SoftMax layer)
Dropout	Inapplicable
Batch normalization	Inapplicable
L_2 regularization	0.0001
Minibatch size	128
Epochs	200
Learning rate	1.0
Learning rate decay	0.9
Optimizer	Adam

2.2 Results

After a hyperparameter optimization procedure similar to the previous case, the parameters shown in Table 2 were used at the end. Since a larger number of epochs were used for this part, validation patience of 20 epochs was used for early stopping in case validation accuracy did not improve.

Using the hyperparameters mentioned in Table 2, a training curve similar to what is shown in Fig 1 is established. This curve does not show any overfitting or underfitting, but this model is susceptible to overfitting. In the presence of a wider/deeper RNN network and/or a deeper fully-connected section, the model quickly overfits, requiring Dropout and further L_2 regularization.

When hyperparameters shown in Table 2 are used for training, the **training accuracy is often between 79-82%**, and the **validation accuracy is between 79-81%**. When evaluated on the entirety of the dev set text, this model achieves a **test set accuracy of between 54-58%** and a **perplexity of between 4.5-5.5**, which is smaller than 7, as requested by the assignment. This shows that the model achieves acceptable test set results for the character-level language modeling task and obtains a good perplexity, meaning it does not have too much entropy in predicting the following characters given the previous one.

3 Conclusion

The present assignment is an exercise in utilizing recurrent neural networks for language modeling and sequence prediction. Firstly, a sequential binary classification model is built using an LSTM network, which given a context, can detect whether the following character will be a vowel or a consonant. Secondly, another LSTM network was built for character-level language modeling. As such, given a context, the model can predict the next character. Both models achieved acceptable

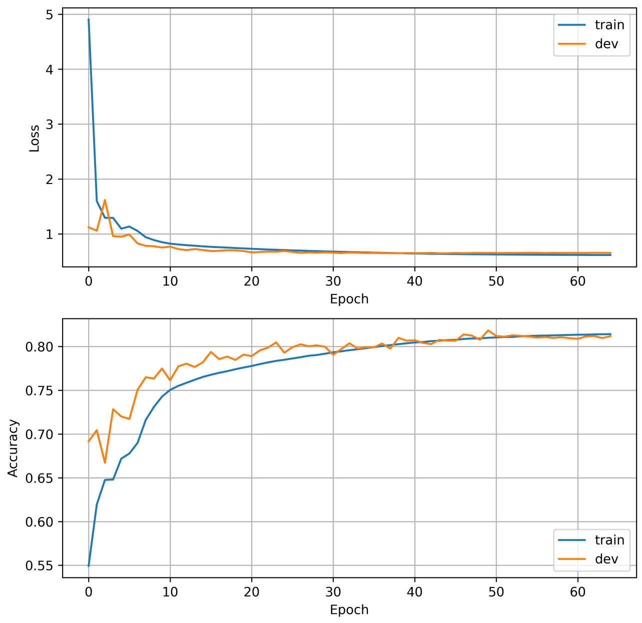

Figure 1: Training progress in the character-level language modeling task

performance after some hyperparameter tuning, proving the capabilities of recurrent neural networks in sequential modeling tasks.