CS 410 Technology Review – RNNs for Language Modeling

Tyler VanderLey

As we have seen throughout this course, language modeling is a fundamental task in the domains of text retrieval and text mining. In particular, a language model allows us to assign a probability distribution over any sequence of words, thereby allowing for the ability to probabilistically generate sentences or assess the likelihood of a given body of text. So far, the main language modeling approach we've analyzed is the unigram language model, in which we make the simplifying assumption that each word within a text is generated independently. More generally, a unigram model falls under the class of N-gram language models, in which the probability of the next word "depends only on the preceding n-1 words" (Chen, 2020). In this review, we will analyze the use of recurrent neural networks (RNNs) for language modeling, an approach that has greater modeling power than that of N-grams. Specifically, this review will examine how RNNs work, their advantages and drawbacks, possible variations, and existing implementations in software packages.

From a high-level, one can view a recurrent neural network as being composed of 3 structures: an input sequence (a sequence of words), a series of hidden states, and an output sequence. In the context of language modeling, the input sequence could be a sentence whose probability we want to compute, or a sequence of words where we want to predict the next word. Notably, a RNN "can process any length input" (Chen, 2020). Next, the hidden states are what truly make a RNN powerful. We define the hidden states over the course of a sequence of timesteps, where at each step, the hidden state takes in the current input from the input sequence and the hidden state from the previous timestep. Then, we perform a computation using these values to define a new hidden state for the current timestep, and then use this hidden state to produce a new output for the current step. Mathematically, we can write this relation as $\boldsymbol{h}(t) = f(W\boldsymbol{x}(t) + U\boldsymbol{h}(t-1))$, where $\boldsymbol{h}(t)$ represents the hidden state at step t, $\boldsymbol{x}(t)$ represents the input at step t, W and U represent learned weight matrices, and f represents an activation function that depends on the design of the neural network, but is often chosen to be functions like Tanh, Sigmoid, or ReLU (Le, 2019). A bias term may also be included in the expression. To define the output at step t, we can take a softmax of the new hidden state $\boldsymbol{h}(t)$, multiplied by some other learned weight matrix, to generate a probability distribution over our vocabulary of words. These

probabilities are what then allow us to use a RNN as a language model, as we can assess the probability of the next word in the sentence from the output distribution at each timestep.

Using the approach outlined above, RNNs have some powerful advantages that make them a very effective method for language modeling. Most notably, by using the previous hidden state to compute the new hidden state, RNNs can capture the entire context of a sentence as they are able to "use information from many steps back" (Chen, 2020). This feature contrasts with N-gram language models that can only utilize the previous n-1 words when assessing the probability of the next word. Therefore, RNNs are a highly expressive modeling technique that can maximize the amount of information available to make accurate, informed predictions about the likelihood of words/sentences. Additionally, there are some structural advantages that make RNNs flexible and easy to use in practice. For instance, they can "process any length input" and the "model size doesn't increase for longer input" since the learned parameters of the model don't change across timesteps, but instead remain consistent over time (Chen, 2020). Both features allow RNNs to be effective approaches across many types of use cases. Although there are many advantages to RNNs, there are also some disadvantages with the approach outlined above. The most apparent problem is the possibility that the earlier words may effectively be forgotten when making predictions about later words in a sentence. This is known as the "vanishing gradient problem" where during training of the model, deeper layers of the network may see an issue where "gradients flowing back in the back propagation step become smaller" to the point that they can no longer capture "long-term dependencies of the language", so predictions will rely only on recent words instead of earlier ones (Le, 2019). When this problem arises, the resulting modeling power of the RNN will be decreased since it will no longer be able to utilize the full capability of its recurrent structure that, theoretically, allows for the use of the entire preceding context of the sentence. To address this problem, there are two popular modifications of the RNN architecture described above that can assist in capturing long-range dependencies – LSTMs and GRUs.

LSTMs (Long Short-Term Memory) address the vanishing gradient problem by introducing a new state at each timestep called the cell state $\boldsymbol{c(t)}$. This new cell state "stores long-term information", and its use enables a LSTM to "erase, write, and read information from the cell" (Chen, 2020). At each timestep, a new intermediate cell state value is computed using the hidden state from the previous timestep and the current input, much like we did before when

computing the hidden state at each step. At each step, we also use three gates that come with their own learnable parameters to determine how much of the previous information to retain. Specifically, a "forget gate" tells the model how much of the previous cell state to use, an "input gate" controls how much of the new intermediate cell state to retain. With these two gates, we can then define a new finalized cell state, and then lastly we use an "output gate" to define how much of this cell state to use when computing the hidden state for this timestep (Chen, 2020). Together, these components allow the model to continuously retain information from previous steps of the recurrence, thereby mitigating the effects of the vanishing gradient problem and enabling long-range dependencies within the text to be captured. GRUs (Gated Recurrent Units) apply a similar idea to that of LSTMs but are a bit simpler and use fewer parameters. In particular, they use just two gates – a "reset gate" to determine how much of the previous hidden state should be retained when computing an intermediate hidden state, and an "update gate" to control how much of this intermediate state should be used compared to the previous hidden state when computing the new hidden state (Chen, 2020). This approach also allows for the retainment of the context of earlier parts of a sentence and is similarly effective to the LSTM implementation. Thus, these variations of the original RNN architecture illustrate how RNNs are a flexible modeling technique that can be extended to improve their functionality and modeling power.

With this theoretical framework for RNNs established, it is worth exploring existing software packages that make them usable in practice. For example, Python has multiple libraries that provide effective implementations of popular RNN architectures. From the PyTorch documentation, one can use the "torch.nn.RNN" to define a RNN with the original architecture described earlier, in which a user can specify parameters like the input size, number of hidden state features, activation function, number of recurrent layers (in order to form a stacked RNN), and whether to make the RNN bidirectional (in which we stack an additional RNN that processes the input in the reverse direction). Similar options for customizability are offered by the "torch.nn.LSTM" and "torch.nn.GRU" classes, which implement the variations described in the previous paragraph. Another competing option is TensorFlow, which has its own implementations of these recurrent architectures. As defined by the TensorFlow documentation, the "tf.keras.layers.RNN" class defines the base RNN, as well as the "tf.keras.layers.LSTM" and "tf.keras.layers.GRU" classes for the variations. TensorFlow offers many of the same options to

specify different parameters as PyTorch, with one apparent difference being that bidirectionality is implemented via a separate class ("tf.keras.layers.Bidirectional") in TensorFlow, whereas it is a parameter in PyTorch. There are certain architectural differences between the two libraries that go beyond the scope of this paper, but in general, both are highly effective implementations that will be well-suited for a large variety of applications.

In sum, recurrent neural networks offer a powerful language modeling approach with their ability to accept variable-length input sequences and capture long-range dependencies by feeding the previous hidden state through the network. The main disadvantage with the standard implementation is the vanishing gradient problem, but this can be addressed with either LSTMs or GRUs, both of which utilize the idea of gating to control how much of the previous input and hidden state to remember. All three of these approaches are implemented in the popular PyTorch and TensorFlow libraries for Python, thereby allowing them to be easily used in any application that requires a language model. Thus, due to their effective modeling approach and availability in trusted software libraries, RNNs make an excellent choice of a language modeling technique.

References

Chen, Qiurui. “Language Models and RNN.” *Medium*, Medium, 11 May 2020, https://medium.com/@rachel_95942/language-models-and-rnn-c516fab9545b.

Le, James. “Recurrent Neural Networks: The Powerhouse of Language Modeling.” *Built In*, 13 May 2019, https://builtin.com/data-science/recurrent-neural-networks-powerhouse-language-modeling.

“Recurrent Neural Networks (RNN) with Keras.” *TensorFlow v2.10.0*, TensorFlow, https://www.tensorflow.org/guide/keras/rnn#built-in_rnn_layers_a_simple_example.

“Torch.nn.” *PyTorch 1.12 Documentation*, PyTorch, https://pytorch.org/docs/stable/nn.html.