Text Generation with Recurrent Neural Networks

Mohit Deshpande, Benjamin Trevor, Brad Pershon

Abstract

In this tutorial, you will be introduced to **recurrent neural networks (RNNs)** as a neural model for language modeling. First, we will briefly review the problem of language modeling. After this review, we can discuss recurrent neural networks as well as how to sample from them. This discussion will also include how the input data should be structured as well as the outputs the RNN produces. After training a model, we will learn how to sample from it. Finally, we will learn about a better variant of the vanilla RNN that will improve the quality of the generated text. Then, we will train a character-based RNN on Shakespearean text.

1 Introduction

Language modeling is the problem of assigning a probability to a word sequence. More formally, given a sequence of words $w_1, \dots, w_n$, we wish to assign some joint probability to the sequence $p(w_1, \dots, w_n)$ that tells us how likely we are to find this sequence in the given language. For example $p(\textit{the cat sat on the mat})$ will have a high probability since that word sequence forms a correct sentence. On the other hand, we expect $p(\textit{mat sat cat the the on})$ to have a very low probability because that word sequence is not grammatically correct.

2 Recurrent Neural Networks

One neural way to learn a language model would be to use a feedforward neural network. However, these cannot handle variable-length sequences of vectors. Instead, we need a recurrence relation. Hence, we use a **recurrent neural network** or **RNN**, shown in Figure 1. Additionally, we can "unroll" this RNN, shown in Figure 2.

There are five learnable parameters in an RNN: W^{xh}, W^{hh}, W^{hy}, b_h, and b_y. h_t represents the hidden state or latent space of the input whose size is a hyperparameter. We initialize h_0 to be the zero vector. Hence, W^{xh} takes our input vector and maps it into the hidden space, much like the hidden layer of a neural network. However, the difference is that we also have an additional set of weights W^{hh} that fold in the previous hidden state, which is a function of the previous hidden state and so on. This allows us to process variable-length

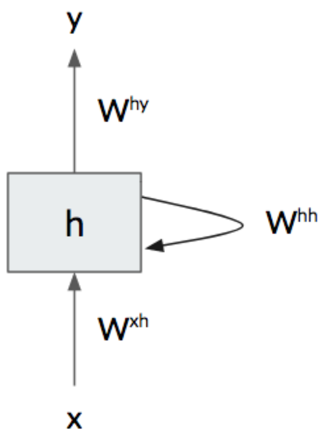

Figure 1: Visual representation of an RNN.

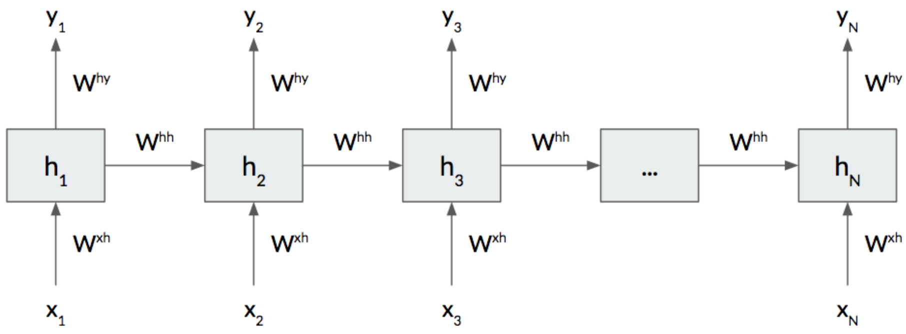

Figure 2: "Unrolled" version of an RNN.

sequences of vectors. (though, for training, we pad these sequences so that we can stack them all together into one matrix for efficiency.)

There are different types of RNNs, shown in Figure 3. Hence, it is not necessary that at each time step the RNN must produce an output or take in an input. For example, for machine translation, we would prefer the last architecture: encoder-decoder many-to-many architecture. For sequence classification, like what we will be doing, we choose the many-to-one architecture.

2.1 Long Short-Term Memory Cell

The largest challenge plain RNNs suffer from is the vanishing gradient problem. Intuitively, this corresponds to its inability to learn long-term dependencies. One solution to this is the Long Short-Term Memory cell, shown in Figure 4.

The most important part of the LSTM cell is the **cell state**, shown as C in the figure. Note that we perform very minimal operations to this cell state: only

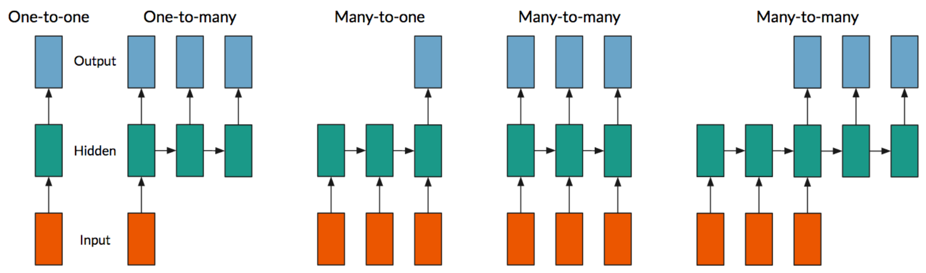

Figure 3: Different variants of RNNs.

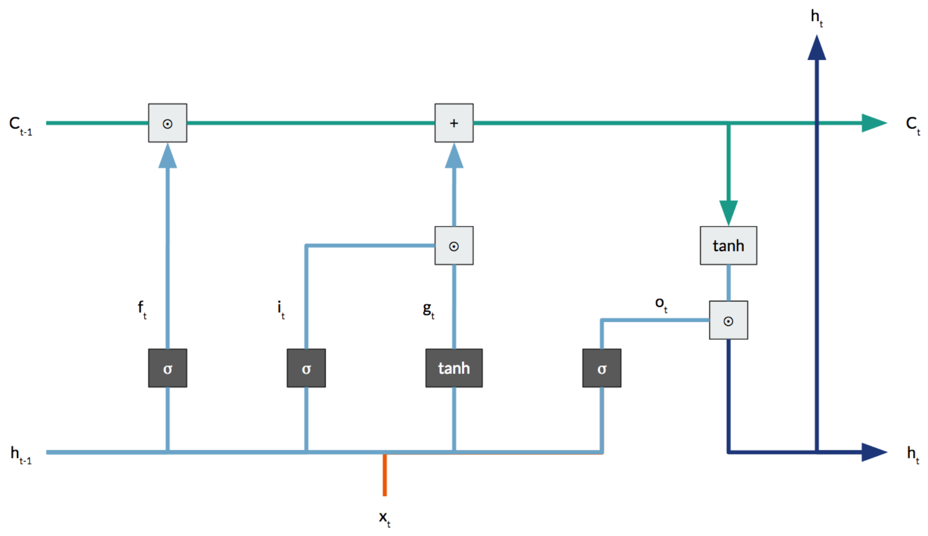

Figure 4: A single long short-term memory cell.

addition and element-wise multiplication. When we perform backpropagation, these two minimal operations just copy the gradient, which doesn't change it, and multiply it by a factor, respectively. This allows most of the unchanged gradient to flow backwards through the time steps which allows the LSTM to learn long-term dependencies.

3 Learning a Generative Language Model

Now we'll actual construct and train an RNN on several different datasets.

Go to this link
`colab.research.google.com/drive/1a4G_ghV6IfW3GOL4tLiGhSqlqQ4FTj5W`
to see the code on Google Colaboratory. The exact same set of lab instructions are also written in the Colaboratory notebook as well.

3.1 Creating Training Data

Since we are using a character-based model, the goal for our network is to classify the next character given the previous 50 characters. So we need to create training data where the input data is 50 characters and the output is the next character.

Split the corpus into overlapping blocks (simply shift over by 1 character each time) of 50 characters into `sentences`. Then extract the next character after those 50 into `next_chars`. The `sentences` variable should be a list of strings, and `next_chars` should be a list of single characters.

(Hint: use string slicing! e.g., `string[x:x+10]` extracts 10 characters starting at position `x`)

Now that we have string representations of the characters, we need to convert these string/character representations into numerical representations using our character ”embeddings”. These ”embeddings” are really just one-hot character embeddings. Remember that `X[i]` refers to a particular sentence, i.e., a sequence of character vectors. And `X[i,j]` refers to a particular character vector in a particular sentence. Use the `char_to_idx` dictionary to convert from characters to their index in the character vector.

Note: If you get stuck there are answers at the end of the tutorial separated by section.

4 Building the Model

Now that we have our training data, we need to create our RNN model. For this tutorial, we will be using Keras (`https://keras.io/`), an API for working with neural networks in Python.

Our base model will be a sequential model, which is a linear sequence of layers. Our model will have two layers, the first of which will be our RNN. For this model, rather than the simple RNN, we will use a Long Short-Term Memory Unit or LSTM. Read the tutorial for more information on LSTMs. In Keras, the output of this layer will simply be the output at the final time step, which is customary for sequence classification. The next layer will be a fully-connected layer that performs the classification over the characters: this is the classification layer that will produce a probability distribution over all of the unique characters.

Steps:

1. Create the base sequential model
2. Add a 256-unit LSTM layer to the model (make sure to use the correct input shape!)

3. Add a dense/fully connected layer to perform classification (make sure to use the correct number of neurons and the correct activation function!)

4. Compile the model using categorical cross-entropy as the loss function and the Adam optimizer with the default parameters of the Adam optimizer.

(Hint: In Keras this will be 3 lines of code)

For reference:

- `https://keras.io/getting-started/sequential-model-guide/`
- `https://keras.io/layers/recurrent/`
- `https://keras.io/layers/core/`

5 Sampling from an RNN

Once our model is built and trained – we'll get to training in a second – we need to be able to use it to generate text. That is, after all, why we came here in the first place. We will do this by sampling from the model with an initial seed.

Remember that our model works by taking a "sentence" of the previous 50 characters and, based on that input, predicts the next character. When we start predicting, however, we don't have any characters yet. So, to get the prediction started, we take a "seed sentence" from our corpus and use that for our first prediction. Each character we predict will be concatenated onto the seed, and then, once we've predicted at least 50 characters, our future predictions will be based on our actual generated text rather than text from the original corpus.

Steps:

1. Randomly select a "seed sentence" (sequence of 50 characters) from the corpus (or you may supply your own!)

2. Generate a one-hot character embedding from the seed sentence

3. Have the model predict the next character based on the preceding 50 characters. Note that the prediction will not be a single character – it will be an array of probabilities for each possible next character. Choose the one with the highest probability, i.e., the argmax!

4. Add the highest probability character to the end of our generated text, then add it's one-hot character embedding to the end of the running generated text

5. Repeat steps 3 & 4 until you've built up a string of predicted characters equal to the sample length

Hints:

- The seed should be of size `(1, 50, vocab_size)` where 1 represents the batch size.
- Use `model.predict(...)` with the runnning seed as input and use `idx_to_char` to convert it back to a character
- You'll need to encode that character as a one-hot character (the size should be `(1, 1, vocab_size)`)
- Use `np.concatenate` in combination with numpy array slicing to attach the model's prediction to the seed while retaining the `(1, 50, vocab_size)` size.

Next, create a subclass of Keras's `Callback` class called `SamplerCallback`. A `Callback` class is used to provide some sort of feedback during model training. These callbacks have hooks that we can use to run code during training, e.g., `on_epoch_{begin/end}`, `on_batch_{begin/end}`, and `on_training_{begin/end}`. In this case, we can create one that will give us sample text after each training epoch. If our model is being trained properly, at the end of each training epoch, the sampled text from the model should look more and more like real text – up until a certain point, of course! We can't train a perfect model, after all.

To do this, simply define a function in your Callback class named `on_epoch_end` that takes three parameters: `self`, `epoch`, and `logs` which will be called automatically at the end of each epoch. This function should sample text from the model using our previously-defined `sample_from_model` function and then print it. To access the current model being trained, use `self.model` rather than our global model. (This aids in re-usability!) Then we just need to create an instance of this class and pass it as a `Callback` argument when we train our model.

6 Training the RNN

Finally, we need to train our RNN model by fitting it to our training data, which, in case you've forgotten, consists of a sequence of "sentences" and a sequence of next characters. We will train our model to learn weights such that, given the "sentence" input, it produces the correct next character as output. Our model will train for a certain number of epochs – each epoch is one full pass over all the training data – using the specified optimizer to adjust the weights in the model to optimize the specified loss function when applied to the training data.

To fit our model to the training data, we will need to specify the following parameters:

- Training input (a numpy array)
- Training output (a numpy array)
- Number of training epochs (try 30)

- A batch size indicating the number of training samples taken in for each update of the model weights (try 256)
- A list of callbacks (see previous section)

For reference, see: `https://keras.io/models/model/`

7 Sample from the Trained Model

Now that our model is trained, we can see how well it did. Sample 1000 characters from the model using the `sample_from_model` function and print it. How does it look? Does it look like Shakespeare? Does it look like English? Is it pronouncable?

Hopefully the answer to all these questions is yes. Even this simple character-based model should be able to produce mostly actual English words, and Shakespearean sounding ones at that. The grammar might be a little questionable, but it shouldn't be terrible either. Given that this is a character-based model with no inherent knowledge of the concept of "words," that's pretty amazing!

8 Extra Fun

If you've gotten this far, you can:

- Try training your model with a different input dataset. We know that LSTMs can write Shakespeare, but what about C code? Use the other dataset and re-run all of the sections to see how well character-level models fair against code. **What if, instead of a character-level model, we tried to use a word-level model for C code? What do you expect the quality of this to be compared to a character-level model on C code?**
- Play around with the model parameters to see how they affect your output. What happens if you increase/decrease the number of hidden units in your LSTM? Or use a different optimizer? In the beginning, we organized our training data so that our model could predict the next character based on the previous 50 characters. What if it was 100 characters? What if it was 10? Do you think the sampled output would get better or worse? What are the tradeoffs to using a larger sequence?
- Recent literature seems to favor GRUs and 1D convolutional networks over LSTM because they produce similar quality results but are more efficient to train. Try these models. Do they perform better than the LSTM?

9 Answers

9.1 Creating Training Data

```
sentence_length = 50
sentences = []
next_chars = []

for i in range(data_size - sentence_length):
    sentences.append(corpus[i: i + sentence_length])
    next_chars.append(corpus[i + sentence_length])
```

9.2 Numerical Representations

```
for i, sentence in enumerate(sentences):
    for t, char in enumerate(sentence):
        X[i, t, char_to_idx[char]] = 1
    y[i, char_to_idx[next_chars[i]]] = 1
```

9.3 Building the Model

```
model.add(LSTM(256, input_shape=(sentence_length, vocab_size)))
model.add(Dense(vocab_size, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam')
```

9.4 Sample from Model

```
def sample_from_model(model, sample_length=100):
    generated_text = ''
    seed = randint(0, data_size - sentence_length)
    seed_sentence = corpus[seed: seed + sentence_length]

    X_pred = np.zeros((1, sentence_length, vocab_size), dtype=np.bool)
    for t, char in enumerate(seed_sentence):
        X_pred[0, t, char_to_idx[char]] = 1

    for i in range(sample_length):
        prediction = np.argmax(model.predict(X_pred))

        generated_text += idx_to_char[prediction]

        embedded_char = np.zeros((1, 1, vocab_size), dtype=np.bool)
        embedded_char[0, 0, prediction] = 1
        X_pred = np.concatenate((X_pred[:, 1:, :], embedded_char),
            axis=1)
    return generated_text
```

9.5 Sampler Callback

```
class SamplerCallback(Callback):
    def on_epoch_end(self, epoch, logs):
        generated_text = sample_from_model(self.model)
        print('\nGenerated text')
        print('-' * 32)
        print(generated_text)

sampler_callback = SamplerCallback()
```

9.6 Training the RNN

```
model.fit(X, y, epochs=30, batch_size=256, callbacks=[sampler_callback])
```

9.7 Sample from the Trained Model

```
generated_text = sample_from_model(model, sample_length=1000)
print('\nGenerated text')
print('-' * 32)
print(generated_text)
```