

[Get started](#)[Open in app](#)[Follow](#)

579K Followers



This is your **last** free member-only story this month. [Sign up for Medium and get an extra one](#)

[PART 1](#) | [PART 2](#) | [PART 3](#) | NATURAL LANGUAGE PROCESSING

Create Your Own Artificial Shakespeare in 10 Minutes with Natural Language Processing

Generate Shakespearean style literary work using recurrent neural networks and Shakespeare corpus



Orhan G. Yalçın Jan 27 · 11 min read ★

[Get started](#)[Open in app](#)

Figure 1. Photo by [Jessica Pamp](#) on [Unsplash](#)

I am sure you've come across dozens of newspaper articles talking about AI-generated text from news articles to poems, from novels to narrations. Wouldn't it be cool if you can actually build a model that does that? Well, this is what we are gonna do in this post. We will build an RNN network that can generate text.

The research shows that one of the most effective artificial neural network types for Natural Language Processing tasks is Recurrent Neural Networks (RNNs). RNNs are widely used in NLP tasks such as machine translation, text generation, image captioning. In NLP tasks, we usually use NLP tools and methods to process the text data into vectors and then feed them into a selected artificial neural network such as RNN, CNN, or even a feedforward neural network to complete a task. In our post, we will follow these two standardized steps: (i) process the text into vectors and (ii) train a neural network with these vectors.

[Get started](#)[Open in app](#)

sequential data since text data has a sequential nature.

A recurrent neural network (RNN) is a class of artificial neural networks where connections between neurons form a temporal sequence.

RNNs capable of capturing dynamic temporal information (temporal memory).

Recurrent Neural Networks are derived from Feedforward Neural Networks, but they offer much more.

Why Not Feedforward Neural Networks?

There are three main limitations of plain feedforward neural networks that make them unsuitable for sequence data:

- A feedforward neural network cannot take the order into account;
- A feedforward neural network requires a fixed input size;
- A feedforward neural network cannot output predictions in different lengths.

One of the fundamental characteristics of text data is the significance of its order. Just like rearranging the order of monthly sales can lead us from an increasing trend to a decreasing trend, rearranging the order of the words in a sentence can change or distort its meaning entirely. This is where the feedforward neural network's limitation surfaces. In a feedforward neural network, the order of the data cannot be taken into account due to this limitation. Rearranging the order of monthly sales would give the same result, which proves that they cannot make use of the order of the inputs.

Get started

Open in app

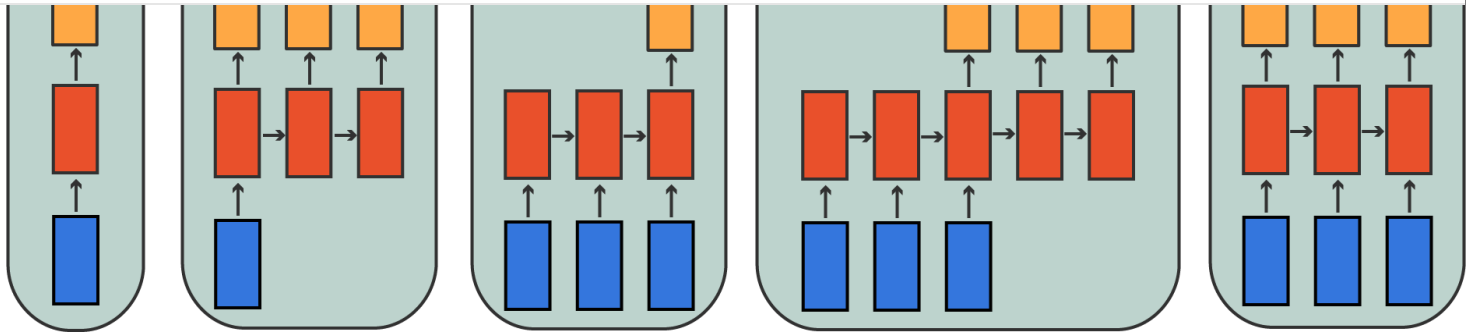


Figure 2. Potential Sequence Data Tasks in Deep Learning (Figure by Author)

Recurrent Neural Networks to the Rescue

RNNs use previous information by keeping them in memory, which is saved as a “state” within an RNN neuron. There are several types of RNNs, such as Vanilla RNN, GRU, and LSTM.

Let’s talk about the memory structure with a basic weather forecasting example. We would like to guess if it will rain by using the information provided in a sequence. This sequence of data may be derived from text, speech, or video. After each new information, we slowly update the probability of rainfall and reach a conclusion in the end. Here is the visualization of this task in Figure 3:

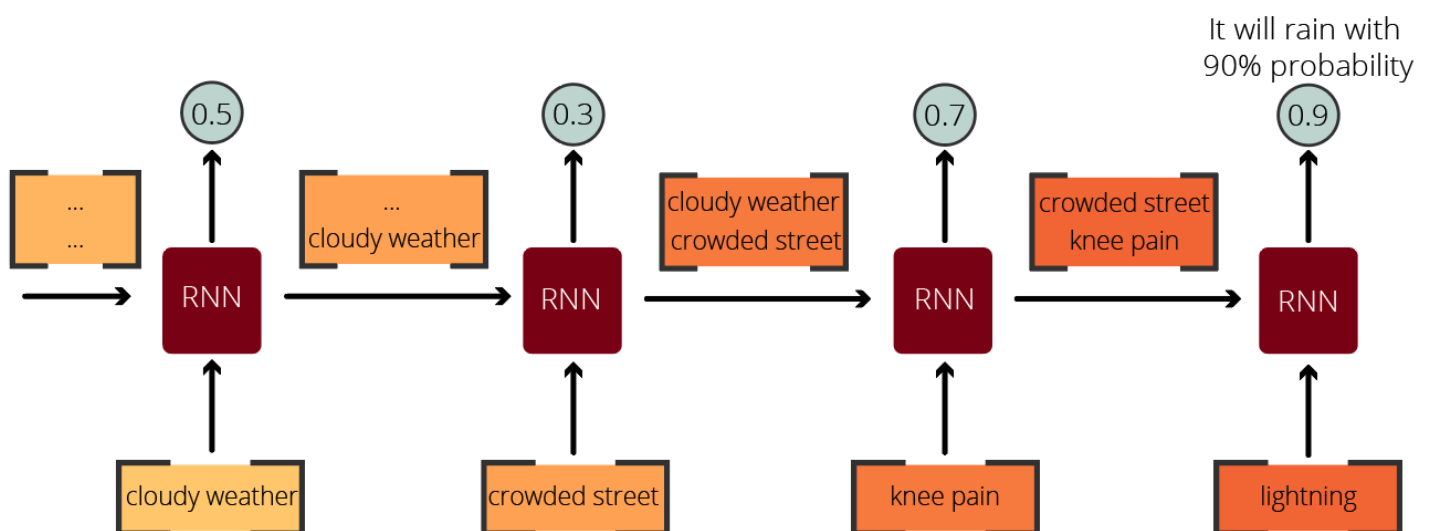


Figure 3. A Simple Weather Forecasting Task: Will it Rain? (Figure by Author)

In Figure 3, we first record that there is cloudy weather. This single information might

Get started

Open in app



outside, which means less likelihood of rainfall and, therefore, our estimation drops to 30% (or 0.3). Then, we are provided with more information: knee pain. It is believed that people with rheumatism feel knee pain before it rains. Therefore, my estimation rises to 70% (or 0.7). Finally, when our model takes lightning as the latest information, the collective estimation increases to 90% (or 0.9). At each time interval, our neuron uses its memory -containing the previous information and adds the new information on top of this memory to calculate the likelihood of rainfall. The memory structure can be set at the layer level as well as at the cell level. Figure 4 shows a cell level RNN mechanism, (i) folded version on the left and (ii) unfolded version on the right.

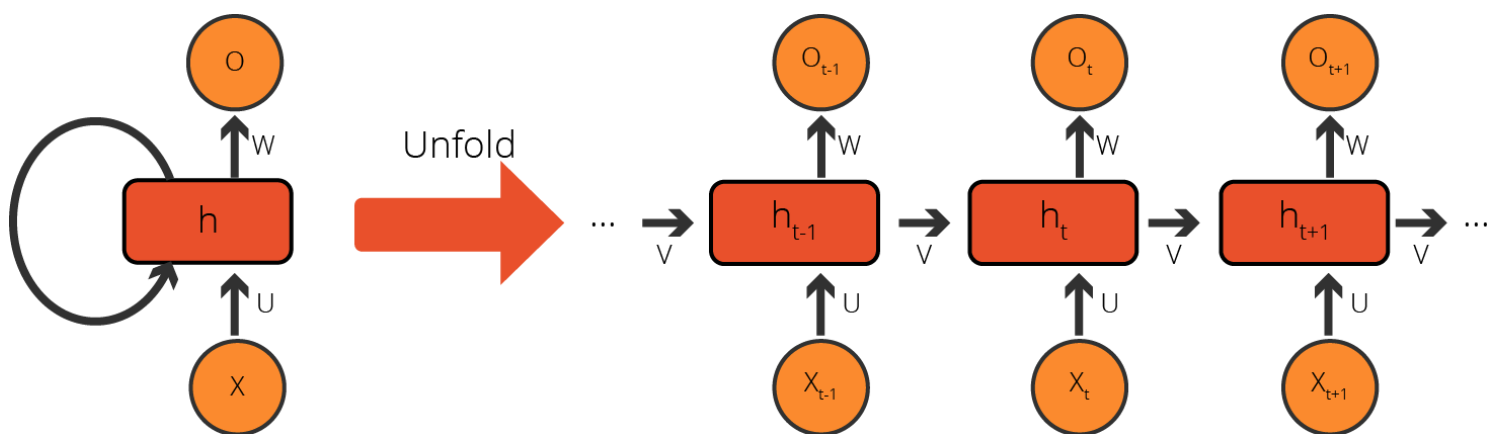


Figure 4. A Cell-Based Recurrent Neural Network Activity (Figure by Author)

RNN Types

As mentioned earlier, there are many different variants of RNNs. In this section, we will cover three RNN types we encounter often:

- Simple (Simple) RNN
- Long Short-term Memory (LSTM) Networks
- Gated Recurrent Unit (GRU) Networks

You can find the visualization of these alternative RNN cells in Figure 5:

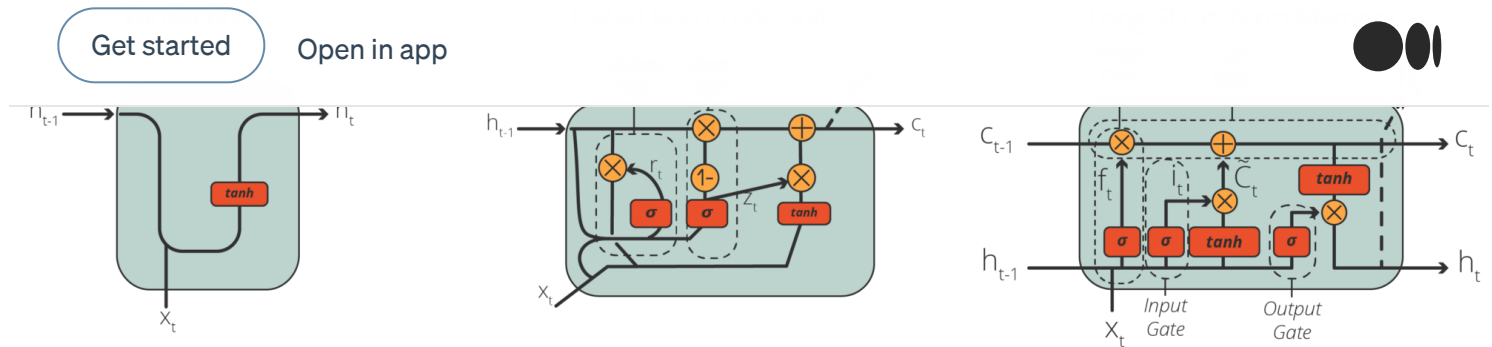


Figure 5.3 Popular Variations of Recurrent Neural Networks: Vanilla RNN, GRU, LSTM (Figure by Author)

Gated Recurrent Units (GRUs) — Gated Recurrent Units are introduced in 2014 by Kyunghyun Cho. Just as LSTMs, GRUs are also gating mechanism in RNNs to deal with sequence data. However, to simplify the calculation process, GRUs use two gates: (i) Reset Gate and (ii) Update Gate. GRUs also use the same values for hidden state and cell state. Figure 5 shows the inner structure of a Gated Recurrent Unit:

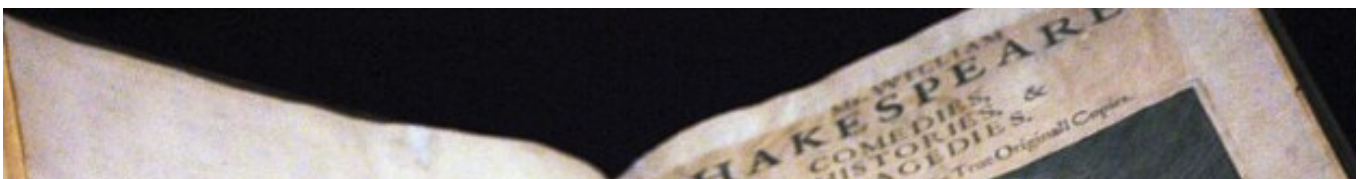
Neural Text Generation with Shakespeare Corpus

In this case study, our goal is to train an RNN capable of generating meaningful text from characters. An RNN can generate text from words as well as from characters, and we select to use characters to generate text for this case study.

When we build a new RNN with no training, it combines a bunch of meaningless characters, which does not mean anything. However, if we feed our RNN with a lot of text data, it starts to imitate these texts' style and generate meaningful text using characters.

So, if we feed the model a lot of didactic text, our model would generate educational materials. If we feed our model with lots of poems, our model will generate poems, so we would have an artificial poet. These are all viable options, but we will feed our model with something else: A long text dataset containing Shakespeare's writings. Therefore, we will create a Shakespeare bot.

Shakespeare Corpus



[Get started](#)[Open in app](#)

Figure 6. A Shakespeare Book on [Wikimedia](#)

Shakespeare Corpus is a text file containing 40,000 lines of Shakespeare's writing, cleaned and prepared by Karpathy and hosted by the TensorFlow team. I strongly recommend you take a look at the `.txt` file to understand the text we are dealing with. The file contains the conversational content where each character's name is placed before the corresponding part, as shown in Figure 7.

Second Citizen:

Consider you what services he has done for his country?

First Citizen:

Very well; and could be content to give him good report fort, but that he pays himself with being proud.

Second Citizen:

Nay, but speak not maliciously.

First Citizen:

I say unto you, what he hath done famously, he did it to that end: though soft-conscienced men can be content to say it was for his country he did it to please his mother and to be partly proud; which he is, even till the altitude of his virtue.

Second Citizen:

What he cannot help in his nature, you account a

[Get started](#)[Open in app](#)

Initial Imports

In this case study, the required libraries are TensorFlow, NumPy, and os, which we can import with the following code:

```
1 import tensorflow as tf
2 import numpy as np
3 import os
```

nlp_inimp.py hosted with ❤ by GitHub

[view raw](#)

To load a dataset from an online directory, we can use the Keras API's util module in TensorFlow. For this task, we will use the `get_file()` function, which downloads a file from a URL if it not already in the cache, with the following code:

```
1 path_to_file = tf.keras.utils.get_file('shakespeare.txt',
2                                         'https://storage.googleapis.com/download.tensorflow.org/d...
```

nlp_loadcorpus.py hosted with ❤ by GitHub

[view raw](#)

After downloading our file, we can read the file from the cache with the following Python code. Now, we successfully saved the entire corpus in the Colab notebook's memory as a variable. Let's see how many characters there in the corpus are and what's the first 100 characters, with the code below:

```
1 # Read, then decode for py2 compat.
2 text = open(path_to_file, 'rb').read()
3 text = text.decode(encoding='utf-8')
4 print ('Total number of characters in the corpus is:', len(text))
5 print('The first 100 characters of the corpus are as follows:\n', text[:100])
```

nlp_readcorpus.py hosted with ❤ by GitHub

[view raw](#)

```
Total number of characters in the corpus is: 1115394
The first 100 characters of the corpus are as follows:
First Citizen:
```


[Get started](#)[Open in app](#)

All:
Speak, speak.

First Citizen:
You

Figure 8. The first 100 characters of the corpus (Figure by Author)

Our entire corpus is accessible via a Python variable named `text`. And now we can start vectorizing it.

Vectorizing the Text

Text Vectorization is a fundamental NLP method to transform text data into a meaningful vector of numbers so that a machine can understand. There are various approaches to text vectorization. In this case study, step by step, this is how we go about this:

- Give an index number to each unique character;
- Run a for loop in the corpus and index every character in the whole text.

To give each unique character an index number, we first have to find all the unique characters in the text file. This is very easy with the built-in `set()` function, which converts a list object to a set object.

The difference between set and list data structures is that lists are ordered and allow duplicates while sets are unordered and don't allow duplicate elements. So, when we run the `set()` function -as shown in the below code-, it returns a set of unique characters in the text file.

```
1 # The unique characters in the corpus
2 vocab = sorted(set(text))
3 print('The number of unique characters in the corpus is', len(vocab))
4 print('A slice of the unique characters set:\n', vocab[:10])
```

nlp_uniques.py hosted with ❤ by GitHub

[view raw](#)

Get started

Open in app



The number of unique characters in the corpus is 65
 A slice of the unique characters set:
 ['\n', ' ', '!', '\$', '&', '"', ',', '-', '.', '3']

Figure 9. A Slice of the Unique Characters List (Figure by Author)

We also need to give each character an index number. The following code assigns a number to each set item, then creates a dictionary of the set items with their given numbers. We also make a copy of the unique set elements in the NumPy array format for later use in decoding the predictions. Then, we can vectorize our text with a simple for loop where we go through each character in the text and assign their corresponding index value and save all the index values as a new list, with the following code:

```
1 # Create a mapping from unique characters to indices
2 char2idx = {u:i for i, u in enumerate(vocab)}
3 # Make a copy of the unique set elements in NumPy array format for later use in the decoding the
4 idx2char = np.array(vocab)
5 # Vectorize the text with a for loop
6 text_as_int = np.array([char2idx[c] for c in text])
```

nlp_vectorize.py hosted with ❤ by GitHub

[view raw](#)

Creating the Dataset

At this point, we have our `char2idx` dictionary to vectorize the text and `idx2char` to de-vectorize (*i.e., decode*) the vectorized text. Finally, we have our `text_as_int` as our vectorized NumPy array. We can now create our dataset.

Firstly, we will use the `from_tensor_slices` method from the Dataset module to create a TensorFlow Dataset object from our `text_as_int` object, and we will split them into batches. The length of each input of the dataset is limited to 100 characters. We can achieve all of them with the following code:

```
1 # Create training examples / targets
2 char_dataset = tf.data.Dataset.from_tensor_slices(text_as_int)
3 # for i in char_dataset.take(5):
```

[Get started](#)[Open in app](#)

```
7 sequences = char_dataset.batch(seq_length+1, drop_remainder=True)
8 # for item in sequences.take(5):
9 #     print(repr(''.join(idx2char[item.numpy()])))
```

nlp_datacr1.py hosted with ❤ by GitHub

[view raw](#)

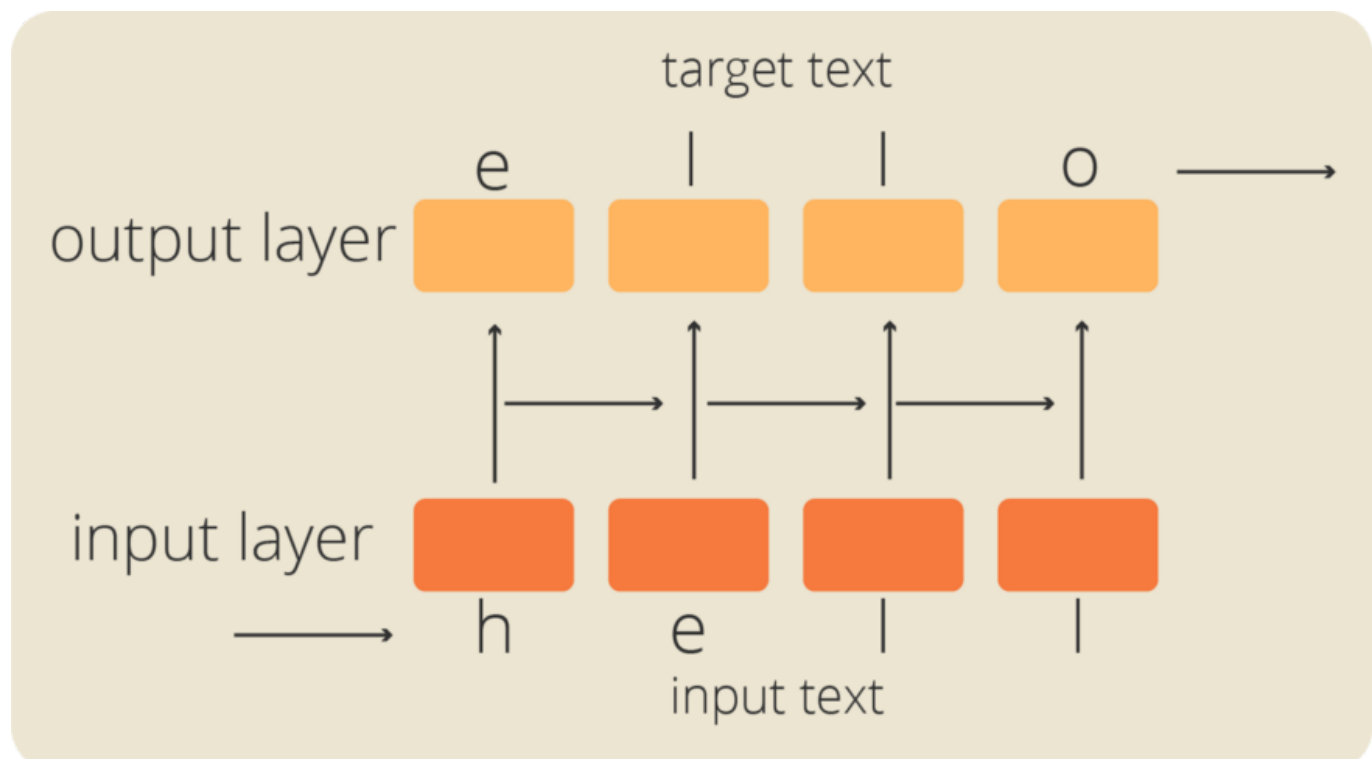
Our sequence object contains sequences of characters, but we have to create a tuple of these sequences to feed into the RNN model. We can achieve this with the custom mapping function below:

```
1 def split_input_target(chunk):
2     input_text = chunk[:-1]
3     target_text = chunk[1:]
4     return input_text, target_text
5
6 dataset = sequences.map(split_input_target)
```

nlp_datacr2.py hosted with ❤ by GitHub

[view raw](#)

The reason that we generated these tuples is that for RNN to work, we need to create a pipeline, as shown in Figure 10:



[Get started](#)[Open in app](#)

Finally, we shuffle our dataset and split it into 64 sentence batches with the following lines:

```
1  BUFFER_SIZE = 10000 # TF shuffles the data only within buffers
2
3  BATCH_SIZE = 64 # Batch size
4
5  dataset = dataset.shuffle(BUFFER_SIZE).batch(BATCH_SIZE, drop_remainder=True)
6
7  print(dataset)
```

nlp_datacr3.py hosted with ❤ by GitHub

[view raw](#)

Building the Model

Our data is ready to be fed into our model pipeline. Let's create our model. We want to train our model and then make new predictions. Firstly, let's set some parameters with the following code:

```
1  # Length of the vocabulary in chars
2  vocab_size = len(vocab)
3  # The embedding dimension
4  embedding_dim = 256
5  # Number of RNN units
6  rnn_units = 1024
```

nlp_setparams.py hosted with ❤ by GitHub

[view raw](#)

Now, what is important about this is that our training pipeline will feed 64 sentences at each batch. Therefore, we need to build our model to accept 64 input sentences at a time. However, after we trained our model, we would like to input single sentences to generate new tasks. So, we need different batch sizes for pre-training and post-training models. To achieve this, we need to create a function, which allows us to reproduce models for different batch sizes. The following code does this:

```
1  def build_model(vocab_size, embedding_dim, rnn_units, batch_size):
2      model = tf.keras.Sequential([
3          tf.keras.layers.Embedding(vocab_size, embedding_dim,
```

[Get started](#)[Open in app](#)

```

6         return_sequences=True,
7         stateful=True,
8         recurrent_initializer='glorot_uniform'),
9     tf.keras.layers.Dense(vocab_size)
10 ])
11 return model

```

nlp_buildmodel.py hosted with ❤ by GitHub

[view raw](#)

There are three layers in our model:

- **An Embedding Layer:** This layer serves as the input layer, accepting input values (*in number format*) and convert them into vectors.
- **A GRU layer:** An RNN layer filled with 1024 Gradient Descent Units
- **A Dense layer:** To output the result, with `vocab_size` outputs.

Now we can create our model for training with the following code:

```

1 model = build_model(
2     vocab_size = len(vocab), # no. of unique characters
3     embedding_dim=embedding_dim, # 256
4     rnn_units=rnn_units, # 1024
5     batch_size=BATCH_SIZE) # 64 for the traning
6
7 model.summary()

```

nlp_buildmodel2.py hosted with ❤ by GitHub

[view raw](#)

Here is the summary of our model in Figure 11:

Model: "sequential"

Layer (type)	Output Shape	Param #
embedding (Embedding)	(64, None, 256)	16640
gru (GRU)	(64, None, 1024)	3938304
dense (Dense)	(64, None, 65)	66625

[Get started](#)[Open in app](#)

Non-trainable params: 0

Figure 11. The Summary View of the Training Model (Figure by Author). Note the 64 in the output shapes, which must be 1 for individual predictions after training

Compiling and Training

To compile our model, we need to configure our optimizer and loss function. For this task, we select `Adam` as our optimizer and sparse categorical crossentropy function as our loss function.

Since our output is always one of the 65 characters, this is a multiclass categorization problem. Therefore, we have to choose a categorical crossentropy function. However, in this example, we select a variant of categorical crossentropy: Sparse Categorical Crossentropy. We use sparse categorical crossentropy because even though they use the same loss function, their output formats are different. Remember we vectorized our text as integers (e.g., `[0]`, `[2]`, `[1]`), not in one-hot encoded format (e.g., `[0,0,0]`, `[0,1,]`, `[1,0,0]`). To be able to output integers, we must use a sparse categorical crossentropy function.

To be able to set the customize our loss function, we are creating a basic function containing sparse categorical crossentropy loss:

Now we can set our loss function and optimizer with the following code:

```
1 def loss(labels, logits):
2     return tf.keras.losses.sparse_categorical_crossentropy(labels, logits, from_logits=True)
3
4 # example_batch_loss = loss(target_example_batch, example_batch_predictions)
5 # print("Prediction shape: ", example_batch_predictions.shape, " (batch_size, sequence_length, voc
6 # print("scalar_loss:      ", example_batch_loss.numpy().mean())
7
8 model.compile(optimizer='adam', loss=loss)
```

nlp_losscomp.py hosted with ❤ by GitHub

[view raw](#)

Get started

Open in app



```

1  # Directory where the checkpoints will be saved
2  checkpoint_dir = './training_checkpoints'
3  # Name of the checkpoint files
4  checkpoint_prefix = os.path.join(checkpoint_dir, "ckpt_{epoch}")
5
6  checkpoint_callback=tf.keras.callbacks.ModelCheckpoint(
7      filepath=checkpoint_prefix,
8      save_weights_only=True)

```

nlp_ckpt.py hosted with ❤ by GitHub

[view raw](#)

Our model and checkpoint directory are configured. We will train our model for 30 epochs and save the training history to a variable called history, with the following code:

```

1  EPOCHS = 30
2  history = model.fit(dataset,
3                      epochs=EPOCHS,
4                      callbacks=[checkpoint_callback])

```

nlp_train.py hosted with ❤ by GitHub

[view raw](#)

```

Epoch 24/30
172/172 [=====] - 11s 57ms/step - loss: 0.7550
Epoch 25/30
172/172 [=====] - 11s 57ms/step - loss: 0.7340
Epoch 26/30
172/172 [=====] - 11s 58ms/step - loss: 0.7176
Epoch 27/30
172/172 [=====] - 11s 58ms/step - loss: 0.7007
Epoch 28/30
172/172 [=====] - 11s 58ms/step - loss: 0.6880
Epoch 29/30
172/172 [=====] - 11s 58ms/step - loss: 0.6743
Epoch 30/30
172/172 [=====] - 11s 58ms/step - loss: 0.6628

```

Figure 12. The Last Eight Epochs of the Model Training (Figure by Author)

Thanks to the model's simplicity and how we encode our model, our training does not take too long (*around 3–4 minutes*). Now we can use the saved weights and build a

[Get started](#)[Open in app](#)

GENERATING NEW TEXT

To be able to view the location of our latest checkpoint, we need to run the following code:

```
1 tf.train.latest_checkpoint(checkpoint_dir)
```

nlp_retrieveckpt.py hosted with ❤ by GitHub

[view raw](#)

Now we can use the custom function we created earlier to build a new model with `batch_size=1`, `load_weights` using the weights saved in the *latest_checkpoint*, use the `build` function to build the model based on input shapes received (i.e., `[1, None]`). We can achieve all of these and `summarize()` the new model with the following code below:

```
1 model = build_model(vocab_size, embedding_dim, rnn_units, batch_size=1)
2 model.load_weights(tf.train.latest_checkpoint(checkpoint_dir))
3 model.build(tf.TensorShape([1, None]))
4 model.summary()
```

nlp_predmodel.py hosted with ❤ by GitHub

[view raw](#)

Model: "sequential_1"

Layer (type)	Output Shape	Param #
=====		
embedding_1 (Embedding)	(1, None, 256)	16640

gru_1 (GRU)	(1, None, 1024)	3938304

dense_1 (Dense)	(1, None, 65)	66625
=====		
Total params: 4,021,569		
Trainable params: 4,021,569		
Non-trainable params: 0		

Figure 13. The Summary View of the Newly Created Model (Figure by Author). Now it accepts single inputs.

Our model is ready to make predictions, and all we need is a custom function to prepare our input for the model. We have to set the following:

[Get started](#)[Open in app](#)

vectorizing the input (from string to numbers),

- An empty variable to store the result,
- A temperature value to manually adjust variability of the predictions,
- Devectorizing the output and also feeding the output to the model again for the next prediction,
- Joining all the generated characters to have a final string.

This custom function below does all of these:

```
1  def generate_text(model, num_generate, temperature, start_string):
2      input_eval = [char2idx[s] for s in start_string] # string to numbers (vectorizing)
3      input_eval = tf.expand_dims(input_eval, 0) # dimension expansion
4      text_generated = [] # Empty string to store our results
5      model.reset_states() # Clears the hidden states in the RNN
6
7      for i in range(num_generate): #Run a loop for number of characters to generate
8          predictions = model(input_eval) # prediction for single character
9          predictions = tf.squeeze(predictions, 0) # remove the batch dimension
10
11         # using a categorical distribution to predict the character returned by the model
12         # higher temperature increases the probability of selecting a less likely character
13         # lower --> more predictable
14         predictions = predictions / temperature
15         predicted_id = tf.random.categorical(predictions, num_samples=1)[-1,0].numpy()
16
17         # The predicted character as the next input to the model
18         # along with the previous hidden state
19         # So the model makes the next prediction based on the previous character
20         input_eval = tf.expand_dims([predicted_id], 0)
21         # Also devectorize the number and add to the generated text
22         text_generated.append(idx2char[predicted_id])
23
24     return (start_string + ''.join(text_generated))
```

nlp_gentext.py hosted with ❤ by GitHub

[view raw](#)

[Get started](#)[Open in app](#)

```
1 generated_text = generate_text(  
2     model,  
3     num_generate=500,  
4     temperature=1,  
5     start_string=u"ROMEO")  
6 print(generated_text)
```

nlp_gentext2.py hosted with ❤ by GitHub

[view raw](#)**ROMEO:**

Sir,--God forbid I am past all thine
first: therefore comes, we will, my lord:
Either they see to us have pluck'd
And take you all fust would do and reprobake
All this shall feel their frost:
I am I am in this great duty?

ROMEO:

By a falling traitor from the sea,
My warges will make up my estate marches well,
Thou most noble cousin Hereford's regit against
My green in plate, to the E:
I say, your mistake--
How melt him in his thanks and what doing thy hearts,
The hour's night, and witnell spe

Figure 14. A 500 character-long text generated by our model (Figure by Author)

Final Words


Using Gated Recurrent Unit and Shakespeare Corpus, you built yourself a Shakespearean bot capable of generating text in any length.

Note that our model uses characters, so the miracle of the model is that it learned to create meaningful words from characters. So, do not think that it adds a bunch of unrelated words together. It goes over thousands of words and learns the

[Get started](#)[Open in app](#)

Feel free to play around with temperature to see how you can change the output from more proper words to more distorted words. A higher temperature value would increase the chances of our function to choose less likely characters. When we add them all up, we would have less meaningful results. On the other hand, a low temperature would cause the function to generate text that is simpler and more of a copy of the original corpus.

Let's Get in Touch and Connect

Besides my latest content, I also share my [Google Colab notebooks](#) with my subscribers, containing full codes for every post I published. If you liked this post, consider subscribing to the newsletter: [Subscribe to the Newsletter!](#) 

Since you are reading this article, I am sure that we share similar interests and are/will be in similar industries. So let's connect via [Linkedin](#)! Please do not hesitate to send a contact request! [Orhan G. Yalçın — Linkedin](#)

Thanks to The Startup.

Sign up for The Variable

By Towards Data Science

Every Thursday, the Variable delivers the very best of Towards Data Science: from hands-on tutorials and cutting-edge research to original features you don't want to miss. [Take a look.](#)

[Get this newsletter](#)[NLP](#)[TensorFlow](#)[Recurrent Neural Network](#)[Artificial Intelligence](#)[Text Generation](#)

[Get started](#)[Open in app](#)

Get the Medium app

