blog.floydhub.com

Beginner's Guide on Recurrent Neural Networks with PyTorch

Ajay Uppili Arasanipalai

24-31 minutes

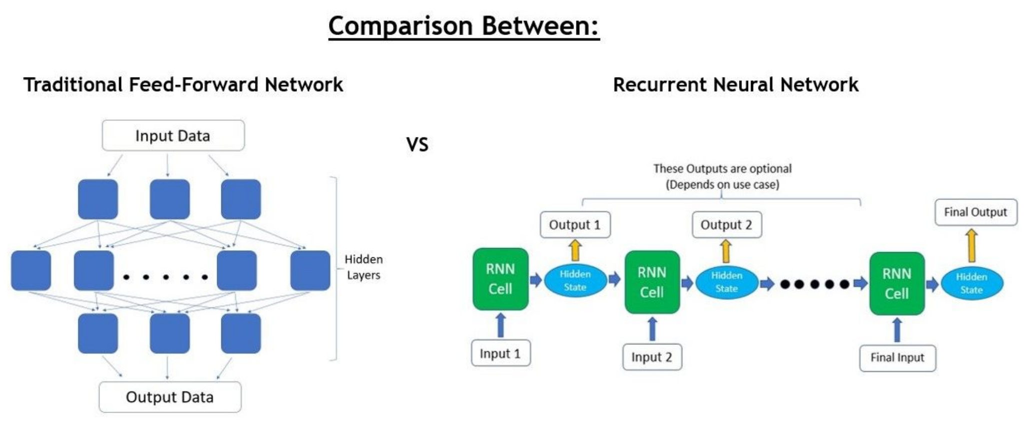
Recurrent Neural Networks(RNNs) have been the answer to most problems dealing with sequential data and Natural Language Processing(NLP) problems for many years, and its variants such as the LSTM are still widely used in numerous state-of-the-art models to this date. In this post, I'll be covering the basic concepts around RNNs and implementing a plain vanilla RNN model with PyTorch to generate text.

Although the content is introductory, the post assumes that you at least have a basic understanding of normal feed-forward neural nets.

Without further ado, let's jump right into it!

Basic Concepts

What exactly are RNNs? First, let’s compare the architecture and flow of RNNs vs traditional feed-forward neural networks.

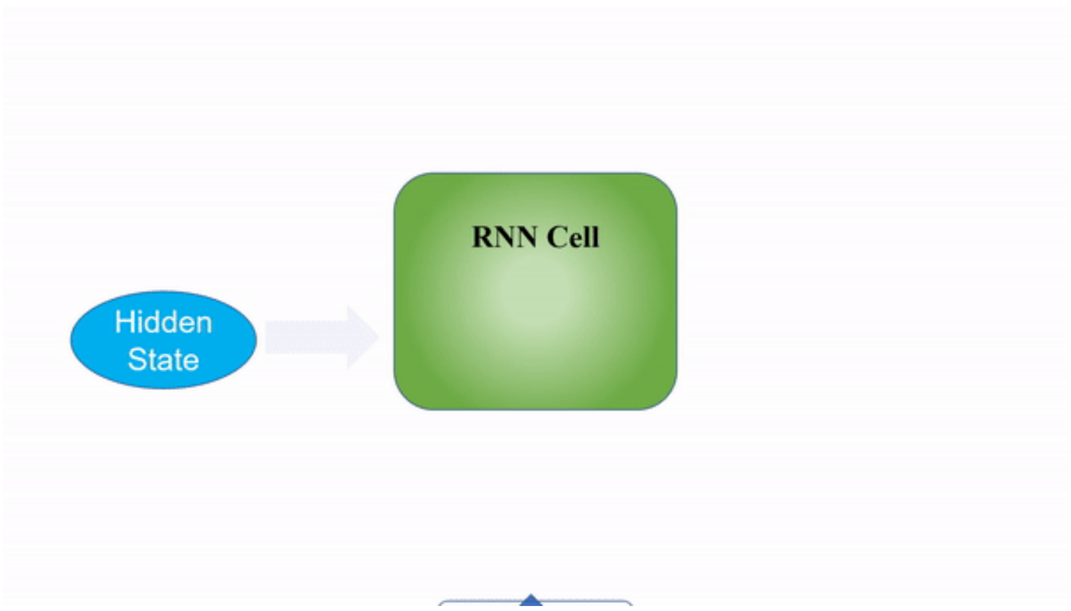


Overview of the feed-forward neural network and RNN structures

The main difference is in how the input data is taken in by the model.

Traditional feed-forward neural networks take in a fixed amount of input data all at the same time and produce a fixed amount of output each time. On the other hand, RNNs do not consume all the input data at once. Instead, they take them in one at a time and in a sequence. At each step, the RNN does a series of calculations before producing an output. The output, known as the hidden state, is then combined with the next input in the sequence to produce another output. This process continues until the model is programmed to finish or the input sequence ends.

Still confused? Don't anguish yet. Being able to visualize the flow of an RNN really helped me understand when I started on this topic.



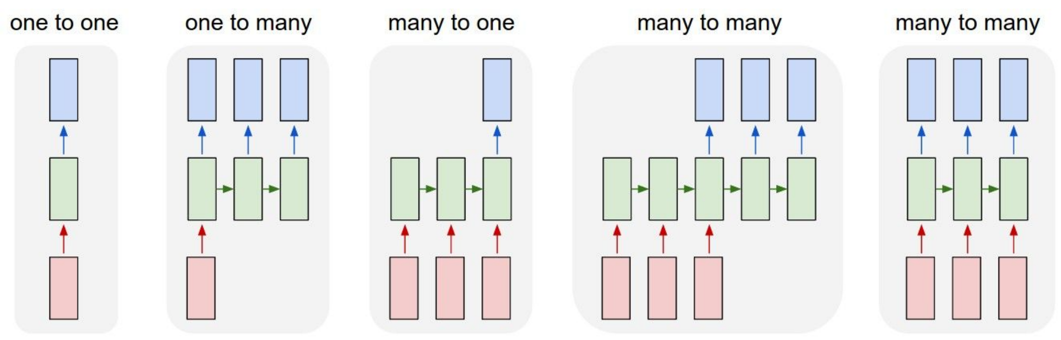
Simple process flow of an RNN cell

As we can see, the calculations at each time step consider the context of the previous time steps in the form of the hidden state. Being able to use this contextual information from previous inputs is the key essence to RNNs' success in sequential problems.

While it may seem that a different RNN cell is being used at each time step in the graphics, the underlying principle of Recurrent Neural Networks is that **the RNN cell is actually the exact same one and reused throughout.**

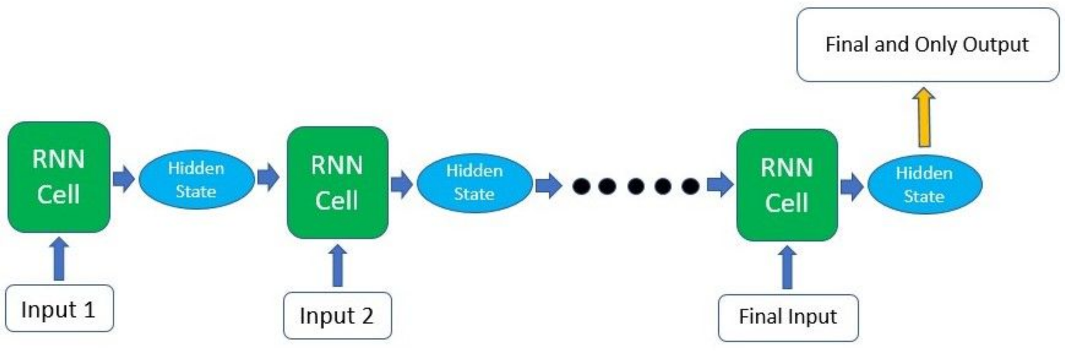
Processing RNN Outputs?

You might be wondering, which portion of the RNN do I extract my output from? This really depends on what your use case is. For example, if you're using the RNN for a classification task, you'll only need one final output after passing in all the input - a vector representing the class probability scores. In another case, if you're doing text generation based on the previous character/word, you'll need an output at every single time step.



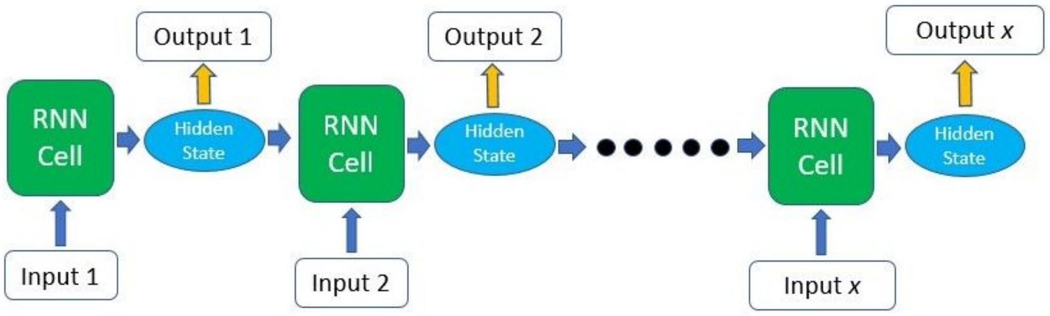
This image was taken from Andrej Karpathy's blog post

This is where RNNs are really flexible and can adapt to your needs. As seen in the image above, your input and output size can come in different forms, yet they can still be fed into and extracted from the RNN model.



Many inputs to one output

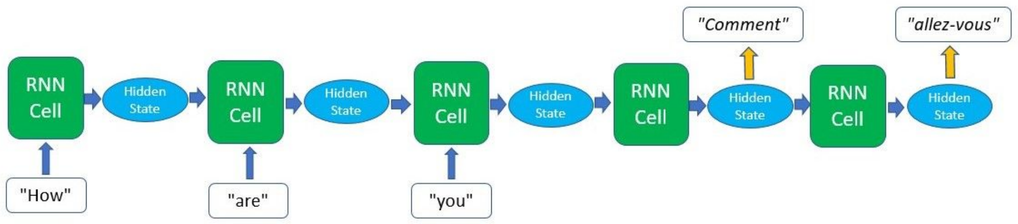
For the case where you'll only need a single output from the whole process, getting that output can be fairly straightforward as you can easily take the output produced by the last RNN cell in the sequence. As this final output has already undergone calculations through all the previous cells, the context of all the previous inputs has been captured. This means that **the final result is indeed dependent on all the previous computations and inputs**.



Many inputs to many outputs

For the second case where you'll need output information from the intermediate time steps, this information can be taken from the hidden state produced at each step as shown in the figure above. The output produced can also be fed back into the model at the next time step if necessary.

Of course, the type of output that you can obtain from an RNN model is not limited to just these two cases. There are other methods such as Sequence-To-Sequence translation where the output is only produced in a sequence after all the input has been passed through. The diagram below depicts what that looks like.



Structure of a sequence-to-sequence model

Inner Workings

Now that we have a basic understanding and a bird's eye view of how RNNs work, let's explore some basic computations that the RNN's cells have to do to produce the hidden states and outputs.

$$\text{hidden}_t = \text{F}(\text{hidden}_{t-1}, \text{input}_t)$$

In the first step, a hidden state will usually be seeded as a matrix of zeros, so that it can be fed into the RNN cell together with the first input in the sequence. In the simplest RNNs, the hidden state and the input data will be multiplied with weight matrices initialized via a scheme such as Xavier or Kaiming(you can read more on this topic here). The result of these multiplications will then be passed through an activation function(such as a tanh function) to introduce non-linearity.

$$\text{hidden}_t = \tanh(\text{weight}_{hidden} * \text{hidden}_{t-1} + \text{weight}_{input} * \text{input}_t)$$

Additionally, if we require an output at the end of each time step we can pass the hidden state that we just produced through a linear layer or just multiply it by another weight matrix to obtain the desired shape of the result.

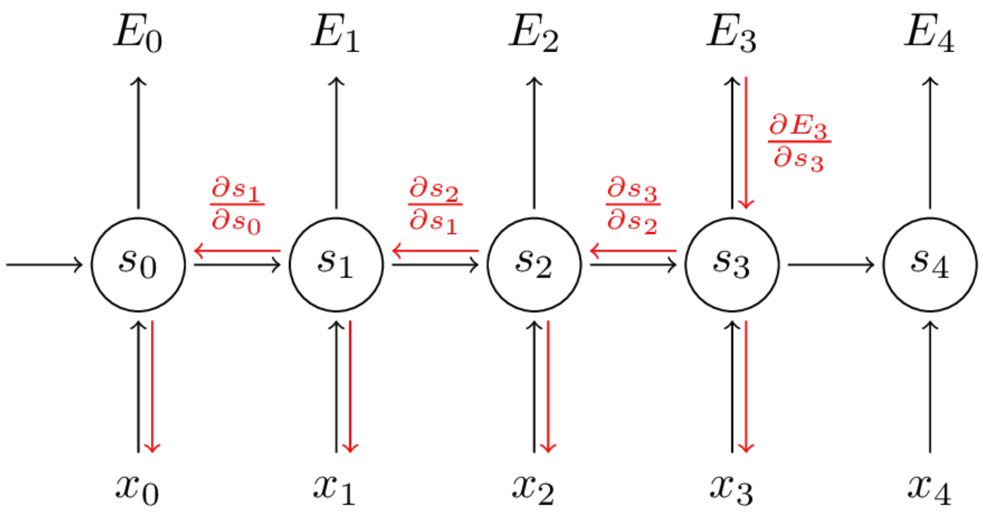
$$\text{output}_t = \text{weight}_{output} * \text{hidden}_t$$

The hidden state that we just produced will then be fed back into the RNN cell together with the next input and this process continues until we run out of input or the model is programmed to stop producing outputs.

As mentioned earlier, these computations presented above are just simple representations of how RNN cells do their calculations. For the more advanced RNN structures such as LSTMs, GRUs, etc., the computations are generally much more complicated.

Training and Back-propagation

Similar to other forms of neural networks, RNN models need to be trained in order to produce accurate and desired outputs after a set of inputs are passed through.


E_0
E_1
E_2
E_3
E_4
$\frac{\partial E_3}{\partial s_3}$
$\frac{\partial s_1}{\partial s_0}$
$\frac{\partial s_2}{\partial s_1}$
$\frac{\partial s_3}{\partial s_2}$
s_0
s_1
s_2
s_3
s_4
x_0
x_1
x_2
x_3
x_4

How weights are updated through back-propagation. Taken from WildML's blog post

During training, for each piece of training data we'll have a corresponding ground-truth label, or simply put a"correct answer" that we want the model to output. Of course, for the first few times that we pass the input data through the model, we won't obtain outputs that are equal to these correct answers. However, after receiving these outputs, what we'll do during training is that we'll calculate the loss of that process, which measures how far off the model's output is from the correct answer. Using this loss, we can calculate the gradient of the loss function for back-propagation.

With the gradient that we just obtained, we can update the weights in the model accordingly so that future computations with the input data will produce more accurate results. The weight here refers to the weight matrices that are multiplied with the input data and hidden states during the forward pass. This entire process of calculating the gradients and updating the weights is called back-propagation. Combined with the forward pass, back-propagation is looped over and again, allowing the model to become more accurate with its outputs each time as the weight matrices values are modified to pick out the patterns of the data.

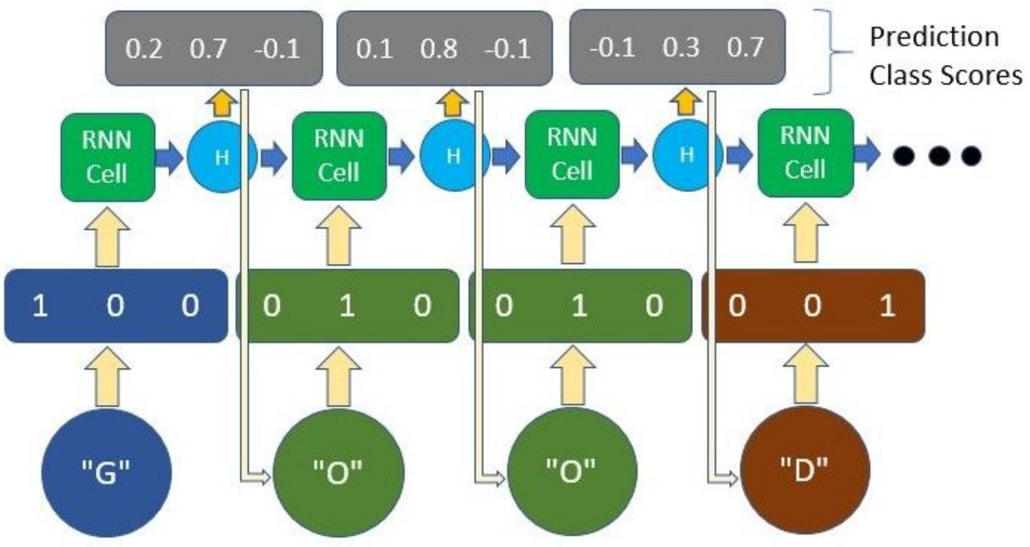
Although it may look as if each RNN cell is using a different weight as shown in the graphics, **all of the weights are actually the same as that RNN cell is essentially being re-used throughout the process**. Therefore, only the input data and hidden state carried forward are unique at each time step.

Textual Input Data

Unlike humans, neural networks are generally much less proficient at handling textual data. Therefore in most Natural Language Processing (NLP) tasks, text data is usually converted to a set of numbers, such as embeddings, one-hot encodings, etc. such that the network can parse the data better. In our implementation later in this post, I'll be using one-hot encoding to represent our characters. Therefore I'll give a brief view of what it encompasses.

As with most machine learning or deep learning projects, data pre-processing more often than not takes up a significant portion of the time of a project. In our example later, we'll be pre-processing our textual data into a simple representation - one-hot encodings at the character level.

This form of encoding is basically giving each character in the text a unique vector. For example, if our text only contains the word "GOOD", there are only 3 unique characters and thus our vocabulary size is only 3. We will allocate each unique character a unique vector, where all the items are zero except one at an index assigned to that character. This is how we represent each character to the model.



Inputs are being converted to one-hot representations and outputs are the corresponding class scores

The output may be something similar as well, where we can take the highest number in the vector and take it as the predicted character.

The output may be something similar as well, where we can take the highest number in the vector and take it as the predicted character.

Hands-On (Time for the code!)

We've gone through most of the basics of RNNs. While you may still have some concepts that you're uncertain of, sometimes reading and implementing it in the code may help clear things up for you!

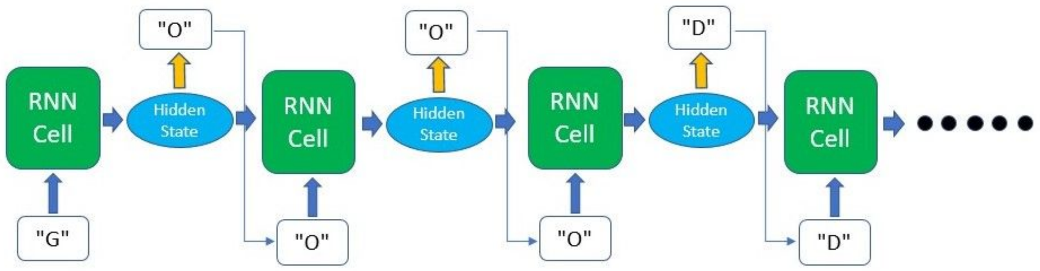
You can run the code we're using on FloydHub by clicking the button below and creating the project.

Run on FloydHub

Alternatively, here's there link to the notebook on GitHub:

https://github.com/gabrielloye/RNN-walkthrough/blob/master/main.ipynb

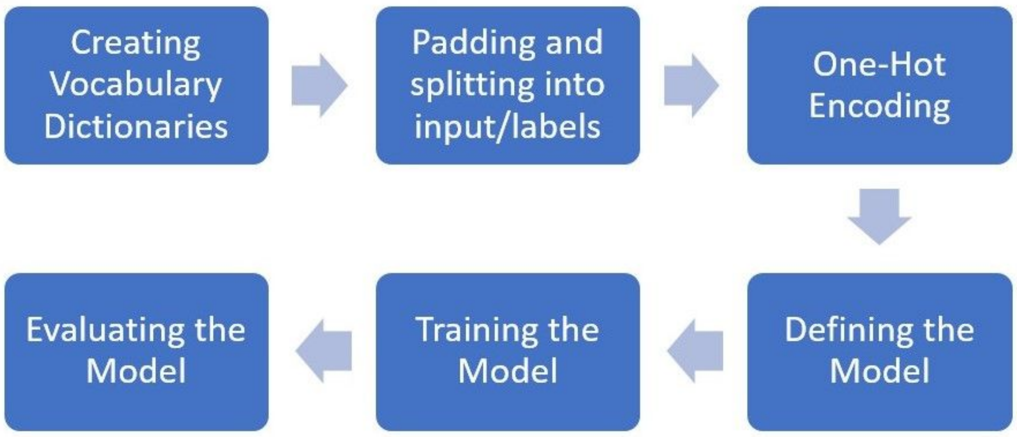
In this implementation, we'll be using the PyTorch library, a deep learning platform that is easy to use and widely utilized by top researchers. We will be building a model that will complete a sentence based on a word or a few characters passed into it.



How our model will be processing input data and producing outputs

The model will be fed with a word and will predict what the next character in the sentence will be. This process will repeat itself until we generate a sentence of our desired length.

To keep this short and simple, we won't be using any large or external datasets. Instead, we'll just be defining a few sentences to see how the model learns from these sentences. The process that this implementation will take is as follows:



Flow of the implementation

We'll start off by importing the main PyTorch package along with the *nn* package which we will use when building the model. In addition, we'll only be using NumPy to pre-process our data as Torch works really well with NumPy.

```
import torch
from torch import nn

import numpy as np
```

First, we'll define the sentences that we want our model to output when fed with the first word or the first few characters.

Then we'll create a dictionary out of all the characters that we have in the sentences and map them to an integer. This will allow us to convert our input characters to their respective integers (*char2int*) and vice versa (*int2char*).

```
text = ['hey how are you','good i am
fine','have a nice day']

# Join all the sentences together and
extract the unique characters from the
combined sentences
chars = set(''.join(text))

# Creating a dictionary that maps integers
to the characters
int2char = dict(enumerate(chars))

# Creating another dictionary that maps
characters to integers
char2int = {char: ind for ind, char in
int2char.items()}
```

The *char2int* dictionary will look like this: It holds all the letters/symbols that were present in our sentences and maps each of them to a unique integer.

```
[Out]: {'f': 0, 'a': 1, 'h': 2, 'i': 3, 'u':
4, 'e': 5, 'm': 6, 'w': 7, 'y': 8, 'd': 9,
'c': 10, ' ': 11, 'r': 12, 'o': 13, 'n': 14,
'g': 15, 'v': 16}
```

Next, we'll be padding our input sentences to ensure that all the sentences are of standard length. While RNNs are typically able to take in variably sized inputs, we will usually want to feed training data in batches to speed up the training process. In order to used batches to train on our data, we'll need to ensure that each sequence within the input data is of equal size.

Therefore, in most cases, padding can be done by filling up sequences that are too short with **0** values and trimming sequences that are too long. In our case, we'll be finding the length of the longest sequence and padding the rest of the sentences with blank spaces to match that length.

```
# Finding the length of the longest string
in our data
maxlen = len(max(text, key=len))

# Padding

# A simple loop that loops through the list
of sentences and adds a ' ' whitespace until
the length of
# the sentence matches the length of the
longest sentence
for i in range(len(text)):
  while len(text[i])<maxlen:
      text[i] += ' '
```

As we're going to predict the next character in the sequence at each time step, we'll have to divide each sentence into:

- Input data
- The last input character should be excluded as it does not need to be fed into the model
- Target/Ground Truth Label
- One time-step ahead of the Input data as this will be the "correct answer" for the model at each time step corresponding to the input data

```
# Creating lists that will hold our input
and target sequences
input_seq = []
target_seq = []

for i in range(len(text)):
    # Remove last character for input
sequence
  input_seq.append(text[i][:-1])

    # Remove first character for target
sequence
  target_seq.append(text[i][1:])
  print("Input Sequence: {}\nTarget
Sequence: {}".format(input_seq[i],
target_seq[i]))
```

Our input sequence and target sequence will look like this:

- Input Sequence: hey how are yo
- Target Sequence: ey how are you

The target sequence will always be one-time step ahead of the input sequence.

Now we can convert our input and target sequences to sequences of integers instead of a sequence of characters by mapping them using the dictionaries we created above. This will allow us to one-hot-encode our input sequence subsequently.

```
for i in range(len(text)):
    input_seq[i] = [char2int[character] for
character in input_seq[i]]
    target_seq[i] = [char2int[character] for
character in target_seq[i]]
```

Before encoding our input sequence into one-hot vectors, we'll define 3 key variables:

1. *dict_size*: Dictionary size - The number of unique characters that we have in our text

- This will determine the one-hot vector size as each character will have an assigned index in that vector

2. *seq_len*: The length of the sequences that we're feeding into the model

- As we standardized the length of all our sentences to be equal to the longest sentences, this value will be the max length - 1 as we removed the last character input as well

3. *batch_size*: The number of sentences that we defined and are going to feed into the model as a batch

```
dict_size = len(char2int)
seq_len = maxlen - 1
batch_size = len(text)
```

```
def one_hot_encode(sequence, dict_size,
seq_len, batch_size):
    # Creating a multi-dimensional array of
zeros with the desired output shape
    features = np.zeros((batch_size,
seq_len, dict_size), dtype=np.float32)

    # Replacing the 0 at the relevant
character index with a 1 to represent that
character
    for i in range(batch_size):
        for u in range(seq_len):
            features[i, u, sequence[i][u]] =
1
    return features
```

We also defined a helper function that creates arrays of zeros for each character and replaces the corresponding character index with a **1**.

```
# Input shape --> (Batch Size, Sequence
Length, One-Hot Encoding Size)
input_seq = one_hot_encode(input_seq,
dict_size, seq_len, batch_size)
```

Since we're done with all the data pre-processing, we can now move the data from NumPy arrays to PyTorch's very own data structure - **Torch Tensors.**

```
input_seq = torch.from_numpy(input_seq)
target_seq = torch.Tensor(target_seq)
```

Now we've reached the fun part of this project! We'll be defining the model using the Torch library, and this is where you can add or remove layers, be it fully connected layers, convolutional layers, vanilla RNN layers, LSTM layers, and many more! In this post, we'll be using the basic *nn.rnn* to demonstrate a simple example of how RNNs can be used.

Before we start building the model, let's use a built-in feature in PyTorch to check the device we're running on (CPU or GPU). This implementation will not require GPU as the training is really simple. However, as you progress on to large datasets and models with millions of trainable parameters, using the GPU will be very important to speed up your training.

```
# torch.cuda.is_available() checks and returns a Boolean True if a GPU is available, else it'll return False
is_cuda = torch.cuda.is_available()

# If we have a GPU available, we'll set our device to GPU. We'll use this device variable later in our code.
if is_cuda:
    device = torch.device("cuda")
    print("GPU is available")
else:
    device = torch.device("cpu")
    print("GPU not available, CPU used")
```

To start building our own neural network model, we can define a class that inherits PyTorch's base class(*nn.module*) for all neural network modules. After doing so, we can start defining some variables and also the layers for our model under the constructor. For this model, we'll only be using 1 layer of RNN followed by a fully connected layer. The fully connected layer will be in charge of converting the RNN output to our desired output shape.

We'll also have to define the forward pass function under `forward()` as a class method. The forward function is executed sequentially, therefore we'll have to pass the inputs and the zero-initialized hidden state through the RNN layer first, before passing the RNN outputs to the fully-connected layer. Note that we are using the layers that we defined in the constructor.

The last method that we have to define is the method that we called earlier to initialize the hidden state - *init_hidden().* This basically creates a tensor of zeros in the shape of our hidden states.

```
class Model(nn.Module):
    def __init__(self, input_size,
output_size, hidden_dim, n_layers):
        super(Model, self).__init__()

        # Defining some parameters
        self.hidden_dim = hidden_dim
        self.n_layers = n_layers

        #Defining the layers
        # RNN Layer
        self.rnn = nn.RNN(input_size,
hidden_dim, n_layers, batch_first=True)
        # Fully connected layer
        self.fc = nn.Linear(hidden_dim,
output_size)

    def forward(self, x):

        batch_size = x.size(0)

        # Initializing hidden state for
first input using method defined below
        hidden =
self.init_hidden(batch_size)

        # Passing in the input and hidden
state into the model and obtaining outputs
        out, hidden = self.rnn(x, hidden)

        # Reshaping the outputs such that it
can be fit into the fully connected layer
        out = out.contiguous().view(-1,
self.hidden_dim)
        out = self.fc(out)

        return out, hidden

    def init_hidden(self, batch_size):
        # This method generates the first hidden state of zeros which we'll use in the forward pass
        # We'll send the tensor holding the hidden state to the device we specified earlier as well
        hidden = torch.zeros(self.n_layers, batch_size, self.hidden_dim)
        return hidden
```

After defining the model above, we'll have to instantiate the model with the relevant parameters and define our hyper-parameters as well. The hyper-parameters we're defining below are:

- `n_epochs`: Number of Epochs --> Number of times our model will go through the entire training dataset
- `lr`: Learning Rate --> Rate at which our model updates the weights in the cells each time back-propagation is done

For a more in-depth guide on hyper-parameters, you can refer to this comprehensive article.

Similar to other neural networks, we have to define the optimizer and loss function as well. We’ll be using *CrossEntropyLoss* as the final output is basically a classification task and the common *Adam* optimizer.

```
# Instantiate the model with hyperparameters
model = Model(input_size=dict_size, output_size=dict_size, hidden_dim=12, n_layers=1)
```

```
# We'll also set the model to the device
that we defined earlier (default is CPU)
model.to(device)

# Define hyperparameters
n_epochs = 100
lr=0.01

# Define Loss, Optimizer
criterion = nn.CrossEntropyLoss()
optimizer =
torch.optim.Adam(model.parameters(), lr=lr)
```

Now we can begin our training! As we only have a few sentences, this training process is very fast. However, as we progress, larger datasets and deeper models mean that the input data is much larger and the number of parameters within the model that we have to compute is much more.

```
# Training Run
for epoch in range(1, n_epochs + 1):
    optimizer.zero_grad() # Clears existing
gradients from previous epoch
    input_seq.to(device)
    output, hidden = model(input_seq)
    loss = criterion(output,
target_seq.view(-1).long())
    loss.backward() # Does backpropagation
and calculates gradients
    optimizer.step() # Updates the weights
accordingly
```

```
    if epoch%10 == 0:
        print('Epoch:
{}/{}.............'.format(epoch, n_epochs),
end=' ')
        print("Loss:
{:.4f}".format(loss.item()))
```

```
[Out]:  Epoch: 10/100............. Loss:
2.4176
        Epoch: 20/100............. Loss:
2.1816
        Epoch: 30/100............. Loss:
1.7952
        Epoch: 40/100............. Loss:
1.3524
        Epoch: 50/100............. Loss:
0.9671
        Epoch: 60/100............. Loss:
0.6644
        Epoch: 70/100............. Loss:
0.4499
        Epoch: 80/100............. Loss:
0.3089
        Epoch: 90/100............. Loss:
0.2222
        Epoch: 100/100............. Loss:
0.1690
```

Let’s test our model now and see what kind of output we will get. As a first step, we'll define some helper function to convert our model output back to text.

```
# This function takes in the model and
character as arguments and returns the next
character prediction and hidden state
def predict(model, character):
    # One-hot encoding our input to fit into
the model
    character = np.array([[char2int[c] for c
in character]])
    character = one_hot_encode(character,
dict_size, character.shape[1], 1)
    character = torch.from_numpy(character)
    character.to(device)

    out, hidden = model(character)

    prob = nn.functional.softmax(out[-1],
dim=0).data
    # Taking the class with the highest
probability score from the output
    char_ind = torch.max(prob, dim=0)
[1].item()

    return int2char[char_ind], hidden
```

```
# This function takes the desired output
length and input characters as arguments,
returning the produced sentence
def sample(model, out_len, start='hey'):
    model.eval() # eval mode
    start = start.lower()
    # First off, run through the starting
characters
    chars = [ch for ch in start]
    size = out_len - len(chars)
    # Now pass in the previous characters
and get a new one
    for ii in range(size):
        char, h = predict(model, chars)
        chars.append(char)

    return ''.join(chars)
```

Let's run the function with our model and the starting words *'good'.*

```
sample(model, 15, 'good')
```

```
[Out]: 'good i am fine '
```

As we can see, the model is able to come up with the sentence ‘good i am fine ‘ if we feed it with the words ‘good’. Pretty good for a few lines of code, yea?

Model Limitations

While this model is definitely an over-simplified language model, let’s review its limitations and the issues that need to be addressed in order to train a better language model.

Over-fitting

We only fed the model with 3 training sentences, therefore it essentially “memorized” the sequence of characters of these sentences and thus returned us the exact sentence that we trained it on. However, if a similar model is trained on a larger data-set with some randomness added into it, the model will pick out the general sentence structures and language rules, and it’ll be able to generate its own unique sentences.

Nevertheless, running your models with a single sample or batch acts as a sanity check for your workflow, ensuring that your data types are all correct, your model is learning fine, etc.

Handling of unseen characters

The model is only currently able to process the characters that it has seen before in the training data set. Normally, if the training data set is large enough, all letters and symbols, etc. should appear at least once and will thus be present in our vocabulary. However, it is always good to have a way to handle never seen before characters, such as assigning all unknowns to its own index.

Representation of Textual Data

In this implementation, we used one-hot encoding to represent our characters. While it may be fine for this task due to its simplicity, most of the time it should not be used as a solution in actual or more complex problems. This is because:

- It is computationally too expensive for large datasets
- There is no contextual/semantic information embedded in one-hot vectors

and many other downsides that make this solution less viable.

Instead, most modern NLP solutions rely on word embeddings (word2vec, GloVe) or more recently, unique contextual word representations in BERT, ELMo, and ULMFit. These methods allow the model to learn the meaning of a word based on the text that appears before it, and in the case of BERT, etc., learn from the text that appears after it as well.

Next Steps

This post is just the tip of the iceberg when it comes to Recurrent Neural Networks. While the vanilla RNN is rarely used in solving NLP or sequential problems, having a good grasp of the basic concepts of RNNs will definitely aid in your understanding as you move towards the more popular GRUs and LSTMs.

Ready for Long Short-Term Memory?! Here's the sequel!

Special thanks to Alessio for his constant guidance and the rest of the FloydHub team for providing this amazing platform and allowing me to give back to the deep learning community. Stay awesome!

About Gabriel Loye

Gabriel is an Artificial Intelligence enthusiast and web developer. He’s currently exploring various fields of deep learning, from Natural Language Processing to Computer

Vision. He is always open to learning new things and implementing or researching on novel ideas and technologies. He will be starting his undergraduate studies in Business Analytics at NUS School of Computing. He is currently an intern at a FinTech start-up, PinAlpha. Gabriel is also a FloydHub AI Writer. You can connect with Gabriel on LinkedIn and GitHub.