Generating Tv-Script using Recurrent Neural Network

INSE 6421

Project

Concordia University

Submitted By:

Meghaksh Brahmbhatt | 40166917

Binit Kumar | 40172005

April, 14th 2022

Table of Contents

What is TV Script Generation Project?

A project done by us to generate a fake Tv-script using a RNN model. We are using a Tv script from a popular Tv show named Seinfeld. The neural network takes the following mentioned tv-script, learns the text sequences from it to generate a new fake tv-script.

- Our code repository location: https://github.com/binit92/INSE-6421
- Seinfeld dataset from season 9. Dataset source: https://www.kaggle.com/thec03u5/seinfeld-chronicles#scripts.csv
- Seinfeld: https://en.wikipedia.org/wiki/Seinfeld

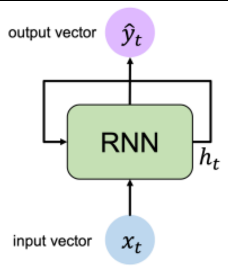
Expected Input Script and output Script

Input Sample	Output Sample
george: so, i was in the contrary. george: so, i guess i was a woman, and the defendants was a good boy. elaine: i don't know where it is. jerry: so i was thinking about this one? hoyt: i thought i was a little adjustment. jerry: what is that? hoyt: yes, yes. yes. i got a pee on this. you were in the middle of a plane, and i have a little adjustment. george: what happened to him?	george: so you want to see how you could do that? hoyt: i don't want to see you in a hotel. elaine: you want to get the car on the street and a wheelchair? hoyt: what do you think? george: yeah, i guess i was wondering about it. jerry: what are you doing with this girl? elaine: no, i got to tell him. george: i can't believe this is the most exciting thing.

Why using Recurrent Neural Network?

Our script generations take a word and determine the next word in a sentence. This requires keeping a sequence or order of words in the neural network. However, In Feed-Forward Network, there is no sense of order in the input. So, the question is how to build the concept of memory in the neural network so that it can learning for the sequence.

Recurrence relations need to apply at each time step t, such that model learns the concepts of memory by updating the hidden state.

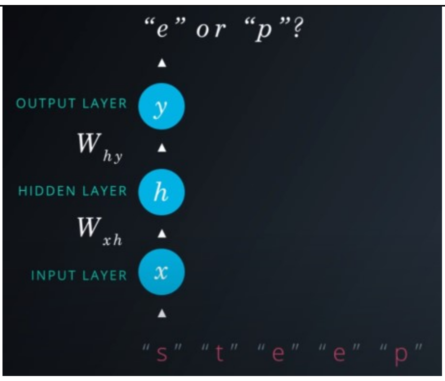
 output vector $\hat{y}_t$ RNN h_t input vector x_t	$\underbrace{\boxed{h_t}}_{\text{cell state}} = \underbrace{\boxed{f_W}}_{\text{function parameterized by W}} (\underbrace{\boxed{h_{t-1}}}_{\text{old state}}, \underbrace{\boxed{x_t}}_{\text{input vector at time step t}})$ • Output vector $\hat{y}_t = g_W\,(h_t)$

Visualizing Character-wise RNN

Our goal here is to predict the next character in word "steep"

- When we pass "s", desired output is "t".
- When we pass "t", desired output is "e".
- When we pass "e", desired output could be "e" or "p". The network doesn't have enough information to determine which character to predict!

To solve this problem, we need to include information about the sequence of characters.

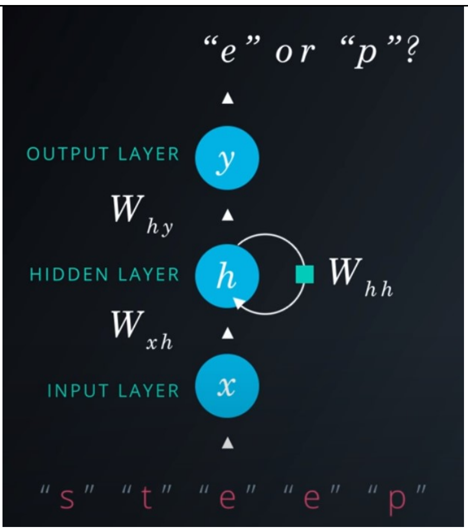


We can solve this problem by routing the hidden layer output from the previous step back into the hidden layer.

- The box in the diagram means the value from the previous sequence, or time step.
- Now the network sees an "e", it knows it saw an "s" and a "t" before, so the next character should be another "e"

This architecture is known as Recurrent Neural Network or RNN

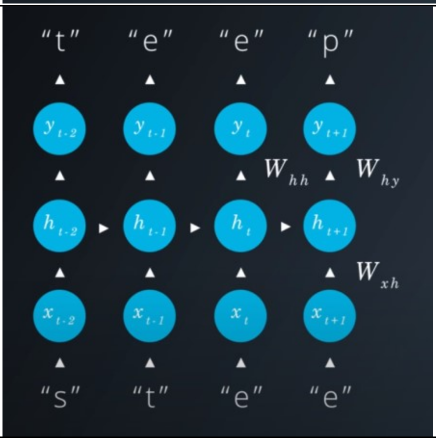
- Now the total input in the hidden layer is the sum of the layered combinations from the input layer and previous hidden layer

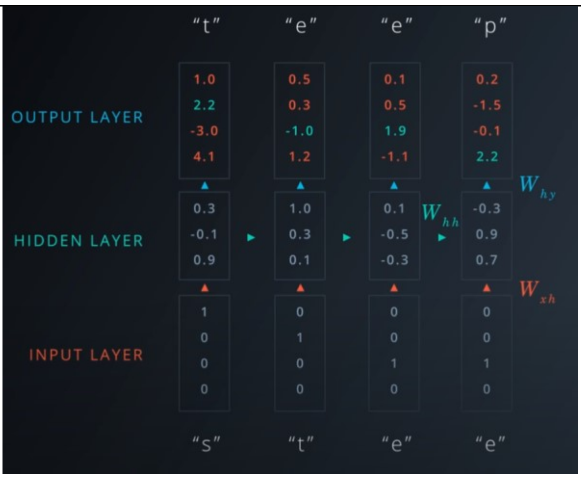


$$h_t = f(W_{hh}h_{t-1} + W_{xh}x_t)$$

We can view our recurrent network as one big graph by unrolling it.

- Now, we have a feed-forward network for each character but connected through the hidden layers.
- Each hidden nodes receives inputs from input node and hidden node from the previous step

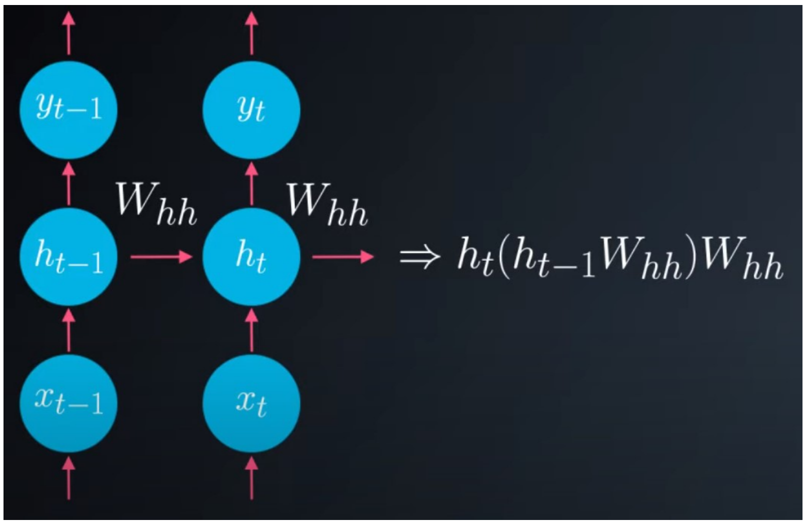

"t" "e" "e" "p"
y_{t-2} y_{t-1} y_t y_{t+1}
W_{hh}
W_{hy}
h_{t-2} h_{t-1} h_t h_{t+1}
W_{xh}
x_{t-2} x_{t-1} x_t x_{t+1}
"s" "t" "e" "e"

Let's visualize by adding some numbers here • Here, we're **one hot encoding** the input characters. • 1000 = "s", 0100 = "t", "0010" = "e", 0010 = "e" • There are three units in the hidden layer and the output layer is showing the logits • We pass the logits into Softmax function to get prediction and to train with a cross entropy loss. This is basic architecture for Character-Wise RNN.	 "t" "e" "e" "p" OUTPUT LAYER 1.0 2.2 -3.0 4.1 0.5 0.3 -1.0 1.2 0.1 0.5 1.9 -1.1 0.2 -1.5 -0.1 2.2 W_{hy} HIDDEN LAYER 0.3 -0.1 0.9 1.0 0.3 0.1 0.1 -0.5 -0.3 -0.3 0.9 0.7 W_{hh} W_{xh} INPUT LAYER 1 0 0 0 0 1 0 0 0 0 1 0 0 0 1 0 "s" "t" "e" "e"

Problem with RNN Architecture

We can include information from a sequence of data using a recurrent connection on the hidden layer. This connection goes through these weights, *Whh* After enrolling the network, we say the hidden layer at step t is a function of the previous hidden state multiplied by those weights. The output of that layer is again multiplied by *Whh*. For every step we have in the network, we are multiplying by the weights again and again.

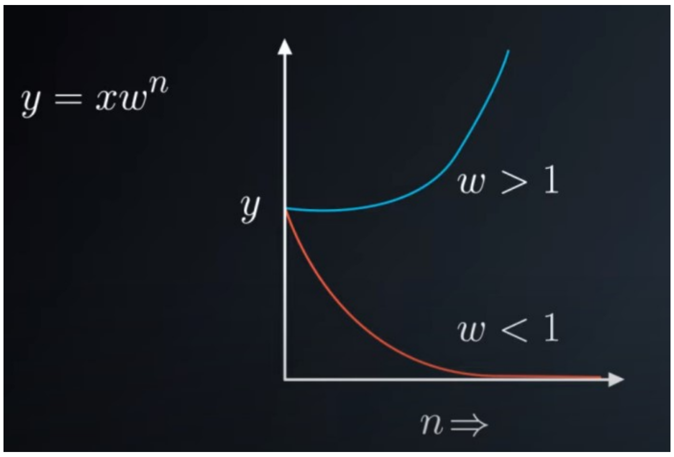
And when we do backpropagation, that's even more multiplication. These leads to problem where gradients going through network either get really small and vanish or get really large and explode.



Vanishing or Exploding Gradients

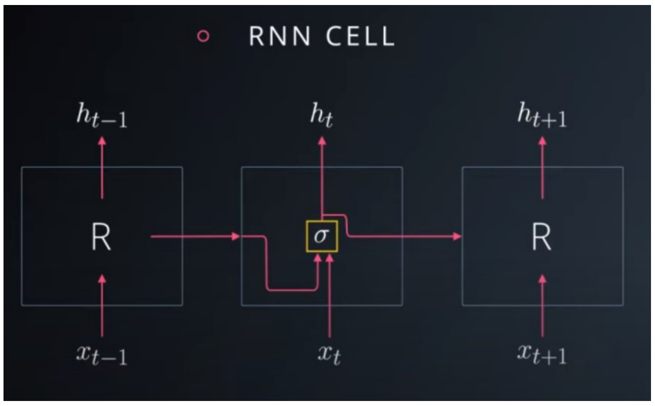
If we multiply by some number a bunch of times, we will get two results except a couple of special cases. If that number is less than 1, we will end up at 0. If it greater than 1, we will head towards infinity. This happens to gradient in normal RNN, where they either vanish or explode.

Resulting in making it difficult for RNNs to learn long range interactions.



RNN Cell

We can think of RNNs as a bunch of cells with inputs and outputs. Inside the cell, we have network layers, such as the sigmoid layer labelled with a sigma here. To solve the problem of the vanishing gradients, we can use more complicated cells called long short-term memory or LSTM



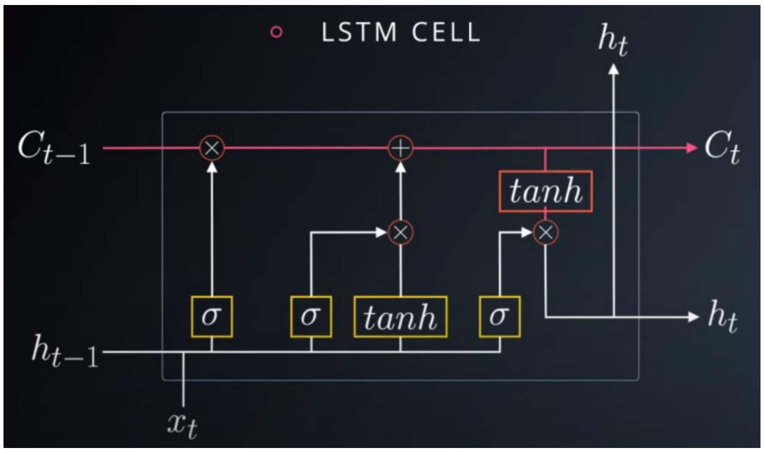
LSTM Cell

Let’s break down LSTM to understand:

The key addition here is the cell state labelled C. In this cell, there are four network layers shown as yellow boxes. Each of them with their own weights. The layers labelled with sigma are sigmoid and tanh is the hyperbolic tangent function. Tanh is similar to a sigmoid in that it squashes input, but the output is between -1 to 1 instead of 0 and 1

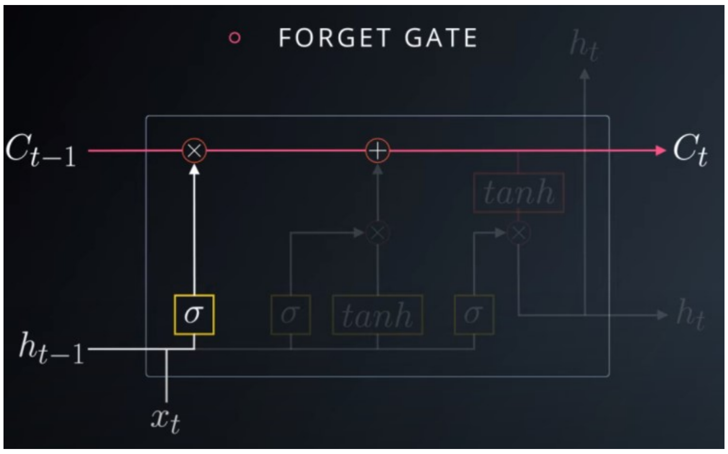
The red circles are point-wise or element-wise operations i.e. they operate on matrices element by element. The main improvement here is through the cell state. The cell state goes through LSTM cell with little interaction allowing information to flow easily through the cells. The cell state is modified only through element-wise operation which functions as gates.

And the hidden state is now calculated through cell state, then passed on



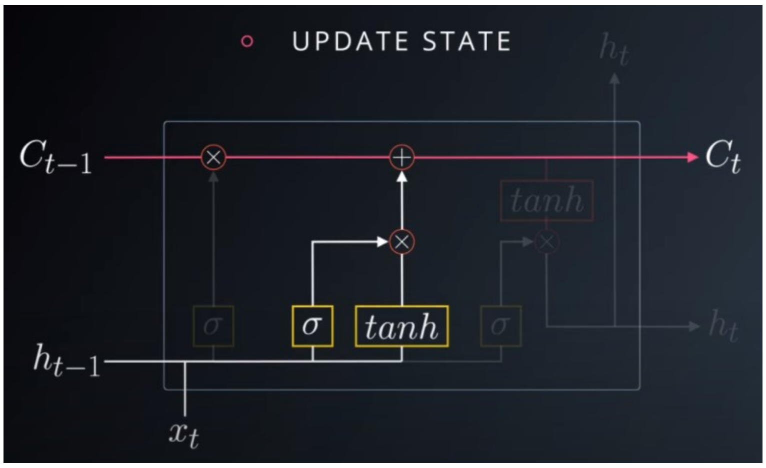
Forget Gate

The first gate is the forget gate. The values coming out of sigmoid layer are between 0 and 1. Then they are multiplied element-wise with the cell state. So the values from this layer close to 0 will shut off certain elements in the cell state. Effectively, forgetting that information going forward. Conversely, values close to 1 will allow information to pass through unchanged. This is helpful, because the network can learn to forget information that causes incorrect predictions. On the other hand, long range information that are helpful is allowed to flow through freely.



Update Gate

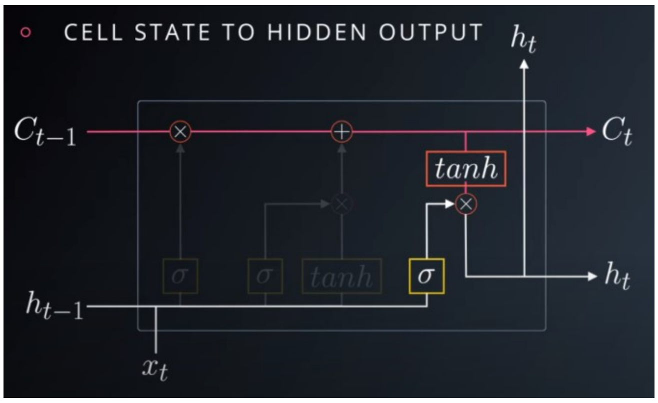
The next gate updates the cell state from the input and previous hidden state. The tanh layer output is added to the cell state and again gated by a sigmoid layer. In this way, the cell state can be updated in the step and passed along to the next cell



Cell state to hidden output

Here, the cell state is used to produce the hidden state which is sent to the next hidden cell as well as to higher layers. It's the arrow pointing up here

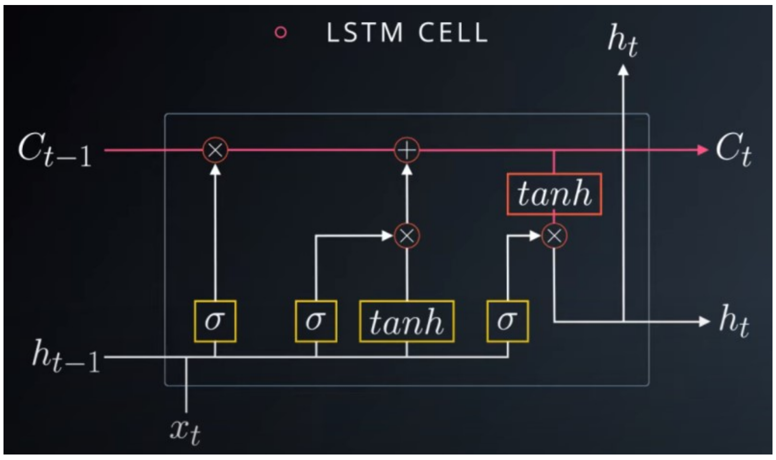
The cell state is passed through another tanh then gated again with another sigmoid layer. All these sigmoid gates let the network learn which information to keep and which information to get rid of.



Putting it all together

Putting all this together, the LSTM cell consists of a cell state with a bunch of gates used to update it, and leak it out to the hidden state.

This is just the basic LSTM. There are multiple variations and lot of ongoing experimentation into improving these. They are also stacked into deeper layer. We just send the output from one cell to the input of another

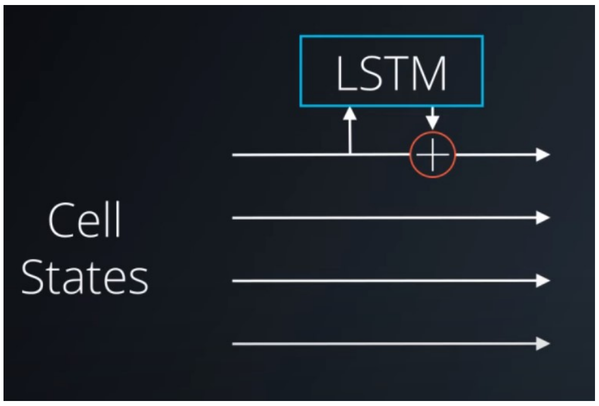


How to fix gradient problem?

Since the cell state is allowed to flow through the hidden layers with only this linear sum operation. Gradient can easily move through the network without being diminished. We can also get gradients added into the network through the LSTM cells but they are just added to the gradients flowing through

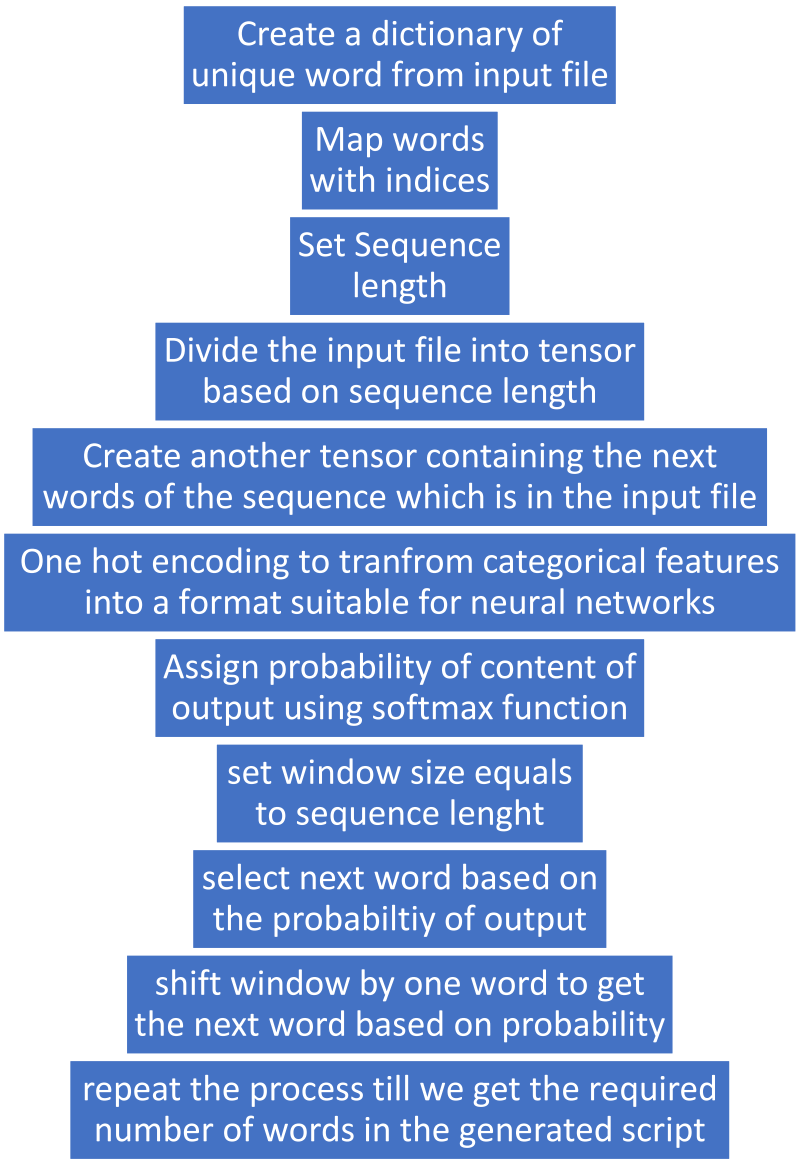
- Math behind LSTMs https://www.youtube.com/watch?v=iX5V1WpxxkY
- Understanding LSTM Networks http://colah.github.io/posts/2015-08-Understanding-LSTMs/

LSTMs are basic unit of RNNs in many applications



Story Scrambler – Automatic Text Generation using Word Level RNN-LSTM

By reading the paper (reference 2), we found the overall flow of how to create the word level RNN-LSTM Text Generation models.


Create a dictionary of unique word from input file
Map words with indices
Set Sequence length
Divide the input file into tensor based on sequence length
Create another tensor containing the next words of the sequence which is in the input file
One hot encoding to tranfrom categorical features into a format suitable for neural networks
Assign probability of content of output using softmax function
set window size equals to sequence lenght
select next word based on the probabiltiy of output
shift window by one word to get the next word based on probability
repeat the process till we get the required number of words in the generated script

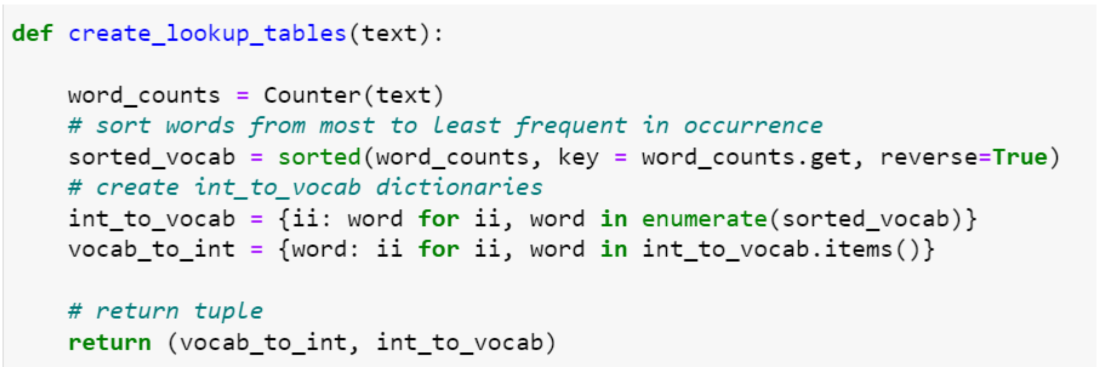
Framework/Library Used

- **Pytorch** – An open-source machine learning framework that accelerates the path from research prototyping to production deployment. (https://pytorch.org/)
- **Numpy** – A library that offers comprehensive mathematical functions, random number generators, linear algebra routines, Fourier transforms and more. (https://numpy.org/)
- **Pickle** – A library to convert Python object hierarchy into byte stream called “pickling”. “unpickling” is the inverse operation where a byte stream is converted back into an object hierarchy. (https://docs.python.org/3/library/pickle.html)

Pre-processing of Input data

Pre-processing is a step before feeding the input to the neural network where the input data is pre-processed to make it suitable for training. This includes

- Changing the entire data set into lowercase
- Splitting the sentences to get all the words
- Creating lookup table to generate word embeddings i.e. transforms the word to integer ids
- *vocab_to_int*: dictionary to go from a word to id
- *int_to_vocab*: dictionary to go from the id to word



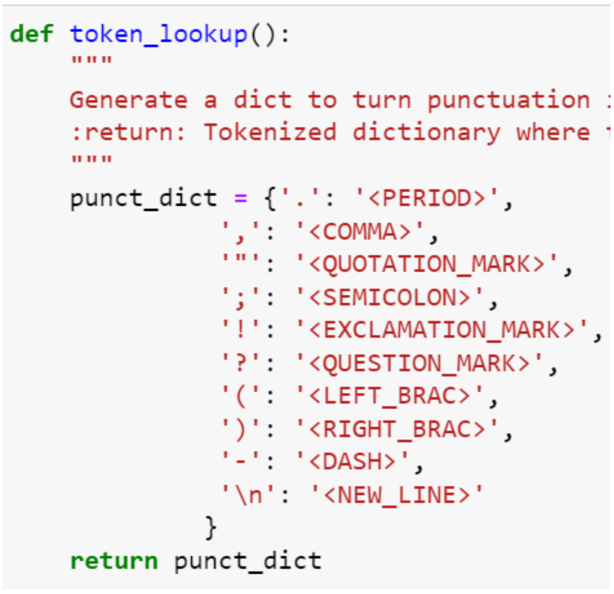
```python
def create_lookup_tables(text):

    word_counts = Counter(text)
    # sort words from most to least frequent in occurrence
    sorted_vocab = sorted(word_counts, key = word_counts.get, reverse=True)
    # create int_to_vocab dictionaries
    int_to_vocab = {ii: word for ii, word in enumerate(sorted_vocab)}
    vocab_to_int = {word: ii for ii, word in int_to_vocab.items()}

    # return tuple
    return (vocab_to_int, int_to_vocab)
```

The above function takes the tv script as input and return two dictionaries as tuple containing word to id and id to word entries.

Tokenize Punctuation

Punctuations like periods and exclamation marks can create multiple ids for the same word. For e.g. bye, bye!

This dictionary will be used to tokenize the symbols and add the delimiter (space) around it. This separates each symbol as its own word, making it easier for the neural network to predict the next word.



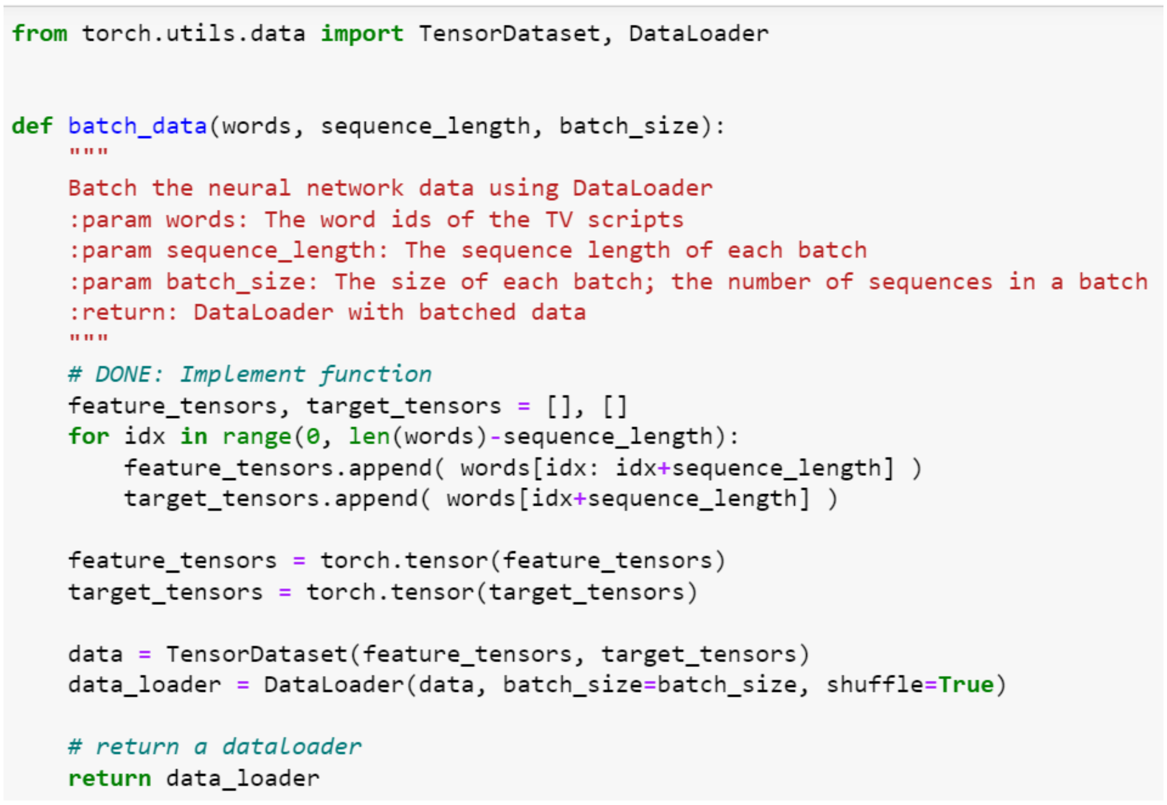
```python
def token_lookup():
    """
    Generate a dict to turn punctuation
    :return: Tokenized dictionary where
    """
    punct_dict = {'.': '<PERIOD>',
                  ',': '<COMMA>',
                  '"': '<QUOTATION_MARK>',
                  ';': '<SEMICOLON>',
                  '!': '<EXCLAMATION_MARK>',
                  '?': '<QUESTION_MARK>',
                  '(': '<LEFT_BRAC>',
                  ')': '<RIGHT_BRAC>',
                  '-': '<DASH>',
                  '\n': '<NEW_LINE>'
                 }
    return punct_dict
```

Creating batches of data

batch_data function to batch words data into chunks of size **batch_size** using Pytorch's DataLoader classes. DataLoader class will help to create **feature_tensors** and **target_tensors** of correct size and content of given **sequence_length** .

E.g. words = [1, 2, 3, 4, 5, 6, 7]

- sequence_length = 4
- First feature_tensor would be: [1, 2, 3, 4]
- The corresponding target_tensor would be: 5
- Second feature_tensor would be: [2, 3, 4, 5]
- And the second target_tensor would be: 6



```python
from torch.utils.data import TensorDataset, DataLoader


def batch_data(words, sequence_length, batch_size):
    """
    Batch the neural network data using DataLoader
    :param words: The word ids of the TV scripts
    :param sequence_length: The sequence length of each batch
    :param batch_size: The size of each batch; the number of sequences in a batch
    :return: DataLoader with batched data
    """
    # DONE: Implement function
    feature_tensors, target_tensors = [], []
    for idx in range(0, len(words)-sequence_length):
        feature_tensors.append( words[idx: idx+sequence_length] )
        target_tensors.append( words[idx+sequence_length] )

    feature_tensors = torch.tensor(feature_tensors)
    target_tensors = torch.tensor(target_tensors)

    data = TensorDataset(feature_tensors, target_tensors)
    data_loader = DataLoader(data, batch_size=batch_size, shuffle=True)

    # return a dataloader
    return data_loader
```

The **sample_x** should be of size (batch_size, sequence_length) or (10, 5) in this case and **sample_y** should just have one dimension: batch_size (10). We can also notice that the target **sample_y** are the next value in the ordered test_text data.

How data-loader looks like

Sample batch of inputs **sample_x** and targets **sample_y** from the data loader. We are shuffling the data in the data loader to get random batches.

```
torch.Size([10, 5])
tensor([[ 28,  29,  30,  31,  32],
        [ 21,  22,  23,  24,  25],
        [ 17,  18,  19,  20,  21],
        [ 34,  35,  36,  37,  38],
        [ 11,  12,  13,  14,  15],
        [ 23,  24,  25,  26,  27],
        [  6,   7,   8,   9,  10],
        [ 38,  39,  40,  41,  42],
        [ 25,  26,  27,  28,  29],
        [  7,   8,   9,  10,  11]])

torch.Size([10])
tensor([ 33,  26,  22,  39,  16,  28,  11,  43,  30,  12])
```

Neural Network Architecture

The below mentioned class RNN inherit the nn.Module class from Pytorch library. __init__ method is the constructor of the class to initialize the Pytorch RNN module with parameters like vocab_size, output_size, embedding_dim, hidden_dim and dropout.

Constructor

```python
class RNN(nn.Module):

    def __init__(self, vocab_size, output_size, embedding_dim, hidden_dim, n_layers, dropout=0.5):
        """
        Initialize the PyTorch RNN Module
        :param vocab_size: The number of input dimensions of the neural network (the size of the vocabulary)
        :param output_size: The number of output dimensions of the neural network
        :param embedding_dim: The size of embeddings, should you choose to use them
        :param hidden_dim: The size of the hidden layer outputs
        :param dropout: dropout to add in between LSTM/GRU layers
        """
        super(RNN, self).__init__()
        # TODO: Implement function

        # set class variables
        self.output_size = output_size
        self.n_layers = n_layers
        self.hidden_dim = hidden_dim

        # define model layers
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.lstm = nn.LSTM(embedding_dim, hidden_dim, n_layers, dropout=dropout, batch_first=True)
        self.dropout = nn.Dropout(dropout)
        self.fc = nn.Linear(hidden_dim, output_size)
```

Forward Propagation

Forward method in the RNN class defines the forward propagation of the neural network.

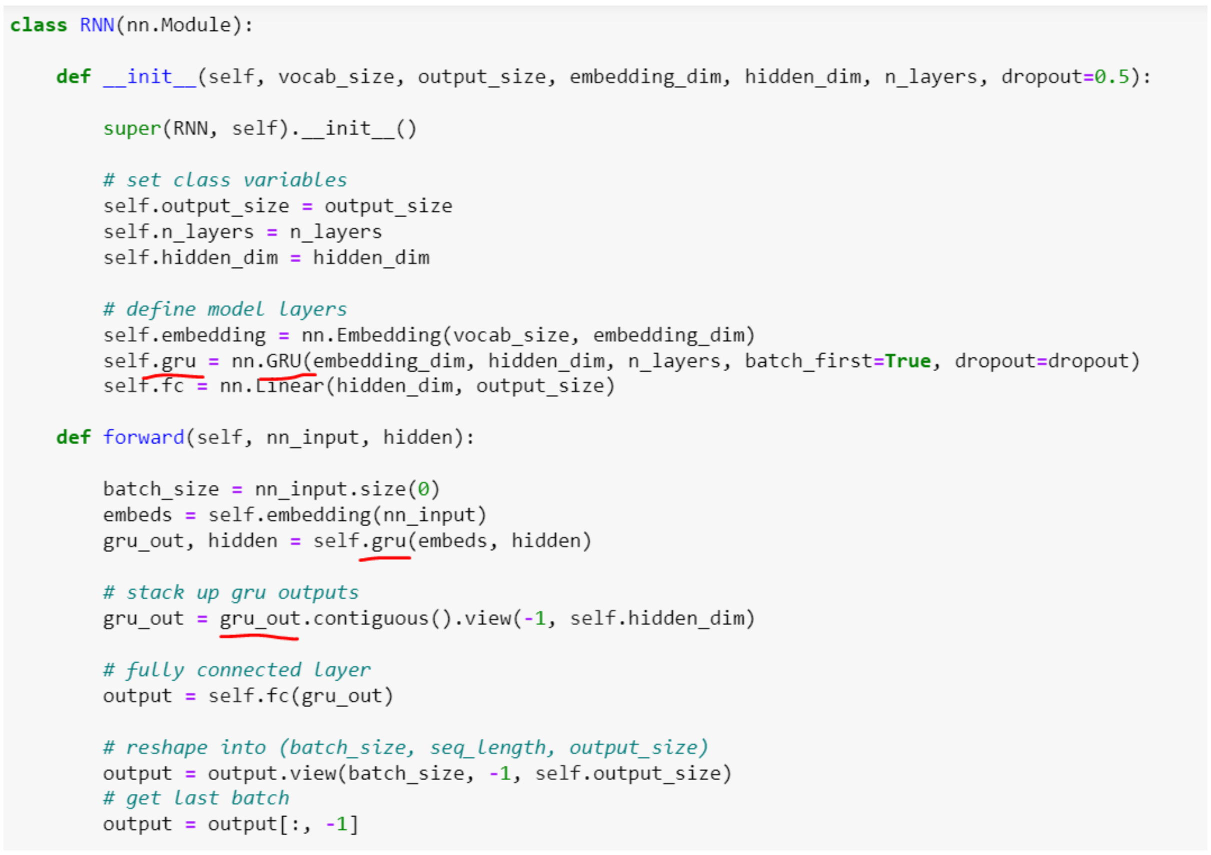
```python
def forward(self, nn_input, hidden):
    """
    Forward propagation of the neural network
    :param nn_input: The input to the neural network
    :param hidden: The hidden state
    :return: Two Tensors, the output of the neural network and the latest hidden state
    """
    # get the batch size
    batch_size = nn_input.size(0)

    # Embedding and LSTM layer
    embeds = self.embedding(nn_input)
    lstm_out, hidden = self.lstm(embeds, hidden)

    # stack up LSTM output to pass through the last FC output layer
    lstm_out = lstm_out.contiguous().view(-1, self.hidden_dim)

    # dropout layer after LSTM
    output = self.dropout(lstm_out)

    # last FC layer
    output = self.fc(output)

    # reshape into (batch_size, seq_length, output_size)
    output = output.view(batch_size, -1, self.output_size)

    # get last batch
    out = output[:, -1]

    # return one batch of output word scores and the hidden state
    return out, hidden
```

Initializing the hidden state of LSTM

```python
def init_hidden(self, batch_size):
    '''
    Initialize the hidden state of an LSTM/GRU
    :param batch_size: The batch_size of the hidden state
    :return: hidden state of dims (n_layers, batch_size, hidden_dim)
    '''
    weight = next(self.parameters()).data

    # initialize hidden state with zero weights, and move to GPU if available
    if train_on_gpu:
        hidden = (weight.new(self.n_layers, batch_size, self.hidden_dim).zero_().cuda(),
                  weight.new(self.n_layers, batch_size, self.hidden_dim).zero_().cuda())

    else:
        hidden = (weight.new(self.n_layers, batch_size, self.hidden_dim).zero_(),
                  weight.new(self.n_layers, batch_size, self.hidden_dim).zero_())

    return hidden
```

Note: Alternatively, we can also use GRU (i.e. Gated Recurrent Units) as well in place of LSTM. GRU is related to LSTM as both are utilizing different way if gating information to prevent vanishing gradient problem.

- The GRU controls the flow of information like the LSTM unit, but without having to use a **memory unit**. It just exposes the full hidden content without any control.
- GRU is relatively new and performance is similar to LSTM but computationally **more efficient** because of less complex structure



```python
class RNN(nn.Module):

    def __init__(self, vocab_size, output_size, embedding_dim, hidden_dim, n_layers, dropout=0.5):

        super(RNN, self).__init__()

        # set class variables
        self.output_size = output_size
        self.n_layers = n_layers
        self.hidden_dim = hidden_dim

        # define model layers
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.gru = nn.GRU(embedding_dim, hidden_dim, n_layers, batch_first=True, dropout=dropout)
        self.fc = nn.Linear(hidden_dim, output_size)

    def forward(self, nn_input, hidden):

        batch_size = nn_input.size(0)
        embeds = self.embedding(nn_input)
        gru_out, hidden = self.gru(embeds, hidden)

        # stack up gru outputs
        gru_out = gru_out.contiguous().view(-1, self.hidden_dim)

        # fully connected layer
        output = self.fc(gru_out)

        # reshape into (batch_size, seq_length, output_size)
        output = output.view(batch_size, -1, self.output_size)
        # get last batch
        output = output[:, -1]
```

Backpropagation

Use the above RNN class to apply forward and back propagation

This function will be called iteratively, in the training loop

```python
def forward_back_prop(rnn, optimizer, criterion, inp, target, hidden):
    """
    Forward and backward propagation on the neural network
    :param decoder: The PyTorch Module that holds the neural network
    :param decoder_optimizer: The PyTorch optimizer for the neural network
    :param criterion: The PyTorch loss function
    :param inp: A batch of input to the neural network
    :param target: The target output for the batch of input
    :return: The loss and the latest hidden state Tensor
    """

    # move data to GPU, if available
    if train_on_gpu:
        inp, target = inp.cuda(), target.cuda()

    # create new variables for hidden/cell states, otherwise
    # we'd backprop through entire training history
    hidden = tuple([c.data for c in hidden])

    # clear gradients
    rnn.zero_grad()

    # forward pass to get the output and new hidden/cell states
    out, hidden = rnn(inp, hidden)

    # output loss
    loss = criterion(out, target)

    # perform backpropagation and optimization
    loss.backward()
    nn.utils.clip_grad_norm_(rnn.parameters(), 5)
    optimizer.step()

    # return the loss over a batch and the hidden state produced by our model
    return loss.item(), hidden
```

Training loop

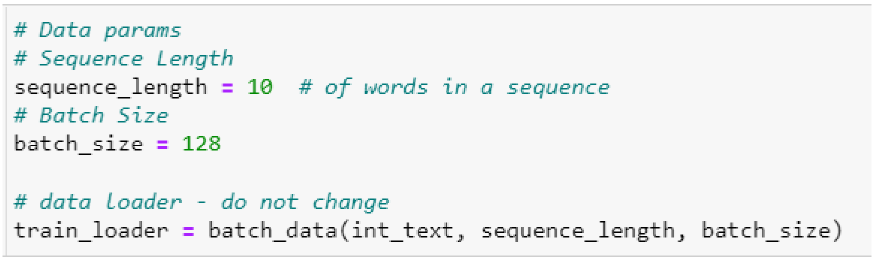
This function train the network over all the batches for the number of epochs given.

The model progress is printed every number of batches

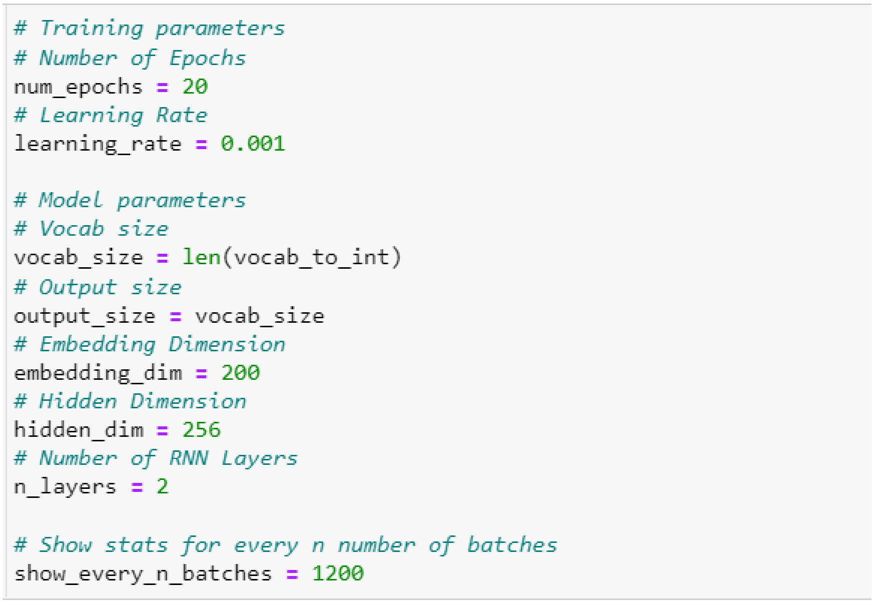
```
def train_rnn(rnn, batch_size, optimizer, criterion, n_epochs, show_every_n_batches=100):
    batch_losses = []

    rnn.train()

    print("Training for %d epoch(s)..." % n_epochs)
    for epoch_i in range(1, n_epochs + 1):

        # initialize hidden state
        hidden = rnn.init_hidden(batch_size)

        for batch_i, (inputs, labels) in enumerate(train_loader, 1):

            # make sure you iterate over completely full batches, only
            n_batches = len(train_loader.dataset)//batch_size
            if(batch_i > n_batches):
                break

            # forward, back prop
            loss, hidden = forward_back_prop(rnn, optimizer, criterion, inputs, labels, hidden)
            # record loss
            batch_losses.append(loss)

            # printing loss stats
            if batch_i % show_every_n_batches == 0:
                print('Epoch: {:>4}/{:<4}  Loss: {}\n'.format(
                    epoch_i, n_epochs, np.average(batch_losses)))
                batch_losses = []

    # returns a trained rnn
    return rnn
```

Hyperparameters

- **Sequence Length**: to set the length of the sequence
- **Batch Size** : to set the size of batches
- **Number of epochs** : to set the number of epochs to train for
- **Learning Rate**: to set the learning rate of Adam Optimizer
- **Vocab Size** : to set the number of unique tokens in vocabulary
- **Output Size** : to set the desired size of output
- **Embedding Dim** : to set the embedding dimension, smaller than vocab_size
- **Hidden Dim** : to set the hidden dimension of RNN
- **N Layer**: to set the number of layers in the RNN
- **Show N Batches**: to set the number of batches at which neural network should print progress



```
# Data params
# Sequence Length
sequence_length = 10  # of words in a sequence
# Batch Size
batch_size = 128

# data loader - do not change
train_loader = batch_data(int_text, sequence_length, batch_size)
```

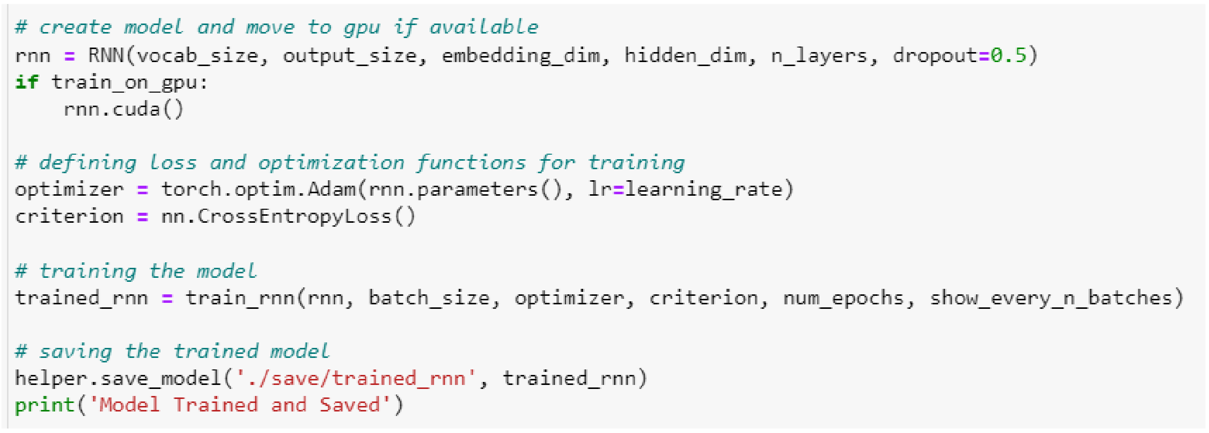


```
# Training parameters
# Number of Epochs
num_epochs = 20
# Learning Rate
learning_rate = 0.001

# Model parameters
# Vocab size
vocab_size = len(vocab_to_int)
# Output size
output_size = vocab_size
# Embedding Dimension
embedding_dim = 200
# Hidden Dimension
hidden_dim = 256
# Number of RNN Layers
n_layers = 2

# Show stats for every n number of batches
show_every_n_batches = 1200
```

Training Result

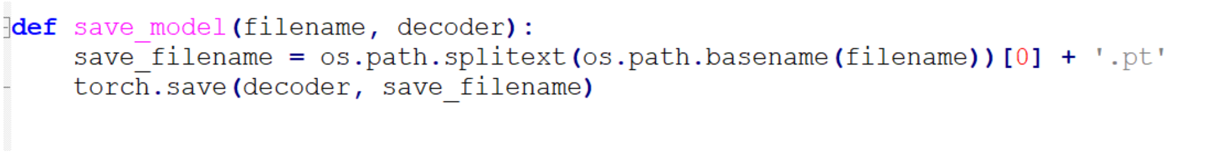
Before starting training, we are initializing:

- The optimizer as Adam using Pytorch's library. Details could be found here: (https://pytorch.org/docs/stable/generated/torch.optim.Adam.html#torch.optim.Adam)
- loss function as Cross Entropy loss function. Details could be found here: (https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html)

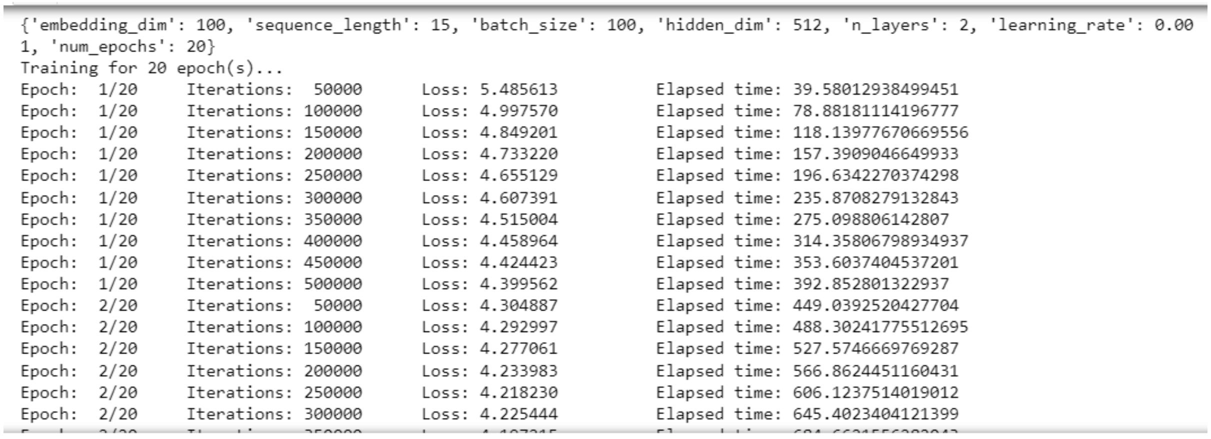


```python
# create model and move to gpu if available
rnn = RNN(vocab_size, output_size, embedding_dim, hidden_dim, n_layers, dropout=0.5)
if train_on_gpu:
    rnn.cuda()

# defining loss and optimization functions for training
optimizer = torch.optim.Adam(rnn.parameters(), lr=learning_rate)
criterion = nn.CrossEntropyLoss()

# training the model
trained_rnn = train_rnn(rnn, batch_size, optimizer, criterion, num_epochs, show_every_n_batches)

# saving the trained model
helper.save_model('./save/trained_rnn', trained_rnn)
print('Model Trained and Saved')
```

Saving the model as a physical checkpoint file after training is complete.



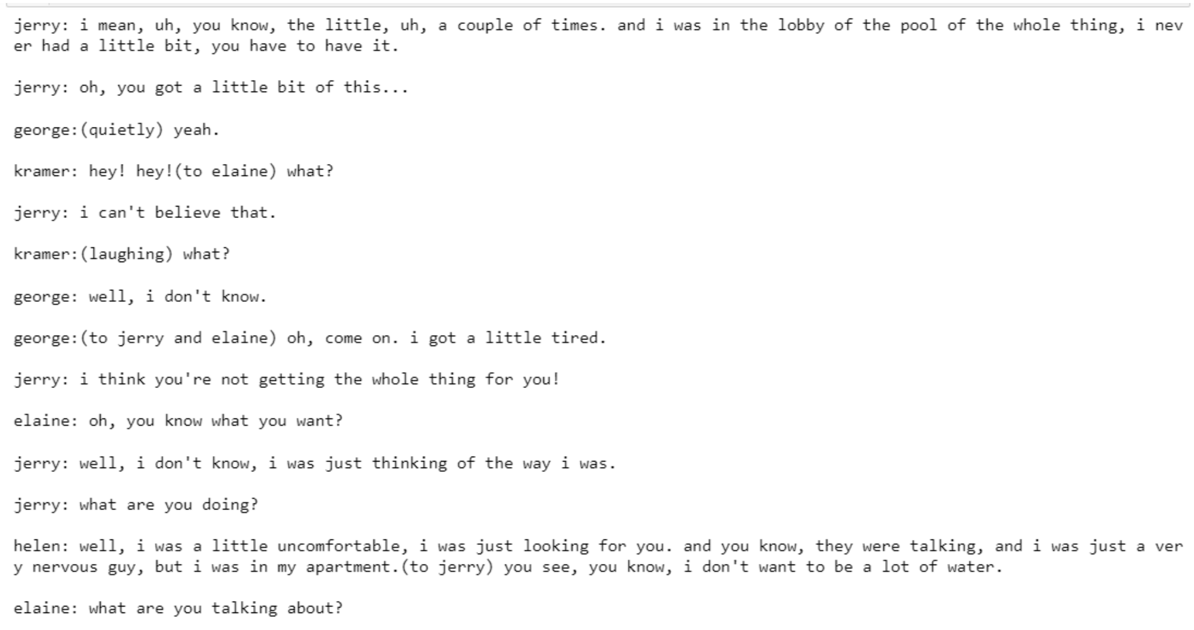
```python
def save_model(filename, decoder):
    save_filename = os.path.splitext(os.path.basename(filename))[0] + '.pt'
    torch.save(decoder, save_filename)
```



```
{'embedding_dim': 100, 'sequence_length': 15, 'batch_size': 100, 'hidden_dim': 512, 'n_layers': 2, 'learning_rate': 0.00
1, 'num_epochs': 20}
Training for 20 epoch(s)...
Epoch:  1/20     Iterations:  50000      Loss: 5.485613         Elapsed time: 39.58012938499451
Epoch:  1/20     Iterations: 100000      Loss: 4.997570         Elapsed time: 78.88181114196777
Epoch:  1/20     Iterations: 150000      Loss: 4.849201         Elapsed time: 118.13977670669556
Epoch:  1/20     Iterations: 200000      Loss: 4.733220         Elapsed time: 157.3909046649933
Epoch:  1/20     Iterations: 250000      Loss: 4.655129         Elapsed time: 196.6342270374298
Epoch:  1/20     Iterations: 300000      Loss: 4.607391         Elapsed time: 235.8708279132843
Epoch:  1/20     Iterations: 350000      Loss: 4.515004         Elapsed time: 275.098806142807
Epoch:  1/20     Iterations: 400000      Loss: 4.458964         Elapsed time: 314.35806798934937
Epoch:  1/20     Iterations: 450000      Loss: 4.424423         Elapsed time: 353.6037404537201
Epoch:  1/20     Iterations: 500000      Loss: 4.399562         Elapsed time: 392.852801322937
Epoch:  2/20     Iterations:  50000      Loss: 4.304887         Elapsed time: 449.0392520427704
Epoch:  2/20     Iterations: 100000      Loss: 4.292997         Elapsed time: 488.30241775512695
Epoch:  2/20     Iterations: 150000      Loss: 4.277061         Elapsed time: 527.5746669769287
Epoch:  2/20     Iterations: 200000      Loss: 4.233983         Elapsed time: 566.8624451160431
Epoch:  2/20     Iterations: 250000      Loss: 4.218230         Elapsed time: 606.1237514019012
Epoch:  2/20     Iterations: 300000      Loss: 4.225444         Elapsed time: 645.4023404121399
```

Observations

Script Generation



```
jerry: i mean, uh, you know, the little, uh, a couple of times. and i was in the lobby of the pool of the whole thing, i nev
er had a little bit, you have to have it.

jerry: oh, you got a little bit of this...

george:(quietly) yeah.

kramer: hey! hey!(to elaine) what?

jerry: i can't believe that.

kramer:(laughing) what?

george: well, i don't know.

george:(to jerry and elaine) oh, come on. i got a little tired.

jerry: i think you're not getting the whole thing for you!

elaine: oh, you know what you want?

jerry: well, i don't know, i was just thinking of the way i was.

jerry: what are you doing?

helen: well, i was a little uncomfortable, i was just looking for you. and you know, they were talking, and i was just a ver
y nervous guy, but i was in my apartment.(to jerry) you see, you know, i don't want to be a lot of water.

elaine: what are you talking about?
```

LSTM vs GRU vs Bidirectional RNN for Script Generation

According to the paper (Reference 3), the model for Tv script generation is trained in all three models based on LSTM, GRU and Bidirectional RNN. The performance of the models is further analysed to reach the conclusion that LSTM generates text in most efficient way followed by GRU and then Bidirectional RNN. Although the loss is least in Bidirectional RNN, followed by LSTM and then GRU.

References

1. Chung, Junyoung, et al. “Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling.” *Arxiv*, 11 12 2014, https://arxiv.org/pdf/1412.3555v1.pdf. Accessed 13 04 2020.
2. Dipti Pawade, et al. “Story Scrambler – Automatic Text Generation Using Word Level RNN LSTM.” *MECS*, 05 06 2018, https://www.mecs-press.org/ijitcs/ijitcs-v10-n6/IJITCS-V10-N6-5.pdf . Accessed 14 04 2020.
3. Sandhya Mangal, et al. “LSTM vs GRU vs Bidirectional RNN for script generation” *Arxiv,* 12 08 2019, https://arxiv.org/ftp/arxiv/papers/1908/1908.04332.pdf Accessed 15 04 2020.
4. Title: Deep Learning from Scratch: Building with Python from First Principles Publisher: O'Reilly Media, Incorporated Published year:2019 Author: Seth Weidman