University of New Haven

TAGLIATELA COLLEGE OF ENGINEERING

ECECS Department

DSCI 6004 Natural Language Processing

Final Project

A Simple Chatbot

By:

Srimanikanta Arjun Karimalammanavar

Course Instructor:

Dr. Muhammad Aminul Islam

TABLE OF CONTENTS

SECTION	DESCRIPTION	PAGE
1	Introduction	3
2	Project Motivation	3
3	Data	3
4	Model	10
5	Building the Model	34
6	Training the Model	38
7	Chatbot	42
8	Summary	43
9	Conclusion	44

1. Introduction

A chatbot is a software that provides a real conversational experience to the user. There are closed domain chatbots and open domain (generative) chatbots. Closed domain chatbot is a chatbot that responses with predefined texts. A generative chatbot generates a response as the name implies. In the Closed domain chatbots, the responses would fixed and machine learning would help to select the correct response given in the user's question. But here, we are not going to select from pre-defined responses but instead, we will generate a response based on the training corpus. We are going to use the encoder-decoder (seq2seq) model for this approach.

2. Project Motivation

1. Initially, learn the steps that would lead to build the building of a closed domain chatbot.
2. Prepare the data such that the model could use it to generate text instead of pre-defined responses.
3. Build seq2seq model (also called as encoder-decoder model).
4. Use LSTM units for text generation from the training corpus

3. Data Pre-Processing

3.1 Data

To create a chatbot there are numerous dataset available of various formats. Picking the dataset to train our chatbot depends on the purpose of creating the chatbot. We could use conversations between banker and investor to create an **E-commerce chatbot**, or just command following chatbot like **Amazon Alexa**, etc., The chatbot that we are going to designing is just a conversational chatbot. More than the data, the idea is to create a Deep Learning model that can be used to design any kind of user friendly chatbot.

We have used the data from Cornell Movie Dialogs Corpus. This corpus contains a large collection of fictional conversations extracted from movie scripts:

- 220,000+ conversational exchanges between 10,000+ pairs of movie characters
- Involves 9000+ characters from 600+ movies
- 300,000+ utterances

3.2 Lists of Data

We will be reading our data into a list splitting it by **"\n".** This will give us a list of conversations carried out by characters. From the below example, we can infer that "**Bianca"** carried out a conversation which has an **"ID" =** "**L1045"** in the movie **"m0"** and the character has a unique id – **"u0".** This data is from **"movie_lines.txt"**.

```
lines = open("movie_lines.txt", encoding = "UTF-8", errors = "ignore").read().split("\n")
lines[:5]
```

```
['L1045 +++$+++ u0 +++$+++ m0 +++$+++ BIANCA +++$+++ They do not!',
 'L1044 +++$+++ u2 +++$+++ m0 +++$+++ CAMERON +++$+++ They do to!',
 'L985 +++$+++ u0 +++$+++ m0 +++$+++ BIANCA +++$+++ I hope so.',
 'L984 +++$+++ u2 +++$+++ m0 +++$+++ CAMERON +++$+++ She okay?',
 "L925 +++$+++ u0 +++$+++ m0 +++$+++ BIANCA +++$+++ Let's go."]
```

Similarly, we will be reading the **"movie_conversations.txt"**. This will give us a list of conversations carried between characters. From the below example, we can infer that a conversation is carried out between **"u0"** and **"u2"** in the movie **"m0"** and the lines said are - **["L194", "L195", "L196", "L197].**

```
conversations = open('movie_conversations.txt', encoding = 'UTF-8', errors = 'ignore').read().split('\n')
conversations[:5]
```

```
["u0 +++$+++ u2 +++$+++ m0 +++$+++ ['L194', 'L195', 'L196', 'L197']",
 "u0 +++$+++ u2 +++$+++ m0 +++$+++ ['L198', 'L199']",
 "u0 +++$+++ u2 +++$+++ m0 +++$+++ ['L200', 'L201', 'L202', 'L203']",
 "u0 +++$+++ u2 +++$+++ m0 +++$+++ ['L204', 'L205', 'L206']",
 "u0 +++$+++ u2 +++$+++ m0 +++$+++ ['L207', 'L208']"]
```

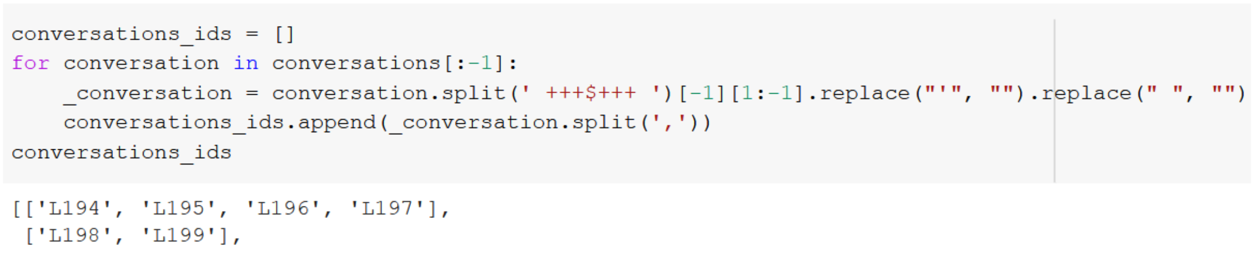
3.3 Dictionaries of Data

The next step would be creating a dictionary of conversations between characters with **dialog ID** as the **"keys"** and the **dialog** by the character as **"values".** We have a string **" +++$+++ "** which is not necessary. From below example we can infer that **"L1045"** said the line **"They do not",**

```
id2line = {}
for line in lines:
    _line = line.split(' +++$+++ ')
    if len(_line) == 5:
      id2line[_line[0]] = _line[4]
id2line
```

```
{'L1045': 'They do not!',
 'L1044': 'They do to!',
```

We are creating a list of lists of conversations by characters. From the below example we can infer that this list has 4 dialogs between 2 characters **[“L194”, “L195”, “L196”, “L197”],**

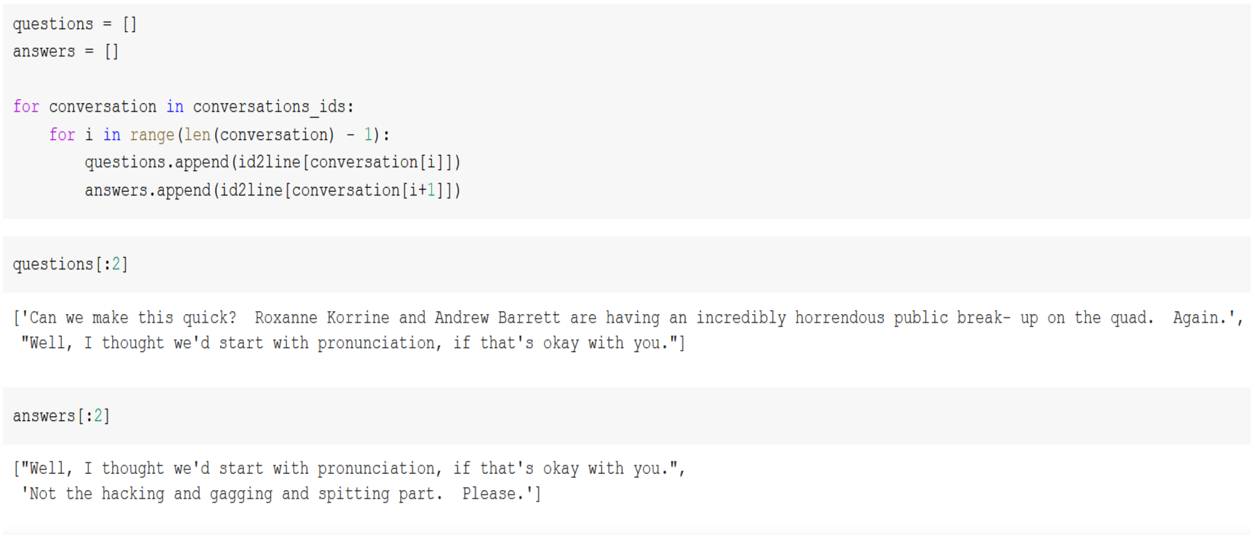


```
conversations_ids = []
for conversation in conversations[:-1]:
    _conversation = conversation.split(' +++$+++ ')[-1][1:-1].replace("'", "").replace(" ", "")
    conversations_ids.append(_conversation.split(','))
conversations_ids
```

```
[['L194', 'L195', 'L196', 'L197'],
 ['L198', 'L199'],
```

3.4 Questions and Answers

Now we need the list of questions and answers to feed our chatbot. By this we mean to just take the conversations and put them into 2 different lists. The questions list will have the **“ith”** conversation while the answers list will have the **“i+1th”** conversation basically treating the conversation as a question and answer session so our chatbot would be a machine which can chat with user.



```
questions = []
answers = []

for conversation in conversations_ids:
    for i in range(len(conversation) - 1):
        questions.append(id2line[conversation[i]])
        answers.append(id2line[conversation[i+1]])
```

```
questions[:2]
```

```
['Can we make this quick?  Roxanne Korrine and Andrew Barrett are having an incredibly horrendous public break- up on the quad.  Again.',
 "Well, I thought we'd start with pronunciation, if that's okay with you."]
```

```
answers[:2]
```

```
["Well, I thought we'd start with pronunciation, if that's okay with you.",
 'Not the hacking and gagging and spitting part.  Please.']
```

3.5 Cleaning Questions and Answers

We are cleaning the questions to remove punctuations and still make sure the sentences don’t lose the meaning. We need to apply this process to the list of questions and answers.

```
clean_questions = []
for question in questions:
    clean_questions.append(clean_text(question))
clean_questions[:2]
```

```
['can we make this quick  roxanne korrine and andrew barrett are having an incredibly horrendous public break up on the quad  again',
 'well i thought we would start with pronunciation if that is okay with you']
```

```
clean_answers = []
for answer in answers:
    clean_answers.append(clean_text(answer))
clean_answers[:2]
```

```
['well i thought we would start with pronunciation if that is okay with you',
 'not the hacking and gagging and spitting part  please']
```

3.6 Filtering the Questions and Answers

We would be creating 2 new lists **short_questions,** and **short_answers.**

We are filtering out the questions and answers and create a new list of questions and answers based on the length of the question/answer. Iterating through the list of **clean_questions** if the length of the question is between 2 and 25 as an ideal question length falls between these numbers, we are appending the question to the list of **short_questions** and the corresponding answers to the list of **short_answers.**

```
short_questions = []
short_answers = []
i = 0
for question in clean_questions:
    if 2 <= len(question.split()) <= 25:
        short_questions.append(question)
        short_answers.append(clean_answers[i])
    i += 1
short_questions[:2]
```

```
['can we make this quick  roxanne korrine and andrew barrett are having an incredibly horrendous public
 'well i thought we would start with pronunciation if that is okay with you']
```

```
short_answers[:2]
```

```
['well i thought we would start with pronunciation if that is okay with you',
 'not the hacking and gagging and spitting part  please']
```

Similarly, we are going to replace the elements from the **clean_questions** and **clean_answers** list. Similar to the above process we are going to iterate over **short_answers,** if the length of the answer is between 2 and 25, we are appending the answer to the list of **clean_answers** and the corresponding question to **clean_questions.**

```
clean_questions = []
clean_answers = []
i = 0
for answer in short_answers:
    if 2 <= len(answer.split()) <= 25:
        clean_answers.append(answer)
        clean_questions.append(short_questions[i])
    i += 1
```

```
clean_answers[:2]
```

```
['well i thought we would start with pronunciation if that is okay with you',
 'not the hacking and gagging and spitting part  please']
```

```
clean_questions[:2]
```

```
['can we make this quick  roxanne korrine and andrew barrett are having an incredibly horrendous
 'well i thought we would start with pronunciation if that is okay with you']
```

3.7 Word and its count

We are going to create a dictionary with **word** as the **"keys"** and their **count** as the **"values".** We would be iterating through the list **clean_questions** and **clean_answers** and add the element to the dictionary and increase its count if the word is encountered again.

```
word2count = {}
for question in clean_questions:
    for word in question.split():
        if word not in word2count:
            word2count[word] = 1
        else:
            word2count[word] += 1
for answer in clean_answers:
    for word in answer.split():
        if word not in word2count:
            word2count[word] = 1
        else:
            word2count[word] += 1
word2count
```

```
{'can': 9280,
 'we': 24122,
 'make': 3588,
 'this': 19390,
```

3.8 Mapping Word to an Integer

We are going to map a unique integer to a word. By this, each word will have a unique integer associated with it. We are going to use the words from **word2count** dictionary. Iterating through the dictionary we are going to get all the words which have a count of more than 15. So, any word which has a count of more than 15 will be added as keys to a new dictionary **questionwords2int** and add a unique integer beginning at **0** and going along until we reach the end of the **word2count** dictionary. This way every word will be mapped to unique integer.

```
threshold_answers = 15
answerswords2int = {}
word_number = 0
for word, count in word2count.items():
    if count >= threshold_answers:
        answerswords2int[word] = word_number
        word_number += 1
threshold_questions = 15
questionswords2int = {}
word_number = 0
for word, count in word2count.items():
    if count >= threshold_questions:
        questionswords2int[word] = word_number
        word_number += 1
questionswords2int
```

```
{'can': 0,
 'we': 1,
 'make': 2,
 'this': 3,
 'quick': 4,
 'and': 5,
```

3.9 Adding Tokens

Now we are going to define our necessary tokens that we need for our encoder and decoder in our seq2seq model. We are going to define a list of tokens - **['<PAD>', '<EOS>', '<OUT>', '<SOS>']**. We would be adding these tokens to the **questionswords2int** and **answerswords2int** dictionaries and mapping the tokens to unique integer using similar method that we followed in the previous steps.

```
tokens = ['<PAD>', '<EOS>', '<OUT>', '<SOS>']

for token in tokens:
    questionswords2int[token] = len(questionswords2int) +1

for token in tokens:
    answerswords2int[token] = len(answerswords2int) +1
```

3.10 Mapping Questions and Answers to integers

We are going to create a list which would contain questions and answers as integers using the **questionswords2int** and **answerswords2int.** Iterating through the **clean_questions,** we are adding the integer equivalent values of each word in the place of the word. This way we get a list of questions and answers equivalent integers.

```
questions_into_int = []

for question in clean_questions:
    ints= []
    for word in question.split():
        if word not in questionswords2int:
            ints.append(questionswords2int['<OUT>'])
        else:
            ints.append(questionswords2int[word])
    questions_into_int.append(ints)
questions_into_int[0]
```

```
answers_into_int = []

for answer in clean_answers:
    ints = []
    for word in answer.split():
        if word not in answerswords2int:
            ints.append(answerswords2int['<OUT>'])
        else:
            ints.append(answerswords2int[word])
    answers_into_int.append(ints)
```

3.11 Sorting Questions and Answers

Now we are creating two new lists **sorted_clean_questions** and **sorted_clean_answers** which contain the questions and answers (respectively) sorted according to the length of the questions in the **questions_into_int** list.

```
sorted_clean_questions = []
sorted_clean_answers = []

for length in range(1, 25+1):
    for i in enumerate(questions_into_int):
        if len(i[1]) == length:
            sorted_clean_questions.append(questions_into_int[i[0]])
            sorted_clean_answers.append(answers_into_int[i[0]])
```

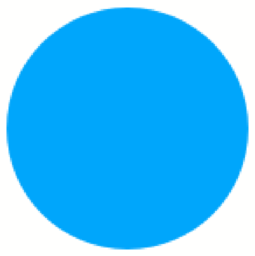
4. MODEL

4.1 Recurrent Neural Networks (RNNs)

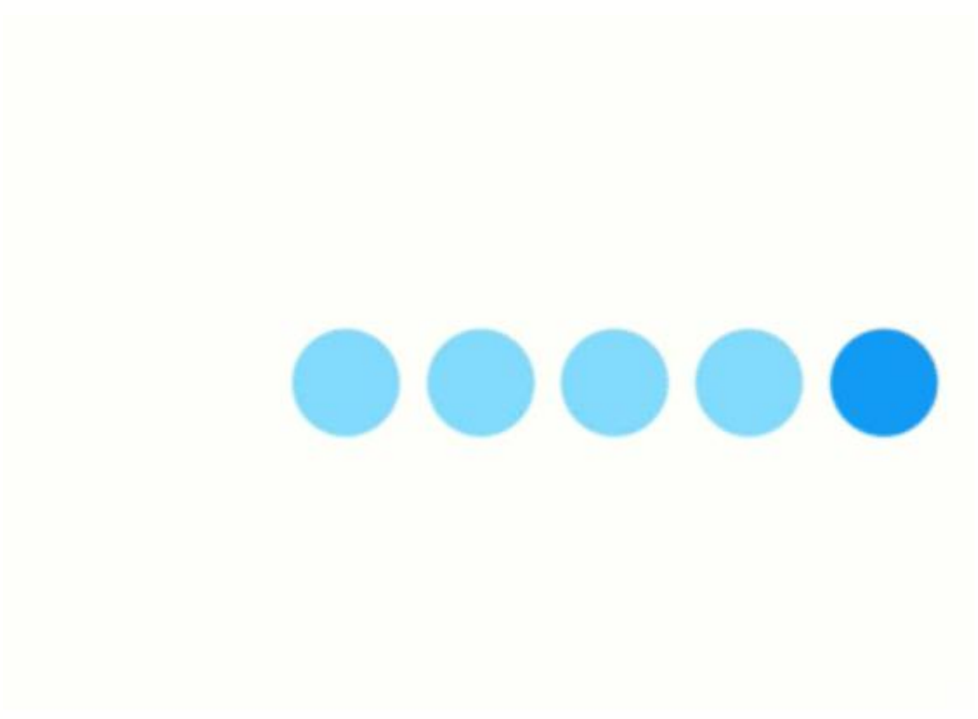
Getting into Deep Learning, Recurrent Neural Networks (RNNs) are a powerful technique that are important to understand. If you use a smartphone or frequently surf the internet, odds are you've used applications that leverages RNN's. Recurrent neural networks are used in speech recognition, language translation, stock predictions; It's even used in image recognition to describe the content in pictures.

4.1.1 Sequence Data

RNNs are neural networks that are good at modeling sequence data. To understand what that means let's do a thought experiment. Say you take a still snapshot of a ball moving in time.



Let's say you want to predict the direction that the ball was moving. It would be hard to do so with a static ball and any prediction would be a random guess and might be wrong. Without the knowledge of where the ball has been, we wouldn't have enough data to predict where its going. If you record many snapshots of the ball's position in succession, you will have enough information to make a better prediction.



So, this is a sequence, a particular order in which one thing follows another. With this information, you can now see that the ball is moving to right.

Sequence data comes in many forms. Audio is a natural sequence. We can chop up an audio spectrum and feed it into RNNs. Text is another form of sequence. We can break up Text into a sequence of characters or sequence of words.

4.1.2 Sequential Memory

Ok so, RNN's are good at processing sequence data for predictions. But how??

Well, they do that by having a concept I like to call sequential memory. To get a good intuition behind what sequential memory means…

I want to invite you to say the alphabet in your head.

That was easy right. If you were taught this specific sequence, it should come quickly to you.

Now try saying the alphabet backward.

Z Y X W V U T S R Q P O N M L K J I H G F E D C B A

I bet this is much harder. Unless you've practiced this specific sequence before, you'll likely have a hard time.

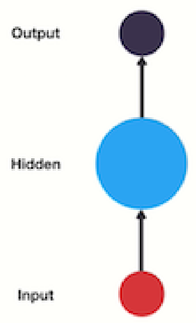
Here's a fun one, start at the letter M and try completing the sequence...,

At first, you'll struggle with the first few letters, but then after your brain picks up the pattern, the rest will come naturally.

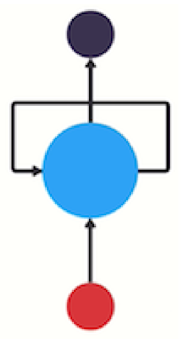
So, there is a very logical reason why this can be difficult. You learn the alphabet as a sequence. Sequential memory is a mechanism that makes it easier for your brain to recognize sequence patterns.

4.1.3 Working of Recurrent Neural Network

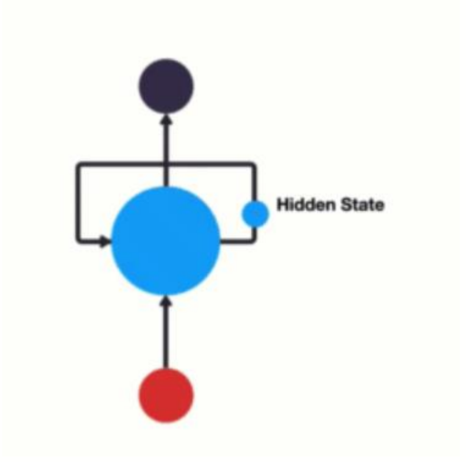
Alright so RNN's have this abstract concept of sequential memory, but how the heck does an RNN replicate this concept? Well, let's look at a traditional neural network also known as a feed-forward neural network. It has its input layer, hidden layer, and the output layer.



How do we get a feed-forward neural network to be able to use previous information to effect later ones? What if we add a loop in the neural network that can pass prior information forward?



And that's essentially what a recurrent neural network does. An RNN has a looping mechanism that acts as a highway to allow information to flow from one step to the next.



This information is the hidden state, which is a representation of previous inputs. Let's run through an RNN use case to have a better understanding of how this works.

Let's say we want to build a chatbot. They're popular nowadays. Let's say the chatbot can classify intentions from the users inputted text.

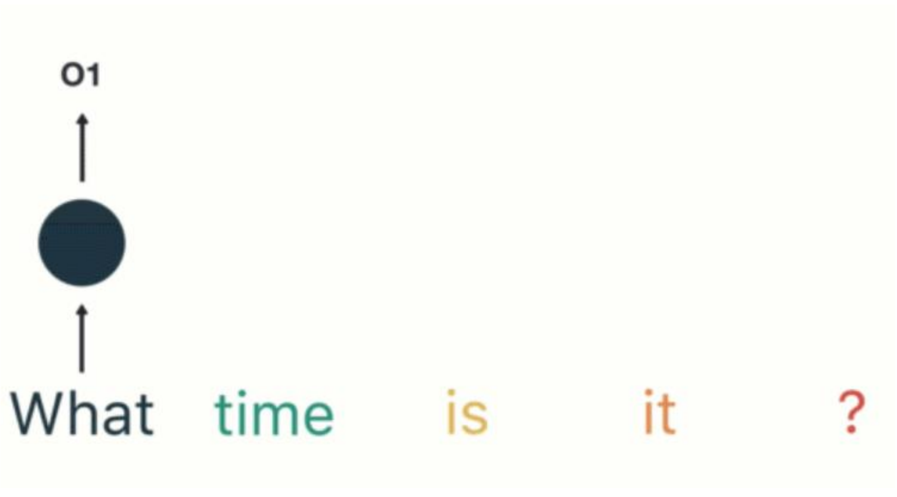
To tackle this problem. First, we are going to encode the sequence of text using an RNN. Then, we are going to feed the RNN output into a feed-forward neural network which will classify the intents.

Ok, so a user types in… ***what time is it?*** To start, we break up the sentence into individual words. RNN's work sequentially so we feed it one word at a time.

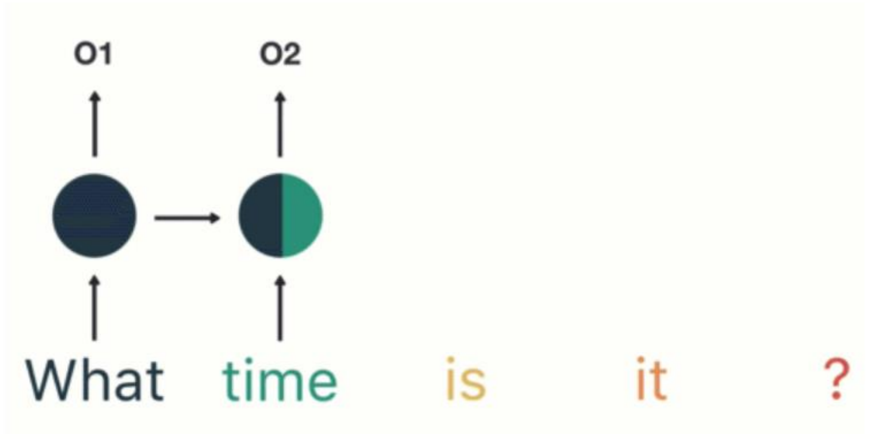
What
time
is
it
?

Breaking up a sentence into word sequences

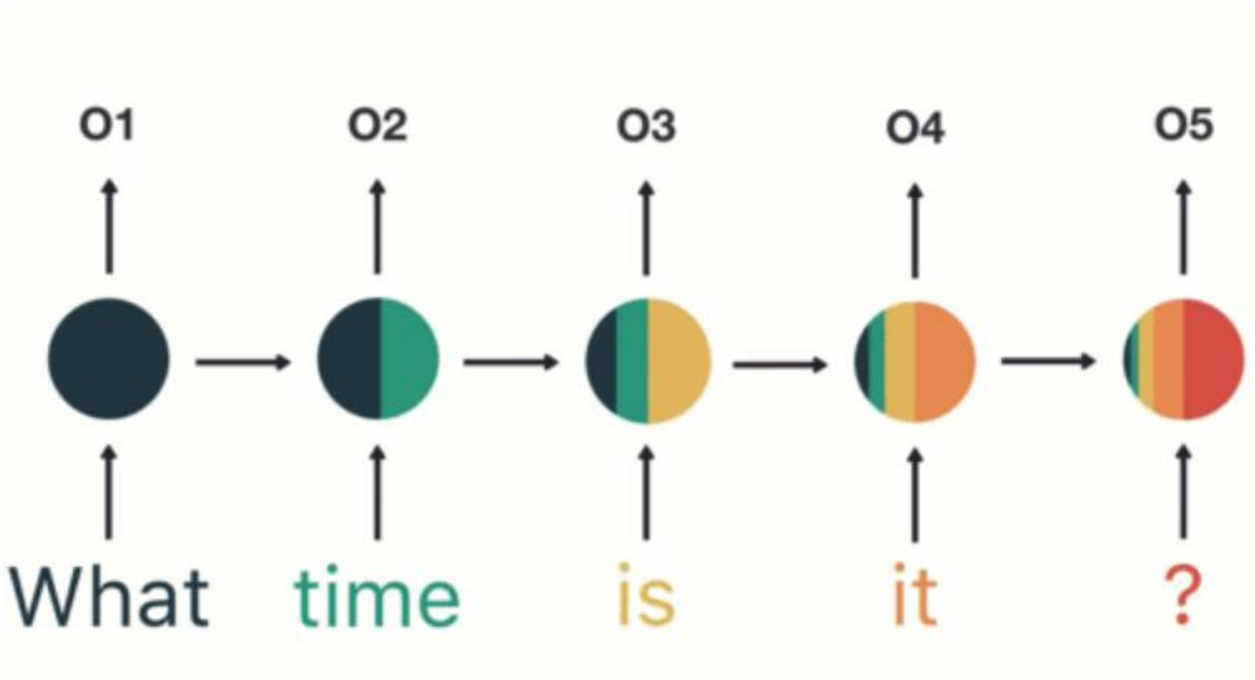
The first step is to feed **"What"** into the RNN. The RNN encodes "What" and produces an output.



For the next step, we feed the word **"time"** and the hidden state from the previous step. The RNN now has information on **both** the word **"What"** and **"time".**

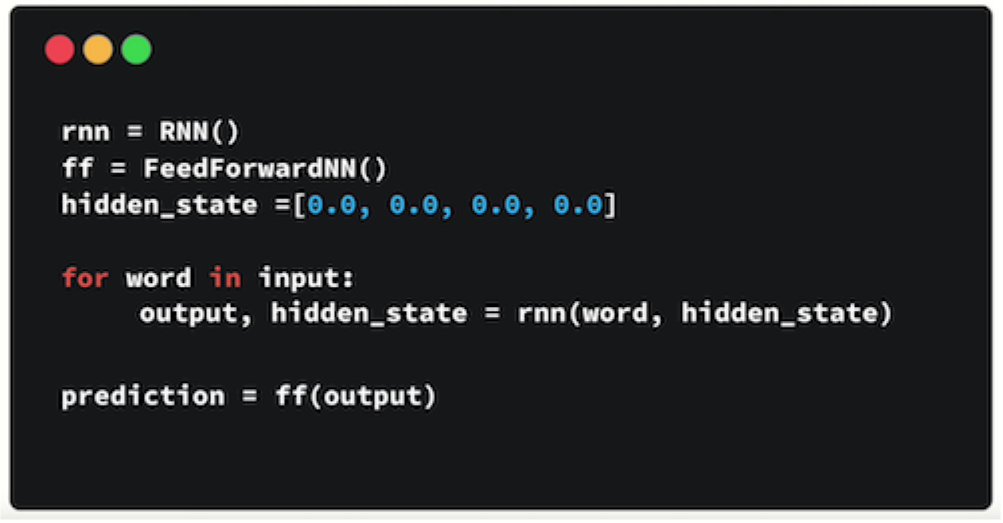


We repeat this process, until the final step. You can see by the final step the RNN has encoded information from all the words in previous steps.



Since the final output was created from the rest of the sequence, we should be able to take the final output and pass it to the feed-forward layer to classify an intent.

A Python Pseudo-code showcasing the control flow…,

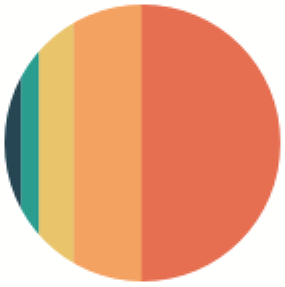


```
rnn = RNN()
ff = FeedForwardNN()
hidden_state =[0.0, 0.0, 0.0, 0.0]

for word in input:
     output, hidden_state = rnn(word, hidden_state)

prediction = ff(output)
```

First, you initialize your network layers and the initial hidden state. The shape and dimension of the hidden state will be dependent on the shape and dimension of your recurrent neural network. Then you loop through your inputs, pass the word and hidden state into the RNN. The RNN returns the output and a modified hidden state. You continue to loop until you're out of words. Last you pass the output to the feedforward layer, and it returns a prediction. And that's it! The control flow of doing a forward pass of a recurrent neural network is a for loop.

4.1.4 Vanishing Gradient

You may have noticed the odd distribution of colors in the hidden states. That is to illustrate an issue with RNN's known as short-term memory.

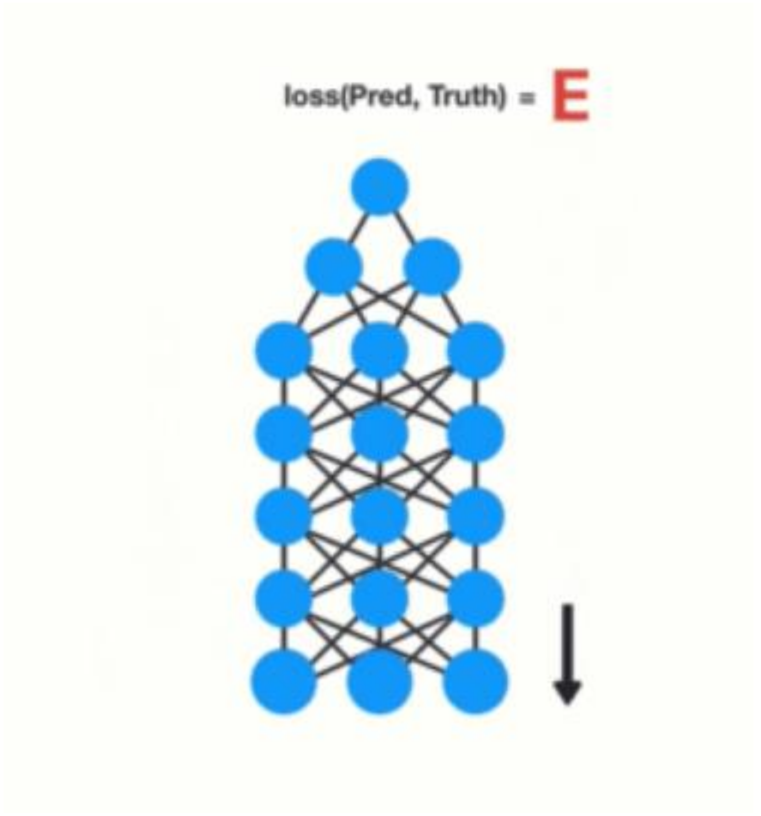


Final Hidden State of the RNN

Short-term memory is caused by the infamous vanishing gradient problem, which is also prevalent in other neural network architectures. As the RNN processes more steps, it has troubles retaining information from previous steps. As you can see, the information from the word “what” and “time” is almost non-existent at the final time step. Short-Term memory and the vanishing gradient are due to the nature of back-propagation; an algorithm used to train and optimize neural networks. To understand why this is, let's look at the effects of back propagation on a deep feed-forward neural network.

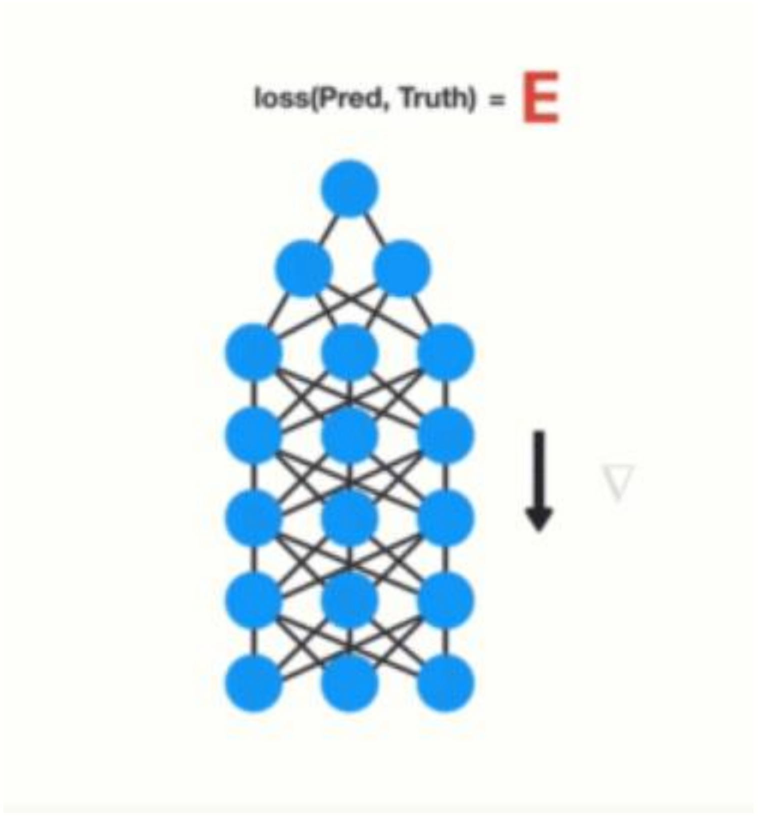
Training a neural network has three major steps.

1. It does a forward pass and makes a prediction.
2. It compares the prediction to the ground truth using a loss function. The loss function outputs an error value which is an estimate of how poorly the network is performing.
3. Last, it uses that error value to do back propagation which calculates the gradients for each node in the network.



The gradient is the value used to adjust the networks internal weights, allowing the network to learn. The bigger the gradient, the bigger the adjustments and vice versa. Here is where the problem lies. When doing back propagation, each node in a layer calculates its gradient with respect to the effects of the gradients, in the layer before it. So, if the adjustments to the layers before it is small, then adjustments to the current layer will be even smaller.

That causes gradients to exponentially shrink as it back propagates down. The earlier layers fail to do any learning as the internal weights are barely being adjusted due to extremely small gradients. And that’s the vanishing gradient problem.



Gradient Shrinks as it back-propagates down

Let’s see how this applies to recurrent neural networks. You can think of each time step in a recurrent neural network as a layer. To train a recurrent neural network, you use an application of back-propagation called back-propagation through time. The gradient values will exponentially shrink as it propagates through each time step.

Again, the gradient is used to make adjustments in the neural networks weights thus allowing it to learn. Small gradients mean small adjustments. That causes the early layers not to learn.

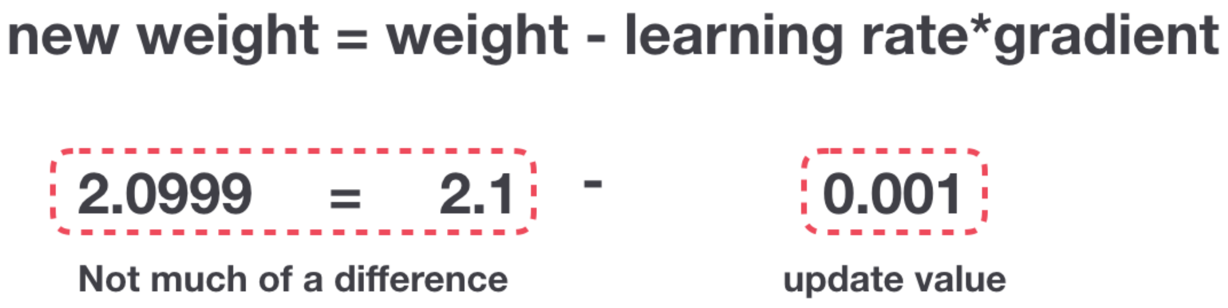
Because of vanishing gradients, the RNN doesn’t learn the long-range dependencies across time steps. That means that there is a possibility that the word “what” and “time” are not considered when trying to predict the user’s intention. The network then must make the best guess with “is it?”. That’s ambiguous and would be difficult even for a human. So not being able to learn on earlier time steps causes the network to have a short-term memory.

Ok so RNN’s suffer from short-term memory, so how do we combat that? To mitigate short-term memory, two specialized recurrent neural networks were created. One called Long Short-Term Memory or LSTM’s for short. The other is Gated Recurrent Units or GRU’s. LSTM’s and GRU’s essentially function just like RNN’s, but they’re capable of learning long-term dependencies using mechanisms called “gates.” These gates are different tensor operations that can learn what information to add or remove to the hidden state. Because of this ability, short-term memory is less of an issue for them.

4.1.5 The Problem, Short-Term Memory

Recurrent Neural Networks suffer from short-term memory. If a sequence is long enough, they’ll have a hard time carrying information from earlier time steps to later ones. So, if you are trying to process a paragraph of text to do predictions, RNN’s may leave out important information from the beginning.

During back propagation, recurrent neural networks suffer from the vanishing gradient problem. Gradients are values used to update a neural networks weight. The vanishing gradient problem is when the gradient shrinks as it back propagates through time. If a gradient value becomes extremely small, it doesn’t contribute too much learning.



So, in recurrent neural networks, layers that get a small gradient update stops learning. Those are usually the earlier layers. So, because these layers don’t learn, RNN’s can forget what it seen in longer sequences, thus having a short-term memory.

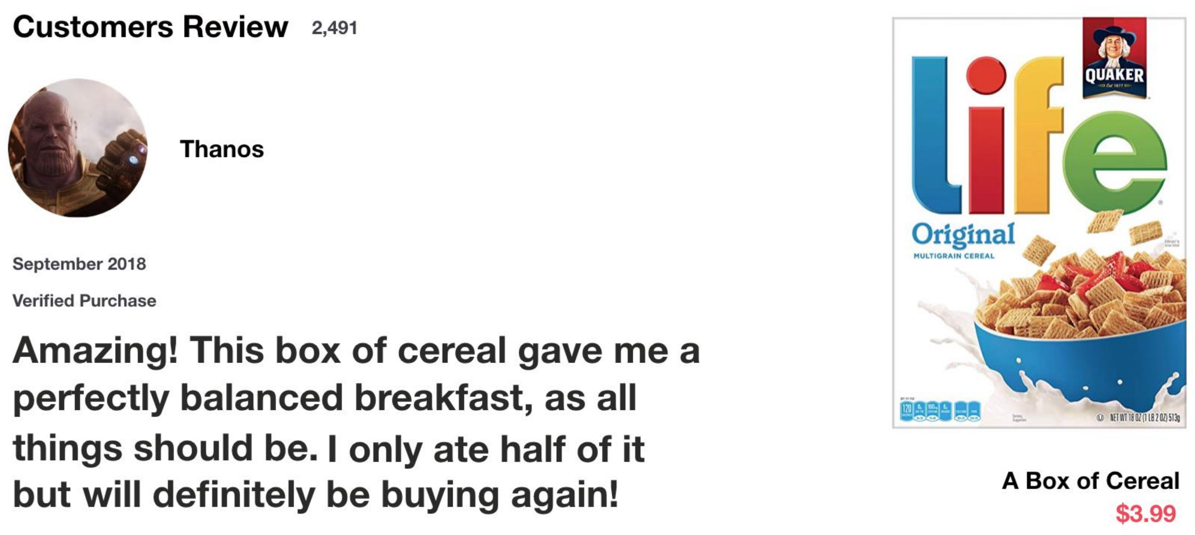
4.1.6 LSTMs and GRUs as solution

LSTM ’s and GRU’s were created as the solution to short-term memory. They have internal mechanisms called gates that can regulate the flow of information.

These gates can learn which data in a sequence is important to keep or throw away. By doing that, it can pass relevant information down the long chain of sequences to make predictions. Almost all state-of-the-art results based on recurrent neural networks are achieved with these two networks. LSTM’s and GRU’s can be found in speech recognition, speech synthesis, and text generation. You can even use them to generate captions for videos.

4.1.7 Intuition for LSTM

Ok, Let’s start with a thought experiment. Let’s say you’re looking at reviews online to determine if you want to buy Life cereal (don’t ask me why). You’ll first read the review then determine if someone thought it was good or if it was bad.



Customers Review 2,491

Thanos

September 2018

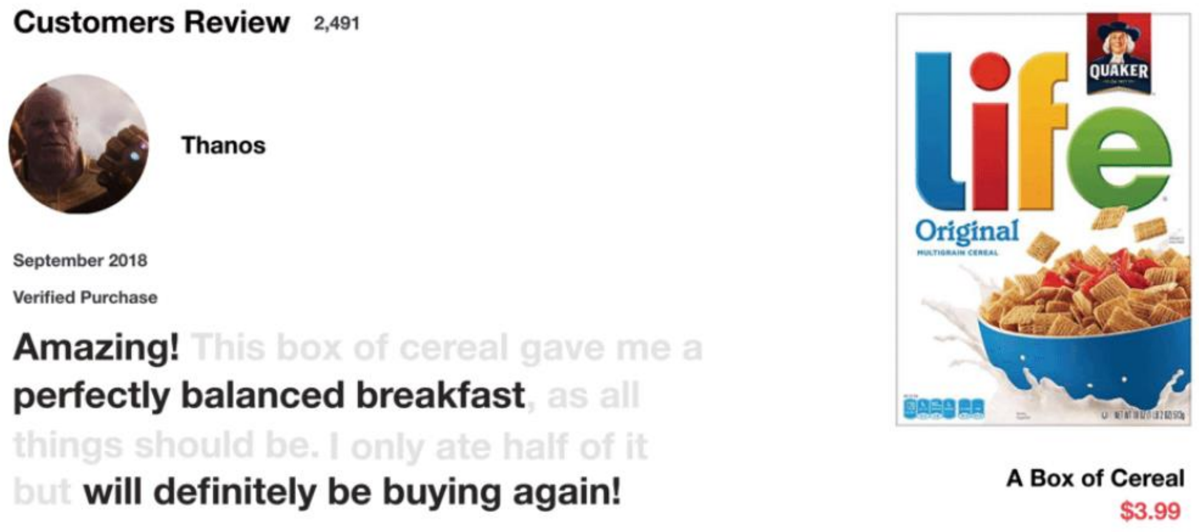
Verified Purchase

Amazing! This box of cereal gave me a perfectly balanced breakfast, as all things should be. I only ate half of it but will definitely be buying again!

A Box of Cereal

$3.99

When you read the review, your brain subconsciously only remembers important keywords. You pick up words like “amazing” and “perfectly balanced breakfast”. You don’t care much for words like “this”, “gave“, “all”, “should”, etc. If a friend asks you the next day what the review said, you probably wouldn’t remember it word for word. You might remember the main points though like “will definitely be buying again”. If you’re a lot like me, the other words will fade away from memory.



Customers Review 2,491

Thanos

September 2018

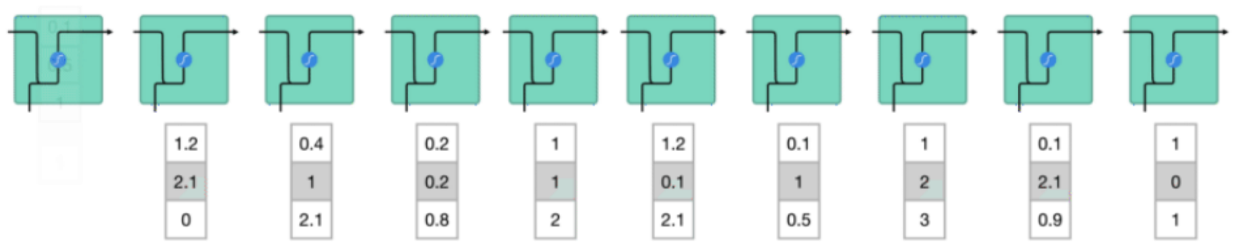
Verified Purchase

Amazing! This box of cereal gave me a **perfectly balanced breakfast**, as all things should be. I only ate half of it but **will definitely be buying again!**

And that is essentially what an LSTM or GRU does. It can learn to keep only relevant information to make predictions and forget non relevant data. In this case, the words you remembered made you judge that it was good.

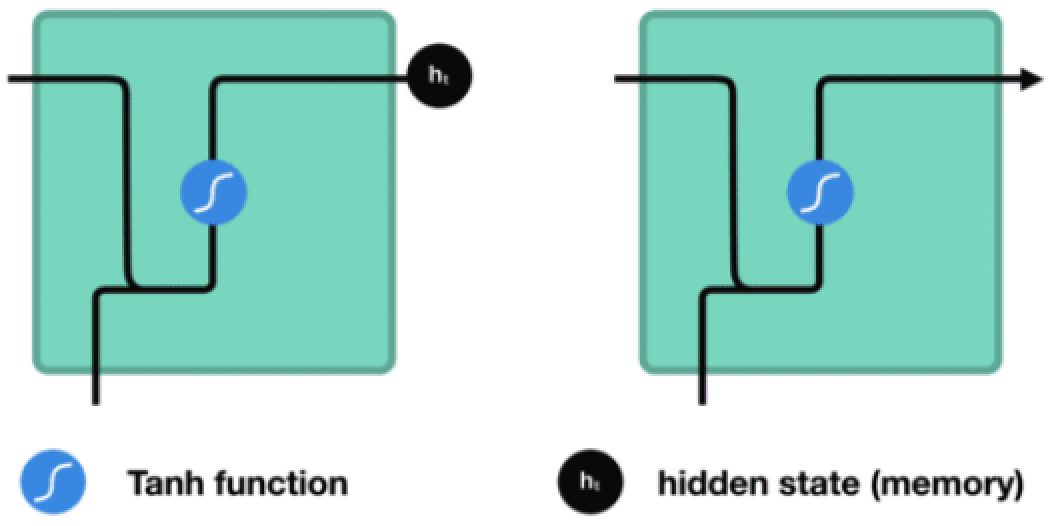
4.1.8 Review of RNNs

To understand how LSTM’s or GRU’s achieves this, let’s review the recurrent neural network. An RNN works like this; First words get transformed into machine-readable vectors. Then the RNN processes the sequence of vectors one by one.



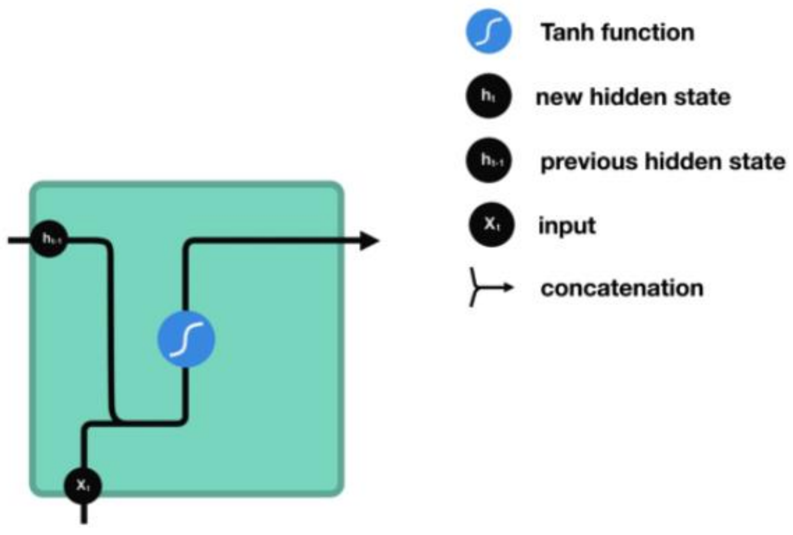
Processing sequence one by one

While processing, it passes the previous hidden state to the next step of the sequence. The hidden state acts as the neural networks’ memory. It holds information on previous data the network has seen before.



Passing hidden state to next time step

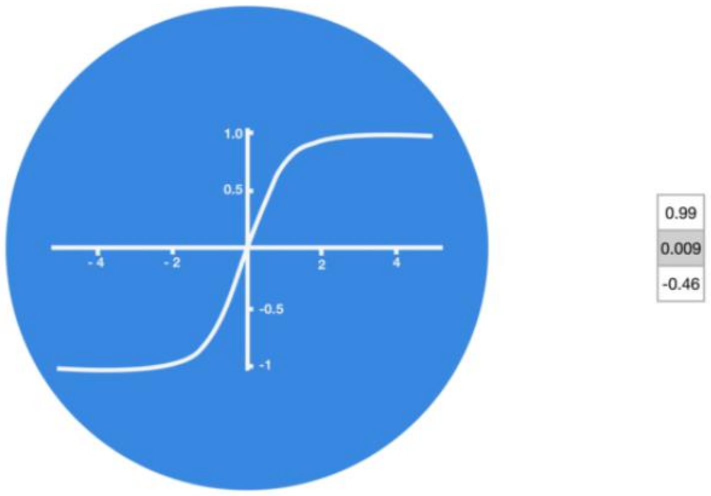
Let’s look at a cell of the RNN to see how you would calculate the hidden state. First, the input and previous hidden state are combined to form a vector. That vector now has information on the current input and previous inputs. The vector goes through the tanh activation, and the output is the new hidden state, or the memory of the network.



RNN Cell

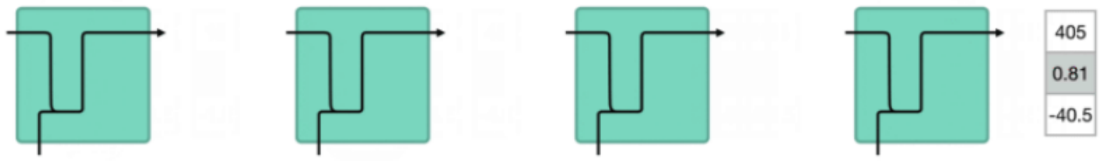
4.1.9 Tanh Activation

The tanh activation is used to help regulate the values flowing through the network. The tanh function squishes values to always be between -1 and 1.



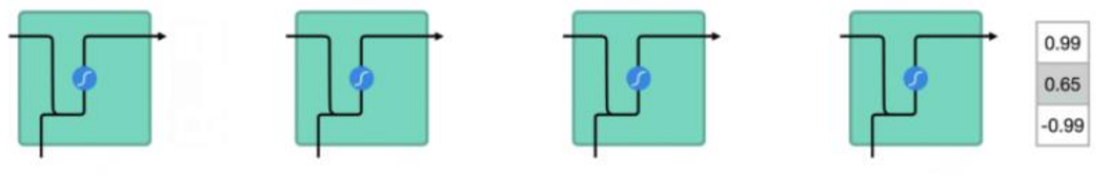
Tanh squishes values to be between -1 and 1

When vectors are flowing through a neural network, it undergoes many transformations due to various math operations. So, imagine a value that continues to be multiplied by let's say ***3***. You can see how some values can explode and become astronomical, causing other values to seem insignificant.



vector transformations without tanh

A tanh function ensures that the values stay between -1 and 1, thus regulating the output of the neural network. You can see how the same values from above remain between the boundaries allowed by the tanh function.

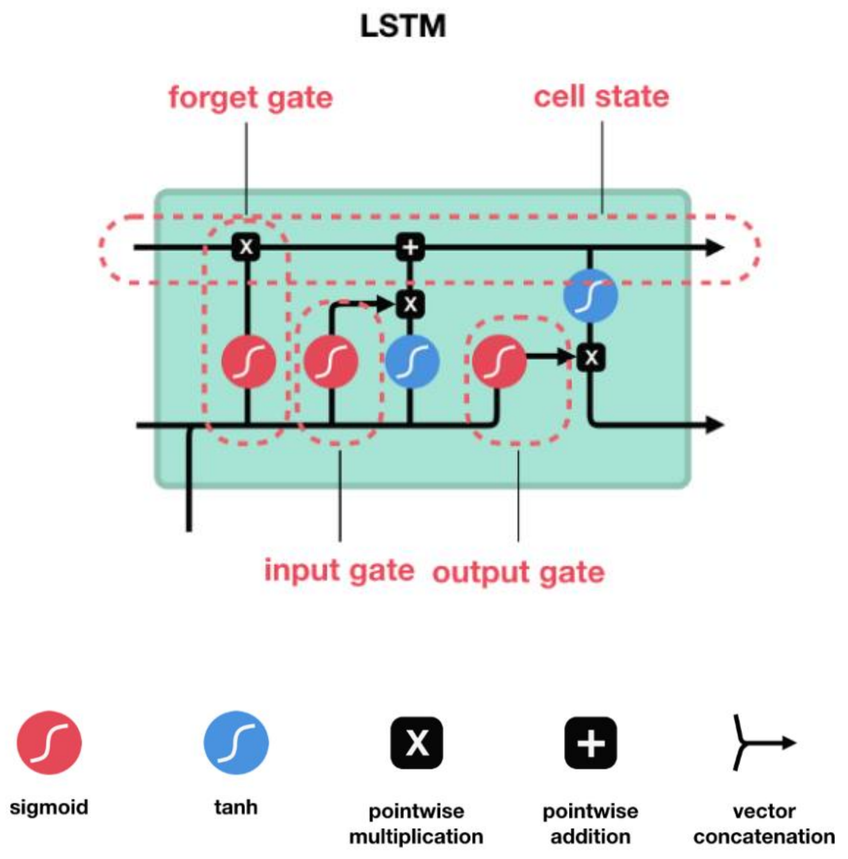


vector transformations with tanh

So that’s an RNN. It has very few operations internally but works well given the right circumstances (like short sequences). RNN’s uses a lot less computational resources than its evolved variants, LSTM’s and GRU’s.

4.2 LSTM

An LSTM has a similar control flow as a recurrent neural network. It processes data passing on information as it propagates forward. The differences are the operations within the LSTM’s cells.



These operations are used to allow the LSTM to keep or forget information. Now looking at these operations can get a little overwhelming, so we'll go over this step by step.

4.2.1 Core Concept

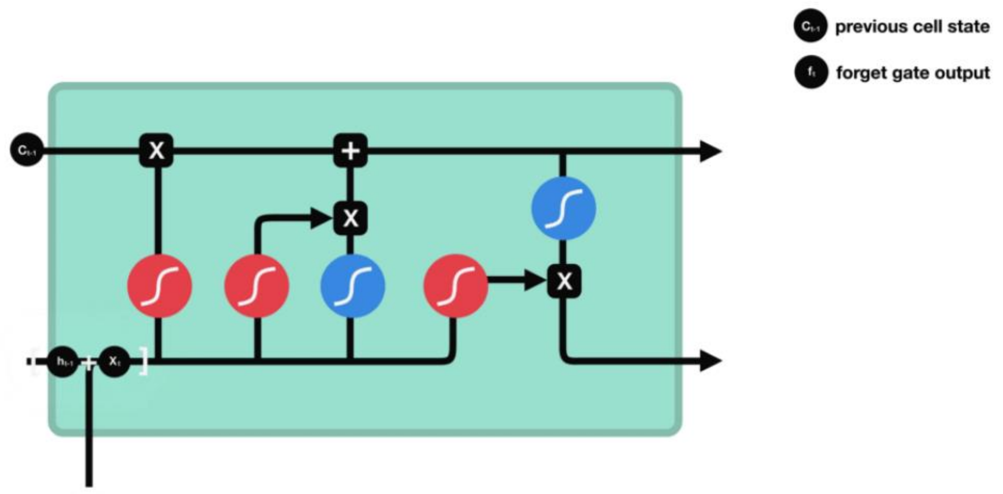
The core concept of LSTM's is the cell state, and its various gates. The cell state act as a transport highway that transfers relative information all the way down the sequence chain. You can think of it as the "memory" of the network. The cell state, in theory, can carry relevant information throughout the processing of the sequence. So even information from the earlier time steps can make its way to later time steps, reducing the effects of short-term memory. As the cell state goes on its journey, information gets added or removed to the cell state via gates. The gates are different neural networks that decide which information is allowed on the cell state. The gates can learn what information is relevant to keep or forget during training.

4.2.2 Sigmoid

Gates contains sigmoid activations. A sigmoid activation is like the tanh activation. Instead of squishing values between -1 and 1, it squishes values between 0 and 1. That is helpful to update or forget data because any number getting multiplied by 0 is 0, causing values to disappears or be "forgotten." Any number multiplied by 1 is the same value therefore that value stay's the same or is "kept." The network can learn which data is not important therefore can be forgotten or which data is important to keep.

4.2.3 Forget State

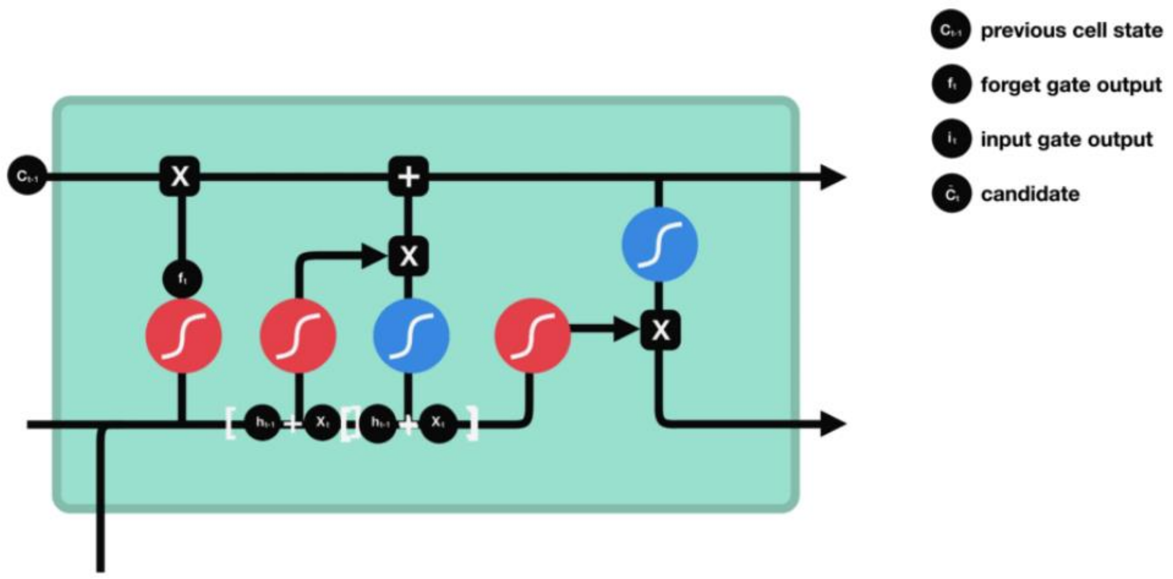
First, we have the forget gate. This gate decides what information should be thrown away or kept. Information from the previous hidden state and information from the current input is passed through the sigmoid function. Values come out between 0 and 1. The closer to 0 means to forget, and the closer to 1 means to keep.



Forget gate operations

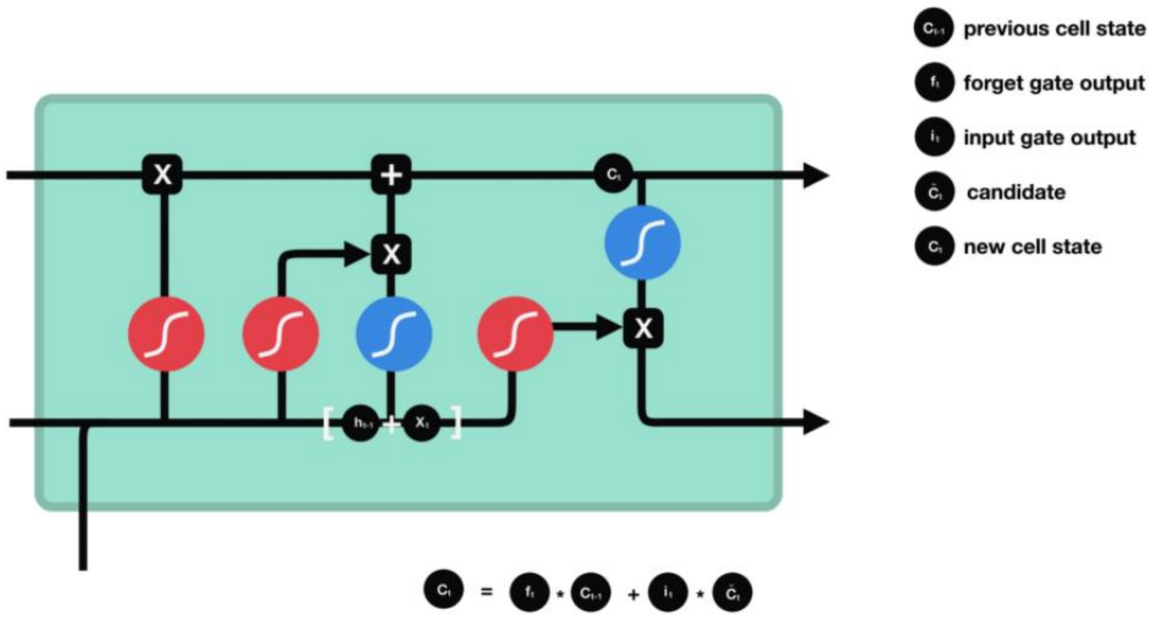
4.2.4 Input Gate

To update the cell state, we have the input gate. First, we pass the previous hidden state and current input into a sigmoid function. That decides which values will be updated by transforming the values to be between 0 and 1. 0 means not important, and 1 means important. You also pass the hidden state and current input into the tanh function to squish values between -1 and 1 to help regulate the network. Then you multiply the tanh output with the sigmoid output. The sigmoid output will decide which information is important to keep from the tanh output.



4.2.5 Cell State

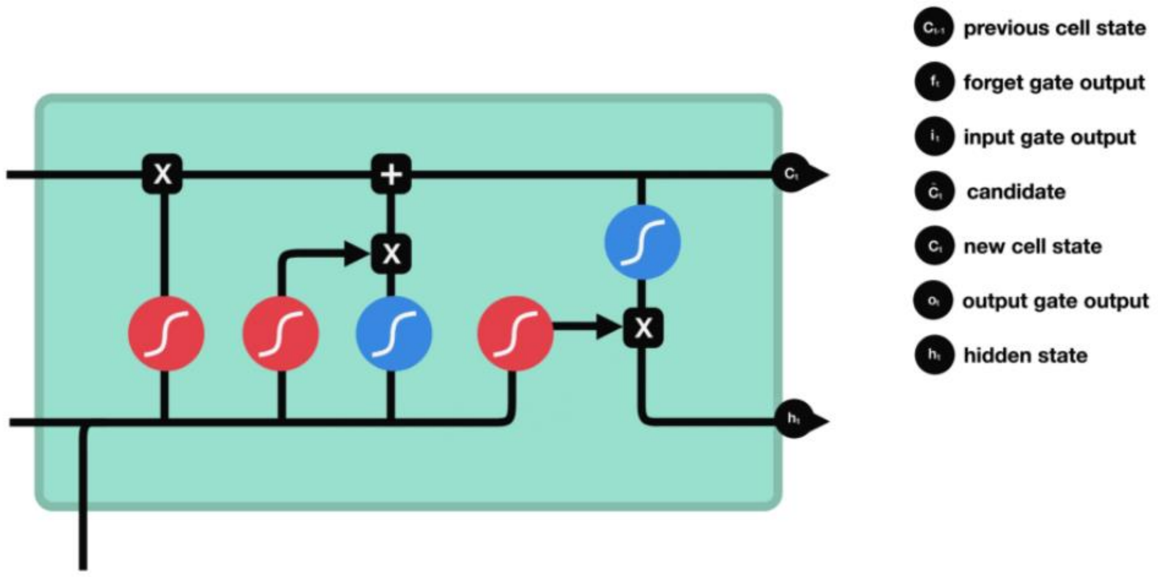
Now we should have enough information to calculate the cell state. First, the cell state gets pointwise multiplied by the forget vector. This has a possibility of dropping values in the cell state if it gets multiplied by values near 0. Then we take the output from the input gate and do a pointwise addition which updates the cell state to new values that the neural network finds relevant. That gives us our new cell state.



Calculating cell state

4.2.6 Output Gate

Last, we have the output gate. The output gate decides what the next hidden state should be. Remember that the hidden state contains information on previous inputs. The hidden state is also used for predictions. First, we pass the previous hidden state and the current input into a sigmoid function. Then we pass the newly modified cell state to the tanh function. We multiply the tanh output with the sigmoid output to decide what information the hidden state should carry. The output is the hidden state. The new cell state and the new hidden is then carried over to the next time step.



output gate operations

To review, the Forget gate decides what is relevant to keep from prior steps. The input gate decides what information is relevant to add from the current step. The output gate determines what the next hidden state should be.

4.2.7 Python Pseudo-Code

```
def LSTMCELL(prev_ct, prev_ht, input):

    combine = prev_ht + input

    ft = forget_layer(combine)

    candidate = candidate_layer(combine)

    it = input_layer(combine)

    Ct = prev_ct * ft + combine * it

    ot = output_layer(combine)

    ht = ot * tanh(Ct)

    return ht, Ct

ct = [0, 0, 0]

ht = [0, 0, 0]
```

```
for input in inputs:
        ct, ht = LSTMCELL(ct, ht, input)
```

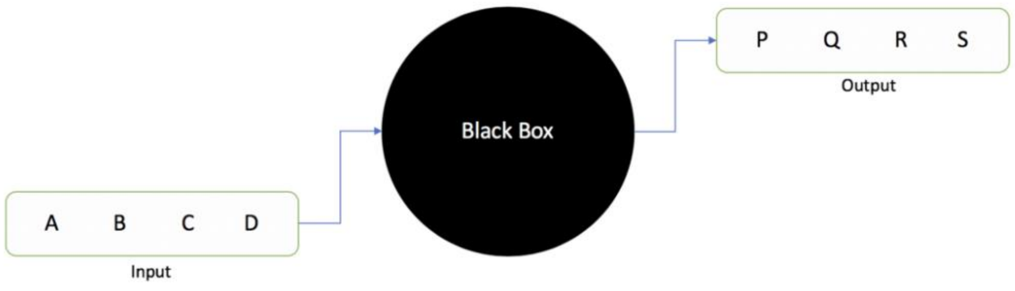
1. First, the previous hidden state and the current input get concatenated. We'll call it *combine*.
2. *Combine* gets fed into the forget layer. This layer removes non-relevant data.
3. A candidate layer is created using *combine*. The candidate holds possible values to add to the cell state.
4. *Combine* also gets fed into the input layer. This layer decides what data from the candidate should be added to the new cell state.
5. After computing the forget layer, candidate layer, and the input layer, the cell state is calculated using those vectors and the previous cell state.
6. The output is then computed.
7. Pointwise multiplying the output and the new cell state gives us the new hidden state.

The control flow of an LSTM network are a few tensor operations and a for loop. You can use the hidden states for predictions. Combining all those mechanisms, an LSTM can choose which information is relevant to remember or forget during sequence processing.

4.3 Seq2Seq Model

4.3.1 What is a seq2seq model?

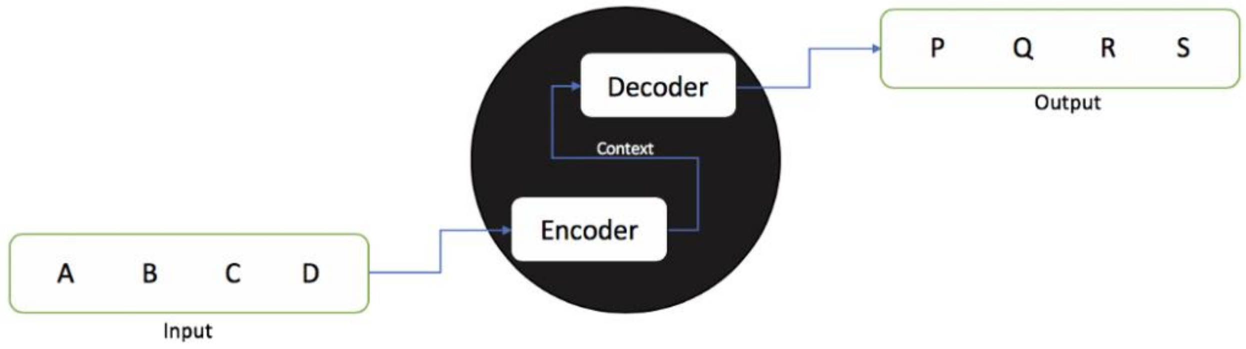
A Seq2Seq model is a model that takes a sequence of items (words, letters, time series, etc) and outputs another sequence of items.



Seq2Seq Model

In the case of Chatbots, the input is a series of words, and the output is also a series of words (responses).

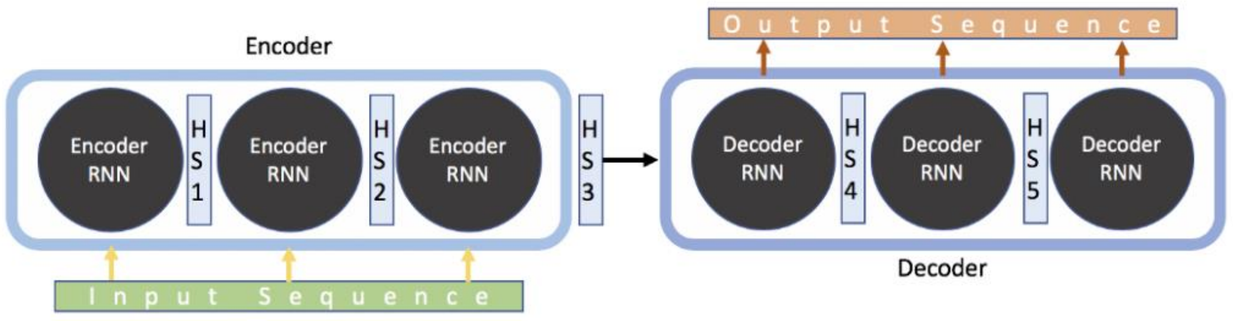
Now let's work on reducing the blackness of our black box. The model is composed of an *encoder* and a *decoder*. The encoder captures the *context* of the input sequence in the form of a *hidden state vector* and sends it to the decoder, which then produces the output sequence. Since the task is sequence based, both the encoder and decoder tend to use some form of RNNs, LSTMs, GRUs, etc. The hidden state vector can be of any size, though in most cases, it's taken as a power of 2 and a large number (256, 512, 1024) which can in some way represent the complexity of the complete sequence as well as the domain.



4.3.2 RNNs in seq2seq model

RNNs by design, take two inputs, the current example they see, and a representation of the previous input. Thus, the output at time step *t* depends on the current input as well as the input at time *t-1*. This is the reason they perform better when posed with sequence related tasks. The sequential information is preserved in a hidden state of the network and used in the next instance.

The Encoder, consisting of RNNs, takes the sequence as an input and generates a final embedding at the end of the sequence. This is then sent to the Decoder, which then uses it to predict a sequence, and after every successive prediction, it uses the previous hidden state to predict the next instance of the sequence.



Encoder-Decoder Model for Seq2Seq Modelling

Drawback: The output sequence relies heavily on the context defined by the hidden state in the final output of the encoder, making it challenging for the model to deal with long sentences. In the case of long sequences, there is a high probability that the initial context has been lost by the end of the sequence.

4.3.3 What is ATTENTION and why do we need Attention Mechanisms for the seq2seq model

Let's consider two scenarios, scenario one, where you are reading an article related to the current news. The second scenario where you are preparing for a test. Is the level of attention the same or different in both situations?

You will be reading with considerable attention when preparing for the test compared to the news article. While preparing for the test, you will learn with a greater focus on keywords to help you remember a simple or a complex concept. The same implies to any deep learning task where we want to focus on a particular area of interest.

"SEQ2SEQ MODEL MODELS USE ENCODER-DECODER ARCHITECTURE"

Seq2Seq model maps a source sequence to the target sequence. The source sequence in the case of chatbots could be **user input**, and the target sequence can be **chatbot response based on training data.**

We pass a user input to an encoder; the **encoder encodes the complete information of the source sequence into a single real-valued vector, also known as the context vector**. This context vector is then passed on the decoder to produce an output sequence (chatbot response). The context vector has the responsibility to summarize the entire input sequence into a single vector.

The basic idea of Attention mechanism is to avoid attempting to learn a single vector representation for each sentence, instead, it pays attention to specific input vectors of the input sequence based on the attention weights.

At every decoding step, the decoder will be informed how much "attention" needs to be paid to each input word using a set of ***attention weights**.* These attention weights provide contextual information to the decoder.

4.3.4 Attention Mechanism

Bahdanau et al. proposed an attention mechanism that **learns to align and translate jointly**. It is also known as **Additive attention** as it performs a **linear combination** of **encoder states and the decoder states**.

1. All hidden states of the encoder(forward and backward) and the decoder are used to generate the context vector, unlike how just the last encoder hidden state is used in seq2seq without attention.
2. The attention mechanism aligns the input and output sequences, with an alignment score parameterized by a feed-forward network. It helps to pay attention to the most relevant information in the source sequence.
3. The model predicts a target word based on the context vectors associated with the source position and the previously generated target words.

Seq2Seq model with an attention mechanism consists of an encoder, decoder, and attention layer.

Attention layer consists of

- Alignment layer
- Attention weights
- Context vector

4.3.4.1 Alignment Score

The alignment score maps how well the inputs around position ***"j"*** and the output at position "***i"*** match. The score is based on the previous decoder's hidden state, **$s_{(i-1)}$** just before predicting the target word and the hidden state, h_j of the input sentence.

$$e_{ij}. = a(s_{i-1}, h_j) \qquad \text{Alignment Score}$$

The decoder decides which part of the source sentence it needs to pay attention to, instead of having encoder encode all the information of the source sentence into a fixed-length vector.

4.3.4.2 Attention Weights

We apply a softmax activation function to the alignment scores to obtain the attention weights.

$$\alpha_{i_j} = {\exp(e_{ij})}\Big/{\sum_{k=1}^{Tx} \exp(e_{ik})} \quad \text{Attention weight}$$

Softmax activation function will get the probabilities whose sum will be equal to 1, This will help to represent the weight of influence for each of the input sequence. Higher the attention weight of the input sequence, the higher will be its influence on predicting the target word.

4.3.4.3 Context Vector

The context vector is used to compute the final output of the decoder**.** The context vector c_i is the weighted sum of attention weights and the encoder hidden states (h_1, h_2, …,h_{tx}), which maps to the input sentence.

$$C_i = \sum_{j=1}^{Tx} \alpha_{ij}\, h_j \quad \text{Context vector}$$

4.3.4.4 Predicting the Target Word

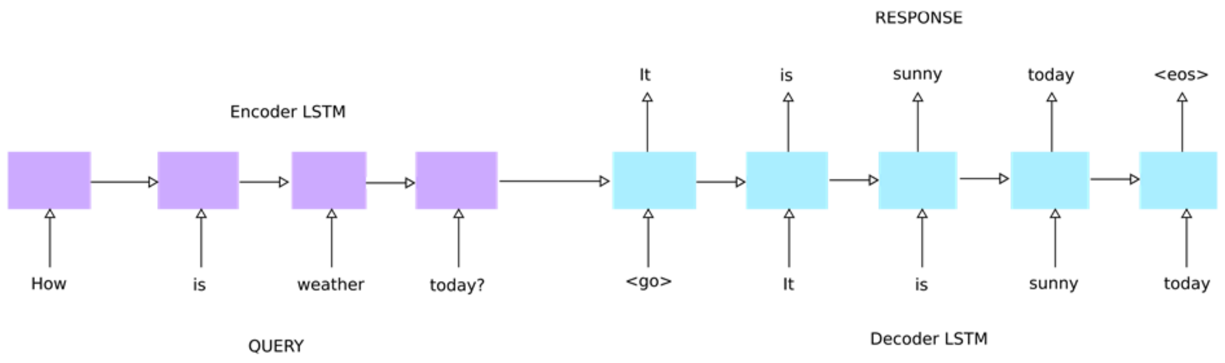
To predict the target word, the decoder uses

- Context vector(c_i),
- Decoder's output from the previous time step (y_{i-1})and
- Previous decoder's hidden state(**s_{i-1})**

$$s_i = f(s_{i-1}, c_i, y_{i-1})$$

5. Building the Model

5.1 Path to Building the Model



Here we are going to see the steps that would lead us in building the model:

1. Defining the inputs
2. Pre-processing the targets
3. Encoder RNN
4. Decoding Train Set
5. Decoding Test Set
6. Decoder RNN
7. Sequence-to-Sequence model

5.2 Defining the Inputs

Placeholders are primitive way of storing data in TensorFlow. It allows us to create our operations and build our computation graph. More about placeholders here.

We are using **tf.placeholder** to create our **inputs, targets, learning rate, dropout rate**. Dropout consists in randomly setting a fraction rate of input units to 0 at each update during training time, which helps **prevent overfitting**. The units that are kept are scaled by 1/(1-rate), so that their sum is unchanged at training time and inference time.

5.3 Pre-processing the targets

In this step, we are going to create tensors using the targets and the **questionswords2int** dictionary.

We have used **tf.fill** to create a tensor with scalar value with dimensions of batch size and value being the **questionswords2int** dictionary. This would give us a tensor of shape of batch_size with **dtype = int32**. This tensor is stored in **left_side.**

We have used **tf.strided_slice** to create a similar tensor using the **targets** and we would store this tensor in **right_side.**

We now use **tf.concat** to concatenate the two tensors that we created in the previous step.

```
def preprocess_targets(targets, word2int, batch_size):
    left_side = tf.fill([batch_size, 1], word2int['<SOS>'])
    right_side = tf.strided_slice(targets, [0,0], [batch_size, -1], [1,1])
    preprocessed_targets = tf.concat([left_side, right_side], 1)
    return preprocessed_targets
```

5.4 Encoder RNN

We are building an encoder RNN using LSTM. We use the **tf.contrib.rnn** to build a basic LSTM cell.

We then use the **DropOutWrapper** attribute to add dropout to input and output of the RNN cell (LSTM in our case).

We then define the encoder cell composed of multiple simple cells using the **MultiRNNCell** attribute which takes in the LSTM cell (**lstm_dropout)** after dropout has been added.

Finally, we use the **bidirectional_dynamic_rnn** attribute to output the LSTM outputs and last hidden state using the encoder cell as forward and backward direction.

```
def encoder_rnn(rnn_inputs, rnn_size, num_layers, keep_prob, sequence_length):
    lstm = tf.contrib.rnn.BasicLSTMCell(rnn_size)
    lstm_dropout = tf.contrib.rnn.DropoutWrapper(lstm, input_keep_prob = keep_prob)
    encoder_cell = tf.contrib.rnn.MultiRNNCell([lstm_dropout] * num_layers)
    encoder_output, encoder_state = tf.nn.bidirectional_dynamic_rnn(cell_fw = encoder_cell,
                                                                    cell_bw = encoder_cell,
                                                                    sequence_length = sequence_length,
                                                                    inputs = rnn_inputs,
                                                                    dtype = tf.float32)
    return encoder_state
```

5.5 Decoding Train/Test Set

We are going to be decoding the train and test set in a similar step, but the output would be different.

We are going to extract **attention keys,** which needs to be compared with target states, **attention values** – to be used to construct context vectors, **attention score function** – to compute similarity between key and target states **and attention construct function** – to build attention states using the **prepare_attention** attribute from the **seq2seq** module. We are using the **Bahdanau** attention mechanism for our decoder to assign attention weights.

From decoding training set we get the output using the **dynamic_rnn_decoder** and we apply **dropout** to our decoder output also using **tf.nn.dropout**.

From decoding the test set, we get the predictions obtained using the test set using the **dynamic_rnn_decoder.**

5.6 Decoder RNN

We are defining a context manager for defining ops that creates variables(layers). We assign **"decoding"** as the decoding scope name. With decoding as our scope, we create a similar range of steps as RNN. We initiate a **BasicLSTMCell**, add dropout to our LSTM, create a **decoder_cell** using **MultiRNNCell** with the **lstm_dropout** as the RNN cell.

We define the weights using **truncated_normal_initializer** with a Standard deviation value of 0.1 (weights).

We then define the **output_function** using **fully_connected** which creates a variable called weights, representing a fully connected weight matrix which is multiplied by the inputs to produce a Tensor of hidden units. We have used the **decoding_scope** as scope, **biases** as biases_initializer and weights as weight_initializer.

We get the **training_predictions** and **test_predictions** from the **decode_training_set** and **decode_training_set** respectively.

So, we get the training_predictions and test_predictions from our decoder RNN cell.

5.7 Sequence-to-Sequence Model

We are creating the encoder RNN inputs using the **embed_sequence** attribute with the inputs from the **model_inputs()** function. Every word becomes embedded into a vectore of size – **encoding_embedding_size**. The RNN inputs will be of the dimension **batch_size * encoding_embedding_size * number_of_indices**. (Link).

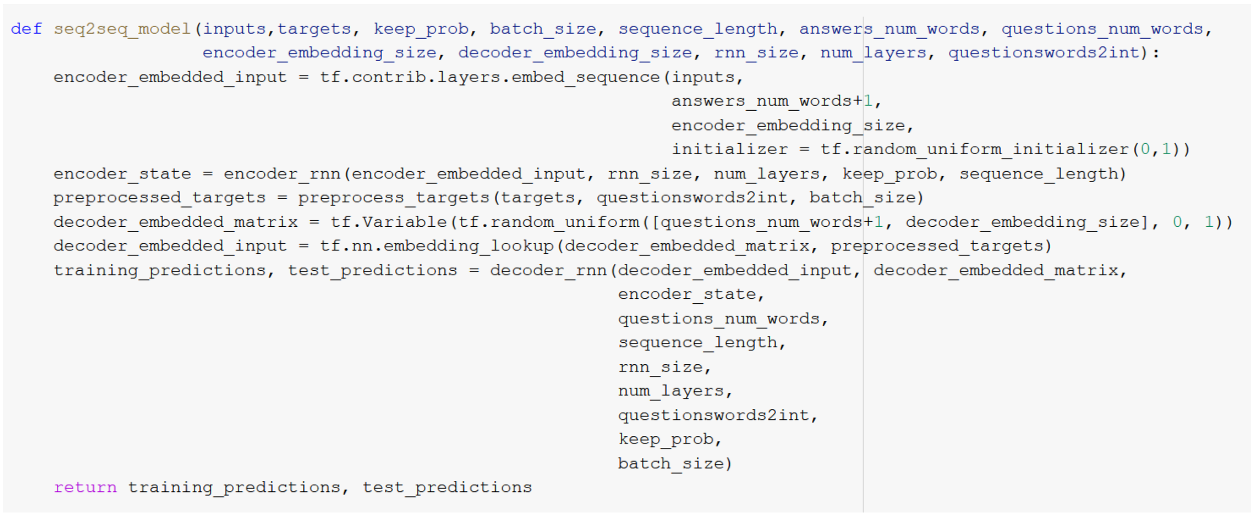
The **encoder_state** is created by the **encoder_rnn** getting the encoder_embedded_input, rnn_size, number_of_layers, dropout rate, and sequence_length which we will be getting using the placeholder attribute later in the training part.

We get the preprocessed targets from the **preprocess_targets** function using targets, questionswords2int and batch_size.

We create a **decoder_embedded_matrix** by creating a tensor of shape of length of questionswords2int + 1 (6998 + 1) and decoding_embedding_size = 1024 with random values from a uniform distribution.

We create the embedded input for the decoder using **tf.nn.embedding_lookup** which returns a tensor related to our **preprocessed targets** with the same type as the tensors in **decoder_embedded_matrix.**

Finally, we get the test and training predictions from the **decoder_rnn** using the decoder_embedded_input, decoder_embedded_matrix, encoder_state, length of questionswords2int, sequence_length, rnn_size, number_of_layers, questionswords2int dictionary, and batch_size



```
def seq2seq_model(inputs,targets, keep_prob, batch_size, sequence_length, answers_num_words, questions_num_words,
                  encoder_embedding_size, decoder_embedding_size, rnn_size, num_layers, questionswords2int):
    encoder_embedded_input = tf.contrib.layers.embed_sequence(inputs,
                                                              answers_num_words+1,
                                                              encoder_embedding_size,
                                                              initializer = tf.random_uniform_initializer(0,1))
    encoder_state = encoder_rnn(encoder_embedded_input, rnn_size, num_layers, keep_prob, sequence_length)
    preprocessed_targets = preprocess_targets(targets, questionswords2int, batch_size)
    decoder_embedded_matrix = tf.Variable(tf.random_uniform([questions_num_words+1, decoder_embedding_size], 0, 1))
    decoder_embedded_input = tf.nn.embedding_lookup(decoder_embedded_matrix, preprocessed_targets)
    training_predictions, test_predictions = decoder_rnn(decoder_embedded_input, decoder_embedded_matrix,
                                                         encoder_state,
                                                         questions_num_words,
                                                         sequence_length,
                                                         rnn_size,
                                                         num_layers,
                                                         questionswords2int,
                                                         keep_prob,
                                                         batch_size)
    return training_predictions, test_predictions
```

6. Training the Model

6.1 Hyperparameters

6.1.1 EPOCH:

In terms of Neural Networks, an epoch refers to one cycle through the full training dataset. Usually, training a neural network takes more than a few epochs. In other words, if we feed a neural network the training data for more than one epoch in different patterns, we hope for different generalization when given a new input.

We have set our number of epochs to 100. The reason behind this is to make sure that our model is trained best on all data points as our data set is very large and we need the best responses for our chatbot.

6.1.2 BATCH_SIZE:

The batch size is a hyperparameter that defines the number of samples to work through before updating the internal model parameters.

Think of a batch as a for-loop iterating over one or more samples and making predictions. At the end of the batch, the predictions are compared to the expected output variables and an error is calculated. From this error, the update algorithm is used to improve the model, e.g. move down along the error gradient.

A training dataset can be divided into one or more batches.

We have defined our batch_size to be 32.

6.1.3 RNN_SIZE:

Default : 256

Size of RNN hidden state. Rule of thumb for RNN hidden state size: “the optimal size of the hidden layer is usually between the size of the input and size of the output layers.

Rnn_size = 1024

6.1.4 NUMBER OF LAYERS:

Default : 2

The situations in which performance improves with a second (or third…) hidden layer are very few.

Num_layers = 3

6.1.5 DECAY RATE

Default : 0.97

Decay rate for rmsprop. After each update, the weights are multiplied by a factor slightly less than 1. This prevents the weights from growing too large, and can be seen as gradient descent on a quadratic regularization term. Weight Decay specifies regularization in the neural network

learning_rate_decay = 0.9

6.1.6 LEARNING RATE

Default : 0.002

RNN learning rate. If the learning rate is low, then training is more reliable, but optimization will take a lot of time because steps towards the minimum of the loss function are tiny. If the learning rate is too high, then training may not converge or even diverge. Weight changes can be so big that the optimizer overshoots the minimum and makes the loss worse.

learning_rate = 0.001

6.1.7 ENCODING EMBEDDING SIZE

Integer number of dimensions for embedding matrix.

encoding_embedding_size = 1024

6.1.8 DECODING EMBEDDING SIZE

Size of the decoder embedding matrix

decoding_embedding_size = 1024

6.2 Running our Model

We activate our session by running **tf.InteractiveSession().** We load the **inputs, targets, learning_rate, dropout_rate** running **model_inputs().** We get the sequence length using the placeholder attribute of shape 25 as this is the maximum length of our input questions data. The shape of the input tensor would be the shape of the **inputs.**

We get the training and test predictions by running our **seq2seq_model**.

```
tf.reset_default_graph()
session = tf.InteractiveSession()

inputs, targets, lr, keep_prob = model_inputs()

sequence_length = tf.placeholder_with_default(25, None, name = 'sequence_length')

input_shape = tf.shape(inputs)

training_predictions, test_predictions = seq2seq_model(tf.reverse(inputs, [-1]),
                                                       targets,
                                                       keep_prob,
                                                       batch_size,
                                                       sequence_length,
                                                       len(answerswords2int),
                                                       len(questionswords2int),
                                                       encoding_embedding_size,
                                                       decoding_embedding_size,
                                                       rnn_size,
                                                       num_layers,
                                                       questionswords2int)
```

6.3 Splitting Data into Training and validation sets

We are defining our loss error using **sequence_loss** which gives us a weighted cross-entropy loss for a sequence of **training_predictions.**

We are going to be using **ADAM OPTIMIZER** as to implement the Adam algorithm – a method for Stochastic optimization. We compute the gradients by passing the loss error to the optimizer.

We apply gradient clipping to prevent exploding gradients in very deep networks usually in RNN. This prevents any gradient to have norm greater than the threshold and thus the gradients are clipped. **tf.clip_by_value** clips the gradient tensor in the gradients by minimum clip value of -5 and maximum clip value of 5. This would give us a clipped tensor.

We apply optimizer to the clipped gradients.

We apply padding with <PAD> token giving the same length for questions and answers.

We are going to be splitting the data into batches of questions and answers. We are going to use the list of questions and answers and split them into batches using a **start_index**. We then apply padding to these lists.

```
def split_into_batches(questions, answers, batch_size):
    for batch_index in range(0, len(questions) // batch_size):
        start_index = batch_index * batch_size
        questions_in_batch = questions[start_index : start_index + batch_size]
        answers_in_batch = answers[start_index : start_index + batch_size]
        padded_questions_in_batch = np.array(apply_padding(questions_in_batch, questionswords2int))
        padded_answers_in_batch = np.array(apply_padding(answers_in_batch, answerswords2int))
        yield padded_questions_in_batch, padded_answers_in_batch
```

Then we split our data into 85% train and 15% validation set.

```
training_validation_split = int(len(sorted_clean_questions) * 0.15)
training_questions = sorted_clean_questions[training_validation_split:]
training_answers = sorted_clean_answers[training_validation_split:]
validation_questions = sorted_clean_questions[:training_validation_split]
validation_answers = sorted_clean_answers[:training_validation_split]
```

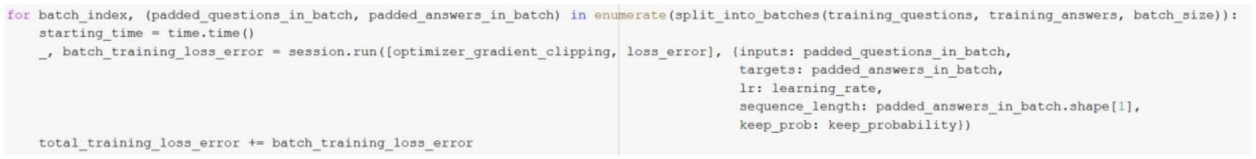
6.4 Training our model

We are going to need the **padded_questions_in_batch** and **padded_answers_in_batch** from the **split_into_batches** using the training_questions and training_answers as inputs

We then run a tf.session with **tf.global_variables_initializer** on **optimizer_gradient_clipping** and **loss_error** with padded_questions_in_batch as inputs, padded_answers_in_batch as targets, 0.001 learning rate, with keep_probability as 0.5.

We repeat this process with validation_questions and validation_answers as well. We get the total_training_loss_error and total_validation_loss_error. We get the average_validation_loss_error by dividing by number of validation questions by batch_size.

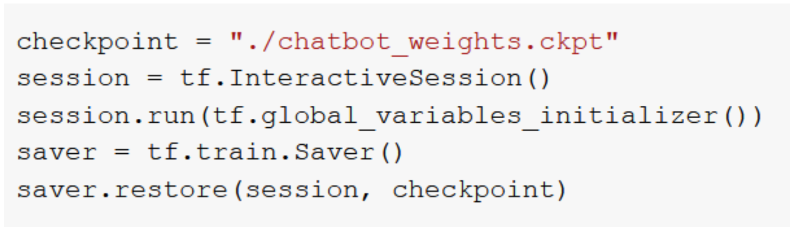
If the average_validation_loss_error becomes minimum,the chatbot prints out **"I speak better now"** through which we can understand that our model's performance is improving gradually. This would be saved in **saver** using **tf.train.Saver().**



```
for batch_index, (padded_questions_in_batch, padded_answers_in_batch) in enumerate(split_into_batches(training_questions, training_answers, batch_size)):
    starting_time = time.time()
    _, batch_training_loss_error = session.run([optimizer_gradient_clipping, loss_error], {inputs: padded_questions_in_batch,
                                                                                            targets: padded_answers_in_batch,
                                                                                            lr: learning_rate,
                                                                                            sequence_length: padded_answers_in_batch.shape[1],
                                                                                            keep_prob: keep_probability})
    total_training_loss_error += batch_training_loss_error
```

7. Chatbot

We are going to use **Checkpoints** to capture the exact value of all parameters (**tf.Variable** objects) used by a model. Checkpoints do not contain any description of the computation defined by the model and thus are typically only useful when source code that will use the saved parameter values is available. So, we have saved a checkpoint of our model in **checkpoint**. And then we defined a **session** using **tf.InteractiveSession()**. We save both **session** and **checkpoint** in **saver.**



```
checkpoint = "./chatbot_weights.ckpt"
session = tf.InteractiveSession()
session.run(tf.global_variables_initializer())
saver = tf.train.Saver()
saver.restore(session, checkpoint)
```

We define a function **convert_string2int** which converts the user input into integers using the **questionwords2int** dictionary and **clean_text** function which we defined in our data processing section.

```
def convert_string2int(question, word2int):
    question = clean_text(question)
    return [word2int.get(word, word2int['<OUT>']) for word in question.split()]
```

We are defining the chatbot using a **while** loop. If the user input is [**Goodbye, bye, quit** or **exit]** we are breaking the chatbot. If the user input is any but these, the first step is to clean the user input using **clean_text** and convert it into integers using **convert_string2int** which would give us a list of unique integer for each token of user input. This list of integers would be of the length of number of tokens in the user input. We apply padding to this list by adding the integer associated with the token **"<PAD>"** by difference between 20 and the length of user input. We then create a **fake_batch** of zeroes matrix with the [**batch_size, 20]** size. Then we assign the padded question as the **fake_batch**'s first element. Now the chatbot has to predict a response from the **test_predictions.**

Hence, the chatbot would run a TensorFlow session using the **checkpoint** created on **test_predictions,** and **fake_batch** and **keep_prob = 0.5**. The keep_prob value is used to control the dropout rate used when training the neural network. Essentially, it means that each connection between layers will only be used with probability 0.5 when training. This **reduces overfitting**. This gives us a **predicted_answer** numpy array. We then use the **np.argmax()** to get the indices of maximum values along axis=1. We then use the **answersints2word** dictionary to map the integer values of user input and let the chatbot print the word corresponding to the integer.

8. Summary

1. We have learned the steps on how to prepare a data for chatbots in general.
2. We learnt the mechanisms and working of RNN and limitations.
3. We understood how LSTM overcomes shortcomings of RNN.
4. We created a seq2seq model.
5. We added encoder RNN and decoder RNN with LSTM units.
6. We used Bahdanau attention mechanism.

9. Conclusion

Closed Domain architectures focus on response selection from a set of predefined responses when the open domain architecture enables us to perform boundless text generation. Closed domain systems use intent classification, entity identification, and response selection. But for an open domain chatbot, intent classification is harder and an immense number of intents are likely. Rather than selecting full responses, the open domain or generative model generates the response word by word, allowing for new combinations of language.

In industries, some companies use the closed domain chatbots to ensure that the user always receives the right response from the predefined ones. The Natural Language Processing- NLP domain is developing and training neural networks for approximating the approach the human brain takes towards language processing. This deep learning strategy allows computers to handle human language much more efficiently.

References

1. Numpy - https://numpy.org/doc/stable/reference/generated/numpy.argmax.html

2. Recurrent Neural Networks - https://towardsdatascience.com/illustrated-guide-to-recurrent-neural-networks-79e5eb8049c9

3. Long Short-Term Memory

4. TensorFlow Placeholders - https://databricks.com/tensorflow/placeholders

5. Attention for seq2seq - https://docs1.w3cub.com/tensorflow~python/tf/contrib/seq2seq/prepare_attention/

6. Sequence-to-Sequence model - https://pytorch.org/tutorials/beginner/chatbot_tutorial.html?highlight=chatbot%20tutorial

7. Attention based Chatbot - https://www.kaggle.com/samsonleegh/chat-bot-word-based-with-attention

8. Understanding Attention - https://www.kaggle.com/tientd95/understanding-attention-in-neural-network