Chapter 6

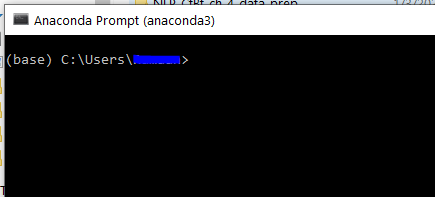
**Project: ChatBot**

**Training the ChatBot**

Note: Create a virtual environment using conda: TensorFlow v1.0.0 with Python 3.5.4

**6.0 Conda Virtual environment**

Open ***anaconda cmd*** from installed Anaconda. Use the following commands. Here we are creating a virtual environment of ***Python 3.5.4*** to use ***TensorFlow 1.0***.



conda create --name py354 python=3.5.4

conda activate py354

pip install protobuf==3.19.4

pip install tensorflow==1.0.0

conda install ipykernel

conda install spyder

**6.1 Setting the HyperPartameters**

We're going to set some values for the ***Hyper-Parameters*** that will be used during the training and later we will be able to change if we want to experiment more and try to improve the results.

* Note that Machine-Learning, Deep Learning, AI and also of course Deep-NLP, a lot of the work is ***experimentation*** & ***tweaking*** the parameters to try to find the *best model* that will lead to the *best accuracy (inour case best ChatBot).*
* HyperPartameters: Following are not the best values, but these are at least relevant values that are most of the time used in the architecture of the neural networks you can find online. At the end we will actually try to improve the model by either tuning the parameters or improving the data processing or even improving the architecture of the Seq2Seq brain of our *ChatBot*.

# *Setting the HyperPartameters: We choosed these names for TensorFlow matching*

epochs = 100    # *or 50*

batch\_size = 64    # *or 128*

rnn\_size = 512

num\_layers = 3

encoding\_embedding\_size = 512

decoding\_embedding\_size = 512

learning\_rate = 0.01

learning\_rate\_decay = 0.9

min\_learning\_rate = 0.0001

keep\_probability = 0.5  # *optimal according to Hinton*

* Single Iteration: A single iteration of training contains.

1. Getting the batches of input into the networks then
2. Forward propagating them inside the Encoder RNN get the Encoders states
3. Forward propagating the Encoders states with the targets inside the Decoder RNN to get the final output then the final answer as predicted by the ChatBot
4. And then back propagating the ***Loss*** generated by the ***outputs*** *and the* ***target*** back into the neural network and updating the weights towards the direction of a better ability for the ChatBot to speak like a human.

* epochs number of epochs. ***epochs*** are the number of the whole iterations of the Training.
* We choose 100, if the training takes too long on your computer, recommend to choose less like 50. But not too low because we really need a lot of epochs to get some correct results.
* batch\_size we'll choose a batch size to start with 64. And again *if the training takes too long* you can try 128. This will get basically *more* ***questions*** *and get* ***answers*** *in your batches* to train the NN.
* rnn\_size the size of the RNN and we'll choose 512. Is the *number of input tensors* of the *encoder*.
* num\_layers number of layers. You want to have in the Encoder-RNN that and also the Decoder RNN. For now we choose 3 we'll probably change that at the end of the implementation when trying to improve the model.
* encoding\_embedding\_size is the *number of columns* in your embeddings matrix that is the *number of columns* you want to have for the Embedding values.
* Where in this matrix *each line* corresponds to *each token* in the *whole corpus of the questions*.
* We're going to choose a size of ***512***, meaning that we're going to have ***512*** columns in this embeddings matrix.
* decoding\_embedding\_size we will also choose 512 for decoding embeddings size

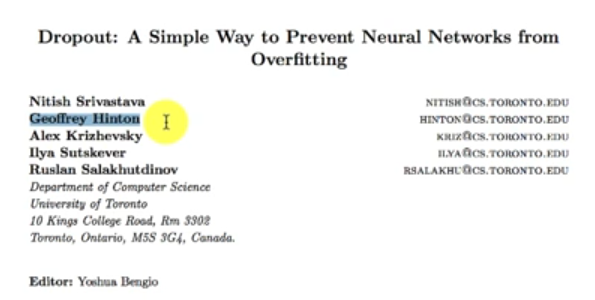
learning\_rate = 0.01

learning\_rate\_decay = 0.9

min\_learning\_rate = 0.0001

keep\_probability = 0.5  # *optimal according to Hinton*

* learning\_rate the learning rate is quite tricky. We never know exactly what to choose.
* It must not be too high it must not be too low.
  + If it is too high, model will learn too fast and will therefore not learn how to speak properly.
  + And if it's too low, it will take ages for the model to learn how to speak properly.
* We're going to start with **0.01**.
* learning\_rate\_decay that's basically by which *percentage* the *learning rate* is reduced over the iterations of the training.
* When we start with a certain learning rate but then we apply decay over the iterations of the training to reduce the learning rate so that it can *learn* in *more depth* the *logic* of *human conversations* or in general the *correlations* identified in the *data set*.
* We're going to choose a common value that you will find in most implementations. We're going to choose a decay of 90% i.e. ***0.9***.
* There are many implementation where decay = 1, meaning that there is no decay.
* min\_learning\_rate = 0.0001 Since we are applying the decay the learning rate will be reduced over the iterations and we have many iterations so we don't want the learning rate to reach a too low value.
* That’s a very low number, but don't worry, besides we will try to add some *early stopping* in case there is *literally no improvement* during the *training*.
* keep\_probability best way to set keep\_prob the dropout value to improve our model. We take a fast look of a paper from *Journal of Machine learning Reasearch 15 (2014):* ***Dropout: A Simple Way to Prevent Neural Networks from Overfitting***.



* In Page 1133 we see that dropping out ***20% of the input*** ***units*** and ***50% of the hidden*** ***units*** was often found to be optimal.
* But here we have to choose value for the ***hidden*** ***units*** not *input units* and therefore we're going to trust Jeffrey Hinton for the first try of our model. We're going to choose a dropout rate of **50%**.
* Instead of ***keep\_prob*** we use the name keep\_probability because this is the name in the ***TensorFlow*** (1.0.0) of ***API*** and therefore you must introduce it this way, keep probability ***keep\_prob*** won't work.

**6.2 Defining a session**

Here we are going to define a **tf.InteractiveSession**() on which all the ***TensorFlow*** training will be run. So to open a session in ***TensorFlow*** we're going to create an object of the **tf.InteractiveSession**() class and that object will be our session.

* But before we create this object we need to reset the ***TensorFlow*** graphs to ensure that the graph is ready for the training.

# *Defining a session*

**tf.reset\_default\_graph**()

session = **tf.InteractiveSession**()

So in general when you open ***TensorFlow*** session to do some training you have to reset the graph first.

**6.3 Loading the model Inputs**

Here we are going to use function **model\_inputs()** which we defined before at the beginning of "building SEQ2SEQ model" part.

* Here we're going to load the inputs of our SEQ2SEQ model and we're going to use this **model\_inputs**() function to return all these model inputs that will be just one line.

# *Loding the model Inputs*

inputs, targets, lr, keep\_prob = **model\_inputs**()

**6.4 Setting the Sequence Length**

* We're going to set the ***sequence length*** to a maximum length which will be 25 (remember that, we already did that previously in *Part 1: data pre-processing*. max\_line\_length = 25, to truncate the sentence at a specific length).

# *Setting the Sequence Length*

sequence\_length = **tf.placeholder\_with\_default**(25, **None**, name = 'sequence\_length')

* This sequence\_length is used in our defined seq2seq\_model(), decoder\_rnn, encoder\_rnn, decode\_training\_set().
* Means that in the training we won't be using questions and the answer that have more than 25 words.
* **tf.placeholder\_with\_default**() is used (instead of the simple placeholder function) because we're going to set a default value to produce when the output is not fed into the RNN.
* First argument is the ***maximum length***, i.e ***25***.
* Second argument is the shape, the ***TensorFlow tensor shape*** or a simple list of integers. And since sequence length is just a simple maximum length, there is no tensor to deal with and therefore will just input ***None***.
* Third argument is just a name we want to give to that sequence length, name = 'sequence\_length' so that we can ***reuse*** it ***afterwards*** when we need it for the ***training***.

**6.5 Shape of the inputs tensor**

Before using seq2seq\_model() function to get the training predictions and the test predictions we just need to get the shape of the input tensor. (i.e size of the input).

# *Getting the shape of the inputs tensor*

input\_shape = **tf.shape**(inputs)

* Right now our inputs are TensorFlow tensor but we need to get the shape of these inputs because it will be one of the arguments of one specific function we'll use for the training.
* This function is actually the ***ones()*** function by TensorFlow basically creates a ***tensor of 1's*** and the dimensions of this tensor is going to be exactly the same as ***input\_shape***.
* It's going to become the argument of this future ***ones()*** function. When we calculate the loss function in the next section.

**6.6 Getting the predictions**

We now use the ***hyperparameters***, initialized ***sequence\_length***, and ***inputs*** from our ***model\_inputs()*** to get the Training and Test predictions.

# *Getting the training predictions and test predictions*

training\_predictions, test\_predictions = **seq2seq\_model**(**tf.reverse**(inputs, [-1]),

                                                       targets,

                                                       keep\_prob,

                                                       batch\_size,

                                                       sequence\_length,

**len**(ansWrd2Int),

**len**(questnWrd2Int),

                                                       encoding\_embedding\_size,

                                                       decoding\_embedding\_size,

                                                       rnn\_size,

                                                       num\_layers,

                                                       questnWrd2Int)

* We're not starting the training yet, that's going to be ***from the moment*** we start the ***for loop*** over the ***epochs***.
* We get the training and test predictions by feeding the loaded model inputs inputs, targets, lr, keep\_prob into the neural network.
* Notice we are not using the hyperparameter keep\_probability.
* batch\_size, encoding\_embedding\_size, decoding\_embedding\_size, rnn\_size, num\_layers are from hyperparameters.
* ***sequence\_length*** is from what we initialized before, *initialized sequence length with default values*.
* Here we use our defined function: **seq2seq\_model()**, since this function returns two predictions, training predictions and training predictions we are going to introduce two new variables training\_predictions, test\_predictions that are going to be our *training predictions* and our *test predictions*.
* Not to be confused with the local variables *traing\_pred, test\_pred* in the **seq2seq\_model()**, definition function, this time it's the *real variables* that we will use *later* on in the *training*.
* Reshaping the tensors: Recall, in Machine Learning and Deep learning introductory tutorials sometimes we reshape our inputs using NumPy ***reshape()*** function. The same thing is happening here, we are now using the similar TensorFlow function ***reverse()***, to reshape dimensions of our input tensor.

predictions = **seq2seq\_model**(**tf.reverse**(inputs, [-1]),

* The second argument [-1] used to reverse the dimension of the tensor.
* We'll actually connect the layer when running the ***session*** with the ***run()*** method of our session object and that will be in the big for loop over the ***epochs*** of the ***training***.

**6.7 Setting Loss Error, the Optimizer and Gradient Clipping**

Here we'll set Loss Error, the Optimizer and Gradient Clipping.

* Gradient Clipping: This is a technique (some operations) that will cap the gradient in the graph between a *minimum* value and a *maximum* value and to avoid some exploding or vanishing gradient issues.
* The loss\_error is going to be based on a Weighted Cross Entropy Loss Error which is the most relevant loss to use when dealing with Sequences as we are doing right now and in general when dealing with deep NLP.
* The Optimizer that we're going to use will be first an **AdamOptimizer** which is one of the best optimizers for Stochastic Gradient Descent. Then we will apply Gradient Clipping to that optimizer to avoid exploding or vanishing great issues.
* Name-scope: We are going to define a **new scope** which will contain two elements that we'll use for the training which are going to be the loss\_error and the Optimizer with Gradient Clipping optimizer\_gradient\_clipping.

**with** **tf.name\_scope**("optimization"):

* **name\_scope** is from TF library. We call the scope "optimization ". In this scope we'll define Loss Error, the Optimizer and Gradient Clipping.
* Loss Error: Here we use **tf.contrib.seq2seq.sequence\_loss**() function from Tensorflow. .

    loss\_error = **tf.contrib.seq2seq.sequence\_loss**(training\_predictions,

                                                  targets,

**tf.ones**([input\_shape[0], sequence\_length]))

* We use training\_predictions which we got from **seq2seq\_model**() in previous section 6.6.
* targets is from returned value of **model\_inputs**().
* **tf.ones**([input\_shape[0], sequence\_length]) corresponds to the weights. We prepare this tensor of weights by initializing them all to 1's with the right dimensions. Here input\_shape[0], sequence\_length are the dimensions of this weight tensor.
* We initialize this weights by **tf.ones**() function of TF library. This initialize all values to 1's.
* input\_shape[0] is from ***input\_shape*** with index **0**. Representing the *number of lines* of this weight tensor.
* sequence\_length is the initialized sequence length with default values. Representing the *number of columns* of this weight tensor.
* Optimizer with Gradient Clipping: We do this in 3-steps.

1. Setting the optimizer.
2. Creating clipped gradients (creating gradients and clipping them).
3. Apply clipped gradients to selected optimizer.

* Optimizer: We select **tf.train.AdamOptimizer** from TF as our optimizer. It takes one argument: the learning rate (as well as all other optimizers do). ***learning\_rate*** is ***0.01*** from our hyperparameters.

optimizer = **tf.train.AdamOptimizer**(learning\_rate)

* Clipped Gradients: The first thing is to compute all the gradients.
* We have one gradient per neuron in NN. For each of the neurons we compute the gradient of the with respect to the weight of that neuron.
* We have one gradient per neuron in the Neural Networks and for each of the neuron we compute the Gradient of the Loss Error with respect to the weight of that neuron. And all *these gradients are into a graph* and are *attached to a variable*.
* To compute the Gradients we use our optimizer **AdamOptimizer**. We're going to use a method **compute\_gradients** from that optimizer by applying the loss\_error.

gradients = **optimizer.compute\_gradients**(loss\_error)

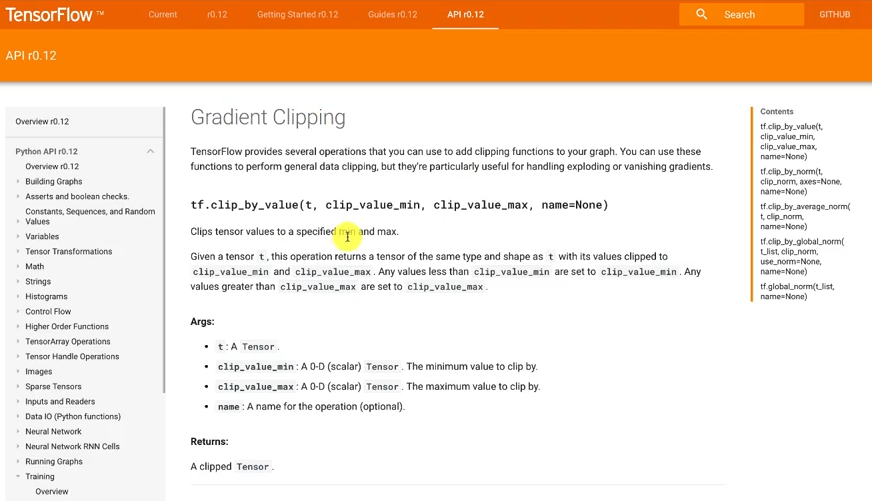
* We always compute the Gradient with respect to the weights, but we *don't have to put the weights as argument* but just the loss\_error of which we want to compute the Gradient.
* Capping each gradients: clipped\_gradients are going to be our gradients clipped by value. We are going to use **clip\_by\_value()** from TF. Also notice that we are using Python's list Comprehension.

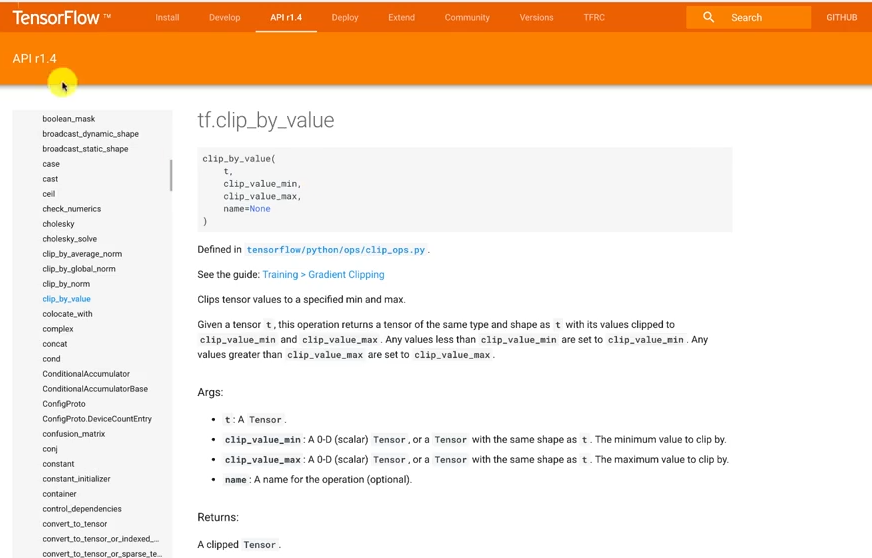
[(**tf.clip\_by\_value**(grad\_tensor, -5., 5.), grad\_variable) **for** grad\_tensor, grad\_variable **in** gradients **if** grad\_tensor **is** **not** **None**]

**tf.clip\_by\_value**(grad\_tensor, -5., 5.)

* This function takes three arguments:
* First one is the tensor of gradient that is each of the *tensor of gradient* among all the gradients in the *graph*.
* Second argument is a minimum value below which the *gradient* value cannot go.
* Third argument is the maximum value above which the *gradient* value cannot go
* We will choose the **minimum =-5** and the **maximum = 5**, that will keep our *Gradients* in a way to avoid vanishing or exploding gradient issues.
* Since we want to do that for each of the gradients of the graph we're going to have to make a For loop to *loop over each one of these gradients*.
* a gradient in the graph is attached to a variable and therefore we are using two new variables: the tensor itself and the variable attached to it.
* we are going to make this for loop in the Python's List Comprehension (is a trick technique to get these gradients directly inside a list).
* We're going to loop over the two elements grad\_tensor, grad\_variable in gradients . grad\_tensor is using inside **tf.clip\_by\_value** and we are checking this inside gradients using this condition: grad\_tensor **is** **not** **None**. To apply **tf.clip\_by\_value** properly.
* That’s how we clip each of the tensor (which are not **None**) inside gradients. At the same time we putting the clipped tensor in a list named clipped\_gradients.

clipped\_gradients = [(**tf.clip\_by\_value**(grad\_tensor, -5.0, 5.0), grad\_variable) **for** grad\_tensor, grad\_variable **in** gradients **if** grad\_tensor **is** **not** **None**]





# *Setting up the Loss Error, the Optimizer and Gradient Clipping*

**with** **tf.name\_scope**("optimization"):

    loss\_error = **tf.contrib.seq2seq.sequence\_loss**(training\_predictions,

                                                  targets,

**tf.ones**([input\_shape[0], sequence\_length]))

    optimizer = **tf.train.AdamOptimizer**(learning\_rate)

    gradients = **optimizer.compute\_gradients**(loss\_error)

    clipped\_gradients = [(**tf.clip\_by\_value**(grad\_tensor, -5.0, 5.0), grad\_variable) **for** grad\_tensor, grad\_variable **in** gradients **if** grad\_tensor **is** **not** **None**]

    optimizer\_gradient\_clipping = **optimizer.apply\_gradients**(clipped\_gradients)

* Applying gradient clipping to our Optimizer: We name this clipped optimizer as ***optimizer\_gradient\_clipping***. And we use our ***clipped\_gradients***. **apply\_gradients()** is a function in our ***AdamOptimizer*** which is from ***tf.train***.

optimizer\_gradient\_clipping = **optimizer.apply\_gradients**(clipped\_gradients)

**6.8 Padding the Sequences**

We're going to finally apply the padding to the sequences with the **<PAD>** token.

* Why Padding: All the sentences in a batch whether they are questions or answers must have the same length.
* What are we going to do exactly with our questions and our answers using **<PAD>**: Notice the following question and answer.

*# Question: ["Who", "are", "you"]*

*# Answer: [<SOS>, "I", "am", "a", "bot",".", <EOS>]*

Notice, question and answer don't have the same length. After we apply the padding this question and answer will become this.

*# Question: ["Who", "are", "you", <PAD>, <PAD>, <PAD>, <PAD>]*

*# Answer: [<SOS>, "I", "am", "a", "bot",".", <EOS>]*

So basically the **<PAD>** tokens are added so that the *length* of the *question sequence* is equal to the *length* of the *answer sequence*. It's a must do in deep NLP.

* We do this for *each batch* and for *each sentence of each batch* we will *complete* the *length* of each sentence with enough **<PAD>** tokens so that each sentence of the batch has the same length. We define the **apply\_padding**(batch\_of\_sequences, word2int)function for this.
* batch\_of\_sequences not single sequences it’s a batch.
* word2int whether it is **questnWrd2Int** or **ansWrd2Int** dictionary, we'll actually use them both.

# *Padding the sequences with the <PAD> token*

*# Question: ["Who", "are", "you"]*

*# Answer: [<SOS>, "I", "am", "a", "bot",".", <EOS>]*

**def** **apply\_padding**(batch\_of\_sequences, word2int):

    max\_sequence\_length = **max**([**len**(sequence) **for** sequence **in** batch\_of\_sequences])

**return** [sequence + [word2int['<PAD>']] \* (max\_sequence\_length - **len**(sequence)) **for** sequence **in** batch\_of\_sequences]

* Getting the Maximum length: **max**([**len**(sequence) **for** sequence **in** batch\_of\_sequences]) Finds the maximum length in the batch. Using the for-loop **for** sequence **in** batch\_of\_sequences inside a list.

max\_sequence\_length = **max**([**len**(sequence) **for** sequence **in** batch\_of\_sequences])

* How we're going to complete them:

**return** [sequence + [word2int['<PAD>']] \* (max\_sequence\_length - **len**(sequence)) **for** sequence **in** batch\_of\_sequences]

* [word2int['<PAD>']] is **id** of the token **<PAD>**.
* [word2int['<PAD>']] \* (max\_sequence\_length - **len**(sequence)) is a list of **<PAD>**'s **id** of length max\_sequence\_length - **len**(sequence).\
* Then we add this list with the corresponding sequence (which is also a list):

sequence + [word2int['<PAD>']] \* (max\_sequence\_length - **len**(sequence))

* We do these for each sequence in the batch using a For-Loop:

**for** sequence **in** batch\_of\_sequences

* Next: Before train the model, We have to do two things:
* First we have to split the data into batches of questions and answers.
* Second step is to split again the questions and answers into training and validation data for cross validation during the training.

**6.9 Splitting the data into BATCHES of Questions and Answers**

Here we split the data in two batches of questions and answers. It is fundamental in Deep-Learning.

* We'll define new function **split\_into\_batches**() that will use the previous function **apply\_padding**() because inside each of these batches we will pad the questions with the **<PAD>** tokens.
* We'll apply this **split\_into\_batches**() in the training to create the batches and feed the Neural Network with these ***batches*** of ***inputs*** and ***targets***.
* Note that the inputs *will correspond to the* questions and the targets *will respond to the* answers.
* We'll make this function **split\_into\_batches**(questions, answers, batch\_size)we input our questions, answers then we group those into batches of batch\_size.
* Total no. of Batches: We consider the Questions, we use its size because we get corresponding answers according the questions.

**len**(questions) // batch\_size

* Here **//** is Pythons Integer Division, to truncate the fractions.

**for** batch\_index **in** **range**(0, **len**(questions) // batch\_size):

* Creating the Batches: When loop runs, start\_index will be: 0, batch\_size, 2.batch\_size, 3.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]

* Selects the questions/answers from their list between the index start\_index and start\_index + batch\_size.
* Apply padding: Returning the Padded Question-Answers Batches:

padded\_questions\_in\_batch = **np.array**(**apply\_padding**(questions\_in\_batch, questnWrd2Int))

padded\_answers\_in\_batch = **np.array**(**apply\_padding**(answers\_in\_batch, ansWrd2Int))

**yield** padded\_questions\_in\_batch, padded\_answers\_in\_batch

* **apply\_padding**(questions\_in\_batch, questnWrd2Int) we use our **apply\_padding()** function to pad all the Questions-Answers in corresponding batch.
* Then we apply np.array() over the ***list-of-sequences*** (Padded Question-Answers Batches) to be compatible with our NN.
* We use Python **yield**. To return padded\_questions\_in\_batch, padded\_answers\_in\_batch the NumPy arrays. ***yield*** in Python is like a ***return*** but it's better to use it when dealing with ***SEQUENCES***.

# *Splitting the data into batches of questions and answers*

**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, questnWrd2Int))

        padded\_answers\_in\_batch = **np.array**(**apply\_padding**(answers\_in\_batch, ansWrd2Int))

**yield** padded\_questions\_in\_batch, padded\_answers\_in\_batch

Next we split the questions and answers into *training* and *validation data*.

**6.10 Splitting the questions and answers into TRAINING and VALIDATION sets**

Here we'll split the *questions* and *answers* into the *training* and *validation* sets. We'll make 4-sets actually

1. a set for the training questions
2. a set for the training answers then
3. a set for the validation questions and
4. a set for the validation answers.

* In Cross-Validation 10% or 15% of the training data won't be used to train the neural network i.e. won't be back propagated.
* Get the index: We'll use our ***sorted\_clean\_questions***, which is the final processed data in data processing part. Basically these are all the questions all cleaned and sorted by the length of the questions and gathering the sequences *of unique* encoding integers.

training\_validation\_split = **int**(**len**(sorted\_clean\_questions) \* 0.15)

* We're going to use this whole list of encoded questions to get the total number of questions.
* Then we just multiply this total number by ***0.15*** to get ***15%***. Notice ***int*** is used to truncate the fractions.
* Splitting for the TRAINING set: We basically select the **last *85%*** portion of the Sorted Clean Questions and answers lists.

training\_questions = sorted\_clean\_questions[training\_validation\_split:]

training\_answers = sorted\_clean\_answers[training\_validation\_split:]

* The index is starting from ***training\_validation\_split*** up to the last index.
* That’s why we used [training\_validation\_split : ]
* Splitting for the VALIDATION set: We basically select the **first *15%*** portion of the Sorted Clean Questions and answers lists.

validation\_questions = sorted\_clean\_questions[:training\_validation\_split]

validation\_answers = sorted\_clean\_answers[:training\_validation\_split]

* The index is starting from beginning index **0** up to the ***training\_validation\_split*** index.
* That’s why we used [ : training\_validation\_split]

# *Splitting the questions and answers into training and validation sets*

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]

**6.11 Training: Part 1 (variables)**

# Training

batch\_index\_check\_training\_loss = 100

batch\_index\_check\_validation\_loss = ((**len**(training\_questions)) // batch\_size // 2) - 1

total\_training\_loss\_error = 0

list\_validation\_loss\_error = []

early\_stopping\_check = 0

early\_stopping\_stop = 1000

checkpoint = "./chatbot\_weights.ckpt"

# For Windows users, replace "chatbot\_weights.ckpt" line of code by: checkpoint = "./chatbot\_weights.ckpt"

**session.run**(**tf.global\_variables\_initializer**())

* We will check the training class every 100 batches.

batch\_index\_check\_training\_loss = 100

* We'll check the validation loss at the halfway and at the end of an epoch. Notice iinteger division "//" is used.

batch\_index\_check\_validation\_loss = ((**len**(training\_questions)) // batch\_size // 2) - 1

* We initialize total\_training\_loss\_error to ***zero*** and that will be used to *compute* the *sum* of the *training losses* on *100 batches* because in previous line we chose to *check* the *loss* every *100 batches*.

total\_training\_loss\_error = 0

* Early Stopping Technique: We initialize the list of Validation Loss Errors and we have to make this list because we're going to use the Early Stopping Technique which consists of *checking* if we managed to reach a *loss* that *is below the minimum* of *all* the *losses* we got and all the losses we get we're going to put them in that list.

list\_validation\_loss\_error = []

* Early stopping to check is going to *correspond* to the *number of checks* each time there is *no improvement* of the *validation*.
* So each time we don't reduce the validation loss, early stopping check is going to be incremented by 1 and once it reaches a certain number early\_stopping\_stop we will stop everything.

early\_stopping\_check = 0

early\_stopping\_stop = 1000 *# we choose 1000 to reach the last epoch i.e. 100-th epoch. Generally we set early\_stopping 100*

* 1000 is a bit high but that's just because I want to make the training last until the last epoch. *Recommended value for early stopping is 100*.
* Check point: That's going to be the file containing the weights.It is to save the weights which we will be able to load whenever we want to chat with our trained chatbot.

checkpoint = "./chatbot\_weights.ckpt"

# *For Windows users, replace "chatbot\_weights.ckpt" line of code by: checkpoint = "./chatbot\_weights.ckpt"*

* Then we're going to run our ***session*** by taking our ***session*** object (which we defined using ***tf.InteractiveSession()*** at section 6.2 of this chapter) and then using the ***run*** method.

**session.run**(**tf.global\_variables\_initializer**())

* Inside this ***run*** method we have to ***initialize*** all the ***global variables*** by **tf.global\_variables\_initializer**().

|  |
| --- |
| **for** epoch **in** **range**(1, epochs + 1):  **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          ending\_time = **time.time**()          batch\_time = ending\_time - starting\_time  **if** batch\_index % batch\_index\_check\_training\_loss **==** 0:  **print**('Epoch: {:>3}/{}, Batch: {:>4}/{}, Training Loss Error: {:>6.3f},  Training Time on 100 Batches: {:d} seconds'.**format**(epoch,  epochs,  batch\_index,  **len**(training\_questions) // batch\_size,  total\_training\_loss\_error / batch\_index\_check\_training\_loss,  **int**(batch\_time \* batch\_index\_check\_training\_loss)))              total\_training\_loss\_error = 0  **if** batch\_index % batch\_index\_check\_validation\_loss **==** 0 **and** batch\_index **>** 0:              total\_validation\_loss\_error = 0              starting\_time = **time.time**()  **for** batch\_index\_validation, (padded\_questions\_in\_batch, padded\_answers\_in\_batch)  **in** **enumerate**(**split\_into\_batches**(validation\_questions, validation\_answers, batch\_size)):                  batch\_validation\_loss\_error = **session.run**(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: 1})                  total\_validation\_loss\_error += batch\_validation\_loss\_error              ending\_time = **time.time**()              batch\_time = ending\_time - starting\_time              average\_validation\_loss\_error = total\_validation\_loss\_error / (**len**(validation\_questions) / batch\_size)  **print**('Validation Loss Error: {:>6.3f},  Batch Validation Time: {:d} seconds'.**format**(average\_validation\_loss\_error, **int**(batch\_time)))              learning\_rate \*= learning\_rate\_decay  **if** learning\_rate **<** min\_learning\_rate:                  learning\_rate = min\_learning\_rate  **list\_validation\_loss\_error.append**(average\_validation\_loss\_error)  **if** average\_validation\_loss\_error **<=** **min**(list\_validation\_loss\_error):  **print**('I speak better now!!')                  early\_stopping\_check = 0                  saver = **tf.train.Saver**()  **saver.save**(session, checkpoint)  **else**:  **print**("Sorry I do not speak better, I need to practice more.")                  early\_stopping\_check += 1  **if** early\_stopping\_check **==** early\_stopping\_stop:  **break**  **if** early\_stopping\_check **==** early\_stopping\_stop:  **print**("My apologies, I cannot speak better anymore. This is the best I can do.")  **break**  **print**("Game Over") |

**6.12 Training: Part 2 (Big for-loop)**

Now we start the big for loop to do all the training. It is going to FOR loop over the EPOCH.

**for** epoch **in** **range**(1, epochs + 1):

**for** batch\_index, (padded\_questions\_in\_batch, padded\_answers\_in\_batch)

**in** **enumerate**(**split\_into\_batches**(training\_questions, training\_answers, batch\_size)):

* In each epoch we have to *loop over* all the *batches*, se we used a *nested for loop* too. Recall: an *epoch* is completed when all the batches go *back* and *forth* to *double RNN* in our *Seq2Seq model*.

**for** batch\_index, (padded\_questions\_in\_batch, padded\_answers\_in\_batch) **in** **enumerate**(**split\_into\_batches**(training\_questions, training\_answers, batch\_size)):

* **enumerate** converts them into a list of i, (padded\_questions\_in\_batch, padded\_answers\_in\_batch) format.
* Our defined **split\_into\_batches** will return the padded\_questions\_in\_batch, padded\_answers\_in\_batch.
* Hence we used batch\_index, (padded\_questions\_in\_batch, padded\_answers\_in\_batch) as loop variable, where is a tuple.
* starting\_time = **time.time**() to measure the *time* of the *training of each batch* . It will calculate the difference between the end time and starting time to get the training-time at the bench.
* Training Loss Error Of Specific Batch: We'll use **session.run()** to run our session. To get this batch\_training\_loss\_error, we're going to ***run*** our ***session*** with our:
* optimizer and the loss\_error that we'll put into a list
* Then we input following as a dictionary:
  + *inputs*
  + *targets*
  + *learning rate*
  + *sequence length*
  + *keep probability*

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})

* Since the return two outputs and we'll need last one, we've used **\_** for the first.
* padded\_answers\_in\_batch.shape[1] gets the second ***dim*** of padded\_answers\_in\_batch.
* *learning\_rate*, *keep\_probability* are from the hyper-parameters.
* Total Training Loss Error: Now we've just got our batch\_training\_loss\_error, we're going to add it to the total\_training\_loss\_error (because we want to get the total training loss error on 100 batches, then we'll do an average).

total\_training\_loss\_error += batch\_training\_loss\_error

* Ending Time & Batch time: We again use time.time() to compute ending time, then we use previously calculated starting\_time to get the batch\_time.

        ending\_time = **time.time**()

        batch\_time = ending\_time - starting\_time

* Training Loss Error: We have batch\_index\_check\_training\_loss =100, i.e after *each 100 batches* we print the following (using ***python formatting***):
* Current epoch & epoch
* Current batch and total batch
* Training Loss Error
* Training Time on 100 Batches
* total\_training\_loss\_error = 0, after printing, we re-initialize total\_training\_loss\_error for the next 100 batches:

        ending\_time = **time.time**()

        batch\_time = ending\_time - starting\_time

**if** batch\_index % batch\_index\_check\_training\_loss **==** 0:

**print**('Epoch: {:>3}/{}, Batch: {:>4}/{}, Training Loss Error: {:>6.3f},

Training Time on 100 Batches: {:d} seconds'.**format**(epoch,

epochs,

batch\_index,

**len**(training\_questions) // batch\_size,

total\_training\_loss\_error / batch\_index\_check\_training\_loss,

**int**(batch\_time \* batch\_index\_check\_training\_loss)))

            total\_training\_loss\_error = 0

**Training loss error: All Code At Once**

**for** epoch **in** **range**(1, epochs + 1):

**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

ending\_time = **time.time**()

batch\_time = ending\_time - starting\_time

**if** batch\_index % batch\_index\_check\_training\_loss **==** 0:

**print**('Epoch: {:>3}/{}, Batch: {:>4}/{}, Training Loss Error: {:>6.3f},

Training Time on 100 Batches: {:d} seconds'.**format**(epoch,

epochs,

batch\_index,

**len**(training\_questions) // batch\_size,

total\_training\_loss\_error / batch\_index\_check\_training\_loss,

**int**(batch\_time \* batch\_index\_check\_training\_loss)))

            total\_training\_loss\_error = 0

* Validation Loss Error: We do the same to calculate validation loss error, inside a new if-condition.
* We're going to compute an average of some validation loss errors on the validation set. Remember we want to check the validation error halfway of an epoch. But we don't want to include the first batch.

**if** batch\_index % batch\_index\_check\_validation\_loss **==** 0 **and** batch\_index **>** 0:

* total\_validation\_loss\_error = 0 initialize the total validation loss error to zero. Exactly as we did before with the total\_training\_loss\_error.
* Then we're going to get starting\_time = **time.time**() because we're going to measure the time of each batch again.
* Next we do a new for loop (it is similar to our previously used: **for** batch\_index, (padded\_questions\_in\_batch, padded\_answers\_in\_batch) but this time we use it for validation set). Basically we copied the previous For-loop and changed it for the validation set.

**if** batch\_index % batch\_index\_check\_validation\_loss **==** 0 **and** batch\_index **>** 0:

            total\_validation\_loss\_error = 0

            starting\_time = **time.time**()

**for** batch\_index\_validation, (padded\_questions\_in\_batch, padded\_answers\_in\_batch)

**in** **enumerate**(**split\_into\_batches**(validation\_questions, validation\_answers, batch\_size)):

                batch\_validation\_loss\_error = **session.run**(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: 1})

                total\_validation\_loss\_error += batch\_validation\_loss\_error

* Inside session.run(), we don’t need optimizer (because in validation, we only do test, not the training). Since we are avoiding the list including optimizer, the session.run() will return only one value, hence we didn't used "**\_**".
* We'll use our parameter-dictionary as we did before.
* No dropout: Another thing to note that, for validation set we *don’t use any dropout-regularization*, hence keep\_prob: 1. *Dropout-regularization* is only used for training.
* Now we are done with this for-loop, we calculate the ***ending\_time***, ***batch\_time***, ***average\_validation\_loss\_error*** out-side the for-loop. Then we print: Validation Loss Error, Batch Validation Time.

            ending\_time = **time.time**()

            batch\_time = ending\_time - starting\_time

            average\_validation\_loss\_error = total\_validation\_loss\_error / (**len**(validation\_questions) / batch\_size)

**print**('Validation Loss Error: {:>6.3f},

Batch Validation Time: {:d} seconds'.**format**(average\_validation\_loss\_error, **int**(batch\_time)))

* Decay to the learning rates: Here we'll apply some decay to the learning rates and we do that jus by a multiplication as follows:

learning\_rate \*= learning\_rate\_decay

* Checking against the min\_learning\_rate:

**if** learning\_rate **<** min\_learning\_rate:

                learning\_rate = min\_learning\_rate

* Apply Early Stopping: First we append the current average\_validation\_loss\_error to the list of validation loss error list\_validation\_loss\_error. Then we check the current average\_validation\_loss\_error against the minimum from the list\_validation\_loss\_error.
* If average\_validation\_loss\_error improved, we continue.
* Also we save the model by using:

                saver = **tf.train.Saver**()

**saver.save**(session, checkpoint)

* If average\_validation\_loss\_error not improved, we apply the early stopping. When we reach to the maximum, i.e. early\_stopping\_stop, we break out from this loop.

**list\_validation\_loss\_error.append**(average\_validation\_loss\_error)

**if** average\_validation\_loss\_error **<=** **min**(list\_validation\_loss\_error):

**print**('I speak better now!!')

                early\_stopping\_check = 0

                saver = **tf.train.Saver**()

**saver.save**(session, checkpoint)

**else**:

**print**("Sorry I do not speak better, I need to practice more.")

                early\_stopping\_check += 1

**if** early\_stopping\_check **==** early\_stopping\_stop:

**break**

**Validation loss error: All Code At Once**

**if** batch\_index % batch\_index\_check\_validation\_loss **==** 0 **and** batch\_index **>** 0:

            total\_validation\_loss\_error = 0

            starting\_time = **time.time**()

**for** batch\_index\_validation, (padded\_questions\_in\_batch, padded\_answers\_in\_batch)

**in** **enumerate**(**split\_into\_batches**(validation\_questions, validation\_answers, batch\_size)):

                batch\_validation\_loss\_error = **session.run**(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: 1})

                total\_validation\_loss\_error += batch\_validation\_loss\_error

            ending\_time = **time.time**()

            batch\_time = ending\_time - starting\_time

            average\_validation\_loss\_error = total\_validation\_loss\_error / (**len**(validation\_questions) / batch\_size)

**print**('Validation Loss Error: {:>6.3f},

Batch Validation Time: {:d} seconds'.**format**(average\_validation\_loss\_error, **int**(batch\_time)))

            learning\_rate \*= learning\_rate\_decay

**if** learning\_rate **<** min\_learning\_rate:

                learning\_rate = min\_learning\_rate

**list\_validation\_loss\_error.append**(average\_validation\_loss\_error)

**if** average\_validation\_loss\_error **<=** **min**(list\_validation\_loss\_error):

**print**('I speak better now!!')

                early\_stopping\_check = 0

                saver = **tf.train.Saver**()

**saver.save**(session, checkpoint)

**else**:

**print**("Sorry I do not speak better, I need to practice more.")

                early\_stopping\_check += 1

**if** early\_stopping\_check **==** early\_stopping\_stop:

**break**

* Stopping the Main for loop: If we reach to the early stop value ***early\_stopping\_stop***. We'll break our main for-loop " **for** epoch **in** **range**(1, epochs + 1)", *before completing all the epochs*.

**if** early\_stopping\_check **==** early\_stopping\_stop:

**print**("My apologies, I cannot speak better anymore. This is the best I can do.")

**break**

**print**("Game Over")

**All Code at once: The training**

# *Building a ChatBot with Deep-NLP*

'''

NOTE: Need Specific environment. Install following:

    conda create --name py354 python=3.5.4

    conda activate py354

    pip install protobuf==3.19.4

    pip install tensorflow==1.0.0

    conda install ipykernel

    conda install spyder

 '''

# *Impoting the libraries*

**import** numpy **as** np

**import** tensorflow **as** tf

**import** re

**import** time

# *==============================     Part 1 : Data Preprocessing     ==============================*

# *Importing the dataset*

lines\_in\_cnvrstn = **open**("../data\_set/movie\_lines.txt", encoding='utf-8', errors='ignore').**read**().**split**('\n')

cnvrstn = **open**("../data\_set/movie\_conversations.txt", encoding='utf-8', errors='ignore').**read**().**split**('\n')

# *Creating a dictionary that maps each linea and its ID*

id\_2\_line = {};

**for** lyn **in** lines\_in\_cnvrstn:

    \_line = **lyn.split**(" +++$+++ ")

**if** **len**(\_line) **==** 5:

        id\_2\_line[\_line[0]] = \_line[4] # *creates the dicttionary*

        # *\_line[0] is id "key in dictionary" and \_line[4] is the line "as value of the key"*

# *Creating the list of all of the conversations*

cnvrstn\_ids = []

**for** cvstn **in** cnvrstn[:-1]:

    \_cnvrstn =  **cvstn.split**(" +++$+++ ")[-1][1:-1].**replace**("'", "").**replace**(" ", "")

**cnvrstn\_ids.append**(**\_cnvrstn.split**(","))

# *Getting seperately the questions and answers*

raw\_questn = []

raw\_ans = []

**for** cvstn **in** cnvrstn\_ids:

**for** i **in** **range**(**len**(cvstn) - 1):

**raw\_questn.append**(id\_2\_line[cvstn[i]])  # *using the "id\_2\_line" dictionary, by id-key*

**raw\_ans.append**(id\_2\_line[cvstn[i+1]])

        # *range(len(cvstn) - 1) is used because of cvstn[i+1]*

        # *notice we are using both "cnvrstn\_ids" and "id\_2\_line"*

# *Doing the first cleaning of the texts*

**def** **clean\_text**(text):

    text = **text.lower**();

    text = **re.sub**(r"i'm", "i am", text)

    text = **re.sub**(r"he's", "he is", text)

    text = **re.sub**(r"she's", "she is", text)

    text = **re.sub**(r"that's", "that is", text)

    text = **re.sub**(r"what's", "what is", text)

    text = **re.sub**(r"where's", "where is", text)

    text = **re.sub**(r"\'ll", " will", text)

    text = **re.sub**(r"\'ve", " have", text)

    text = **re.sub**(r"\'re", " are", text)

    text = **re.sub**(r"\'d", " would", text)

    text = **re.sub**(r"won't", "will not", text)

    text = **re.sub**(r"can't", "cannot", text)

    text = **re.sub**(r"[-()#/@;:<>{}+=~|.?,]", "", text)

**return** text

# *Cleaning the questions*

clean\_questn = []

**for** qes **in** raw\_questn:

**clean\_questn.append**(**clean\_text**(qes))

# *Cleaning the answers*

clean\_ans = []

**for** aNs **in** raw\_ans:

**clean\_ans.append**(**clean\_text**(aNs))

# *create a dictionary that maps each word to its number of occurrences.*

word2count = {}

**for** questn **in** clean\_questn:

**for** word **in** **questn.split**():

**if** word **not** **in** word2count:

            word2count[word] = 1

**else**:

            word2count[word] += 1

**for** ans **in** clean\_ans:

**for** word **in** **ans.split**():

**if** word **not** **in** word2count:

            word2count[word] = 1

**else**:

            word2count[word] += 1

# *Creating two dictionaries that map the questions words and the answers words to a unique integers*

threshold = 20

word\_number = 0

questnWrd2Int = {}

# *'word' is "key" and 'count' is "value"*

**for** word, count **in** **word2count.items**():

**if** count **>=** threshold:

        questnWrd2Int[word] = word\_number

        word\_number += 1

word\_number = 0

ansWrd2Int = {}

**for** word, count **in** **word2count.items**():

**if** count **>=** threshold:

        ansWrd2Int[word] = word\_number

        word\_number += 1

# *Adding the last tokens to these two dictionaries*

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

**for** tkn **in** tokens:

    # *adding 'token' as "key" and 'unique int' as "value"*

    questnWrd2Int[tkn] = **len**(questnWrd2Int) + 1

**for** tkn **in** tokens:

    # *adding 'token' as "key" and 'unique int' as "value"*

    ansWrd2Int[tkn] = **len**(ansWrd2Int) + 1

# *Creating inverse dictionary of the ansWrd2Int, notice ':' is used to crete dictionary*

ans\_Int\_2\_Wrd = {w\_i: w **for** w, w\_i **in** **ansWrd2Int.items**()}

# *Adding EOS tokens to the end of every answers*

**for** i **in** **range**(**len**(clean\_ans)):

    clean\_ans[i] += " <EOS>" # *notice a space is added to seperate <EOS>*

# *Translating all the Questions and the Answers into integers*

# *and replacing all the words that were filtered out by our token <OUT>*

questions\_to\_int = []

**for** question **in** clean\_questn:

    ints = []

**for** word **in** **question.split**():

**if** word **not** **in** questnWrd2Int:

            # *Checking filtered word and adding <OUT>'s token*

**ints.append**(questnWrd2Int["<OUT>"])

**else**:

            # *Adding word's corresponding token*

**ints.append**(questnWrd2Int[word])

    # *Finally "ints" is containing a tokenized question sentence,*

    # *we append this to "questions\_to\_int"*

**questions\_to\_int.append**(ints)

# *We do same for the Answers*

answers\_to\_int = []

**for** answer **in** clean\_ans:

    ints = []

**for** word **in** **answer.split**():

**if** word **not** **in** ansWrd2Int:

**ints.append**(ansWrd2Int["<OUT>"])

**else**:

**ints.append**(ansWrd2Int[word])

**answers\_to\_int.append**(ints)

# *Sorting the both questions and answers by the length of the questions*

max\_line\_length = 25

sorted\_clean\_questions = []

sorted\_clean\_answers = []

# *print(enumerate(questions\_to\_int));*

enumarated = **list**(**enumerate**(questions\_to\_int))

**for** length **in** **range**(1, max\_line\_length+1):

**for** i **in** **enumerate**(questions\_to\_int):

**if** **len**(i[1]) **==** length:

**sorted\_clean\_questions.append**(questions\_to\_int[i[0]])

**sorted\_clean\_answers.append**(answers\_to\_int[i[0]])

# *==============================     Part 2 : building SEQ2SEQ model     ==============================*

# *creating TF placeholder for inputs and target*

**def** **model\_inputs**():

    inpUts = **tf.placeholder**(tf.int32, [**None**, **None**], name="input")

    tarGets = **tf.placeholder**(tf.int32, [**None**, **None**], name="target")

    lr = **tf.placeholder**(tf.float32, name="learning\_rate")

    keep\_prob = **tf.placeholder**(tf.float32, name="keep\_prob")

**return** inpUts, tarGets, lr, keep\_prob

# *preprocessiong the tergets*

**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

# *------------------ Creating the ENCODER RNN layer ----------------*

**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*

    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

# *Decoding the training set*

**def** **decode\_training\_set**(encoder\_state, decoder\_cell, decoder\_embedded\_input, sequence\_length, decoding\_scope, output\_function, keep\_prob, batch\_size):

    attention\_states = **tf.zeros**([batch\_size, 1, decoder\_cell.output\_size])

    attention\_keys, attention\_values, attention\_score\_function, attention\_construct\_function = **tf.contrib.seq2seq.prepare\_attention**(

                                                                                                                    attention\_states,

                                                                                                                    attention\_option = "bahdanau",

                                                                                                                    num\_units = decoder\_cell.output\_size)

    training\_decoder\_function = **tf.contrib.seq2seq.attention\_decoder\_fn\_train**(encoder\_state[0],

                                                                              attention\_keys,

                                                                              attention\_values,

                                                                              attention\_score\_function,

                                                                              attention\_construct\_function,

                                                                              name = "attn\_dec\_train")

    decoder\_output, decoder\_final\_state, decoder\_final\_context\_state = **tf.contrib.seq2seq.dynamic\_rnn\_decoder**(decoder\_cell,

                                                                                                              training\_decoder\_function,

                                                                                                              decoder\_embedded\_input,

                                                                                                              sequence\_length,

                                                                                                              scope = decoding\_scope)

    decoder\_output\_dropout = **tf.nn.dropout**(decoder\_output, keep\_prob)

**return** **output\_function**(decoder\_output\_dropout)

# *Decoding the test/validatin set*

**def** **decode\_test\_set**(encoder\_state, decoder\_cell, decoder\_embedding\_matrix, sos\_id, eos\_id, maximum\_length, num\_words, decoding\_scope, output\_function, keep\_prob, batch\_size):

    attention\_states = **tf.zeros**([batch\_size, 1, decoder\_cell.output\_size])

    attention\_keys, attention\_values, attention\_score\_function, attention\_construct\_function = **tf.contrib.seq2seq.prepare\_attention**(

                                                                                                                    attention\_states,

                                                                                                                    attention\_option = "bahdanau",

                                                                                                                    num\_units = decoder\_cell.output\_size)

    test\_decoder\_function = **tf.contrib.seq2seq.attention\_decoder\_fn\_inference**(

                                                                                output\_function,

                                                                                encoder\_state[0],

                                                                                attention\_keys,

                                                                                attention\_values,

                                                                                attention\_score\_function,

                                                                                attention\_construct\_function,

                                                                                decoder\_embedding\_matrix,

                                                                                sos\_id,

                                                                                eos\_id,

                                                                                maximum\_length,

                                                                                num\_words,

                                                                                name = "attn\_dec\_inf")

    test\_prediction, decoder\_final\_state, decoder\_final\_context\_state = **tf.contrib.seq2seq.dynamic\_rnn\_decoder**( decoder\_cell,

                                                                                                                test\_decoder\_function,

                                                                                                                scope = decoding\_scope)

**return** test\_prediction

# *------------------ Creating the DECODER RNN layer ----------------*

**def** **decoder\_rnn**(decoder\_embedded\_input, decoder\_embedding\_matrix, encoder\_state, num\_words, sequence\_length, rnn\_size, num\_layers, word2int,  keep\_prob, batch\_size):

**with** **tf.variable\_scope**("decoding") **as** decoding\_scope:

        lstm = **tf.contrib.rnn.BasicLSTMCell**(rnn\_size)   # *define a layer*

        lstm\_dropout = **tf.contrib.rnn.DropoutWrapper**(lstm, input\_keep\_prob = keep\_prob)     # *Apply dropout on the layer*

        decoder\_cell = **tf.contrib.rnn.MultiRNNCell**([lstm\_dropout]\*num\_layers)   # *Creating stack of the layer*

        # *Intialize weight*

        weights = **tf.truncated\_normal\_initializer**(stddev = 0.1)

        biases = **tf.zeros\_initializer**()

        output\_function = **lambda** x: **tf.contrib.layers.fully\_connected**(  x,

                                                                        num\_words,

**None**,

                                                                        scope = decoding\_scope,

                                                                        weights\_initializer = weights,

                                                                        biases\_initializer = biases)

        # *Training Predictions*

        training\_predictions = **decode\_training\_set**( encoder\_state,

                                                    decoder\_cell,

                                                    decoder\_embedded\_input,

                                                    sequence\_length,

                                                    decoding\_scope,

                                                    output\_function,

                                                    keep\_prob,

                                                    batch\_size)

        # *Test Predictions*

**decoding\_scope.reuse\_variables**()

        test\_prediciton = **decode\_test\_set**(  encoder\_state,

                                            decoder\_cell,

                                            decoder\_embedding\_matrix,

                                            word2int['<SOS>'],

                                            word2int['<EOS>'],

                                            sequence\_length -1 ,

                                            num\_words,

                                            decoding\_scope,

                                            output\_function,

                                            keep\_prob,

                                            batch\_size)

**return** training\_predictions, test\_prediciton

'''

--- Python Lambda ---

    A lambda function is a small anonymous function. A lambda function can take any number of arguments, but can only have one expression.

Syntax

        lambda arguments : expression

    The expression is executed and the result is returned

Example:

    Add 10 to argument a, and return the result:

    x = lambda a : a + 10

    print(x(5))

'''

# *Building the seq2seq model : Brain of our chatbot*

**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, questnWrd2Int):

    # *assemblage = putting togather*

    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, questnWrd2Int, batch\_size)

    decoder\_embeddings\_matrix = **tf.Variable**(**tf.random\_uniform**([questions\_num\_words+1, decoder\_embedding\_size], 0, 1))

    decoder\_embedded\_input = **tf.nn.embedding\_lookup**(decoder\_embeddings\_matrix, preprocessed\_targets)

    traing\_pred, test\_pred = **decoder\_rnn**(   decoder\_embedded\_input,

                                            decoder\_embeddings\_matrix,

                                            encoder\_state,

                                            questions\_num\_words,

                                            sequence\_length,

                                            rnn\_size,

                                            num\_layers,

                                            questnWrd2Int,

                                            keep\_prob,

                                            batch\_size)

**return** traing\_pred, test\_pred

# *==============================     Part 3 : Trainig SEQ2SEQ model     ==============================*

# *Setting the HyperPartameters: We choosed these names for TensorFlow matching*

epochs = 100    # *or 50*

batch\_size = 64    # *or 128*

rnn\_size = 512

num\_layers = 3

encoding\_embedding\_size = 512

decoding\_embedding\_size = 512

learning\_rate = 0.01

learning\_rate\_decay = 0.9

min\_learning\_rate = 0.0001

keep\_probability = 0.5  # *optimal according to Hinton*

# *Defining a session*

**tf.reset\_default\_graph**()

session = **tf.InteractiveSession**()

# *Loding the model Inputs*

inputs, targets, lr, keep\_prob = **model\_inputs**()

# *Setting the Sequence Length*

sequence\_length = **tf.placeholder\_with\_default**(25, **None**, name = 'sequence\_length')

# *Getting the shape of the inputs tensor*

input\_shape = **tf.shape**(inputs)

# *Getting the training predictions and test predictions*

training\_predictions, test\_predictions = **seq2seq\_model**(**tf.reverse**(inputs, [-1]),

                                                       targets,

                                                       keep\_prob,

                                                       batch\_size,

                                                       sequence\_length,

**len**(ansWrd2Int),

**len**(questnWrd2Int),

                                                       encoding\_embedding\_size,

                                                       decoding\_embedding\_size,

                                                       rnn\_size,

                                                       num\_layers,

                                                       questnWrd2Int)

# *Setting up the Loss Error, the Optimizer and Gradient Clipping*

**with** **tf.name\_scope**("optimization"):

    loss\_error = **tf.contrib.seq2seq.sequence\_loss**(training\_predictions,

                                                  targets,

**tf.ones**([input\_shape[0], sequence\_length]))

    optimizer = **tf.train.AdamOptimizer**(learning\_rate)

    gradients = **optimizer.compute\_gradients**(loss\_error)

    clipped\_gradients = [(**tf.clip\_by\_value**(grad\_tensor, -5., 5.), grad\_variable) **for** grad\_tensor, grad\_variable **in** gradients **if** grad\_tensor **is** **not** **None**]

    optimizer\_gradient\_clipping = **optimizer.apply\_gradients**(clipped\_gradients)

# *Padding the sequences with the <PAD> token*

    # *Question: ["Who", "are", "you"]*

    # *Answer: [<SOS>, "I", "am", "a", "bot",".", <EOS>]*

    # *After we apply the padding this question and answer will become this:*

        # *Question: ["Who", "are", "you", <PAD>, <PAD>, <PAD>, <PAD>]*

        # *Answer: [<SOS>, "I", "am", "a", "bot",".", <EOS>]*

**def** **apply\_padding**(batch\_of\_sequences, word2int):

    max\_sequence\_length = **max**([**len**(sequence) **for** sequence **in** batch\_of\_sequences])

**return** [sequence + [word2int['<PAD>']] \* (max\_sequence\_length - **len**(sequence)) **for** sequence **in** batch\_of\_sequences]

# *Splitting the data into batches of questions and answers*

**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, questnWrd2Int))

        padded\_answers\_in\_batch = **np.array**(**apply\_padding**(answers\_in\_batch, ansWrd2Int))

**yield** padded\_questions\_in\_batch, padded\_answers\_in\_batch

# *Splitting the questions and answers into training and validation sets*

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]

# *Training*

batch\_index\_check\_training\_loss = 100

batch\_index\_check\_validation\_loss = ((**len**(training\_questions)) // batch\_size // 2) - 1

total\_training\_loss\_error = 0

list\_validation\_loss\_error = []

early\_stopping\_check = 0

early\_stopping\_stop = 1000      # *we choose 1000 to reach the last epoch i.e. 100-th epoch. Generally we set early\_stopping 100*

checkpoint = "./chatbot\_weights.ckpt" # *For Windows users, replace "chatbot\_weights.ckpt" line of code by: checkpoint = "./chatbot\_weights.ckpt"*

**session.run**(**tf.global\_variables\_initializer**())

**for** epoch **in** **range**(1, epochs + 1):

**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

        ending\_time = **time.time**()

        batch\_time = ending\_time - starting\_time

**if** batch\_index % batch\_index\_check\_training\_loss **==** 0:

**print**('Epoch: {:>3}/{}, Batch: {:>4}/{}, Training Loss Error: {:>6.3f}, Training Time on 100 Batches: {:d} seconds'.**format**(epoch,

                                                                                                                                       epochs,

                                                                                                                                       batch\_index,

**len**(training\_questions) // batch\_size,

                                                                                                                                       total\_training\_loss\_error / batch\_index\_check\_training\_loss,

**int**(batch\_time \* batch\_index\_check\_training\_loss)))

            total\_training\_loss\_error = 0

        # *validation loss-error*

**if** batch\_index % batch\_index\_check\_validation\_loss **==** 0 **and** batch\_index **>** 0:

            total\_validation\_loss\_error = 0

            starting\_time = **time.time**()

**for** batch\_index\_validation, (padded\_questions\_in\_batch, padded\_answers\_in\_batch) **in** **enumerate**(**split\_into\_batches**(validation\_questions, validation\_answers, batch\_size)):

                batch\_validation\_loss\_error = **session.run**(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: 1})

                total\_validation\_loss\_error += batch\_validation\_loss\_error

            ending\_time = **time.time**()

            batch\_time = ending\_time - starting\_time

            average\_validation\_loss\_error = total\_validation\_loss\_error / (**len**(validation\_questions) / batch\_size)

**print**('Validation Loss Error: {:>6.3f}, Batch Validation Time: {:d} seconds'.**format**(average\_validation\_loss\_error, **int**(batch\_time)))

            # *decay to the learning rates and early stopping*

            learning\_rate \*= learning\_rate\_decay

**if** learning\_rate **<** min\_learning\_rate:

                learning\_rate = min\_learning\_rate

**list\_validation\_loss\_error.append**(average\_validation\_loss\_error)

**if** average\_validation\_loss\_error **<=** **min**(list\_validation\_loss\_error):

**print**('I speak better now!!')

                early\_stopping\_check = 0

                saver = **tf.train.Saver**()

**saver.save**(session, checkpoint)

**else**:

**print**("Sorry I do not speak better, I need to practice more.")

                early\_stopping\_check += 1

**if** early\_stopping\_check **==** early\_stopping\_stop:

**break**

**if** early\_stopping\_check **==** early\_stopping\_stop:

**print**("My apologies, I cannot speak better anymore. This is the best I can do.")

**break**

**print**("Game Over")

# *python prctc\_ctbbt.py*

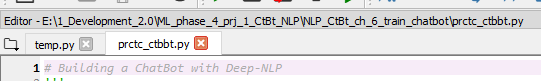
* Running above code into our virtual environment: Open anaconda prompt (search in the windows menu).
* Activate the virtual environment named "py354":

(base) C:\Users\user>conda activate py354

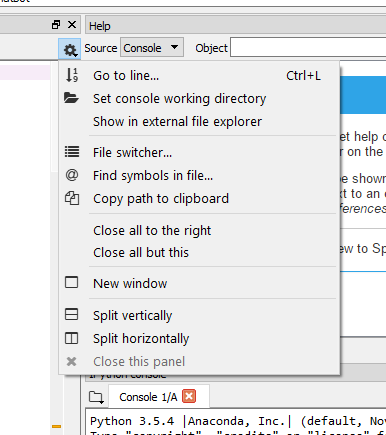
* Open SPYDER IDE from the virtual environment:

(py354) C:\Users\user>spyder

* Open the chatbot.py file from the editor.



* Set the directory as console working directory.



* Execute the code section-by-section.