Chapter 5

**Project: ChatBot**

**Building Seq2Seq model.** Note: TensorFlow v1.0.0 with Python 3.5.4

**5.1 TensorFlow Placeholder**

We're going to make a *function* that will create *placeholders* for the *inputs* and the *target*.

* Placeholder and why do we need to create it: In ***TensorFlow*** all variables are used in tensors. We need to go from the ***NumPy*** arrays to ***tensors***.
* But that's not all because in ***TensorFlow*** all the variables used in tensors must be defined as ***TensorFlow*** placeholders.
* Basically it's an advanced data structure that can contain tensors and also other features. So that's always the first thing to build a deep NN with ***TensorFlow*** .
* We name this function ***model\_inputs*** inside this function we will create a placeholder for the *inputs* and placeholder for the *targets*.
* Then we will add a learning rate and even *more hyper parameters*.
* This function is not going to take any arguments because it will just directly take the input and output and convert them into placeholders.
* The variable ***inputs*** will be *Tensorflow Placeholder* containing the *inputs*.
* ***placeholder()*** creates placeholders for the input. We need to enter three parameters:
* The type of the data: remember that we converted our input into unique integers in part 1 data preprocessing. So the type that will choose here is going to be ***int32***.

# *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")

* Dimensions of the matrix of the input data: Since the inputs are the *lists of the questions* encoded into *unique integers* ie. a list of lists, that’s why we need to specify this as a *two* *dimensional matrix*. [**None**, **None**] Defines this matrix.
* The last argument is just a name that we have to give to the inputs and targets.
* Remember that answers are the *targets*. *Target* is the same as the *initial value* of input at the beginning but it remain *unchanged*. And then we are going to compare the *chatbot* *answers* (i.e. input vector) to the *real answers*.
* Two more TF placeholders:
* ***lr*** which will hold the *learning rate* (a hyper parameter) and
* ***keep\_prob*** is the parameter that controls the *dropout rate*. Dropout is the rate of the *neurons* that you choose to *override during one iteration in the training*. Usually the ***dropout rate*** is ***20%*** so that you ***deactivate 20%*** of the neurons during the different iterations of the training.
* We'll tune these hyperparameters later.

# *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

* Notice that we used type ***tf.float32*** for ***Float*** values. Also we didn't define any 2-D matrices.

**5.2 Preparing Target**

Before we start creating the Encoding and Decoding layer we have to prepare targets because the decoder accept a certain format of the targets to *feed* into the *neural network (LSTM)*.

* Twofold: This format is *twofold*. First the targets must be into *batches*. The RNN of the decoder will not accept *single targets* (i.e. a single answer).
* We will feed the neural network for example a batch of 10 answers at a time.
* First we set a batch size (a hyper-parameter) and then create batches of that size. Each batch will go into the RNN of the Decoder.
* SOS at the beginning: The second important element is that *each of the answers* in the *batch* of *Target* must start with the *SOS* tokens.
* Besides creating ***batches of answers*** we need to put the ***SOS*** ***token*** or at least a unique ***integer encoding SOS*** at the beginning of each of answers inside each batch.
* How to add SOS: Since we need to keep the same size for all the answers you know using the padding, we'll actually take all these answers inside batches and ***remove the last column*** of these ***answers***.
* Then we will make a concatenation to add the ***start of string tokens (SOS-tokens)*** at the beginning of the target in the batches.
* We'll create a function ***preprocess\_targets(targets, word2int, batch\_size)***. It takes 3 parameters,

1. The targets as input because it will return the pre-processed version of this target. That targets is into batches and each target answer will start with the S.O.S token.
2. ***word2int*** represents our dictionary because this is the dictionary that maps the tokens to integers. So that we'll get the ***unique*** ***identifier*** of the ***SOS*** token.
3. The third argument is obviously the batch size to create the batches of targets that we will create.

# *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** preprocess\_targets

* Left side: We're going to create left side of the matrix, contains 10 rows of SOS in 1 column.

left\_side = **tf.fill**([batch\_size, 1], word2int["<SOS>"])

* We used ***fill()*** function to create the matrix, ***[batch\_size, 1]*** means ***batch\_size*** of rows and 1 column. 2nd argument is the filling value ***word2int["<SOS>"]*** unique integer representing SOS.
* Right side: Here we used ***strided\_slice()*** function. It takes the ***targets*** as argument, starts from ***[0,0]*** top-left-corner of the matrix.
* Creates a matrix ***[batch\_size, -1]*** of ***batch\_size*** of rows and excluding last column.
* Note that the last argument ***[1,1]*** means the sliding stride window of size .

right\_side = **tf.strided\_slice**(targets, [0,0], [batch\_size, -1], [1,1])

* Finally we concate these two matrices into one, using

preprocessed\_targets = **tf.concat**([left\_side, right\_side], 1)

* Here **1** means the direction of concatenation. 1 for horizontal concatenation.
* Finally we return ***preprocessed\_targets***.

**return** preprocess\_targets

**5.3 seq2seq Architecture: Encoder**

Now we're going to create the architecture of our seq2seq model. This architecture is twofold. We'll first create the *Encoder RNN layer* then the *Decoder RNN layer*. It will be stacked LSTM which apply DROPOUT to improve the accuracy.

* In this part we're going to make a function that will define the architecture of this *Encoder RNN layer*.
* Note: We can include either LSTM or GRU (Gated Recurrent Unit). At the end of this note we'll talk about a chatbot implementation in PyTorch and GRU (Gated Recurrent Unit).
* But in this one we are going to include an LSTM instead of a GRU. And for this we're going to use the basic LSTM-Cell class by Tensorflow.

# *Creating the ENCODER RNN layer*

**def** **encoder\_rnn\_layer**(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\_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

* This function takes several arguments:

**def** **encoder\_rnn\_layer**(rnn\_inputs, rnn\_size, num\_layers, keep\_prob, sequence\_length):

* rnn\_inputs corresponds to the model inputs. Not to be confused with the questions inputs but inputs like the ***input questions***, the ***target answers***, the ***learningrate***, the ***keep prob***. So that's the model inputs for the RNN.
* rnn\_size is not to be confused with the number of layers in the RNN the rnn\_size is the *number of input tensors* of the *encoder* .
* num\_layers is the number of RNN layers which I'm counting.
* keep\_prob we need this to apply a Dropout Regularization to our LSTM. We will have a stacked LSTM with dropout. We will first make an LSTM and then we will apply dropout.
* sequence\_length which is the *list of the length* of *each question* in the *batch*.
* LSTM cell class:

lstm = **tf.contrib.rnn.BasicLSTMCell**(rnn\_size)

* ***lstm*** represent an object of the **BasicLSTMCell** class. It is inside the ***contrib*** module under the ***rnn*** submodule and then from there we can get the **BasicLSTMCell** class.
* We need to put just one argument which is our ***rnn\_size*** which corresponds to the *number of input tensors* in the layers.
* Dropout: Now we're going to apply dropout to this new ***lstm*** object. It is some kind of wrapper that can wraps an already existing LSTM which is the ***lstm*** object we just created.

lstm\_dropout = **tf.contrib.rnn.DropoutWrapper**(lstm, input\_keep\_prob = keep\_prob)

* We used **DropoutWrapper** class inside the ***contrib*** module under the ***rnn*** submodule. It takes two arguments:
* ***lstm*** which is the same LSTM but on which dropout was applied.
* ***input\_keep\_prob = keep\_prob*** to control the dropout rate. ***input\_keep\_prob*** is not an arbitrary name this is part of the **DropoutWrapper** class.
* Usual dropout rate is 20%. During the training iteration 20% of the neurons weights are not updated (non-exitent/ deactivated in some way).

    encoder\_cell = **tf.contrib.rnn.MultiRNNCell**([lstm\_dropout]\*num\_layers)

* Encoder cell: We are going to create it through a very useful class named **MultiRNNCell** inside the ***contrib*** module under the ***rnn*** submodule.

encoder\_cell = **tf.contrib.rnn.MultiRNNCell**([lstm\_dropout]\*num\_layers)

* Which takes an argument. We use our Dropout applied LSTM object ***lstm\_dropout***.
* We have to use the number of layers we want in our encoder RNN and that's how we put this: ***[lstm\_dropout]*** in square brackets, to make it several layers, we just multiplied that by the number of layers ***num\_layers***.
* Here the encoder cell is composed of several layers.
* Recall intuition notes in "Tutorials On Machine Learning & Deep Learning", RNN part understand the *difference between* ***cells*** *and* ***states***.
* Encoder States: One of the output of the RNN is the states.We're going to get it from the **bidirectional\_dynamic\_rnn** function by the ***nn*** module.
* Since the ***encoder\_state*** is going to be returned by the **bidirectional\_dynamic\_rnn** function and returns two elements. And is the second element so to ignore the first element we used "\_" underscore.

    \_, 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)

* This creates a dynamic version of a Bidirectional Recurrent Neural Network.
* So this kind of dynamic version of a Bidirectional RNN basically takes your input and will build independent forward and backward RNNs.
* Be careful, in this kind of dynamic version of a Bidirectional RNN we have to make sure that the *input size* of the *forward cell* and the *backwards cell* must *match*.

cell\_fw = encoder\_cell specifies forward RNN.

cell\_bw = encoder\_cell specifies backward RNN.

* It is the same encoder cell. So the *forward RNN* and *backward RNN* cells are actually uses same encoder cells.
* sequence\_length is the *list of the length* of *each question* in the *batch*.
* inputs = rnn\_inputs is the model inputs.
* last argument dtype = tf.float32 is to specify the datatypes.
* Finally we return the output of this encoder

**return** encoder\_state

* Note that we are returning the states , not the cells. The cell will be part of the loop in the RNN but will *not be the output*. The output is the states which is in this case encoder\_state.
* Next we build the ***RNN decoder*** and it's not going to be that simple.
* We'll make it in three more steps because we're going to make some *cross-validation* to *separate* some *training*
* *data* and then some cross validated data to reduce overfitting and improve the accuracy of new observations.
* First we'll make a separation between the training data and the cross validation data and then we'll be ready to create our decoder RNN layer.

**5.4 seq2seq Architecture P1: Encoder**

* First step will be to *decode* the *training set*
* Second step will be to *decode* the *validation set*.
* Then we'll be ready to create the *decoder RNN layer*.

# *Creating the ENCODER RNN layer*

**def** **encoder\_rnn\_layer**(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 = "bahdanu", 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)

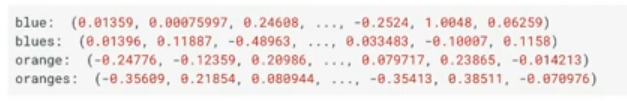
    decoder\_output\_dropout = **tf.nn.dropout**(decoder\_output, keep\_prob)

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

* **decode\_training\_set() Definition:** We're going to make another function **def** **decode\_training\_set**(). It is going to take 8 arguments.

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

* encoder\_state our decoder is getting encoder\_state as input to proceed to the decoding. encoder\_state was returned by the previous function **encoder\_rnn\_layer**().
* decoder\_cell is the cell in the RNN of the decoder.
* decoder\_embedded\_input is the inputs on which were applied embedding.
* An ***embedding*** is a ***mapping*** from discrete objects, such as words, to vectors of real numbers. For example, a 300\*dimensional embedding for English words could include



The individual dimensions in these vectors typically have no inherent meaning. Instead, it s the overall *patterns of location* and *distance* between *vectors* that machine learning takes advantage of.

*Source: tensorflow.org.*

* Basically an embedding is just a *mapping from our words* to *vectors of real numbers*. Each one encoding uniquely the word associated to it.
* Basically it's the kind of format that the decoder is accepting for its inputs.

|  |  |
| --- | --- |
| * sequence\_length is our sequence length. *list of the length* of *each question* in the *batch*. * decoding\_scope visit TenorFlow. It's basically an advanced data-structure that will *wrap* your *TensorFlow variables*. In our case it will be like an object of this ***variable\_scope***. * output\_function is a function we will use to *return* the *decoder outputs* . * keep\_prob we're also going to apply some dropout regularization * batch\_size again we'll work with batches. |  |

* Attention States: We have to initialize these as three dimensional matrices containing only zeroes.

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

* We'll use **tf.zeros**() function similar to **NumPy zero's** function.
* batch\_size is the number of lines, these lines corresponds to the observations and since we're working with batches here so the number of lines is going to be the ***batch\_size***.
* 1 number of columns
* decoder\_cell.output\_size number of elements.
* Keys, Values, The Functions for the attention: And we have to do is prepare the keys, values, the functions for the attention.

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

    attention\_states,

    attention\_option = "bahdanu",

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

* So we're going to get first the attention\_keys then the attention\_values then the attention\_score\_function and lastly attention\_construct\_function.
* All these we will get it in one shot by using **tf.contrib.seq2seq.prepare\_attention()**. Here we're going to *Preprocess* the *Training Data* in order to *Prepare* it to the *Attention Process*.
* Attention features:
* attention\_keys are the *keys* to be *compared* with the *target states*
* attention\_values are the *values* that we'll use to *construct* the *context vectors*. The *context* is returned by the *encoder* and that should be used by the *decoder* as the *first element* of the *decoding*.
* attention\_score\_function is used to compute the *similarity* between the *keys* and the *target states* and
* attention\_construct\_function is a function used to build the *attention* *state*.
* **prepare\_attention()** parameters:
* attention\_states is that we just initialized in the previous line. Right now they're just initialized as zeros.
* attention\_option = "bahdanu" here we choose an *attention option*. We have several *options* to proceed to the *attention process* and one of them is bahdanu.
* decoder\_cell.output\_size is the number of units here.

* Training Decoder Function: It will do the decoding of the training set.

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")

* We'll use another TensorFlow function **attention\_decoder\_fn\_train()** from **tf.contrib.seq2seq** once we prepare the attention correctly, i.e ***attention features***.
* Those ***attention features*** that we created in the previous line of this step, are going to be the arguments of this **attention\_decoder\_fn\_train()**function.
* However we will put one more argument in **attention\_decoder\_fn\_train()** that will be our first argument here and that's the encoder\_state[0]
* encoder\_state[0] note that we got it from the previous function **encoder\_rnn\_layer()** where we used **tf.nn.bidirectional\_dynamic\_rnn()**.

But to get exactly what we want, we need to get the index zero, encoder\_state[0] of this encoder\_state variable.

* Now we input all our ***attention features*** that we created.

attention\_keys,

attention\_values,

attention\_score\_function,

attention\_construct\_function,

* name = "attn\_dec\_train" this last argument here which is the name argument and this *corresponds* to a *name scope* for the *decoding function* .

Basically what we just did here is that we got **attention\_decoder\_fn\_train()** function for the training of our future dynamic RNN-Decoder .

* Decoder Output: Now we get the decoder output, our variable is for that ***decoder\_output***. For this we'll use **tf.contrib.seq2seq.dynamic\_rnn\_decoder()** it is a function that is used to get the ***output*** to ***final state*** and the ***final context state*** of the ***decoder***.

    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)

* Since **dynamic\_rnn\_decoder()**returns three elements, first the ***decoder output*** then the ***decoder final state*** then the ***decoder final context state***, but we are only interested in the ***decoder output***. We can use

decoder\_output, \_, \_ =

But for clear understanding we gonna use the other two elements full name.

decoder\_output, decoder\_final\_state, decoder\_final\_context\_state =

**tf.contrib.seq2seq.dynamic\_rnn\_decoder**(

* Also lets rename **\_** in our previous **def** **encoder\_rnn\_layer**() to encoder\_output.

**\_**, encoder\_state = **tf.nn.bidirectional\_dynamic\_rnn**(

to

**encoder\_output**, encoder\_state = **tf.nn.bidirectional\_dynamic\_rnn**(

* Actually we don't need the first returned element **encoder\_output** for Encoding. And for the Decoding we only need the first returned element ***decoder\_output***.
* Parameters of **dynamic\_rnn\_decoder()**function:
* **decoder\_cell** which is one of our arguments of the current **def** **decode\_training\_set**()function definition.
* training\_decoder\_function This one we got from the previous line.
* decoder\_embeded\_input which is also one of the arguments of the current **def** **decode\_training\_set**()function definition.
* sequence\_length *list of the length* of *each question* in the *batch*.
* scope = decoding\_scope one of the arguments of the current **def** **decode\_training\_set**()function definition..
* Applying Dropout and returning the output value: Now we add a dropout to our ***decoder\_output***. We'll apply Tensorflow's **tf.nn.dropout()** function.

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

* As arguments here, we apply the previous line's output ***decoder\_output*** and dropout rate ***keep\_prob***.
* To return the output value we're going to use our **output\_function** which is one of the arguments of the current **def** **decode\_training\_set**()function definition.

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

* This **output\_function** is going to take what we want to return. We want to return our decoder\_output\_dropout.

Next we're going to tackle the decoding of the validation set. It's going to be quite the same as what we just did. We're just going to make a copy paste and then we'll have a few things to change.

**5.5 seq2seq Architecture P2: Decoder**

* The previous one we made **decode\_training\_set**()function that *decode* the *observations* of the *training set* we also returned the output of the decoder.
* That was for the *observations* of the *training set* that are going *back* into the *neural network* to *update* the *weights* and improve the ability of the *ChatBot* to talk like a *human*.
* Now we do the same for the observations of the Test set and the Validation set. These are new observations that *won't be used* for the *training*.
* The function we're about to make will be used not only to *predict* the *observations* of the *test set* but also the *validation set*.
* We'll create ***validation set*** during the ***training*** to apply some ***cross-validation*** technique. During training we keep ***10%*** of the ***training-set*** separate for ***cross-validation***, to test the predictive power on new observations. It will reduce overfitting and improve the accuracy on new observations.
* So the function we were about to make, will be very similar to the one we just made.
* **attention\_decoder\_fn\_inference:** There is going to be one fundamental change, and that's the following function by TensorFlow.

**attention\_decoder\_fn\_inference**()

* Previously to create **decode\_training\_set**() function we've used the **tf.contrib.seq2seq.attention\_decoder\_fn\_train**() to make training\_decoder\_function.
* But now since we'll have to make a ***test decoding function*** to decode the observations of the Test Set or the Validation set, this time we want to use the **attention\_decoder\_fn\_inference**() function by TensorFlow .
* The inference term means to ***deduce logically***. That perfectly makes sense because once the *ChatBot* is *trained* it has a *logic inside its brain* and therefore it is able to *deduce logically* the *answers* to the *questions* that is being asked based on the logic it *learned* during the *training*.
* So it *understood* its *own logic* during the *training* and therefore after the *training* on new observations, it will *infer* the *answers* to the *questions* that is *being asked*.
* And it make sense here because we're about to *decode* the *Test/Validation* set that is to *decode* some *new observations* that *won't* be used for the *training* and we will use the **attention\_decoder\_fn\_inference**() function.

And that's the main change we'll have to make here.

* Building the **decode\_test\_set()** function:

# *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 = "bahdanu",

                                                            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

* In **def** **decode\_test\_set**() we use almost all previous arguments, but we add five extra arguments such as decoder\_embedding\_matrix, sos\_id, eos\_id, maximum\_length, num\_words and we get rid of decoder\_embedded\_input (do not get confused with it).
* When we make the ***decoder\_rnn\_layer*** and then the ***seq2seq\_model*** later, we will make distinction between these two to decoder\_embedded\_input and the decoder\_embedding\_matrix.
* Checkout more on Tensorflow documentation.
* We add those before sequence\_length. And we don’t need sequence\_length for testing, so we get rid of it.

**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):

* sos\_id start of string, S.O.S token ID,
* eos\_id end of string E.OS token ID ,
* maximum\_length corresponds to the length of the longest entry you can find in the batch.
* num\_words total number of words of all the answers, basically you have to take the answers words to int ***ansWrd2Int*** dictionary, take the length of the dictionary. We'll also use the whole number of words of the questions.
* We need to add these new arguments because the new Tensoflow function **attention\_decoder\_fn\_inference** gonna need those.
* ***attention\_states***; attention features: ***attention\_keys, attention\_values, attention\_score\_function, attention\_construct\_function*** we need all of those so no need to change.
* Because we still need to consider the attention to predict our new observations, the only difference is that these new observations won't be used in the train ( i.e they won't be *back propagated* inside the *NN*) but we still need *attention* because *attention* is part of the *power of* the *prediction*. That's why we need to keep the attention features here, that will be used inside this new function **attention\_decoder\_fn\_inference()**.
* **test\_decoder\_function:** The fundamental change we need to include here is using the function from **contrib.seq2seq**;

**tf.contrib.seq2seq.attention\_decoder\_fn\_inference**()

    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")

This function is going to take almost the same arguments with some new ones:

* The first argument it's going to take is output\_function . The output\_function is needed because *we will not return* our final test predictions using the output\_function.
* The **attention\_decoder\_fn\_inference()** function allows us to input this output\_function as an argument and therefore instead of using it in the end we will use it right here.
* We'll keep all attention features. These attention features that we need to keep because we need to keep the *attention part of the neural network* of the *decoder* because that plays a *powerful role* into *predicting* the *final outcome*. So we keep the attention to *propagate the signal* and get even more *relevant predictions*.

encoder\_state[0],

attention\_keys,

attention\_values,

attention\_score\_function,

attention\_construct\_function,

* After those we add the new arguments

decoder\_embedding\_matrix,

sos\_id,

eos\_id,

maximum\_length,

num\_words,

* Change the scope names: So basically this ***name*** scope is the *name of the mode* of the *decoder* whether it is in *training mode* to *train* the ChatBot on the ***given observations*** or the *inference mode* to *predict* the *outcomes* of ***new observations*** that won't be used for the training. So we need to replace this attn\_dec\_train here by attn\_dec\_inf.
* Modify **dynamic\_rnn\_decoder**: We're still going to use the **dynamic\_rnn\_decoder()** function to get the final output (in this case test\_prediction).

    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

* Since this final output test\_prediction *won't be used for the training* that is *won't be back propagated* into the NN, we use the following arguments only

decoder\_cell,

test\_decoder\_function,

scope = decoding\_scope

* Notice : ***test\_decoder\_function*** is used instead of ***train\_decoder\_function***. Which we got from **contrib.seq2seq.attention\_decoder\_fn\_inference**
* But ***decoder\_embedded\_input*** and ***sequence\_length*** won't be used here because these are for the *training purposes*.
* scope = decoding\_scope we still need that for the test predictions and the validation predictions.
* No-dropout for test: We do-not use any dropout because the dropout is only used for the training to reduce overfishing and improve the accuracy, so we removed this as well.

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

* we also changed

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

to

**return** test\_prediction

* Since previously mentioned, we're not going to use the **output\_function** here in the end to get our test predictions, because we already used this **output\_function** arguments of this **attention\_decoder\_fn\_inference()** function in the previous line, Here we just return test\_prediction directly .
* Renaming the defined encoder-rnn–layer to avoid confusion: Now we are ready to make the **decoder\_rnn\_layer** or more simply the **decoder\_rnn** by the way to avoid confusion with main layer of the seq2seq model we just renamed our previously defined ***encoder\_rnn\_layer*** to simply ***encoder\_rnn***.

Next we will make the function that will build the **decoder\_rnn** and we'll use of course previous two functions ***decode\_train\_set*** and ***decode\_test\_set*** because we will not only get the ***training predictions*** but also the ***test predictions*** for respectively the ***training*** ***set*** and both the ***validation-set*** and the ***test-set***.

**5.6 seq2seq Architecture P3: Decoder**

At ***seq2seq Architecture P1: Encoder*** section we built a function for encoder, initially its name was ***encoder\_rnn\_layer***, and recently we changed it to ***encoder\_rnn***.

Now we built *similar function* for the *decoder*, we call it ***decoder\_rnn()***.

* Resource: Here is where you can find all the resources you'll need: *https://www.superdatascience.com/deep-learning-chatbot/*

# *------------------ 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):

* This function is going to take several arguments.
* **decoder\_embedded\_input**, **decoder\_embedding\_matrix**: we covered these previously.
* **encoder\_state**: It's the ***output*** of the ***encoder*** but that becomes the ***input*** of the ***decoder***.
* **num\_words**: total number of words in our corpus of answers
* **sequence\_lengths**
* rnn\_size is the *number of input tensors* of the *decoder* .
* **num\_layers**: Number of layers we want to have inside the RNN of our decoder (not confused with rnn\_size).
* word2int we're going to need our word to integer dictionaries. It can actually be any dictionary it's just an argument. But that will of course be our *answers words to int dictionary*: **ansWrd2Int** that we made in the preprocessing part.
* **keep\_prob**: we're going to apply some dropout again because there are going to be some training in the implementation of these **decoder\_rnn** .
* **batch\_size**: It's at the heart of any neural network. Either we use ***PyTorch*** or ***TensorFlow*** we must input the data as batches.

# *------------------ 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))

'''

* ***scope = decoding\_scope*** from our definition of def ***decode\_training\_set()***. "***with***" syntax of Python specifies that we're making this **decoder\_rnn** in a decoding scope.
* It's a more advanced TensorFlow variable that will contain several data entities and therefore what we need to introduce this scope.

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

* LSTM layer: It is similar as we did before for **encoder\_rnn()** definition.

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

* Applying DROPOUT: Also similar to **encoder\_rnn()** definition.

lstm\_dropout = **tf.contrib.rnn.DropoutWrapper**(lstm, input\_keep\_prob = keep\_prob)

* Stacked LSTM layers: Since we're building a stacked LSTM we're going to use to multi RNN cell to stack several LSTM layers with Dropout applied. It's exactly what we did beforefor our encoder\_rnn.

decoder\_cell = **tf.contrib.rnn.MultiRNNCell**([lstm\_dropout]\*num\_layers)

Its basically a cell of LSTM layers is an object of the multi RNN cell taken from **tf.contrib.rnn.MultiRNNCell** where lstm\_dropout times the number of layers num\_layers we want to have in this stacked LSTM layer inside the decoder\_cell.

* Initialize weight: We need to initialize some weights that will be *associated* to the *neurons* of the *fully connected layers* inside our decoder (i.e. the last layer). Here we'll use **truncated\_normal\_initializer** initialize, it will generate a Truncated Normal Distribution of the weights.

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

Here standard deviation of ***0.1*** is used. We just initialized here as a Truncated Normal Distribution with Standard Deviation of ***0.1***.

That makes our fully connected the weights initialized.

* Bias's: We just need to initialize them as zeros. We use **tf.zeros\_initializer**() initializer.

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

So now we have our weights our biases for the fully connected layers.

* Output function: We'll create a Python lambda function here.

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

                                                                        num\_words,

**None**,

                                                                        scope = decoding\_scope,

                                                                        weights\_initializer = weights,

                                                                        biases\_initializer = biases)

* This function will return is some fully connected layers that will be used for the last layer of the RNN.
* As we know the fully connected layer comes at the very end of the record neural network. You have the LSTM-cells first (the stacked LSTM in the cell), then at the end of the RNN you have the fully connected layer.
* We have a TensorFlow function that will do that for us. And since we are building the most classic neural network layer it's not going to be taken from RNN it's going to be taken from simply layers **tf.contrib.layers.fully\_connected**.
* **x** is the ***inputs*** and the inputs are of course are viable X at this stage.
* ***num\_words*** is number of outputs. That is the number of words and all the answers.
* next argument is the activation function and will keep the default (which is relu), that’s why we used **None** here.
* Then we specify the ***scope***, ***weights*** and ***biases***.

scope = decoding\_scope,

weights\_initializer = weights,

biases\_initializer = biases

This fully connected layers will get the features from the previous layers of the neural network which is the stacked LSTM. And we'll return the final scores using a legit like a **Soft-Max method**. We will return the *final prediction* which will be the *final answer*.

* Training predictions: Here we just use our previously defined **decode\_training\_set**() function

        # *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: Since we're applying some cross-validation during training, we have to use ***test-predictions*** too. But just before that we have to specify that we want to reuse the variables that were introduced in the decoding scope.

        # *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)

We just use our previously defined **decode\_test\_set**() function

* Finally we return the training\_predictions, test\_prediciton.

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

We just made the **decoder\_rnn()** function that builds the RNN at the decoder. Next we assemble everything that is ***encoder\_rnn()*** and ***decoder\_rnn*** to build the final ***seq2seq*** model.

**5.7 seq2seq Architecture "seq2seq": Decoder – P4**

# *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

* Basically we will just assemble **encoder\_rnn** and **decoder\_rnn** to build ***seq2seq*** model. It's the brain of our checkbook.

**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):

* inputs is the question from our ***data-set*** and after the ***training*** these ***inputs*** will be also the questions we'll ask to the ***ChatBot***.
* targets is the answers, the real answer is that we have in our dataset.
* We'll use other arguments that we used already:

keep\_prob, batch\_size, sequence\_length, answers\_num\_words, questions\_num\_words, encoder\_embedding\_size, decoder\_embedding\_size, rnn\_size, num\_layers, questnWrd2Int

* encoder\_embedding\_size and decoder\_embedding\_size are the number of dimensions of the embedding matrix for the encoder/decoder.
* questnWrd2Int dictionary is used to preprocess the targets. We defined this function at the beginning,

**def** **preprocess\_targets**(targets, word2int, batch\_size)

* This function is supposed to return in the end the training predictions and the test predictions. That's basically what we've returned before, in the ***decoder\_rnn***. We just need to use it here.
* We also need to ***encoder\_rnn*** of course because in order for the ***decoder\_rnn*** to return the training predictions and test predictions it needs to take the ***encoder-States*** returned by the ***encoder\_rnn***.
* Before we get ***encoder-States*** we need the encoder embedded input :

    encoder\_embedded\_input = **tf.contrib.layers.embed\_sequence**(  inputs,

                                                                answers\_num\_words+1,

                                                                encoder\_embedding\_size,

                                                                initializer = **tf.random\_uniform\_initializer**(0, 1))

Since the upeer bount is excluded we used, answers\_num\_words+1

* Encoder-States: We just use our previously defined **encoder\_rnn()**. Since encoder\_state are the output of ***encoder\_rnn***.

encoder\_state = **encoder\_rnn**(encoder\_embedded\_input, rnn\_size, num\_layers, keep\_prob, sequence\_length)

* We're going to feed the RNN of our encoder with the encoder embedded input encoder\_embedded\_input and it will return the encoder\_state which will then become the input of the ***decoder\_rnn()***.
* Pre-Processed Target, decoder\_embeddings\_matrix, decoder\_embedded\_input: Now we are ready to get to our final predictions: ***training predictions***, and the ***test predictions***. But before that we need pre-processed target because we will need them for the training.
* We also need to get the **decoder\_embeddings\_matrix**,which is the embeddings matrix of the decoder.
* That will be used to get **decoder\_embedded\_input** and then that's where we'll be fully ready to apply to ***decoder\_rnn*** function to get the final *training predictions* and *test predictions*.

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)

* **tf.Variable** is a class and therefore the decoder\_embeddings\_matrix will be an object of this **Variable** class and this variable class takes several arguments which are going to be mostly the dimensions of this embedding matrix.
* How the embeddings matrix works: Basically we'll have all the words in lines and then in columns you have the embedding values.
* Since the number of lines corresponds to the total number of words in the whole corpus. Well this *number of lines* is going to be the questions\_num\_words+1.
* For the number of columns the second dimension we use decoder\_embedding\_size.

Basically you can have ***several sizes*** for the ***embeddings*** that is ***several number of columns*** and that is ***determined*** by the ***embedding size***.

* To initialize this matrix we use **tf.random\_uniform** function.
* [questions\_num\_words+1, decoder\_embedding\_size], Specifies dimension .
* We also fill it with random numbers taken from a uniform distribution just to initialize it and then it will be of course populated with learning numbers that will participate into the training of the ChatBot to speak like a human.
* 0, 1 are going to be the lower bound in the upper bound of those random numbers that we're initializing this matrix with. And so we're going to fill this matrix with random numbers from 0 to 1. These random numbers will be taken from a uniform distribution.
* The next step is naturally to use this decoder embeddings matrix and Pre-Processed Target to get that decoder embedded input.Here we used **tf.nn.embedding\_lookup**
* In order to get our ***decoder embedded input*** we need to use ***pre-processed targets*** as they are the answers to the questions.
* **traing\_pred, test\_pred:** Now we have everything we need to feed the RNN of the decoder i.e. ***decoder\_rnn***. Therefore now we're going to get our two ultimate variables that will be returned which are **traing\_pred, test\_pred**
* traing\_pred are predictions of the observations used for the training and
* test\_pred predictions of new observations that won't be used for the training.
* And that will either be used for *cross-validation* to *test* the *predictive power* of the model on *new observations* that *won't be used to train the model more* or some new predictions just to chat with the ***ChatBot***.
* Make sure you understand the difference between these two predictions.
* Now we're going to use our defined ***decoder\_rnn()*** function,

    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