Text summarizer using deep learning made easy

amr zaki Follow
Jan 12 · 8 min read

In this series we will discuss a truly exciting natural language processing topic that is using **deep learning techniques to summarize text** , the code for this series is open source , and is found in a jupyter notebook format , to allow it to run on google colab without the need to have a powerful gpu , in addition all data is open source , and you don't have to download it , as you can connect google colab with google drive and put your data directly onto google drive , without the need to download it locally , read this blog to learn more about google colab with google drive .

To summarize text you have 2 main approaches (i truly like how it is explained in this blog)

1. **Extractive method** , which is choosing specific main words from the input to generate the output , this model tends to work , but won't output a correctly structured sentences , as it just selects words from input and copy them to the output , without actually understanding the sentences , think of it as a highlighter .

2. Abstractive method , which is building a neural network to truly workout the relation between the input and the output , not merely copying words , this series would go though this method , think of it like a pen.

this series is made for whomever feels excited to learn the power of building a deep network that is capable of

- analyzing sequences of input
- understanding text
- outputting sequences of output in form of summarizes

hence the name of seq2seq , sequence of inputs to sequence of outputs , which is the main algorithm that is used here .

This series would go into details on how to

1. build your deep learning network online without the need to have a powerful computer
2. Access your datasets online , without the need to download the datasets to your computer.
3. Build a tensorflow networks to address the task

Multiple research has been done throughout the last couple of years , I am currently researching these new approaches , in this series we would go through some of these approaches.

This series implement its code using google colab , so no need to have a powerful computer to implement these ideas , I am currently working on converting the most recent researches to a google colab notebooks for researches too try them out without the need to have powerful gpus , also all the data can be used without the need to download them , as we would use google drive with google colab , read this blog to learn more about how you can work on google ecosystem for deep learning

All the code would be available on this github repo , which contains modifications on some open source implementations of text stigmatization

these researches mainly include

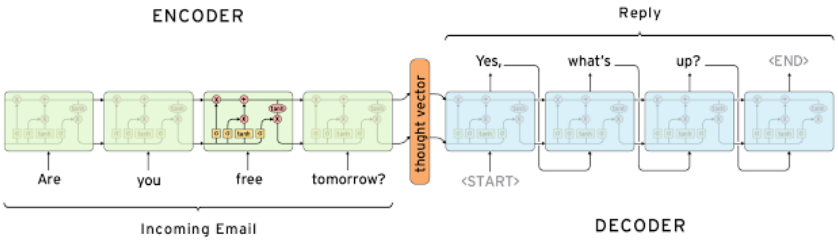


Diagram by Chris Olah

1. implementations using a **seq2seq encoder(bi directional lstm) decoder (with attention)**

this is a crucial implementation , as it is the cornerstone of any recent research for now i have collected different approaches that implement this concept

2. other implementation that i have found truly interesting is a combination of creating new sentences for summarization , with copying from source input , this method is called **pointer generator** , here is my modification in a google colab to the original implementation

3. other implementations that i am currently still researching , is the usage of **reinforcement learning** with deep learning

This series would be built to be easily understandable for any newbie like myself , as you might be the one that introduces the newest architecture to be used as the newest standard for text summarization , so lets begin !!

The following is a quick overview on the series , i hope you enjoy it

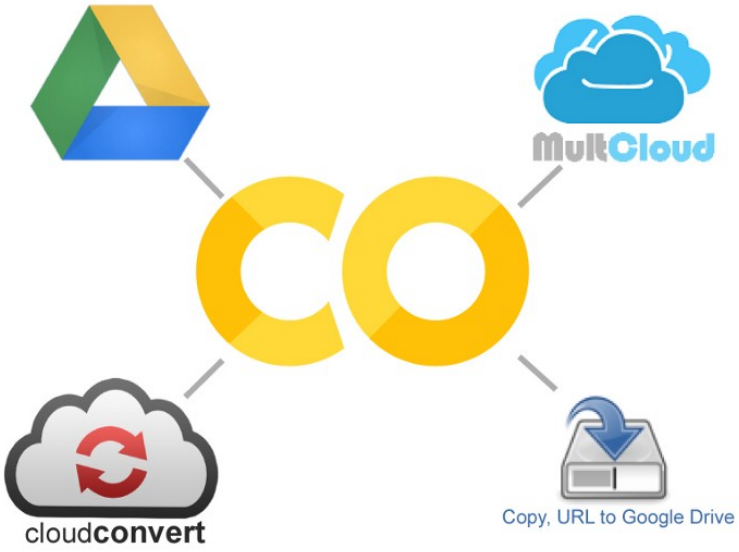
. . .

EazyMind free Ai-As-a-service for text summarization

I have added a text summarization model to a website eazymind so that you can actually try generating your own summaries yourself (**and see what you would be able to build**) , it can be called through simple api calls , and through a python package , so that text summarization can be easily integrated into your application without the hassle of setting up the tensorflow environment) , you can register for free , and enjoy using this api for free .

. . .

1 - Building your deep work online



we would be using google colab for our work , this would enable us to use their free gpu time to build our network , (this blog would give you even more insights on the free ecosystem for your deep project)

you have 2 main options to build your google colab

1. Build a new empty colab
2. Build from github , you can use this repo , which is a collection of different

you can find the details on how to do this in this blog

having your code on google colab enables you to

1. connect to google drive (put your datasets onto google drive)
2. free gpu time

you can find how to connect to google drive in this blog

2- Lets represent words

since our task is a nlp task we would need a way to represent words ,this have 2 main approaches that we would discuses ,

1. either providing the network with a representation for each word , this is called word embedding , which is simply representing a certain word by a an array of numbers , There are multiple already trained word embedding available online , one of them is **Glove vectors**
2. or letting the network understand the representations by itslef

3- The used Datasets

For this task we would use a dataset in form of news and their headers , the most popular is using the CNN/Daily Mail dataset , the news body is used as the input for our model , while the header would be used as the summary target output .

These datasets could be found easily online , we would use 2 main approaches for using these datasets

1. using the raw data itseld , and manually applying processing on them
2. using a prepossessed version for the data , it is currently used in the most recent researches

4 - Models used

Here i would briefly talk about the models that would be included if GOD wills in the coming series , hope you enjoy

A .Corner Stone model

to implement this task , researchers use a deep learning model that consists of 2 parts , an encoder , that understands the input , and represent it in an internal representation , and feed it to another part of the network which is the decoder ,

The main deep learning network that is used for these 2 parts in a LSTM , which stands for long short term memory , which is a modification on the rnn

in the encoder we mainly use a multi-layer bidirectional LSTM , while in the decoder we use an attention mechanism , more on this later

B .Pointer Generator

But researchers found 2 main problems with the above implementation , like discussed in this ACL 2017 paper *Get To The Point: Summarization with Pointer-Generator Networks* , they have a truly amazing blog you need to see

which is

1. **the inability of the network to copy Facts** (like names , and match scores) as it doesn't copy words , it generates them , so it sometimes incapable of generating facts correctly
2. **Repetition of words**

this research builds on these 2 main problems and try to fix them , I have modified their repo to work inside a jupyter notebook on google colab

- my modification
- their repo

C. Using Reinforcement learning with deep learning

I am still researching on this work , but it is a truly interesting research , it is about combing two fields together , it actually uses the pointer generator in its work (like in implementation B) , and uses the same prepossessed version of the data .

This is the research , it uses this repo for its code

they actually are trying to fix 2 main problems with the corner stone implementation which are

1. the decoder in the training , uses the (1 output from the encoder) , (2 the actual summary) , (3 and then uses its current output for the next action) , while in testing it doesn't have a ground truth , as we it is actually needed to be generated , so it only uses (1 output from the encoder) (2 and then uses its current output for the next action) , this causes an **Exposure Problem**
2. the training of the network relies on a metric for measuring the loss , which is different from the metric used in testing , as the metric used in training is the cross entropy loss , while the metric for the testing (like discussed below) is non-differentiable measures such as BLEU and ROUGE

I am currently working on implementing this approach in a jupyter notebook , so if GOD wills it , you would see more updates concerning this in the near future .

4—Summary Evaluation

to evaluate a summary , we use a non-differentiable measures such as BLEU and ROUGE , they simply try to find the common words between the input and the output , the more the better , most of the above approches score from 32 to 38 rouge scores

I hope you enjoyed this quick overview on the series , my main focus in these blogs is to present the topic of text summarization in easy and practical way , providing you with an actual code that is runnable on any computer , without the need to have a powerful GPU , and to connect you to the latest researches about this topic , please sow your support by clapping to this blog , and don't forget to check out the code of these blogs

In the coming blogs if GOD wills it , i would go through the details to build the corner stone implementation , that actually all the modern researches are based apon it , we will use word embedding approach , and we would use the raw data , and manually apply preprocessing

While in later blogs if GOD wills it , we would go through modern approaches like how you would be able to create a pointer generator model , to fix the problems mentioned above , and using reinforcement learning with deep learning .

Text Summarization made easy(2) , Text Representation

amr zaki Follow
Jan 27 · 12 min read

this story is a continuation to the series on how to easily build an abstractive text summarizer , (check out github repo for this series) , today we would go through how you would be able to build **a summarizer able to understand words** , so we would through representing words to our summarizer

my goal in this series to present the latest novel ways of abstractive text summarization in a simple way , (you can check my overview blog) from

1. corner stone method of using seq2seq models with attention
2. to using pointer generator
3. to using reinforcement learning with deep learning

we would use google colab , so you won't have to use a powerful computer , nor would you have to download data to your device , as we would connect google drive to google colab to have a fully integrated deep learning experience (you can check my overview on working on free deep learning ecosystem platforms)

All code can be found online through my github repo

> *This tutorial has been based over the work of https://github.com/dongjun-Lee/text-summarization-tensorflow , they have truly made great work on simplifying the needed work to apply summarization using tensorflow, I have built over their code , to convert it to a python notebook to work on google colab , I truly admire their work*

so lets begin !!

. . .

EazyMind free Ai-As-a-service for text summarization

I have added a text summarization model to a website eazymind so that you can actually try generating your own summaries yourself (and see what you would be able to build) , it can be called through simple api calls , and through a python package , so that text summarization can be easily integrated into your application without the hassle of setting up the tensorflow environment) , you can register for free , and enjoy using this api for free .

. . .

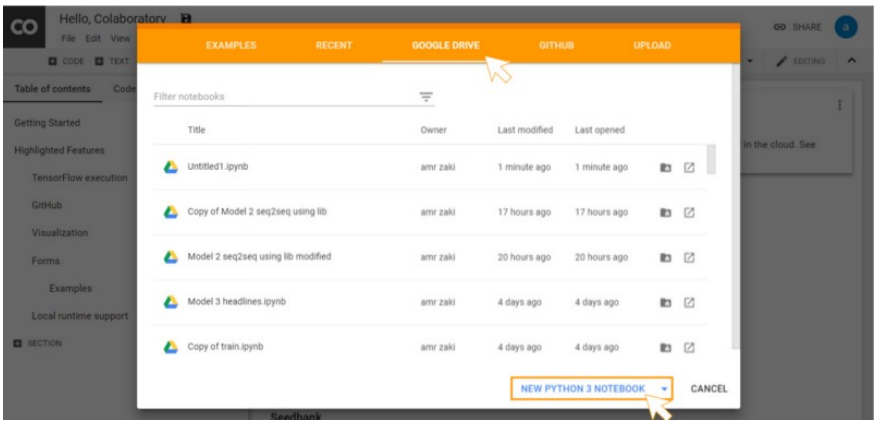
1- Setup

1-A To begin we first create a google colab notebook

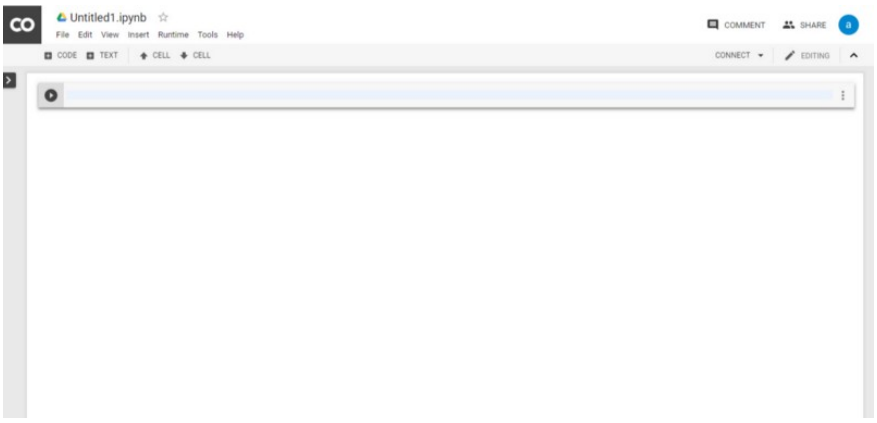
1- go to https://colab.research.google.com

2- select Google Drive Tab (to save your new google colab to google drive)

3- select **New Python 3 Notebook** (you can also select python 2 notebook)

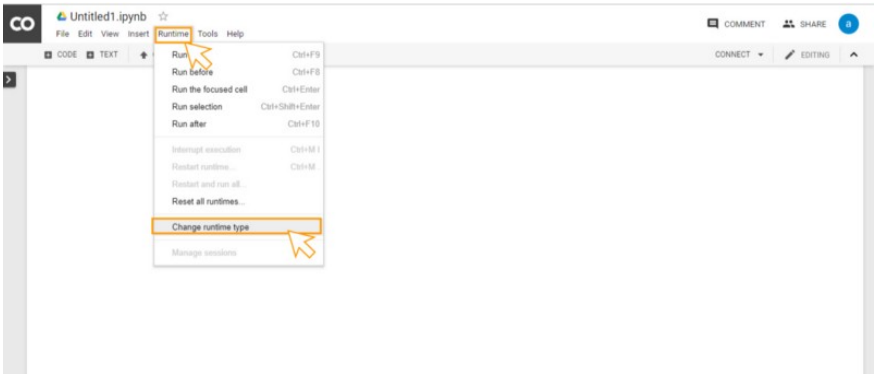


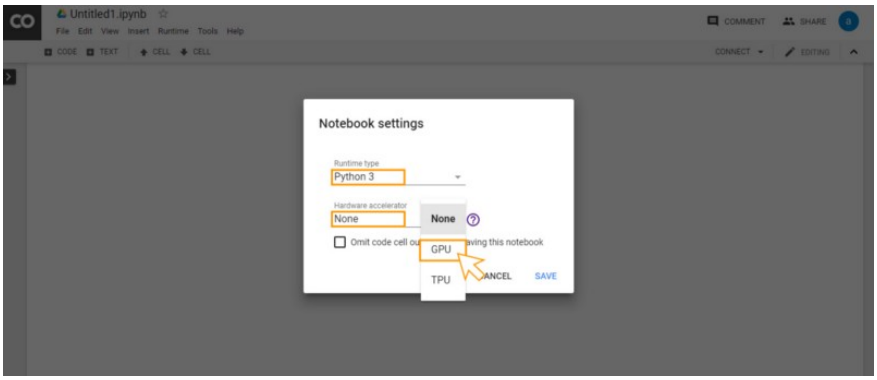
a blank notebook would be created to your google drive , it would look like this



You can change the runtime of your notebook from selecting the runtime button in the top menu , to

1. change which python version you are using
2. choose a hardware accelerator from (GPU , TPU)

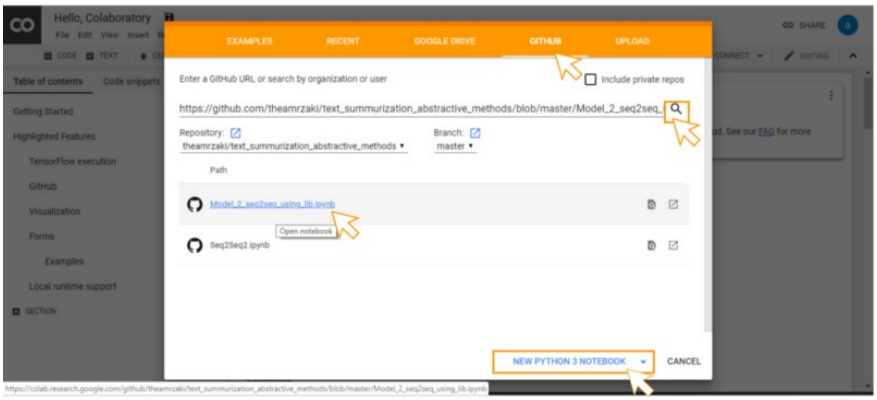




1-A-A or you can clone the code directly from my github repo

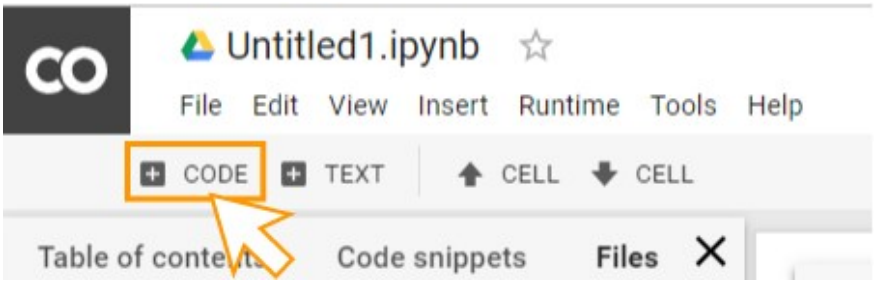
1. go to https://colab.research.google.com , but this time we would select github tab

2. then we just paste the this link , and click upload



1-B Now after we are have created our google colab , lets connect to google drive

in the newly created notebook , add a new code cell



then paste this code in it

```
#https://stackoverflow.com/questions/47744131/colaboratory-can-i-access-to-my-google-drive-folder-and-file
```

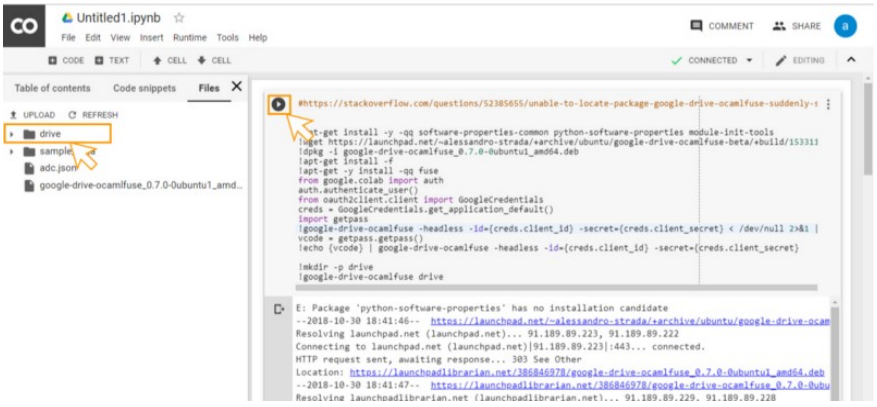
```
!apt-get install -y -qq software-properties-common python-software-properties module-init-tools
!add-apt-repository -y ppa:alessandro-strada/ppa 2>&1 > /dev/null
!apt-get update -qq 2>&1 > /dev/null
!apt-get -y install -qq google-drive-ocamlfuse fuse
from google.colab import auth
auth.authenticate_user()
from oauth2client.client import GoogleCredentials
creds = GoogleCredentials.get_application_default()
import getpass
!google-drive-ocamlfuse -headless -id={creds.client_id} -secret={creds.client_secret} < /dev/null 2>&1 | grep URL
vcode = getpass.getpass()
!echo {vcode} | google-drive-ocamlfuse -headless -id={creds.client_id} -secret={creds.client_secret}
```

```
!mkdir -p drive
!google-drive-ocamlfuse drive
```

this would connect to your drive , and create a folder that your notebook can access your google drive from

It would ask you for access to your drive , just click on the link , and copy the access token , it would ask this twice

after writing this code , you run the code by clicking on the cell (shift enter) or by clicking the play button on the top of your code cell



then you can simply access any file by its path in form of

```
path = "drive/test.txt"
```

1-C Now Lets get the data that we would work on

our data set that we would work on is in form of **news** and their **headlines .**

The input would be **news content** and the output needed would be its summary or in this case would be the **headline**

There are 2 popular dataset for this task

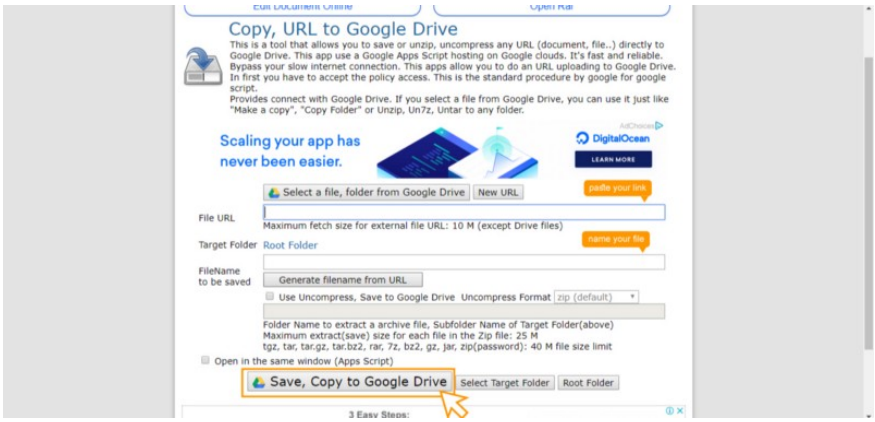
1. Amazon Product Reviews
2. CNN /Daily news dataset **(which we would use in our case)**

you **don't have to download** the data , you can just **copy it to your google drive** , it would just take some seconds not more.

Here is the Link for the folder containing the data .

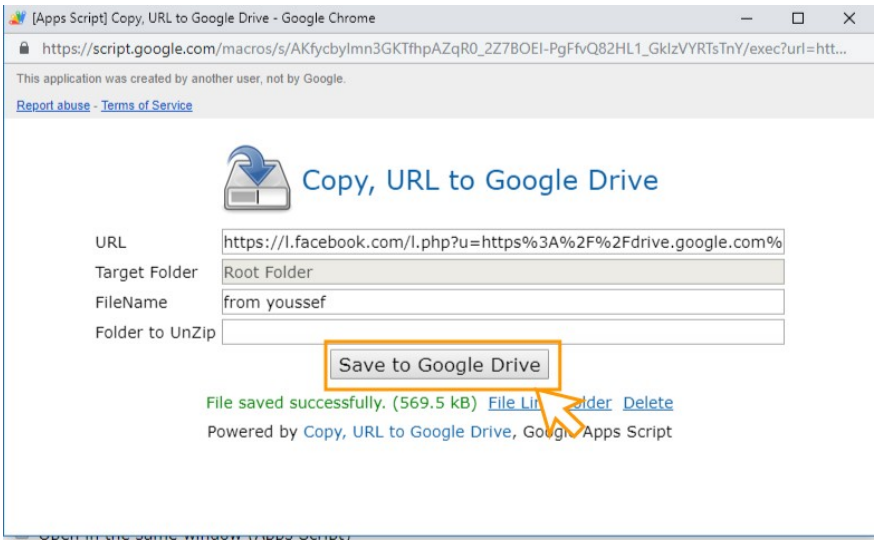
Here we would use Copy, URL to Google Drive , which enables you to easily copy files between different google drives

first you would paste the above Link



paste your link , name it , then save to google drive

then you simply click on Save,Copy to Google Drive (after autentication your google drive)



after authenticating , you just click save to google drive

Now after setup process , we can start our work , so lets Begin !!

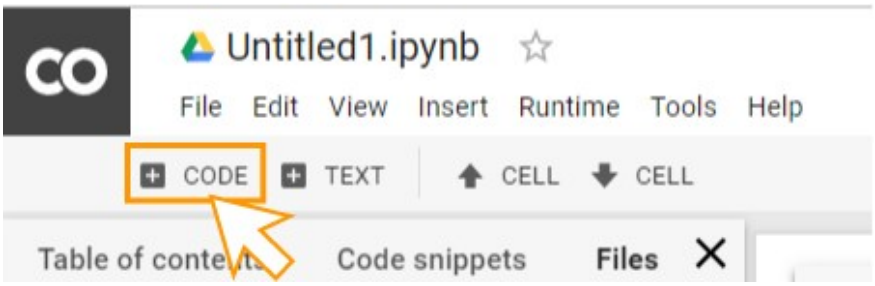
. . .

2- Dependencies and paths

2-a First Lets install needed dependencies

in google colab you are able to install using pip , by simply !pip,

in every code section you simply click on



and then just start writing your code

```
!pip install gensim
!pip install wget

import nltk
nltk.download('punkt')
```

2-b then Lets set needed Dependencies

```
from nltk.tokenize import word_tokenize
import re
import collections
import pickle
import numpy as np
from gensim.models.keyedvectors import KeyedVectors
from gensim.test.utils import get_tmpfile
from gensim.scripts.glove2word2vec import glove2word2vec
```

2-c Then lets define where the data can be found

```
#default path for the folder inside google drive
default_path = "drive/Colab Notebooks/Model 2/"
```

```
#path for training text (article)
train_article_path = default_path +
"sumdata/train/train.article.txt"


#path for training text output (headline)
train_title_path   = default_path +
"sumdata/train/train.title.txt"
```

```
#path for validation text (article)
valid_article_path = default_path +
```

```
"sumdata/train/valid.article.filter.txt"


#path for validation text output(headline)
valid_title_path    = default_path +
"sumdata/train/valid.title.filter.txt"
```

. . .

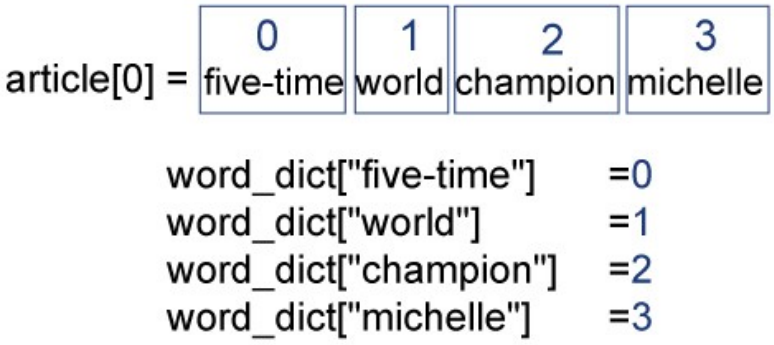
3- Building A Dictionary

for the text summarization to work , you must represent your words in a dictionary format

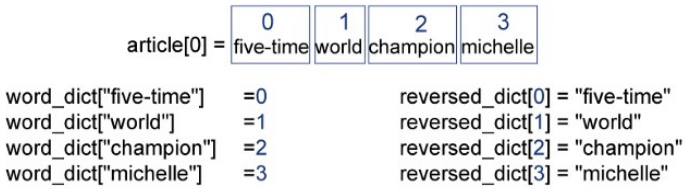
assume we have an article like

> *five-time world champion michelle kwan withdrew from the # us figure skating championships on wednesday , but will petition us skating officials for the chance to compete at the # turin olympics #*

each word would have a representation in a dict



and we would also need the reverse operation also , like



to apply this we would need some helper functions , like

3-A Simple cleaning data function

the goal of this function would be a simple cleaning of data , just by replacing some unneeded characters with #

```
def clean_str(sentence):
    sentence = re.sub("[#.]+", "#", sentence)
    return sentence
```

this substitution of characters is rather simple , you can of course add multiple substitution steps

3-B Function that actually return text

and apply the above cleaning function

```
def get_text_list(data_path, toy):
    with open (data_path, "r", encoding="utf-8") as f:
        if not toy:
            return [clean_str(x.strip()) for x in
f.readlines()][:200000]
        else:
            return [clean_str(x.strip()) for x in
f.readlines()][:50]
```

this function would be called for mltiple cases

1. if you need to load training data
2. or test data
3. or if you just need a sample of any of the above by simply setting **toy = True**

3-C Now lets Build the function that would actually create the needed dictionary

here you would see that we add 4 built-in words , these are essential for the seq2seq algorithim , they are

1. <padding> this would be used to make the sequences of same length
2. <unk> this would be used to identify that the word is not found inside the dict
3. <s> this would be used to identify the beingin of a sentence
4. </s> this would be used to identify the end of a sentence

> *copy the code from github , as here the padding is incorrect due to the editor of medium*

```
def build_dict(step, toy=False):
    if step == "train":
  #First lets load the training data
        train_article_list =
get_text_list(train_article_path, toy)
        train_title_list = get_text_list(train_title_path,
toy)
```

```
#then lets collect all words from the training data
  #by simply tokenizing each text sample to its words
  #here we would use the built-in function imported from
nltk toolkit
  #which simply return a list of words from a sentence
  words = list()
        for sentence in train_article_list +
train_title_list:
            for word in word_tokenize(sentence):
                words.append(word)
```

```
#we would only select the most common words
  word_counter = collections.Counter(words).most_common()
  #first lets set the 4 built-in words
    word_dict = dict()
    word_dict["<padding>"] = 0
    word_dict["<unk>"] = 1
    word_dict["<s>"] = 2
    word_dict["</s>"] = 3
```

```
#then lets build our dict , by simply looping over word_co
        for word, _ in word_counter:
            word_dict[word] = len(word_dict)


#then lets save this to a pickle
  with open(default_path + "word_dict.pickle", "wb") as f:
     pickle.dump(word_dict, f)
```

```
#all of the above was for the training step
#when you are in the validation you can simply load the
```

```
pickles that
#you have just saved


elif step == "valid":
    with open(default_path + "word_dict.pickle", "rb") as f:
     word_dict = pickle.load(f)


#for both of the 2 cases (training , or validation)
#we would create a reversed dict


  reversed_dict = dict(zip(word_dict.values(),
word_dict.keys()))


#then we would simply for the 2 cases (training , or
validation)
#define a max len for article and for the summary


  article_max_len = 50
   summary_max_len = 15
return word_dict, reversed_dict, article_max_len,
summary_max_len
```

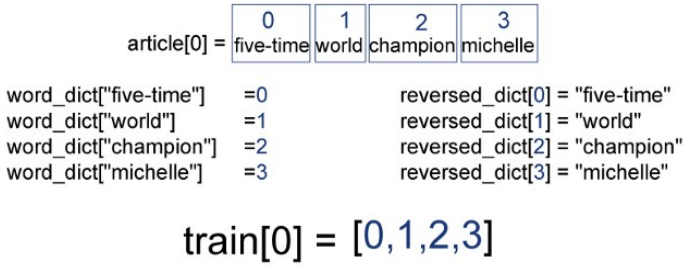
4- Now Lets Build Our Dataset

After building the dict for our data , we would begin to build the actual dataset that would be used in our algorithm

Using the above example of an article ,

> *five-time world champion michelle kwan withdrew from the # us figure skating championships on wednesday , but will petition us skating officials for the chance to compete at the # turin olympics #*

the algorthim would need this to be represented in



which is simply getting the collection of word dict for the words in the given sentence

same would occur on the test data

```
def build_dataset(step, word_dict, article_max_len,
summary_max_len, toy=False):
#---case of train
#---we would load both (article , headline) for training
    if step == "train":
        article_list = get_text_list(train_article_path,
toy)
        title_list = get_text_list(train_title_path, toy)
#---case of valid
#---we only load articles
    elif step == "valid":
        article_list = get_text_list(valid_article_path,
toy)
#---if step is neither (train nor valid) raise error
    else:
        raise NotImplementedError
#---(for each aricle) get list of words
#--- so now x (article) contains list of words
    x = [word_tokenize(d) for d in article_list]
```

```
#---(for each aricle) get index of word from word_dict for
each article
#---if not found , use "<unk>" tokken
#---so now we have our train dataset
    x = [[word_dict.get(w, word_dict["<unk>"]) for w in d]
for d in x]
```

```
#---(for each aricle) limit x to article_max_len
    x = [d[:article_max_len] for d in x]
```

```
#---(for each aricle) if x was less than article_max_len
#--- pad the x by using "<padding>" tokken
    x = [d + (article_max_len - len(d)) * [word_dict["
<padding>"]] for d in x]



    if step == "valid":
        return x
    else:
#-------if step = "train"
#-------we must do the same steps on headline
#-------but here we don't use the concept of padding
        y = [word_tokenize(d) for d in title_list]
        y = [[word_dict.get(w, word_dict["<unk>"]) for w in
d] for d in y]
        y = [d[:(summary_max_len - 1)] for d in y]
        return x, y
```

so lets simply call both (build dict and build dataset)

```
print("Building dictionary...")
word_dict, reversed_dict, article_max_len, summary_max_len =
build_dict("train", False)
```

```
print("Loading training dataset...")
train_x, train_y = build_dataset("train", word_dict,
article_max_len, summary_max_len, False)
```

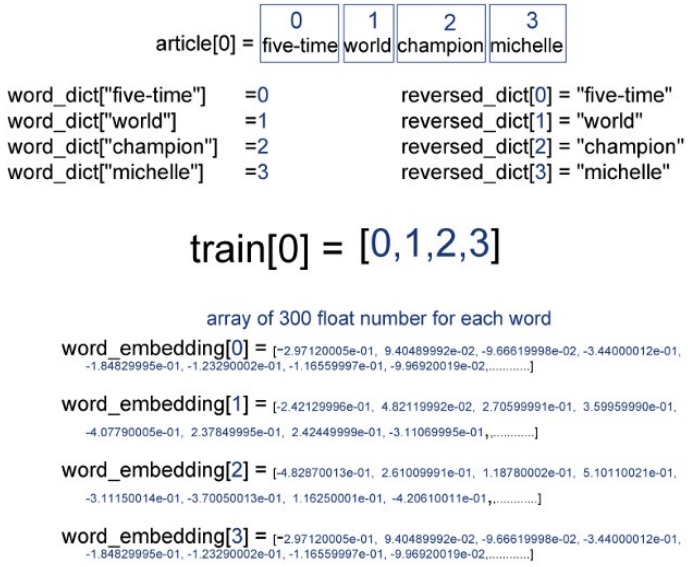
. . .

5- Word Embeddings

But we can't yet feed the our neural network with a list containing the indexes of words , as it would understand them .

We need to represent the word itself in a format that our neural net would understand , and here comes the concept of word embeddings

it is a simple concept , that replaces each word in your dict with a list of numbers , (in our case we would model each word with a 300 float number list)



There are already trained models that have been trained over millions of text to correctly model the words , once you are able to correctly model the words , your neural net would be able to truly understand the text within the article .

A very well known test to identify how well the algorithm understand text after using word embeddings , is applying word similarity on a given word

```
word_vectors.most_similar('man')
```

```
/usr/local/lib/python3.6/dist-packages/
  if np.issubdtype(vec.dtype, np.int):
[('woman', 0.6998662948608398),
 ('person', 0.6443442106246948),
 ('boy', 0.620827853679657),
 ('he', 0.5926738977432251),
 ('men', 0.5819568634033203),
 ('himself', 0.5810033082962036),
 ('one', 0.5779520869255066),
 ('another', 0.5721587538719177),
 ('who', 0.5703631639480591),
 ('him', 0.5670831203460693)]
```

```
word_vectors.most_similar('king')
```

```
/usr/local/lib/python3.6/dist-package
  if np.issubdtype(vec.dtype, np.int)
[('queen', 0.6336469054222107),
 ('prince', 0.619662344455719),
 ('monarch', 0.5899620652198792),
 ('kingdom', 0.5791267156600952),
 ('throne', 0.5606487989425659),
 ('ii', 0.5562329888343811),
 ('iii', 0.5503199100494385),
 ('crown', 0.5224862694740295),
 ('reign', 0.521735429763794),
 ('kings', 0.5066401362419128)]
```

as you can see , the output tells us that the model would now be capable of understanding the relations between words , which is an

extremely important factor in the success of out neural net

5-A lets get the trained model for our work

there is a very well known pretrained model called Glove pre-trained vectors provided by stanford , you can download it from https://nlp.stanford.edu/projects/glove/

or you can simply copy it from my google drive like i have explained before , here is the link for the glove vectors in a pickle format

5-B Build a function to get an array of word embeddings

```
def get_init_embedding(reversed_dict, embedding_size):

print("Loading Glove vectors...")

#---Load glove model which is in a pickle format
    with open( default_path + "glove/model_glove_300.pkl",
'rb') as handle:
        word_vectors = pickle.load(handle)

#---Loop through all words within the reversed_dict
    used_words = 0
    word_vec_list = list()
    for _, word in sorted(reversed_dict.items()):
        try:
#-----------if the word i found in the dict ,
#-----------save its value
            word_vec = word_vectors.word_vec(word)
            used_words += 1
        except KeyError:
#-----------else , generate an array of zeros
#-----------of length = embedding_size
#-----------which in this case would be 300
#-----------this is the case also for <padding> and <unk>
#-----------where <s>, </s> token would be zeros
#-----------like seen below
            word_vec = np.zeros([embedding_size],
dtype=np.float32) #to generate for <padding> and <unk>


#-------add it to the array
#-------remember that we are looping in sorted reversed_dict
#-------so the index of the element inside word_vec_list
#-------would be the same as  index of word
#-------no need of a dict , an array is sufficient
        word_vec_list.append(word_vec)

#---just print out the percentage of knwon words
    print("words found in glove percentage = " +
```

```
str((used_words/len(word_vec_list))*100) )

#----Assign random vector to <s>, </s> token
    word_vec_list[2] = np.random.normal(0, 1,
embedding_size)
    word_vec_list[3] = np.random.normal(0, 1,
embedding_size)


#----then return the array
return np.array(word_vec_list)
```

to call the function we simply call

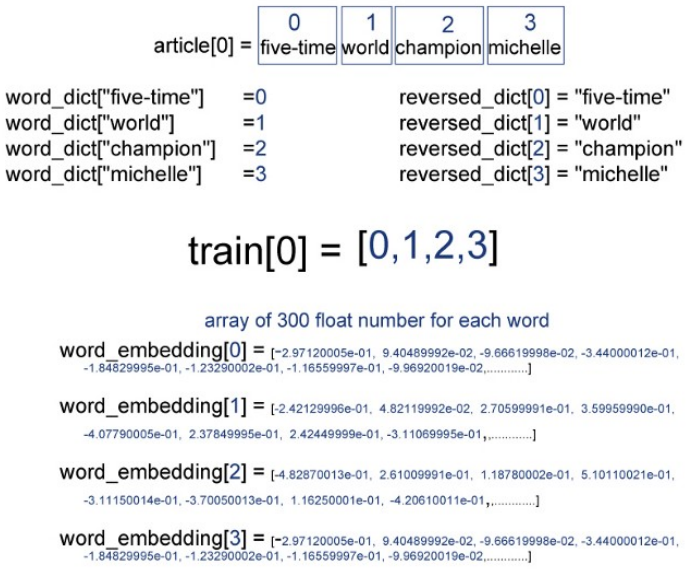
```
word_embedding = get_init_embedding(reversed_dict, 300)
```

. . .

To sum it all UP

so we can say that we have now correctly represented the text for our task of text summarization

so to sum it all up , we have build the code to



by simply calling

```
word_dict, reversed_dict, article_max_len, summary_max_len =
build_dict("train", False)
```

```
train_x, train_y = build_dataset("train", word_dict,
article_max_len, summary_max_len, False)
```

```
word_embedding = get_init_embedding(reversed_dict, 300)
```

the coming steps in the coming tutorial if GOD wills it , we would go through how to build the model itself , we would build a seq2seq encoder decoder model using LSTM , we would go through the very details of building such a model using tensorflow , this would be the corner stone for the next tutorials in the series , that would go through the latest approaches for this problem from

1. using pointer generator model
2. using reinforcement learning with deep learning

don't forget to clone the code for this tutorial from my repo

and you can take a look on the previous tutorial talking about an overview on text summarization

you can also check this blog talking about the eco system of a free deep learning platform

. . .

> *I truly hope you have enjoyed this tutorial , i am waiting for your feedback , and i am waiting for you in the next tutorial if GOD wills it*

. . .

Next Tutorials

- What is seq2seq for text summarization and why (tutorial 3)

(tutorial 3)What is seq2seq for text summarization and why

amr zaki Follow
Feb 16 · 8 min read

This tutorial is the third one from a series of tutorials that would help you build an abstractive text summarizer using tensorflow , today we would discuss the main building block for the text summarization task , begining from RNN why we use it and not just a normal neural network , till finally reaching seq2seq model

About Series

This is a series of tutorials that would help you build an abstractive text summarizer using tensorflow using multiple approaches , ***you don't need to download the data nor you need to run the code locally on your device*** , as **data** is found on **google drive** , (you can simply copy it to your google drive , learn more here) , and the **code** for this series is written in Jupyter notebooks to run on **google colab** can be found here

We have covered so far (code for this series can be found here)

0. Overview on the free ecosystem for deep learning

1. Overview on the text summarization task and the different techniques for the task
2. Data used and how it could be represented for our task

so lets get started

. . .

EazyMind free Ai-As-a-service for text summarization

I have added a text summarization model to a website eazymind so that you can actually try generating your own summaries yourself (and see what you would be able to build) , it can be called through simple api calls , and through a python package , so that text summarization can be easily integrated into your application without the hassle of setting up the tensorflow environment) , you can register for free , and enjoy using this api for free .

. . .

Quick Recap

our task is of text summarization , we call it abstractive as we teach the neural network to generate words not to merely copy words .

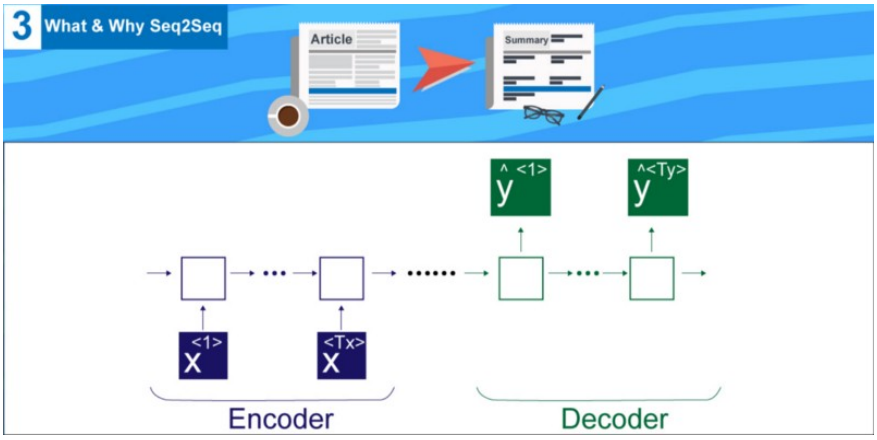
the data that would be used would be news and their headers , it can be found on my google drive , so you just copy it to your google drive without the need to download it (more on this)

We would represent the data using word embeddings , which is simply converting each word to a specific vector , we would create a dictionary for our words (more on this)

there are different approaches for this task , they are built over a corner stone concept , and they keep on developing and building up , they start from a network called seq2seq then they add up to be different networks that increase the overall accuracy , the code for these different approaches can be found here

. . .

Today we would discuss what is seq2seq and why it is used in the first place , so lets start !!

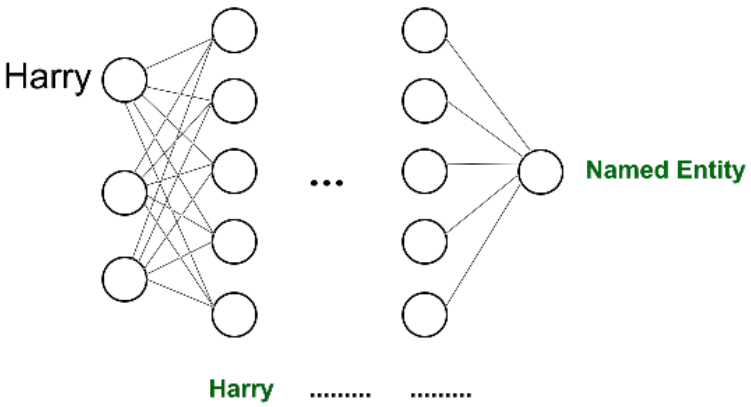


> *This tutorial has been based by the amazing work of **Andrew NG** , his course on RNN has been truly useful , i recommend you to see it*

. . .

1- Why we use complex network structure not a simple neural network

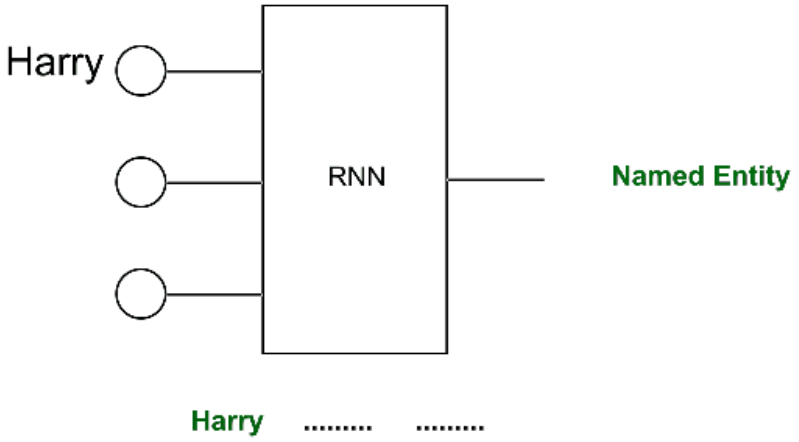
this is truly an important question to be asked , in natural language tasks , it is important for the network to understand the word itself , not to link the word to a specific location , this is what we call (**sharing features accross different parts of text**)



normal neural network inefficient for nlp

assume our task was identifying named entities within a text , as we can see in the previous gif , a normal neural network won't be able to identify the name **Harry** if it is found in different parts of the text

so this is why we would need a new network for this task , this network is called (Recurrent Neural Network) RNN



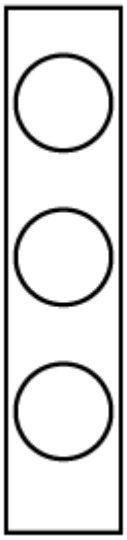
RNN for nlp

here using a RNN , the network was able to identify the name Harry if found in different parts of text .

RNN is the base of seq2seq , as we would see

2- What is RNN (Recurrent Neural Network)

Recurrent Neural Network is a type of neural network that **takes time into consideration** , each box (box with circles as seen in the gif)

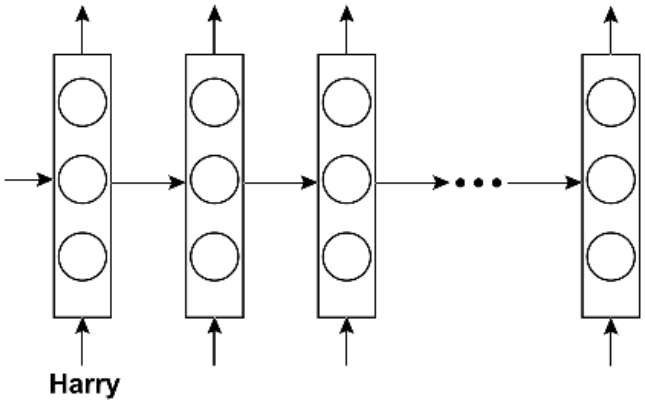


RNN network

is the actually our network , and we use it multiple times , each time , is a step in time , as each time step we would feed it with a word from our sentence , it also takes the output from the previous time step ,

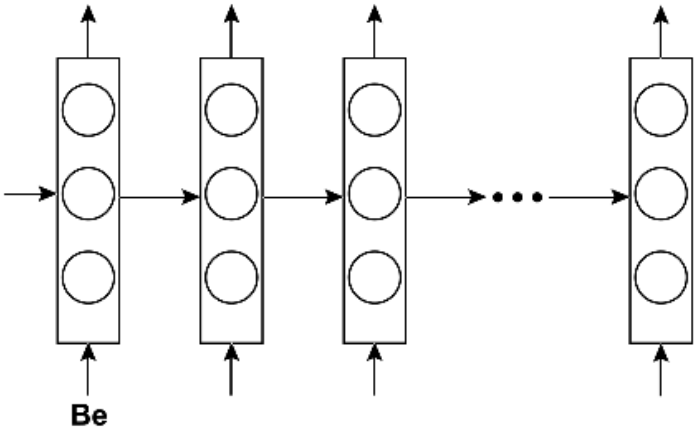
so to recap , RNN is

1. takes time into consideration (runs multiple times in time)
2. takes output from previous step



RNN ex 1

here as we see , it takes the input from previous steps

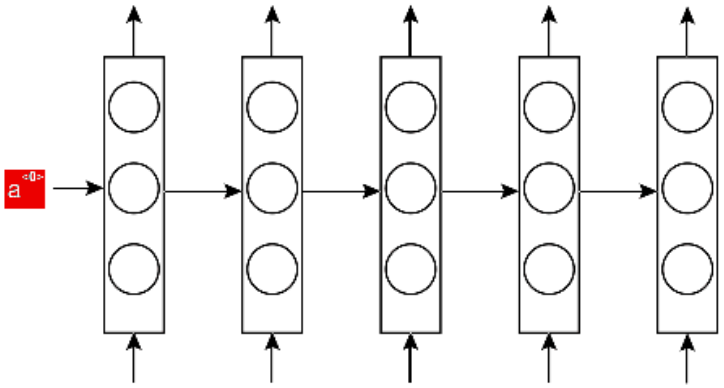


RNN ex 2

and it can understand named entity recognition independent to the location , which is our needed behavior

3- RNN Feed-forward steps

like any other neural network , we would need a feed-forward step

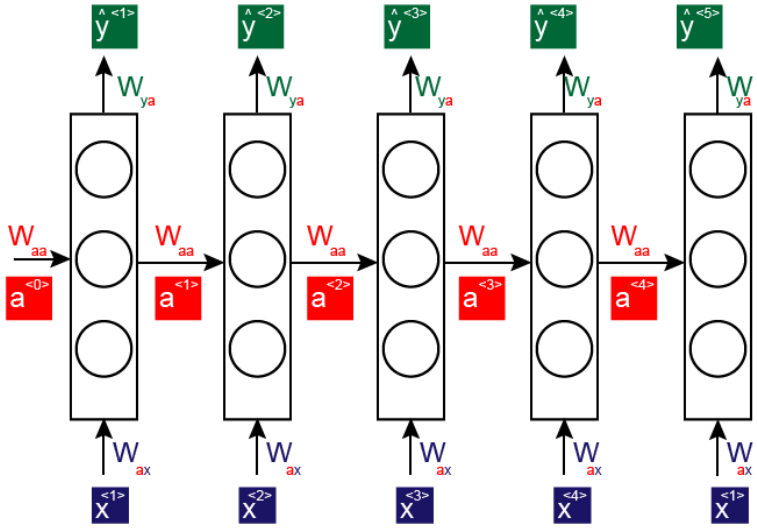


Here we would have

1. X vectors (blue vector) (inputs , which would be words from our sentence)
2. Y vectors (green vector)(outputs , would would be the words exported from each time step
3. A vectors (red vectors) (activaations from each time step)

there are also 3 types of weights

1. Wax vectors (blue) (that would be multiplied by input) , **same for all time steps**
2. Wya vectors (green) ,(that would be multiplied by output) ,**same for all time steps**
3. Waa vectors (red) (that would be multiplied by activations) , **same for all time steps**



the 2 main functions that govern our work are

$$a^{<1>} = g(W_{aa} a^{<0>} + W_{ax} x^{<1>} + b_a)$$

which calculates the next activation parameter using the previous activation parameter and previous input with a bias , here we use activation function g which is mostly either tanh or relu

$$\hat{y}^{<1>} = g(W_{ya} a^{<0>} + b_y)$$

a

the other function is for calculating the output from each time step , here we use the activation parameter , with the bias , with also using a g activation function either tanh or relu

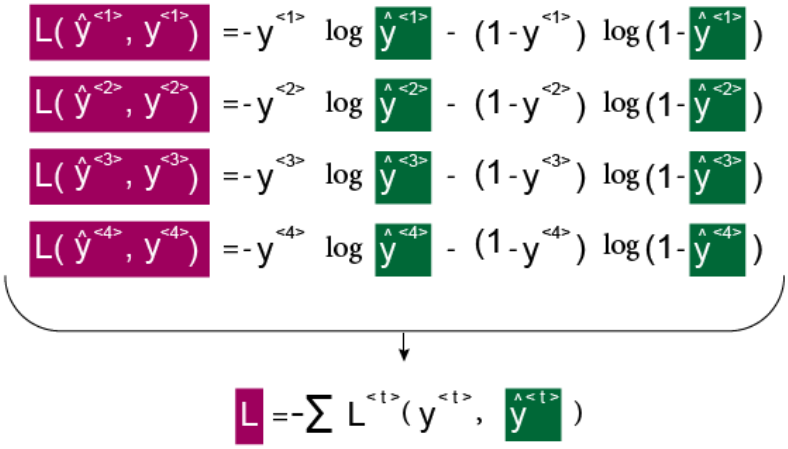
then we would need to calculate loss to be used for back propagation

the main used function is (training Loss)

$$L(\hat{y}^{<1>}, y^{<1>}) = -y^{<1>} \log \hat{y}^{<1>} - (1 - y^{<1>}) \log (1 - \hat{y}^{<1>})$$

here we use the **generated output** yhat with the **given output** y

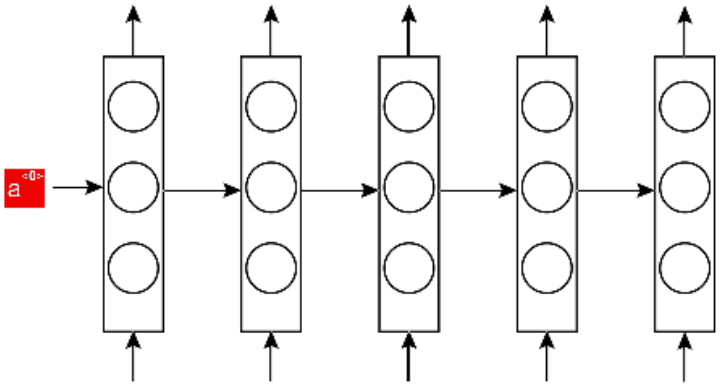
then we simply add them all to get the total loss



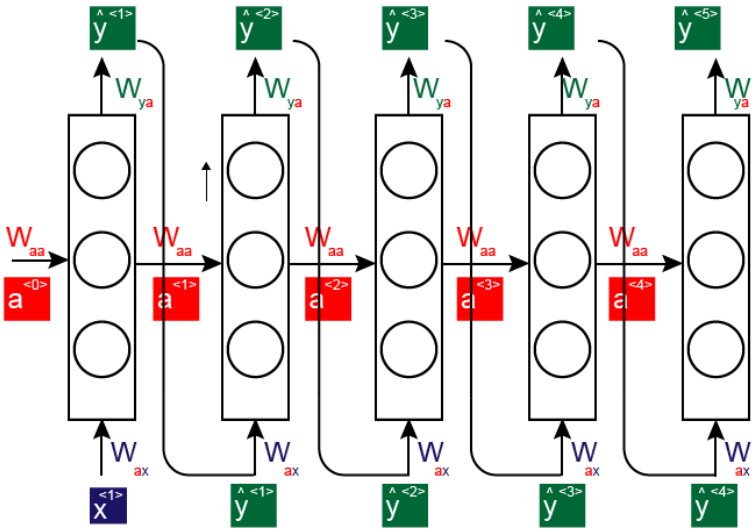
now after that we have talked about training stage , we need to talk about running our network

4- RNN Running stage

now after training our network , we would need to run it ,this stage is also called sampling (*here we would sample random words according to trained language model, for illustration of how rnn runs*)



as we see , the inputs from a time step is forwarded to the other time step till we reach the final output , we would need a tokken <eot> , end of text then we would stop our running.



Here we would calculate the cost of this run

the main function for this is

$$L(\hat{y}^{<1>}, y^{<1>}) = -y^{<1>} \log \hat{y}^{<1>}$$

were we would use both the generated output and the original output

then we simply add them up to get the total output

$$L(\hat{y}^{<1>}, y^{<1>}) = -y^{<1>} \log \hat{y}^{<1>}$$

$$L(\hat{y}^{<1>}, y^{<1>}) = -y^{<1>} \log \hat{y}^{<1>}$$
$$L(\hat{y}^{<2>}, y^{<2>}) = -y^{<2>} \log \hat{y}^{<2>}$$
$$L(\hat{y}^{<3>}, y^{<3>}) = -y^{<3>} \log \hat{y}^{<3>}$$
$$L(\hat{y}^{<4>}, y^{<4>}) = -y^{<4>} \log \hat{y}^{<4>}$$

$$\downarrow$$

$$L = -\sum y^{<t>} \log \hat{y}^{<t>}$$

In All of the above we only talked about one type of RNN , which is many-to-many architectures with same lengths for both input and output , this won't be our case

As for text summarization , we need to have the ability to have different lengths for input and for output , for this we would finally talk about **Seq2Seq**

5- We Finally Reached Seq2Seq

we need a special network that takes input of length (Tx) , and generates another output of another different length (Ty) , this

architecture is called Encoder Decoder .

Both Encoder Decoder here are RNN network , but encoder uses input , and generates an output state that is then used as input to decoder stage

This architecture is used for both tasks

1. Machine translation
2. Text Summarization

. . .

Recap

Today we have discussed

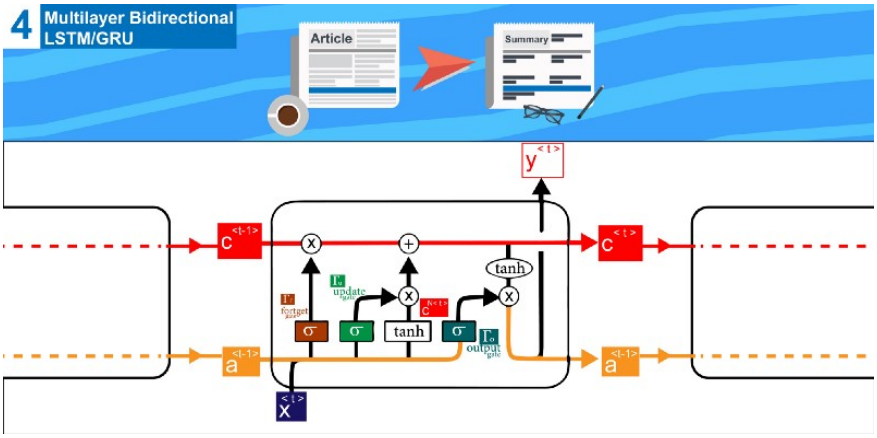
1. why we use RNN for text summarization and not a simple neural network ,
2. what is RNN (feed forward , running)
3. Then we finally reached seq2seq architecture using encoder decoder

. . .

But we can even have a better architecture for text summarization , we can add modifications to RNN to increase its efficiency , and to solve some of its problems , we can also add attention mechanism which proved extremely beneficial for our task , we could also use beam search

Multilayer Bidirectional LSTM/GRU for text summarization made easy (tutorial 4)

amr zaki Follow
Mar 31 · 9 min read



This tutorial is the forth one from a series of tutorials that would help you build an abstractive text summarizer using tensorflow , today we would discuss some useful modification to the core RNN seq2seq model we have covered in the last tutorial

These Modifications are

1. RNN modifications (GRU & LSTM)
2. Bidirectional networks
3. Multilayer networks

. . .

About Series

This is a series of tutorials that would help you build an abstractive text summarizer using tensorflow using multiple approaches , ***you don't need to download the data nor do you need to run the code locally on your device*** , as **data** is found on **google drive** , (you can simply copy it to your google drive , learn more here) , and the **code** for this series is written in Jupyter notebooks to run on **google colab** can be found here

We have covered so far (code for this series can be found here)

0. Overview on the free ecosystem for deep learning (how to use google colab with google drive)

1. Overview on the text summarization task and the different techniques for the task
2. Data used and how it could be represented for our task
3. What is seq2seq for text summarization and why

so lets get started

. . .

EazyMind free Ai-As-a-service for text summarization

I have added a text summarization model to a website eazymind so that you can actually try generating your own summaries yourself (and see what you would be able to build) , it can be called through simple api calls , and through a python package , so that text summarization can be easily integrated into your application without the hassle of setting up the tensorflow environment) , you can register for free , and enjoy using this api for free .

. . .

Quick Recap

Our task is of text summarization, we call it abstractive as we teach the neural network to generate words not just copy words .

The data that would be used would be news and their headers , it can be found on my google drive, so you just copy it to your google drive without the need to download it (more on this)

We would represent the data using word embeddings , which is simply converting each word to a specific vector , we would create a dictionary for our words (more on this)

There are different approaches for this task , they are built over a corner stone concept , and they keep on developing and building up , they start by working on a type of network called RNN , which is arranged in an Encoder/Decoder architecture called seq2seq (more on this), the code for these different approaches can be found here

> This tutorial has been based by the amazing work of **Andrew NG** , his course on RNN has been truly useful, i recommend you to see it

. . .

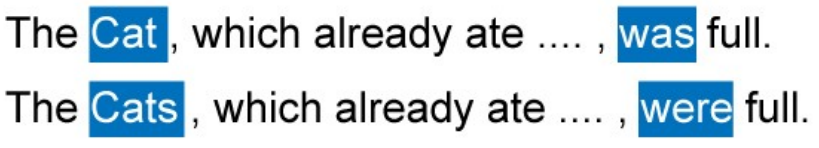
Today we would go through some modifications made to the core component of the encoder/decoder model , these modifications occur on the RNN block itself , to increase its efficiency in the whole model.

1. RNN modifications (LSTM & GRU)

There are 2 main problems with the RNN unit

1. **Exploding Gradients :** Occurs with deep networks (i.e: networks *with many layers* like in our case) , when we apply back propagation, the gradients would get too large . Actually this error can be solved rather easy , using the concept of **gradient clipping** , which is simply setting a specific threshold , that when the gradients exceed it , we would clip it to a certain value .
2. **Vanishing Gradients :** This proves a much harder problem to solve , this also occurs *due to large number of layers* , but this comes from the inability of the normal RNN unit to remember old values that appeared early in the sequence

> *this is quite important when dealing with a nlp problem , as some words depends on words that appeared very early in the sentence like*



Here the word cat/cats which appeared early in the sentence would directly affect choosing either was/were later in the sentence.

to solve this problem we would need a new RNN architecture , here we would discuss 2 main approaches :

1. GRU (Gated Recurrent Unit)
2. LSTM (Long Short term Memory)

1.A) GRU (Gated Recurrent Unit)

Both GRU & LSTM solves the problem of vanishing gradients that normal RNN unit suffers from , they do it by implementing a memory cell within their network , this enables them to store data from early within the sequence to be used later within the sequence.

Here we would talk about GRU (gated recurrent unit) , we begin with the activation equation of RNN (more on this)

$$a^{<t>} = g(W\ a^{<t-1>} + W\ x^{<t>} + b\ )$$

then we would apply some simple modifications to it

$$a^{<t>} = g(W\ a^{<t-1>} + W\ x^{<t>} + b\ )$$

till we finally have

$$c^{N<t>} = \tanh(W_c\ [\ c^{<t-1>}, x^{<t>}\ ] + b_c\ )$$

c here denotes for the memory cell , here it would be the output of the GRU cell .

The **N** sub letter denotes that it is the newly proposed c value (we would use it latter to generate the real c output of the GRU .

so here the new proposed output c (candidate), would depend on the old output c (old candidate) , and the current input at that time

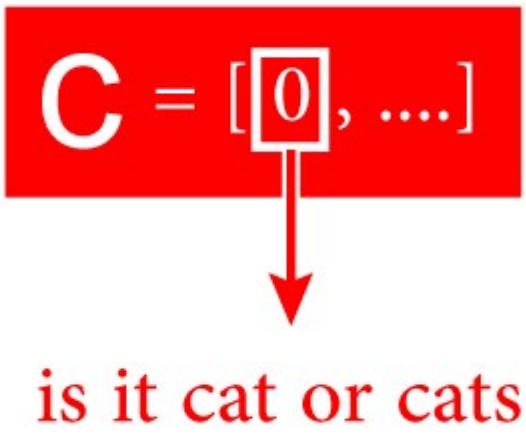
To remember the value of C (candidate), we use another parameter called F (gate update) , this would control whether we would update the value of c or not

$$\Gamma_u = \sigma(W_u [c^{<t-1>} , x^{<t>}] + b_u)$$

here we would use a sigmoid function , we would take into consideration **the old c** , and the **current input X**

so to update the value of C we would use

$$c^{<t>} = \Gamma_u * c^{N<t>} + (1- \Gamma_u) * c^{<t-1>}$$



lets assume that C is a vector , that its first element would **remember important features within the sentence ,** here we would assume that this feature is whether the word is cat or cats

so at first the c vector is empty , till we see the word **cat** , then F would be set to 1 to remember that it is a singular word , and it **would keep** its value until it is used later in the sentence (to generate 'was' not 'were')

$\Gamma_u = 1$

C= [0 ,..]

The

there is just another modification that is needed to build our full GRU unit , it occurs on the function needed to create the new candidate C .

Here we would have a learnable (**Fr**) parameter to learn the relevance between **C new** and **C old**

$$c^{N<t>} = \tanh(W_c\,[\,c^{<t-1>}, x^{<t>}\,] + b_c)$$

$$\downarrow$$

$$c^{N<t>} = \tanh(W_c\,[\,\Gamma_r * c^{<t-1>}, x^{<t>}\,] + b_c)$$

$$\Gamma_r = \sigma(W_r\,[\,c^{<t-1>}, x^{<t>}\,] + b_r)$$

so to sum it all up we have 4 main equations that govern GRU

$$c^{N<t>} = \tanh(W_c\,[\,\Gamma_r * c^{<t-1>}, x^{<t>}\,] + b_c)$$

$$\Gamma_r = \sigma(W_r\,[\,c^{<t-1>}, x^{<t>}\,] + b_r)$$

$$c^{<t>} = \Gamma_u * c^{N<t>} + (1-\Gamma_u) * c^{<t-1>}$$

$$\Gamma_u = \sigma(W_u\,[\,c^{<t-1>}, x^{<t>}\,] + b_u)$$

. . .

1.B) LSTM (Long Short Term Memory)

LSTM is another modification to RNN , it is also build using the same concept of memory , to remember long sequences of data , it was built proposed before GRU , so GRU is actually a simplification to LSTM

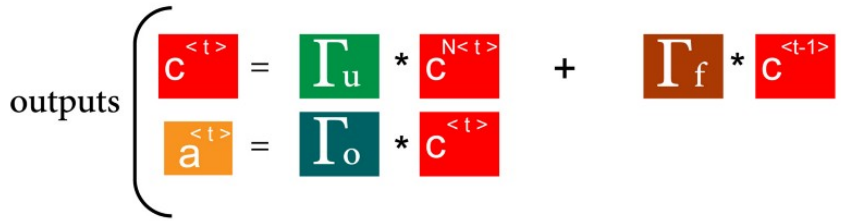
Here in LSTM ,

1. we use activation values , not just C (candidate values) ,
2. we also have 2 outputs from the cell , a new activation , and a new candidate value

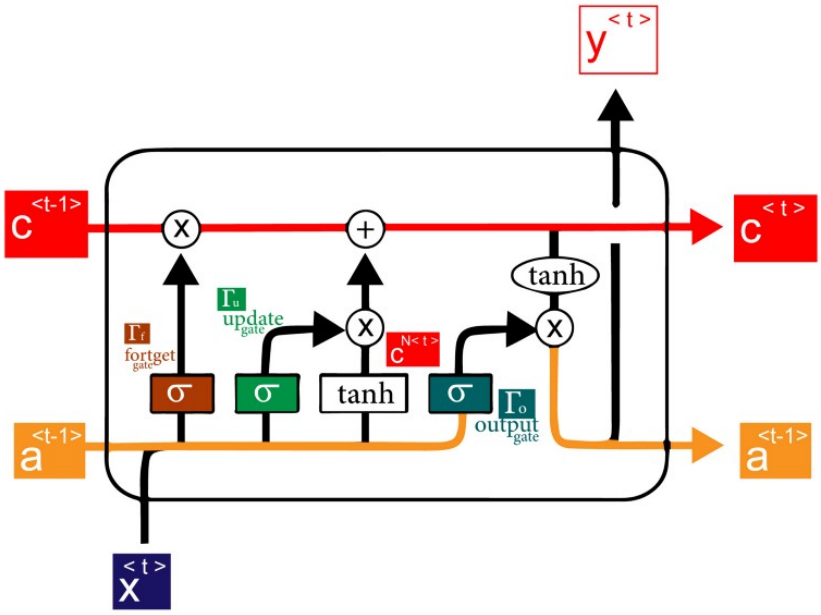
so to calculate the new candidate

here in LSTM we control the memory cell through 3 different gates

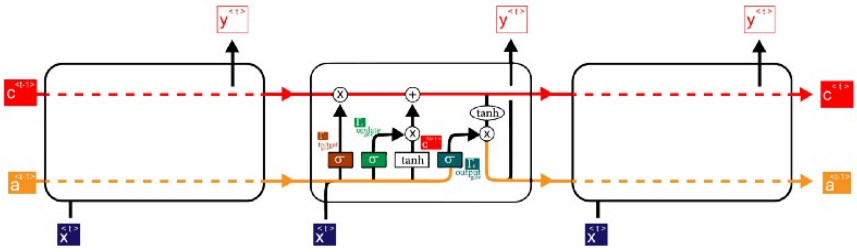
as we said before we have 2 outputs from LSTM , the new candidate and a new activation , in them we would use the previous gates



To combine all of these together



we could also output y prediction from LSTM (by passing them to softmax)



when we connect multiple LSTMs together , we can see that if the network correctly learned the gates parameters , we could pass the candidate values (red values) from early from the sequence to the very end of the sequence , so we can model long dependencies with high accuracy

2. Bidirectional networks

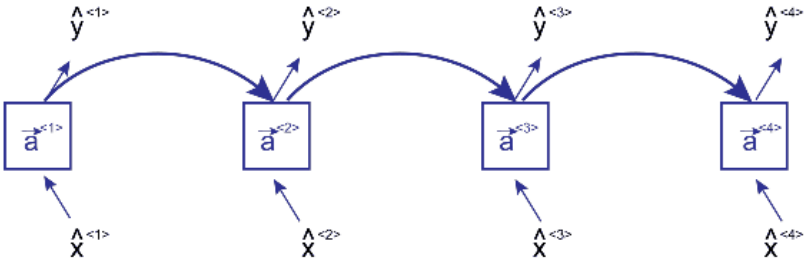
this is a modification made on the normal RNN network to make it able to adjust to an important need in nlp problems ,

> *as in nlp , sometimes to understand a word we need not just to the previous word , but also to the coming word , like in this example*

He said , "Teddy bears are on sale!"

Here to differ between the 2 different meanings of the word **teddy** (one time it is part of a person name , while the other is part of the word bear) we would need to look for the coming word , so this is the reason why we need to apply bidirectional networks

Bidirectional networks is a general architecture that can utilize any RNN model (normal RNN , GRU , LSTM)



forward propagation for the 2 direction of cells

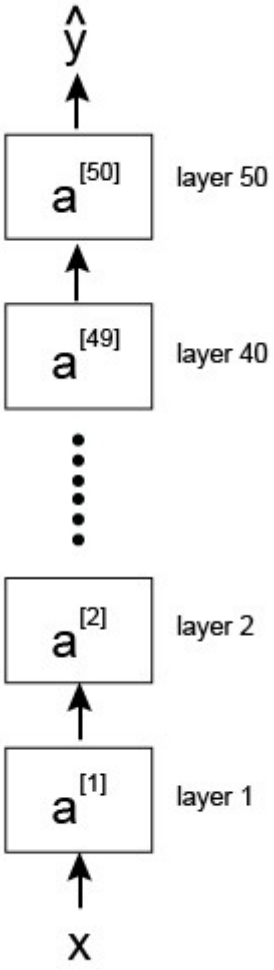
Here we apply forward propagation 2 times , one for the forward cells and one for the backward cells

Both activations (forward , backward) would be considered to calculate the output y^ at time t

$$\hat{y}^{<t>} = g(W_y [\overrightarrow{a}^{<t>}, \overleftarrow{a}^{<t>}] + b_y)$$

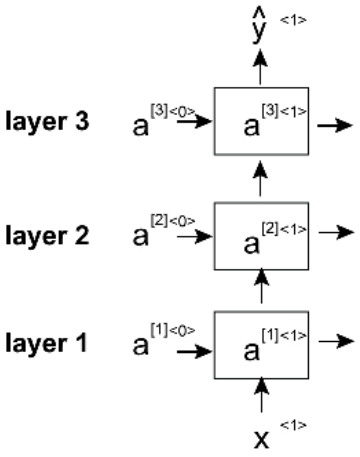
3. Multilayer networks

To achieve even greater results , we can stack multiple RNN(LSTM or GRU or normal RNN) on top of each other , but we must take into consideration that they work with time .



So to get started , here is a normal deep network , we can see that it contains multiple layers (50 in this case) , while when we apply the same concept on RNN , we tend to choose much smaller number of layers , as it would be enough and because it would be computationaly excpensive

now lets see how would we apply the concept of deep networks with RNN

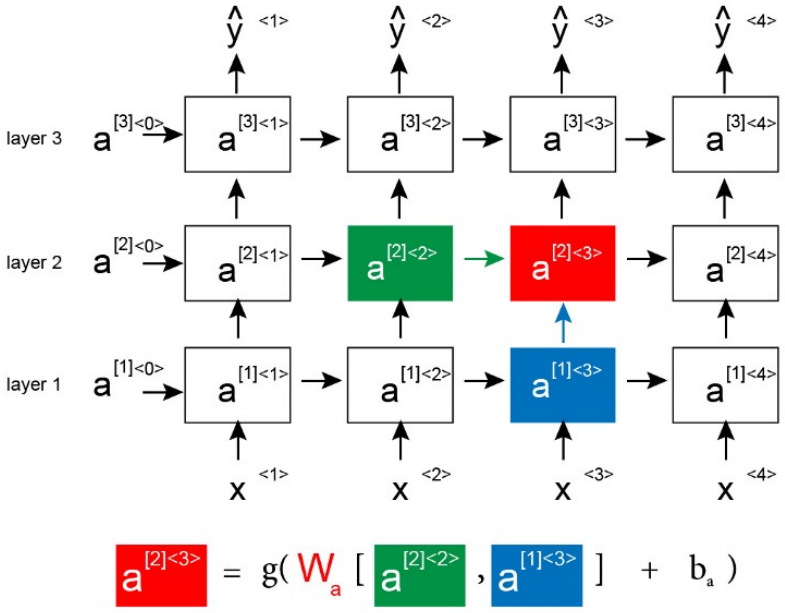


as we can see , since we are working on RNN or its variations , we must take into consideration the time factor , so each vertical column of cells represent a layer , while each progress in time we repeat this column

so our notation would be [layer] <time>

$$a^{[layer]\,<time>}$$

To get the value of any activation layer , we use both



1. Previous activation in time (time 2) from the same layer (layer 2) 💚 **green**
2. previous cell in the same time (time 3) in the previous layer (layer 1) 🔵 **blue**

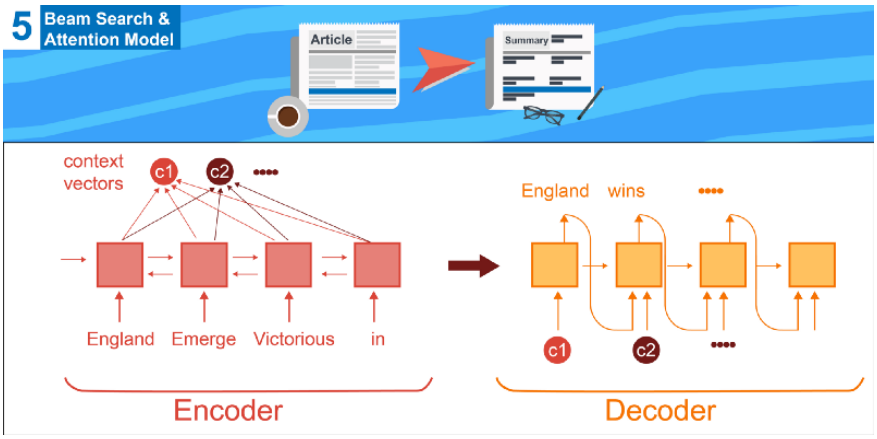
. . .

Next Time if GOD wills it , we would go through how to enhance our architecture even more using the concepts of

1. Beam Search
2. Attention Model

Beam Search & Attention for text Summarization made Easy (Tutorial 5)

amr zaki Follow
Apr 7 · 9 min read



This tutorial is the fifth one from a series of tutorials that would help you build an abstractive text summarizer using tensorflow , today we would discuss some useful modification to the core RNN seq2seq model we have covered in the last tutorial

These Modifications are

1. **Beam Search**
2. **Attention Model**

. . .

About Series

This is a series of tutorials that would help you build an abstractive text summarizer using tensorflow using multiple approaches , ***you don't need to download the data nor do you need to run the code locally on your device*** , as **data** is found on **google drive** , (you can simply copy it to your google drive , learn more here) , and the **code** for this series is written in Jupyter notebooks to run on **google colab** can be found here

We have covered so far (code for this series can be found here)

0. Overview on the free ecosystem for deep learning (how to use google colab with google drive)

1. Overview on the text summarization task and the different techniques for the task
2. Data used and how it could be represented for our task
3. What is seq2seq for text summarization and why
4. Mulitlayer Bidirectional LSTM/GRU

so lets get started

. . .

EazyMind free Ai-As-a-service for text summarization

I have added a text summarization model to a website eazymind so that you can actually try generating your own summaries yourself (and see what you would be able to build) , it can be called through simple api calls , and through a python package , so that text summarization can be easily integrated into your application without the hassle of setting up the tensorflow environment) , you can register for free , and enjoy using this api for free .

. . .

Quick Recap

Our task is of text summarization , we call it abstractive as we teach the neural network to generate words not to merely copy words .

the data that would be used would be news and their headers , it can be found on my google drive , so you just copy it to your google drive without the need to download it (more on this)

We would represent the data using word embeddings , which is simply converting each word to a specific vector , we would create a dictionary for our words (more on this)

There are different approaches for this task , they are built over a corner stone concept , and they keep on developing and building up , they start by working on a type of network called RNN , which is arranged in an Encoder/Decoder architecture called seq2seq (more on this), then we would build the seq2seq in a mulitlayer bidirectional structure , where the rnn cell would be a LSTM cell (more on this) , the code for these different approaches can be found here

> This tutorial has been based by the amazing work of **Andrew NG** , his course on RNN has been truly useful , i recommend you to see it

. . .

Today we would go through some modifications made to the core component of the encoder/decoder model , these modifications helps the network choose the best results from a pool of different possibilities , which is known as **Beam search** , also we would talk about **Attention Model** , which is a simple network added to our architecture to help it pay more attention to specific words to help it output better summarizes .

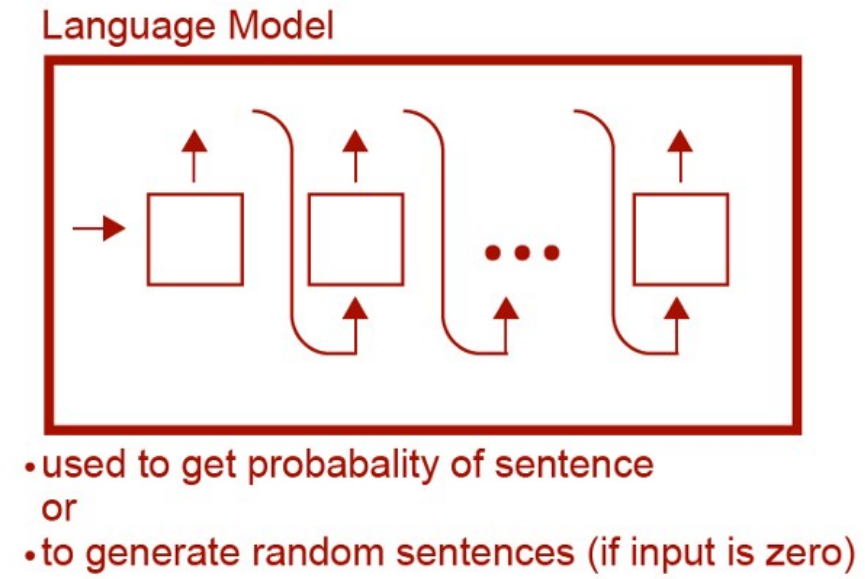
So lets get started !!

. . .

1. Beam Search

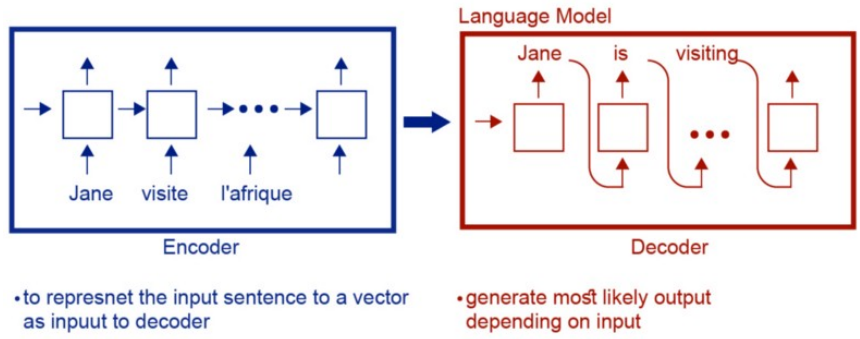
1.A Intuition (why beam search)

Our task of text summarization can be seen as a conditional language model , meaning that we generate an output given an input sentence , so the output is conditioned on the input sentence , so this is why it is called conditional .

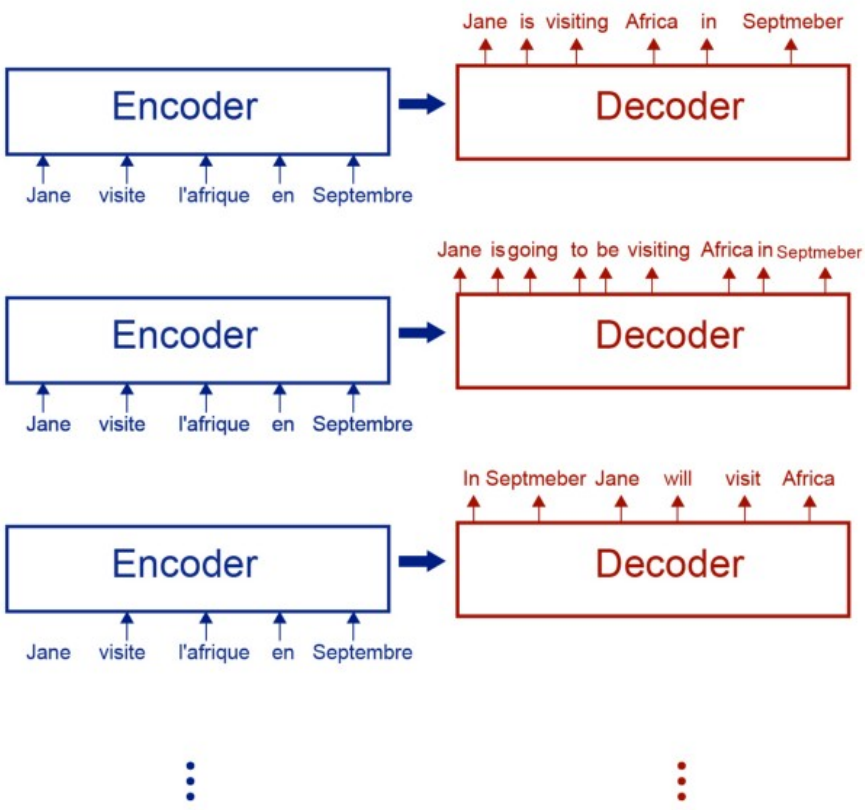


This can be seen as a contrary to a simple language model , as a normal language model only outputs the probability of a certain sentence , this can be used to generate novel sentences ,(if its input can be zero) , but in our case , it would select the most likely output given an input sentence.

So our architecture is actually divided into 2 parts , an encoder , and a decoder (like discussed here) , as the encoder would represent the input sentence as a vector , and pass it to the next part of the architecture (decoder)



But a problem would arise in the decoder , which sentence to select , as there could be a huge number of outputs for a certain input sentence



so how to choose the most likely sentence from this large pool of possible outputs ?

1. Jane is visiting Africa in September
2. Jane is going to be visiting Africa in September
3. In September Jane will visit Africa

One solution could be by generating 1 word at a time , generating the first most likely word , then generating the next most word and so … , this is called **greedy search** , but this tends not to be the most optimized approach , due to 2 main reasons

1. in practice , greedy search doesn't perform well
2. in each step we must take into consideration all possibilities of all words in the dict , if you have 10k words , then in each step you would have to consider 10k^ 10k (10k to the power 10k)

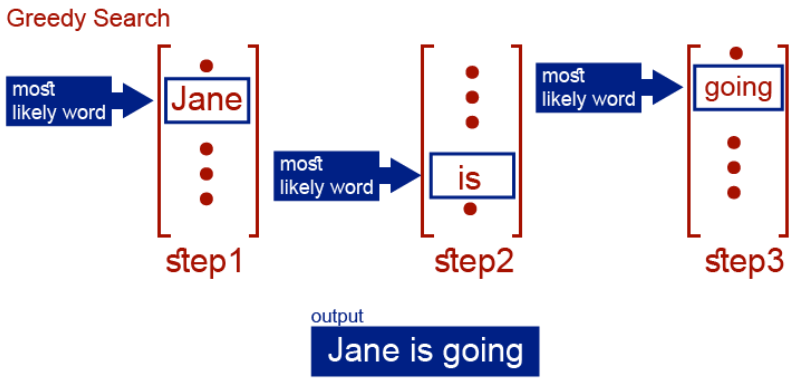
so we would go to a more optimized approximate search approach → **Beam Search**

1.B How to Beam Search

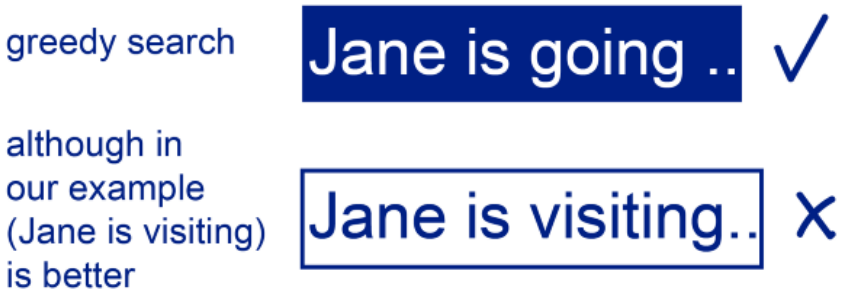
Now that we have understood the basic intuition behind beam search , how to actually implement it ?

Simply beam search differs from basic greedy search **by considering multiple options in every step , not just 1 option** , these number of options is controlled by a variable called **Beam Width**

In greedy search :



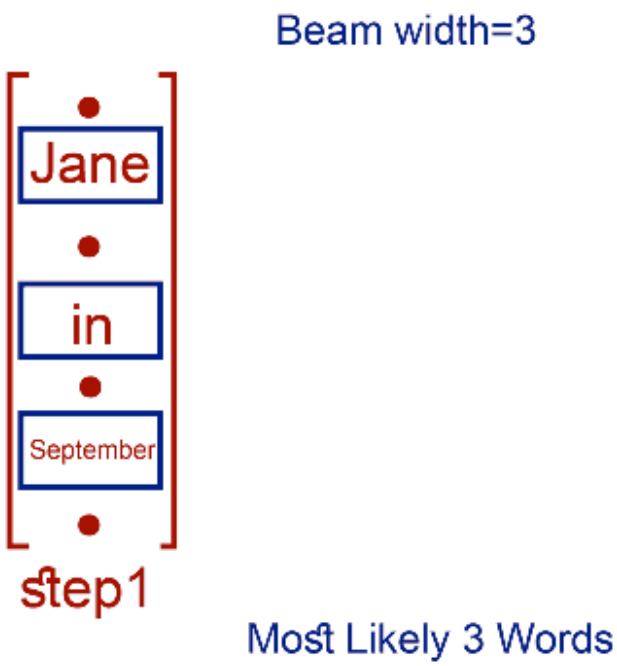
in each step we only consider the most likely output at each step , but this not always guarantee the best solution as



greedy search would output that the best solution is Jane is going , while Jane is visiting is better output

this is because the probability of the word “going” after “is” is bigger than the word “visiting” after is , so this is why we need to consider another approach not greedy search , an approach that would take multiple words into consideration

So in Beam Search it would be **(Beam Width=3)**



in step 1 , we would choose the 3 most likely words , then for each of the 3 selected words , we would get the most 3 likely words , then the same logic would occur on these 3 output sentences till we build our best output.

So if our vocab is 10k , for the each step we tried to look for the most likely word from 10k vocab , so by this we only considered 30k words

1.C Effect of Beam Width

when Beam Width = 3 → consider 3 words each step

when Beam Width = 1 → consider 1 word each step → greedy search

As Beam width increases → better results → but would require more computational resources

. . .

2. Attention Model

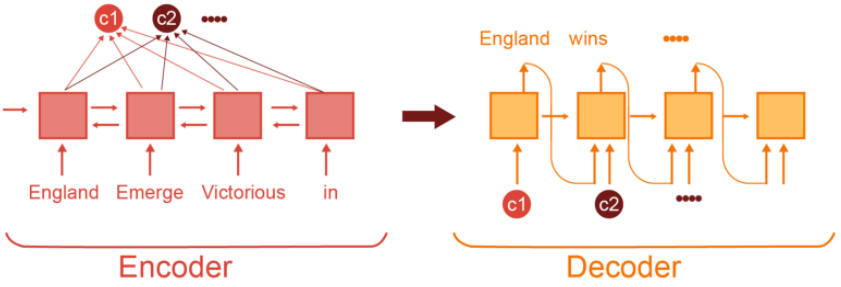
2.A Attention Intuition

When we as humans summarize text , we actually look at couple of words at a time , not the whole text to summarize at a given instance , this is what we are trying to teach our model .

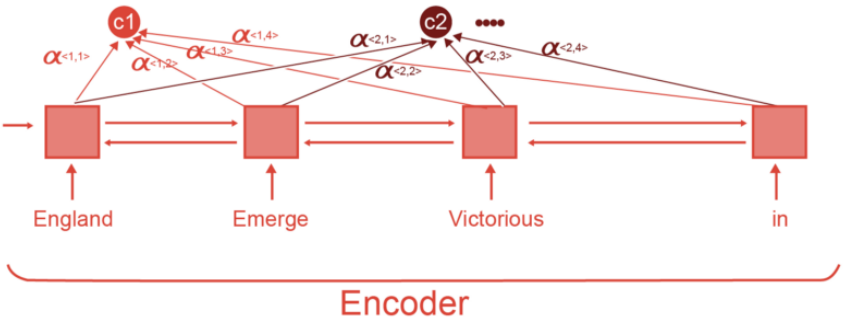
We try to teach our model to only pay attention to the neighboring words not the whole text , but what is the function of this attention , we actually don't know !! , so this is why we would construct a simple neural network for this exact task .

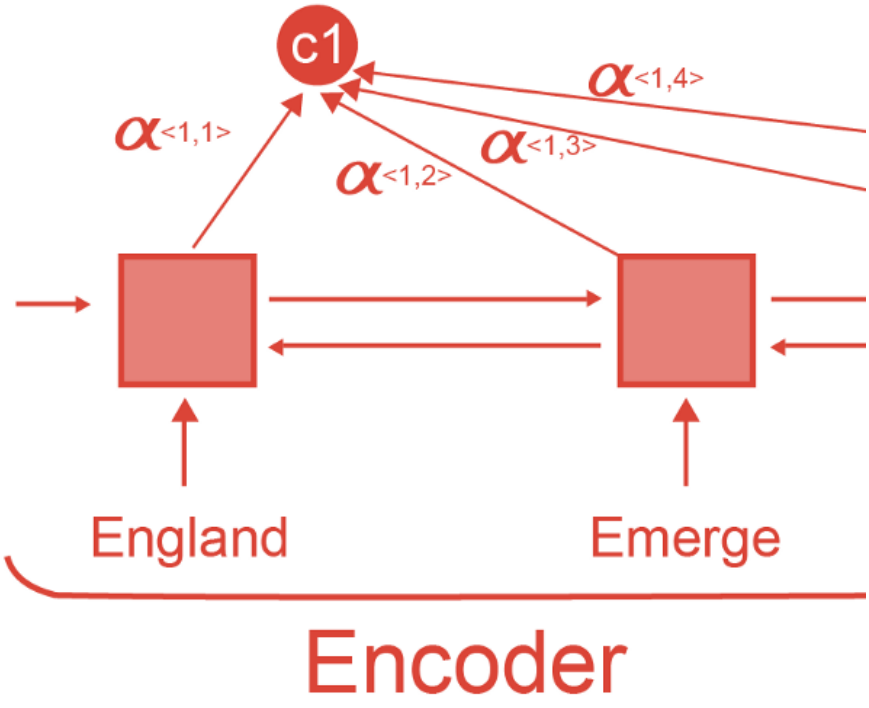
2.B Attention Structure

Here we would work on our seq2seq encoder decoder structure (more on this) , we would work on a bidirectional encoder (more on this) , our work would actually occur on a new interface between the encoder and the decoder , this new interface is called context vector , which actually represent the amount of attention given to words



Encoder would be





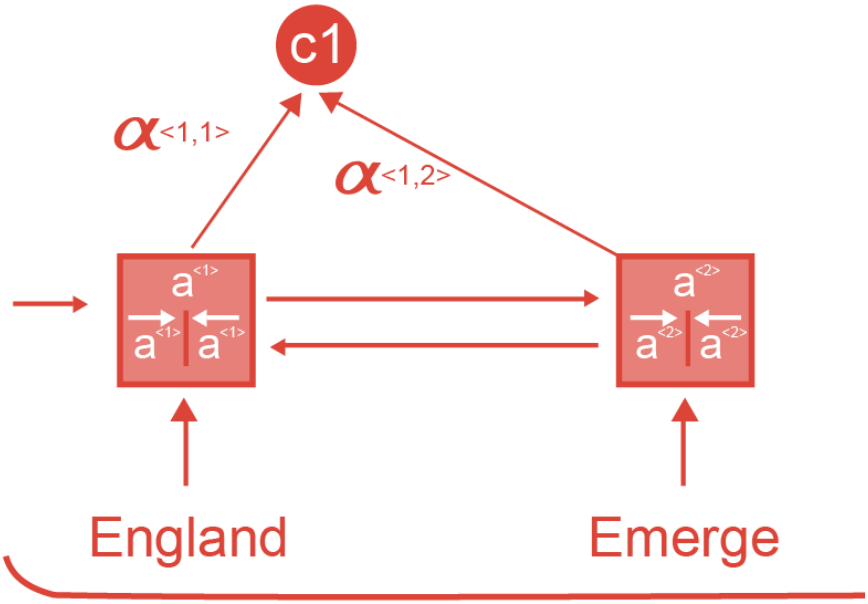
So here to calculate the context C1 that would be the input to the first word in the decoder , we would calculate multiple alpha parameters , so now we would need to know how to get these alpha parameters

2.C Calculate Attention Parameters (context and alpha)

Now that we understood the interface between the encoder and the decoder through the context vector which is actually calculated through attention alpha parameters , we would need to know more about how this is calculated

First : lets go through how to calculate the context vector

don't forget that we are working on bidirectional encoder , so the activation is actually divided into 2 parts left & right (more on this)



Encoder

$$a^{<1>} = (\overrightarrow{a}^{<1>}, \overleftarrow{a}^{<1>})$$

so the attention would be

so the attention have 2 parts , a left and right to form the activation parameter of each cell

$$c1 = \sum \alpha^{<t,1>} a^{<t>}$$

then the context would be weighted sum of the activations (t stands for number of cells in the encoder) where the weights are the alpha parameters , this would occur for each context vector

Second : We need to know how to calculate the alpha itself

Alpha is

$\alpha^{<t,t'>}$ is amount of attention output $y^{<t>}$ should pay to input $a^{<t'>}$

> *t stands for the time step in the output , while t' stands for the time step in input*

> *Sum of all alphas is 1*

there is a formula that ensures that the sum is 1

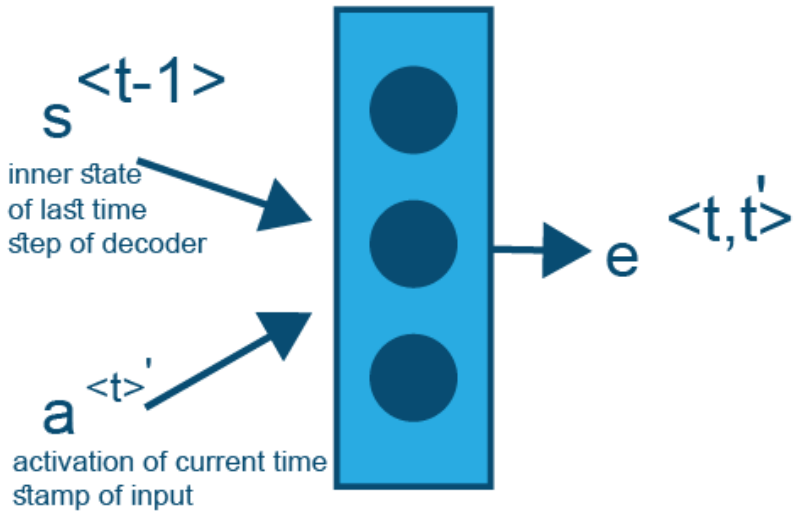
$$\alpha^{<t,t'>} = \frac{\exp(e^{<t,t'>})}{\sum \exp(e^{<t,t'>})}$$

then we just need to know what is 'e'

by intuition we can say that the attention actually depends on

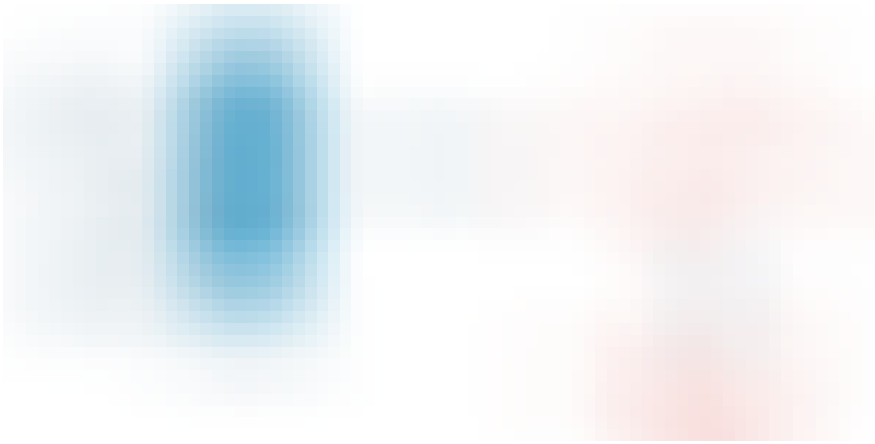
1. the current activation of the input (alpha<t'>)
2. the **previous state of the last time step in the decoder** (s<t-1>)

but we actually don't know the real function between them , so **we simply build a simple neural network to learn this relation** for us. , the output from this network would be 'e' parameter that would be used to calculate the alpha attention parameter



simple neural network

so the whole scenario would be

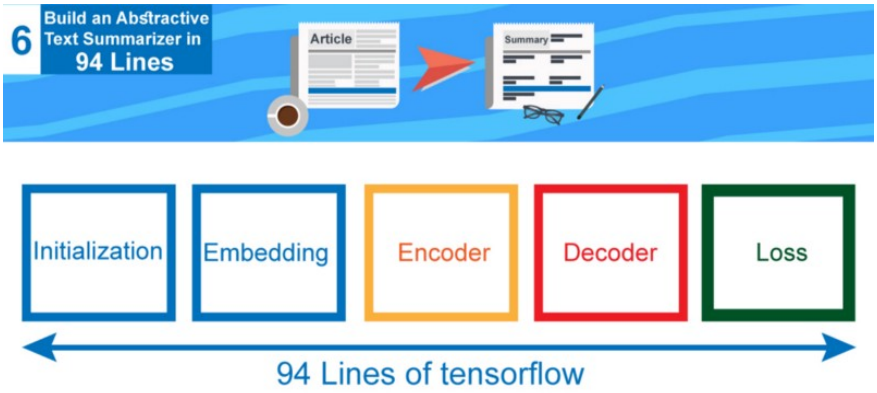


Next Time if GOD wills it , we would go through how to truly implement a text summarization model , we would go through the steps to implement this model which uses implements a multilayer bidirectional LSTM with beam search and attention in tensor flow , the code comes from dongjun-Lee , i have built it into a google colab notebook , and hosted the data to google drive , so that you won't need to download the data nor would you need to run the code on your computer , you simply run on google colab and connect google colab to your google drive .

> I truly hope you have enjoyed reading this tutorial , and i hope i have made these concepts clear , all the code for this series of tutorials are found here , you can simply use google colab to run it , please review

Build an Abstractive Text Summarizer in 94 Lines of Tensorflow !! (Tutorial 6)

amr zaki Follow
Apr 16 · 12 min read



This tutorial is the sixth one from a series of tutorials that would help you build an abstractive text summarizer using tensorflow , today we would build an abstractive text summarizer in tensorflow in an optimized way .

> *Today we would go through one of the most optimized models that has been built for this task , this model has been written by* ***dongjun-Lee*** *, this is* ***the link*** *to his model , I have used his model model on different datasets (in different languages) and it resulted in truly amazing results , so I would truly like to thank him for his effort*

I have made multiple modifications to the model to enable it to enable it to run seamlessly on google colab (link to my model) , and i have hosted the data onto google drive (more on how to link google drive to google colab) , so no need to download neither the code , nor the data , you only need a google colab session to run the code , and copy the data from my google drive to yours (more on this) , and connect google drive to your notebook of google colab

. . .

EazyMind free Ai-As-a-service for text summarization

I have added a text summarization model to a website eazymind so that you can actually try generating your own summaries yourself (and see what you would be able to build) , it can be called through simple api calls , and through a python package , so that text summarization can be easily integrated into your application without the hassle of setting up the tensorflow environment) , you can register for free , and enjoy using this api for free .

. . .

0- Intro

0-A About Series

This is a series of tutorials that would help you build an abstractive text summarizer using tensorflow using multiple approaches , we call it abstractive as we teach the neural network to generate words not to merely copy words .

We have covered so far (code for this series can be found here)

0. Overview on the free ecosystem for deep learning (how to use google colab with google drive)

1. Overview on the text summarization task and the different techniques for the task
2. **Data used and how it could be represented for our task (prerequisites for this tutorial)**
3. What is seq2seq for text summarization and why
4. Mulitlayer Bidirectional LSTM/GRU
5. Beam Search & Attention for text summarization

0-B About the Data Used

The data that would be used would be news and their headers , it can be found on my google drive , so you just copy it to your google drive without the need to download it (more on this)

> *We would represent the data using word embeddings , which is simply converting each word to a specific vector , we would create a dictionary for our words (more on this)* ***(prerequisites for this tutorial)***

0-C About the Model Used

There are different approaches for this task , they are built over a corner stone concept , and they keep on developing and building up .

Today we would start building this corner stone implementation which is a type of network called RNN , which is arranged in an Encoder/Decoder architecture called seq2seq (more on this), then we would build the seq2seq in a mulitlayer bidirectional structure , where the rnn cell would be a LSTM cell (more on this) , then we would add an attention mechanism to better interface the encoder with the decoder (more on this) , then to generate better output we use the ingenious concept of beam search (more on this)

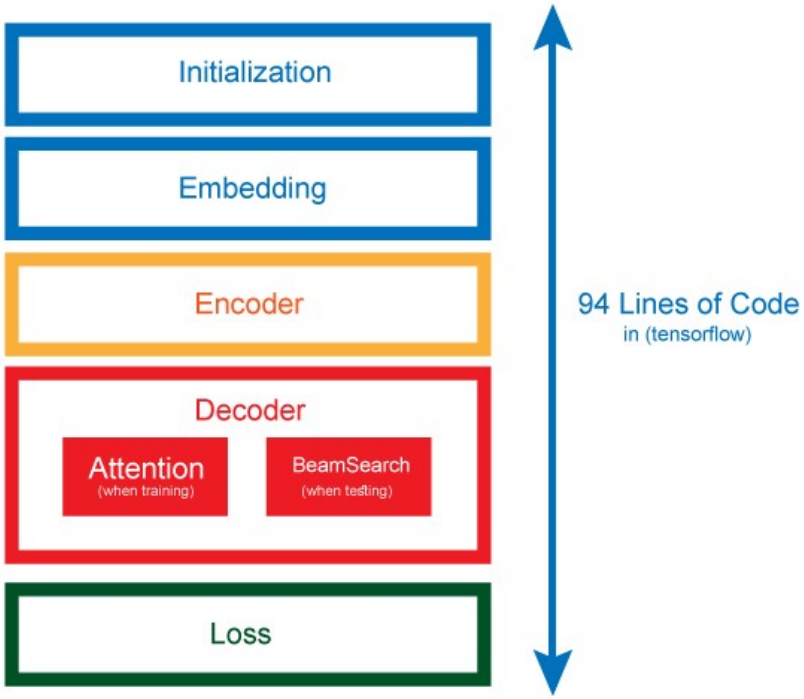
the code for all these different approaches can be found here

so lets get started !!

. . .

Model Structure

our model can be seen to structured into different blocks these blocks are



Initialization Block :

Here we would initialize the needed tensorflow **placeholders** & **variables** , and here would define our **RNN cell** that would be used throughout the model

Embedding Block :

Here we would define the embedding matrix used in both the **encoder** & the **decoder**

Encoder Block :

Here we would define the **multilayer bidirectional RNN** (more on this) that forms the encoder part of our model , and we **output the encoder** state as an input to the decoder part

Decoder Block :

Here the decoder is actually portioned into 2 distinct parts

1. **Attention Mechanism** (more on this) which is used to better interface the encoder with the decoder , this would be used in **training** phase
2. **BeamSearch** (more on this) which is used to generate better output from our model , this would be used in **testing** phase

Loss Block :

This block would only be used in **training** phase , here we would apply clipping to our gradients , and we would actually run our optimizer (Adam Optimizer is used here) , and here is the place where we would apply our gradients to the optimizer.

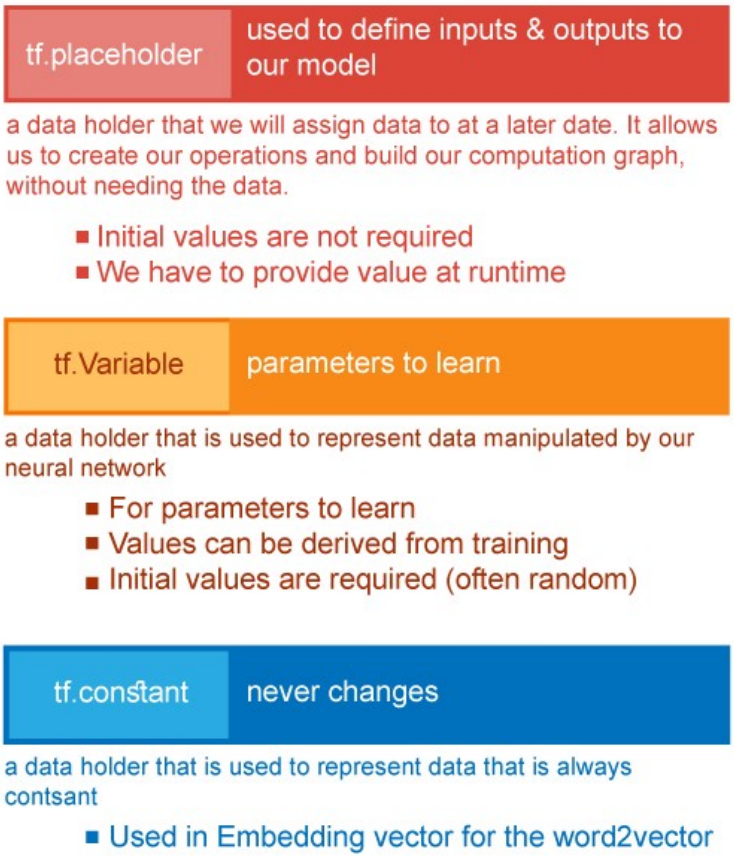
. . .

1- Initialization Block

First we would need to import the libs that we would use

```
import tensorflow as tf
from tensorflow.contrib import rnn  #cell that we would use
```

Before Building our Model Class we need to get define some tensorflow concepts first



So we tend to define placeholders like this

```
X = tf.placeholder(tf.int32, [None, article_max_len])
# here we define the input x as int32 , with promise to
provide its # data in runtime
#
# we also provide its shape , where None is used for a
dimension of # any size
```

and for the variables we tend to define them as

```
global_step = tf.Variable(0, trainable=False)
# a variable must be intialized ,
# and we can set it to either be trainable or not
```

Then lets build our **Model Class**

```
class Model(object):
    def __init__(self, reversed_dict, article_max_len,
summary_max_len, args, forward_only=False):
        self.vocabulary_size = len(reversed_dict)
        self.embedding_size = args.embedding_size
        self.num_hidden = args.num_hidden
        self.num_layers = args.num_layers
        self.learning_rate = args.learning_rate
        self.beam_width = args.beam_width
```

we would pass an obj called args that would actually contain multiple parameters from

1. embedding size (size of word2vector)
2. num_hidden (size of RNN)
3. num_layers (layers of RNN) (more on this)
4. Learning Rate
5. BeamWidth (more on this)
6. Keep Prob

we would also need to initialize the model with other paaremetrs like

1. reversed dict (dict of keys , each key wich is a num points to a specific ord) (more on how to build your reversed dict)
2. article_max_len & article_summary_len (max length of article sentence as input and max length of summary sentenceas output)
3. Forward Only (bool value to indicate training or testing phase) (**Forward Only = False → training phase**)

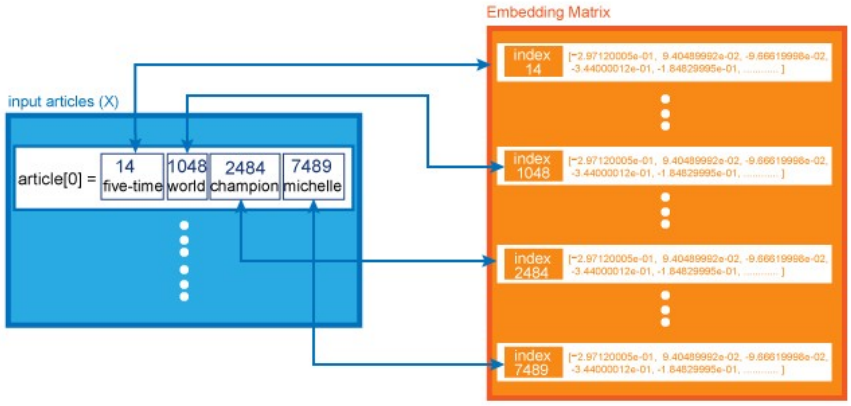
then to continue the initalization

```
if not forward_only: #training phase
            #keep_prob as variable in training phase
            self.keep_prob = args.keep_prob
else: #testing phase
            #keep_prob constant in testing phase
            self.keep_prob = 1.0

        #here we would use LSTM as our cell
        self.cell = tf.nn.rnn_cell.BasicLSTMCell

        #projection layer that would be used in decoder in
both
        #training and testing phase
        with tf.variable_scope("decoder/projection"):
              self.projection_layer =
tf.layers.Dense(self.vocabulary_size, use_bias=False)

        #define batch size(our data would be provided in
batches)
        self.batch_size = tf.placeholder(tf.int32, (),
name="batch_size")

        #X as input , define as length of articles
        self.X = tf.placeholder(tf.int32, [None,
article_max_len])
        self.X_len = tf.placeholder(tf.int32, [None])

        #define decoder (input , target , length)
        #using the summary length
        self.decoder_input = tf.placeholder(tf.int32, [None,
summary_max_len])
        self.decoder_len = tf.placeholder(tf.int32, [None])
        self.decoder_target = tf.placeholder(tf.int32,
[None, summary_max_len])

        #define global step beginning from zero
        self.global_step = tf.Variable(0, trainable=False)
```

. . .

2- Embedding Block :

Here we would represent both our **inputs articles that would be the embedded inputs** and the **decoder inputs** using word2vector (more on this)



mapping between input articles (X) and embedded matrix (word2vector) is done using **tf.nn.embedding_lookup**

we would define our variables for embedding in a variable scope , we would name it embedding

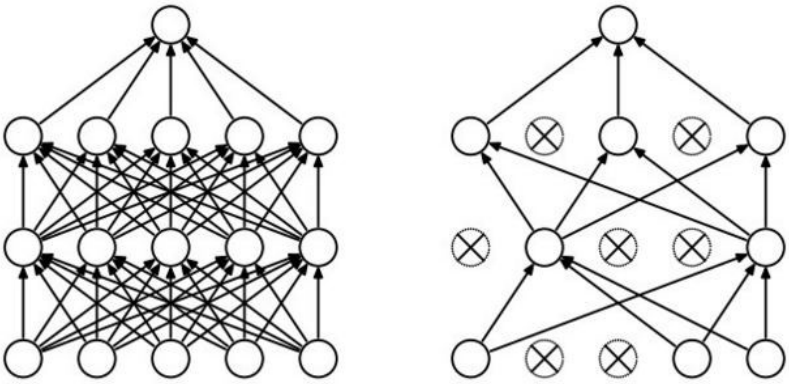
```
with tf.name_scope("embedding"):
            #if training ,
            #and you enable args.glove variable to true
            if not forward_only and args.glove:
                #here we use tf.constant as we won't change
it
                #get_init_embedding is a function
                #that returns the vector for each word in
our dict
                init_embeddings =
tf.constant(get_init_embedding(reversed_dict,
self.embedding_size), dtype=tf.float32)
            else: #else random define the word2vector for
testing
                init_embeddings =
tf.random_uniform([self.vocabulary_size,
self.embedding_size], -1.0, 1.0)
            self.embeddings = tf.get_variable("embeddings",
initializer=init_embeddings)
            #then define for both encoder input
            self.encoder_emb_inp =
tf.transpose(tf.nn.embedding_lookup(self.embeddings,
self.X), perm=[1, 0, 2])
            #and define for decoder input
            self.decoder_emb_inp =
tf.transpose(tf.nn.embedding_lookup(self.embeddings,
self.decoder_input), perm=[1, 0, 2])
```

. . .

3- Encoder Block :

Here we would actually define the multilayer bidirectional lstm for the encoder part of our seq2seq (more on this) , we would define our variables here in a name scope that we would call “encoder”.

Here we would use the concept of **Dropout** , we would use it after each cell in our architecture , it is used to randomly activate a subset of our net, and is used during training for regularization.



(a) Standard Neural Net (b) After applying dropout.

```
with tf.name_scope("encoder"):

            fw_cells = [self.cell(self.num_hidden) for _ in
range(self.num_layers)]

            bw_cells = [self.cell(self.num_hidden) for _ in
range(self.num_layers)]

            fw_cells = [rnn.DropoutWrapper(cell) for cell in
fw_cells]

            bw_cells = [rnn.DropoutWrapper(cell) for cell in
bw_cells]
```

Now after defining the forward and backward cells , we would need to actually connect them together to form the bidirectional structure , so we would use **stack_bidirectional_dynamic_rnn** , which takes all of the following parameters as its inputs

1. forward cells
2. Backward Cells

3. Encoder emb input (input articles in word2vector format)
4. X_len (length of articles)
5. Using time_major = True is a bit more efficient because it avoids transposes at the beginning and end of the RNN calculation.

```
encoder_outputs, encoder_state_fw, encoder_state_bw =
tf.contrib.rnn.stack_bidirectional_dynamic_rnn(
                fw_cells, bw_cells, self.encoder_emb_inp,
                sequence_length=self.X_len, time_major=True,
dtype=tf.float32)
```

Now we would need to actually use the output from this **stack_bidirectional_dynamic_rnn** function , we mainly need 2 main outputs

1. encoder_output (would be used in attention calculation) (more on attention)
2. encoder_state (would be used for the initial state of the decoder)

so to get encoder_output we simply

```
self.encoder_output = tf.concat(encoder_outputs, 2)
```

then to get encoder_state , we would combine both (encoder_state_c) & (encoder_state_h) of both the forward & backward using LSTMStateTuple

```
encoder_state_c = tf.concat((encoder_state_fw[0].c,
encoder_state_bw[0].c), 1)

encoder_state_h = tf.concat((encoder_state_fw[0].h,
encoder_state_bw[0].h), 1)

self.encoder_state = rnn.LSTMStateTuple(c=encoder_state_c,
h=encoder_state_h)
```

. . .

4- Decoder Block :

Here the decoder is divided into 2 parts

1. Training part (to train attention model) (more on attention model)
2. testing/running part (for attention & beam search) (more on beam search)

so lets first define out (name scope) & (variable scope) for both parts , we would also define a multilayer cell structure that would be also used for both parts

```
with tf.name_scope("decoder"), tf.variable_scope("decoder")
as decoder_scope:
            decoder_cell = self.cell(self.num_hidden * 2)
```

4.a Training Part (Attention Model)

First we need to prepare our attention structure , here we would use BahdanauAttention

encoder_output would be used inside the attention calculation (more on attention model)

```
attention_states = tf.transpose(self.encoder_output, [1, 0,
2])
attention_mechanism = tf.contrib.seq2seq.BahdanauAttention(
                    self.num_hidden * 2, attention_states,
memory_sequence_length=self.X_len, normalize=True)
```

then we would further define the decoder cell (as from the first step in decoder , we just defined the decoder cell as a simple multilayer lstm , now we would add attention) , to do this we would use **AttentionWrapper** , which combines attention_mechanism with decoder cell

```
decoder_cell =
tf.contrib.seq2seq.AttentionWrapper(decoder_cell,
attention_mechanism,
attention_layer_size=self.num_hidden * 2)
```

Now we would need to define the inputs to the decoder cell , this input actually comes from 2 sources (more on seq2seq)

1. encoder output (used within initial step)
2. decoder input (summary sentence in the training phase)

so lets first define the initial state that would come from the encoder

```
initial_state = decoder_cell.zero_state(dtype=tf.float32,
batch_size=self.batch_size)
```

```
initial_state =
initial_state.clone(cell_state=self.encoder_state)
```

now we would combine both the initial state with the decoder input (summary sentence) , here to use the BasicDecoder , we need to provide the decoder input through a helper , this helper would combine all of (decoder_emb_inp , decoder_len) together

```
helper =
tf.contrib.seq2seq.TrainingHelper(self.decoder_emb_inp,
self.decoder_len, time_major=True)
```

```
decoder = tf.contrib.seq2seq.BasicDecoder(decoder_cell,
helper, initial_state)
```

```
outputs, _, _ = tf.contrib.seq2seq.dynamic_decode(decoder,
output_time_major=True, scope=decoder_scope)
```

now for the last step of the training phase , we would need to define the outputs (logits)from the decoder , to be used within the loss block for training

```
#just use the rnn_outputs from aall the outputs
self.decoder_output = outputs.rnn_output
```

```
#then get logits , by performing a transpose on decoder
output
self.logits =
tf.transpose(self.projection_layer(self.decoder_output),
perm=[1, 0, 2])
```

```
#then reshape the logits
self.logits_reshape = tf.concat(
                    [self.logits, tf.zeros([self.batch_size,
summary_max_len - tf.shape(self.logits)[1],
self.vocabulary_size])], axis=1)
```

4.b Testing/Running part (Attention & Beam search)

Here in this phase , there are 2 main goals

1. divide the encoder output & encoder states & x_len (article length) to parts to actually perform the beam search methodology (more on beam search)
2. build a decoder independent on decoder input , as in the testing phase we don't have the summary sentence as our input , so we would need to build the decoder in a different way than above

first lets divide encoder output & encoder states & x_len (article length) to parts to actually perform the beam search methodology , here we would use **beam_width** variable that was already defined above

```
tiled_encoder_output = tf.contrib.seq2seq.tile_batch(
                    tf.transpose(self.encoder_output, perm=
[1, 0, 2]), multiplier=self.beam_width)
```

```
tiled_encoder_final_state =
tf.contrib.seq2seq.tile_batch(self.encoder_state,
multiplier=self.beam_width)

tiled_seq_len = tf.contrib.seq2seq.tile_batch(self.X_len,
multiplier=self.beam_width)
```

then lets define the attention mechanism (just like before , but taking the tiled variables into consideration)

```
attention_mechanism = tf.contrib.seq2seq.BahdanauAttention(
                    self.num_hidden * 2,
tiled_encoder_output, memory_sequence_length=tiled_seq_len,
normalize=True)
```

```
decoder_cell =
tf.contrib.seq2seq.AttentionWrapper(decoder_cell,
attention_mechanism,
attention_layer_size=self.num_hidden * 2)
```

```
initial_state = decoder_cell.zero_state(dtype=tf.float32,
batch_size=self.batch_size * self.beam_width)

initial_state =
initial_state.clone(cell_state=tiled_encoder_final_state)
```

then lets define our decoder , but here we would use the **BeamSearchDecoder** , this takes into consideration all of

1. Decoder cell (previously defined)
2. Embedding word2vector (defined in embedding part)
3. projection layer (defined in the beginning of class)
4. decoder initial state (previously defined)
5. beam_width (user defined)
6. start token & end token

```
decoder = tf.contrib.seq2seq.BeamSearchDecoder(
                    cell=decoder_cell,
                    embedding=self.embeddings,
                    start_tokens=tf.fill([self.batch_size],
tf.constant(2)),
                    end_token=tf.constant(3),
                    initial_state=initial_state,
                    beam_width=self.beam_width,
                    output_layer=self.projection_layer )
```

then all what is left to do , is to define the outputs , that would actually directly reflect to the real output from the whole seq2seq architecture , as this phase is where prediction is actually computed

```
outputs, _, _ = tf.contrib.seq2seq.dynamic_decode(
                    decoder, output_time_major=True,
maximum_iterations=summary_max_len, scope=decoder_scope)
```

```
self.prediction = tf.transpose(outputs.predicted_ids, perm=
[1, 2, 0])
```

. . .

5- Loss Block :

This block is where training actually occurs , here training actually occurs through multiple steps

1. calculating loss (more on loss calculation)
2. calculating gradients and applying clipping on gradients (more on exploding gradients)
3. applying optimizer (here we would use Adam optimizer)

First we define our name scope , and we would specify that this block would only work through the training phase

```
with tf.name_scope("loss"):
            if not forward_only:
```

Second we would calculate the loss (more on loss calculation)

```
crossent = tf.nn.sparse_softmax_cross_entropy_with_logits(
                    logits=self.logits_reshape,
labels=self.decoder_target)
```

```
weights = tf.sequence_mask(self.decoder_len,
summary_max_len, dtype=tf.float32)
```

```
self.loss = tf.reduce_sum(crossent * weights /
tf.to_float(self.batch_size))
```

Third we would calculate our gradients , and apply clipping on gradients to solve the problem of exploding gradients (more on exploding gradients)

> ***(from tutorial 4)***

> ***Exploding Gradients :*** *Occurs with deep networks (i.e: networks* with many layers *like in our case) , when we apply back propagation, the gradients would get too large . Actually this error can be solved rather easy , using the concept of* ***gradient clipping*** *, which is simply setting a specific threshold , that when the gradients exceed it , we would clip it to a certain value .*

```
params = tf.trainable_variables()
gradients = tf.gradients(self.loss, params)
clipped_gradients, _ = tf.clip_by_global_norm(gradients,
5.0)
```

Forth we would apply our optimizer , here we would use Adam optimizer , here we would use the previously defined learning_rate

```
optimizer = tf.train.AdamOptimizer(self.learning_rate)
```

```
self.update =
optimizer.apply_gradients(zip(clipped_gradients, params),
global_step=self.global_step)
```

. . .

Next Time if GOD wills it , we would go through

1. the code needed to divide our data into batches
2. needed code to use this model for training

Then after we are done with this core model implementation , if GOD wills it , we would go other modern implementations for text summarization like

1. pointer generator
2. Using reinforcement learning with seq2seq

(more on different implementations for seq2seq for text summarization)

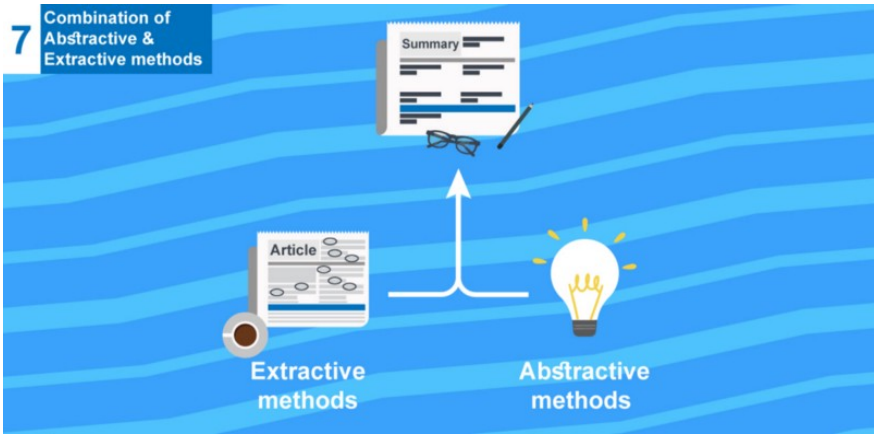
All the code for this tutorial is found as open source here .

> I truly hope you have enjoyed reading this tutorial , and i hope i have made these concepts clear , all the code for this series of tutorials are found here , you can simply use google colab to run it , please review the tutorial and tell me what do you think about it , hope to see you again

Next Tutorials

Combination of Abstractive & Extractive methods for Text Summarization (Tutorial 7)

amr zaki Follow
May 19 · 9 min read



Combining both Abstractive & Extractive methods for text summarization

This tutorial is the seventh one from a series of tutorials that would help you build an abstractive text summarizer using tensorflow .

Today we discover some novel ways of combining both abstractive & extractive methods of copying of words for text summarization , (code can be found here in jupyter notebook format for google colab) , we would combine the concepts of generating new words , with copying of words from the given sentence , we would **learn the reason** this is important , and we would go through how it is actually done !!

EazyMind free Ai-As-a-service for text summarization

You can actually try generating your own summaries using this model easily through eazymind (i have added this model to eazymind , so it can be called through simple api calls , and through a python package ,

so that this model can be easily integrated into your application without the hassle of setting up the tensorflow environment) , you can register for free , and enjoy using this api for free .

. . .

> *Today we would go through concepts discussed in both these papers (Abstractive Text Summarization using Seq) & (Get To The Point: Summarization with Pointer-Generator Networks , their repo , their truly **AMAZING** blog post) , their work has been truly helpful , it has resulted in truly great results , I would really like to thank them for their amazing efforts*

Today we would

1. Discuss how we integrate both worlds of abstractive & extractive methods for text summarization.
2. Quick overview on the code & preprocessing of the data (i have converted this model to a jupyter notebook to run seamlessly on google colab , and the data is found on google drive , so no need to download neither the code , nor the data , you only need a google colab session to run the code , and copy the data from my google drive to yours (more on this) , and connect google drive to your notebook of google colab).
3. This model has been converted to an api (and a python package), that you can simply try it out in your projects without the hassle of actually setting up your tensorflow environment , you can Free Register on eazymind , and use this api for free now .

. . .

0- About Series

This is a series of tutorials that would help you build an abstractive text summarizer using tensorflow in multiple approaches , we call it abstractive as we teach the neural network to generate words not to merely copy words , today we would combine these concepts with extractive concepts to gain the benefits of the 2 worlds.

We have covered so far (code for this series can be found here)

0. Overview on the free ecosystem for deep learning (how to use google colab with google drive)

1. Overview on the text summarization task and the different techniques for the task
2. Data used and how it could be represented for our task (prerequisites for this tutorial)
3. What is seq2seq for text summarization and why
4. Mulitlayer Bidirectional LSTM/GRU
5. Beam Search & Attention for text summarization
6. Building a seq2seq model with attention & beam search

In these tutorials we have built the corner stone model that would enhance over it today , as all the newest approaches build upon this corner baseline model

to lets Begin !!

. . .

1- Why copying ?

Last tutorial , we have built a seq2seq model with attention and beam search capable of abstractive text summarization , the results were truly good , but it would suffer from some problem , **out of vocabulary words** (OOV),

1–1 out of vocabulary words

which are unseen words , actually this problem comes from the fact that we train our model with a limited vocab (as the vocab can never contain all English words) so in testing , our model would face new words that he didn't see before , normally we would model these words as <unk> , but actually this doesn't generate good summaries !!

1–2 Wrong Factual Information

Another problem is that factual information are not generated accurately

> *given a sentence : In last night game ,* Germany beat Argentina **3–2**

> *model would generate :* Germany beat Argentina **2–1**

this comes from the fact that the token **3–2** is actually unique , (not unknown , but unique) , harder for the model to regenerate , so it would be much easier if the model was able to copy the token **3–2** from the original sentence not generate it on its own .

1–3 Replacing names with similar wrong names

Also another problem can be seen with the exact names of people and countries , as our model would actually cluster same countries together using the concept of word embeddings , so we would see that the model actually sees both words (*Delhi* & *Mumbai*) the same , and would see names like (*Anna* & *Emily*) the same , as they would have similar word embeddings .

> *So we would implement a model capable of copying of unique words from original sentence , as it is quite difficult for our model to regenerate these words by himself , this technique is called* ***Pointer generator***

. . .

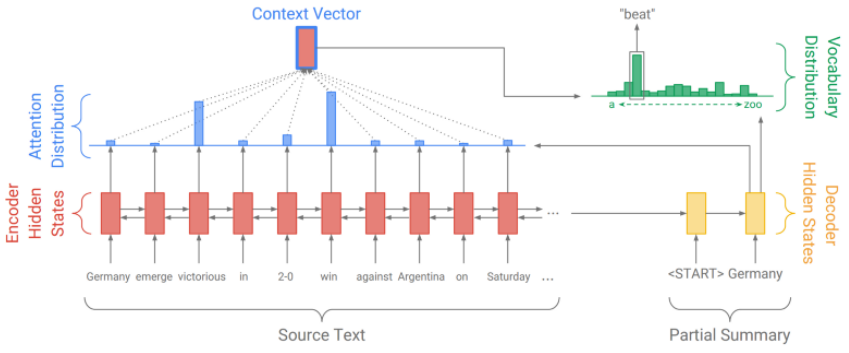
2- What is Pointer generator ?

This is actually a neural network that is trained to learn when to **generate novel words** , and when to **copy words from the original sentence** .

It is called a **pointer generator network** as we use a pointer to point out to the word would be copied from the original sentence .

2–1 Our Basic structure

> *this graph has been borrowed from (Get To The Point: Summarization with Pointer-Generator Networks , their repo , their truly* ***AMAZING*** *blog post)*



Basic structure is built as a seq2seq model (Mulitlayer Bidirectional LSTM Encode & a decoder with Beam Search & Attention) and to generate the output sentence , we use the output from both

1. **Decoder**
2. **Attention** (context vector) (i.e: attention actually tells us which words are important from our input)

from these 2 outputs , we would generate a probability distribution over all our vocab , this is called **Vocabulary Distribution** , this distribution helps us in generating the final output

> *so to keep in mind , we have 2 important distributions here :*

> *1- A **local** distribution (**Attention**) which tells which words are important from input sentence*

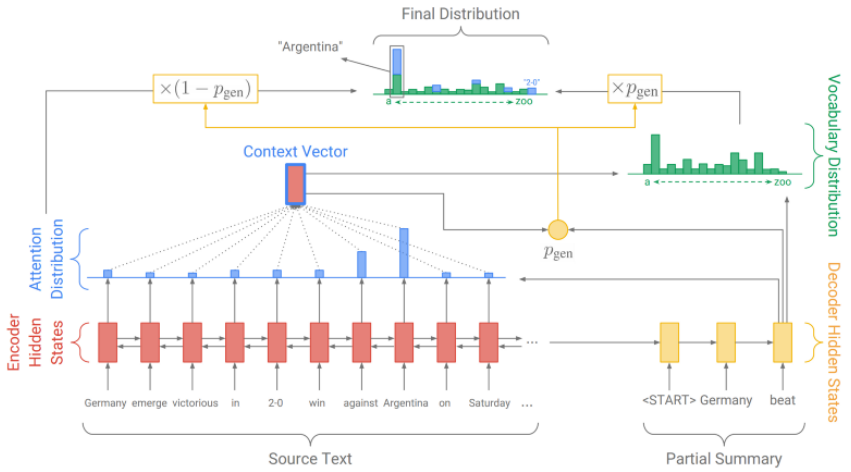
> *2- this **local** distribution (**Attention**) is used to calculate the **Global** distribution (**Vocabulary Distribution**) which tells the probability of relevance of output according to **ALL** words of the vocab*

2–2 Now lets add our Pointer Generator network

Pointer Generator network here would be a neural network trained to choose from where to generate the output , either from

1. the **Global** distribution (**Vocabulary Distribution**) , i.e : generate novel new words
2. or from **local** distribution (**Attention**) , i.e : copy words from original sentence

> *this graph , and formula have been borrowed from (Get To The Point: Summarization with Pointer-Generator Networks , their repo , their truly **AMAZING** blog post)*



$$P_{\text{final}}(w) = p_{\text{gen}} P_{\text{vocab}}(w) + (1 - p_{\text{gen}}) \sum_{i:w_i=w} a_i$$

so we would have a parameter **Pgen** , that would contain the probability of generating the word either from **Vocab distribution** (P vocab), or from **Attention distribution** (sum of attentions of words) , (i.e : either generate a new word , or copy the word from the sentence)

. . .

3- How to build a Pointer Generator network

There have been 2 main approaches for implementing this network , both rely on the same concept with a slight differentiation in the implementation ,

the main inputs would be

1. Decoder inputs
2. Attention inputs

Abstractive Text Summarization using Sequence-to-sequence RNNs and Beyond Paper

$$\begin{aligned} P(s_i = 1) &= \sigma(\mathbf{v}^s \cdot (\mathbf{W}_h^s \mathbf{h}_i + \mathbf{W}_e^s \mathbf{E}[o_{i-1}] \\ &+ \mathbf{W}_c^s \mathbf{c}_i + \mathbf{b}^s)), \end{aligned}$$

P here is Pgen , here we would get it by training a sigmoid layer ,

the inputs would be

1. **hi :** hidden state of the decoder (output of decoder) → **decoder** parameter
2. **E[oi−1]** : previous time step of decoder step → **decoder** parameter
3. **ci** : attention-weighted context vector → **attention** inputs

Ws h ,Ws e ,Ws c , b s and v s are the learnable parameters.

Get To The Point: Summarization with Pointer-Generator Networks

$$p_{\text{gen}} = \sigma\left(w_{h^*}^T h_t^* + w_s^T s_t + w_x^T x_t + b_{\text{ptr}}\right)$$

1. **st :** the decoder state→ **decoder** parameter

2. **xt** : the decoder input→ **decoder** parameter

3. **ht** ∗ : context vector → **attention** inputs

where vectors wh ∗ , ws , wx and scalar bptr are learnable parameters

. . .

4- TensorFlow Implementation

abisee have implemented the paper Get To The Point: Summarization with Pointer-Generator Networks using tensorflow , his code is based on the TextSum code from Google Brain.

I have modified his code (my modification)

- to run in a jupyter notebook to run seamlessly on google colab (more on this)
- and have uploaded the data to google drive , to be easily integrated within google colab (more on this)

so no need to download neither the code , nor the data , you only need a google colab session to run the code , and copy the data from my

google drive to yours (more on this) , and connect google drive to your notebook of google colab

. . .

5- Data representation

this model has been built on CNN/Daily Mail dataset , which is built to have multi summaries for the same story

the data is provided to the model through running it into a script that converts it into chunked binary files to be then provided to the model

I have modified this script (my modification) to be easier (in case you need to reprocess your own data)

the original script expects the data to be provided in a .story format , which is a data file contains both the text and its summary in the same file , so i just edited to be much simpler , now you can provide your data to my script in a csv format

I have also replaced the need to download a specific java script (Stanford CoreNLP) for tokenization , with the simpler nltk tokenizer (hope this proves helpful)

. . .

6- EazyMind API

If you need to try out this model (before trying out the code) , you can easily do so through eazymind , which is a **Free Ai-as-a-service platform** , providing this pointer generator model for abstractive text summarization

You can also resister for free to call this model as an api through either curl

```
curl -X POST \
  http://eazymind.herokuapp.com/arabic_sum/eazysum \
  -H 'cache-control: no-cache' \
  -H 'content-type: application/x-www-form-urlencoded' \
  -d
'key=xxxxxxxxx&sentence=Facebook%20CEO%20Mark%20Zuckerberg%2
C%20left%2C%20makes%20the%20keynote%20speech%20at%20F8%2C%20
the%20Facebook%26%2339%3Bs%20developer%20conference%2C%20Tue
sday%2C%20April%2030%2C%202019%2C%20in%20San%20Jose%2C%20Cal
if.%20(AP%20Photo%2FTony%20Avelar%20)%0AFacebook%20says%20th
at%2C%20unlike%20its%20past%2C%20its%20future%20is%20privacy
%0AA%20trader%20works%20ahead%20of%20the%20closing%20bell%20
on%20the%20floor%20of%20the%20New%20York%20Stock%20Exchange%
20(NYSE)%20on%20April%2012%2C%202019%20in%20New%20York%20Cit
y.%20(Photo%20by%20Johannes%20EISELE%20%2F%20AFP)%20%20%20%2
0%20%20%20%20(Photo%20credit%20should%20read%20JOHANNES%20EI
SELE%2FAFP%2FGetty%20Images)%0AResilience%20is%20still%20the
%20word%20for%20stocks'
```

or through python package

```
pip install eazymind
```

then simply call it

```
from nlp.eazysum import Summarizer
key = "xxxxxxxxxxxxxxxxxxxxx"

sentence = """(CNN)The White House has instructed former
White House Counsel Don McGahn not to comply with a subpoena
for documents from House Judiciary Chairman Jerry Nadler,
teeing up the latest in a series of escalating oversight
showdowns between the Trump administration and congressional
Democrats.      McGahn's decision not to comply     with the
subpoena could push Nadler     to hold McGahn in contempt of
Congress,     just as he's moving to do with Attorney
General William Barr after the Justice     Department defied
a subpoena for the unredacted Mueller report and underlying
evidence."""

summarizer = Summarizer(key)
print(summarizer.run(sentence))
```

. . .

Next Time if GOD wills it , we would go through