Deep Reinforcement Learning for Dialogue Generation

Release 0.1

Kapil Chiravarambath

Apr 04, 2019

CONTENTS

Contents:

CHAPTER

ONE

GETTING STARTED

1.1 Download Source

```
> git clone https://github.com/kc3/Springboard.git
```

1.2 Change directory

```
> cd capstone_2
```

1.3 Create Environment

```
> conda env create -n ‘capstone_2’ -f environment.yml
```

1.4 Test Environment

```
> tox
```

1.5 Download Movie Dialog Dataset

```
> python .srcdatamake_dataset.py ./src/data/raw ./src/data/interim ./src/data/processed
```

1.6 Chat Demo

```
> python -m src.models.predict
```

CHAPTER

TWO

DEEP REINFORCEMENT LEARNING FOR DIALOGUE GENERATION

2.1 Motivation

Conversational bots are one of the most exciting applications of NLP. Services such as Google Assistant, Apple Siri and Amazon Alexa are becoming ubiquitous and improving in capabilities rapidly. Chatbots are typically used in dialog systems for various practical purposes including customer service or information acquisition. Some chatbots use sophisticated natural language processing systems, but many simpler ones scan for keywords within the input, then pull a reply with the most matching keywords, or the most similar wording pattern, from a database.

2.2 Problem Description

Efforts to build statistical dialog systems fall into two major categories, statistical machine translation models and task-oriented dialog systems.

The first treats dialogue generation as a source to target transduction problem and learns mapping rules between input messages and responses from a massive amount of training data. Seq2Seq models [2] or their variants are used for generating responses.

Seq2Seq model works by generating an internal representation of the (sentence, response) pairs in the dataset using a Recursive Neural Network using auto-encoders and then building a decoder to decode the internal representation into a sequence of words that would be the most likely response. The problem faced with this model is a high generation of "I don't know" responses, which is a direct consequence of such occurrences in training text.

The second method focus on solving domain-specific tasks and often rely on carefully limited dialogue parameters, or hand-built templates with state, action and reward signals designed by humans for each new domain, making the paradigm difficult to extend to open-domain scenarios.

There are three problems that conversational dialog systems typically have:

- **Dull and generic responses**: Example,

  A: *How is life?*

  B: *I don't know what you are talking about.*

Another example of this is:

  A: *How old are you?*

  B: *I don't know.*

This can be explained as most systems try to maximize the probability of the next response, and *I don't know.* is an adequate response for most cases. This is further aggravated by the fact that this response shows up a lot in the training data set.

- **Repetitive Responses**: Example,

  A: See you later!

  B: See you later.

  A: See you later.

and so on. Another example is,

  A: Good bye!

  B: Good bye!

  A: Good bye!

and so on.

- **Short-sighted responses**: These systems tend to pick responses that do not increase the length of the conversation or make it interesting or even coherent. An example,

  A: How old are you?

  B: I am 16.

  A: 16?

  B: I don't know what you are talking about?

  A: You don't know what you are saying?

and so on. The first response by agent B is a very close ended response and it is very hard for the agent A to come up with a good response for this. Unfortunately, the results of the response '*I am 16*' does not show up a few turns later. This is the main motivation for using Reinforcement Learning to solve this problem.

2.3 Methodology

The goal of this project would be to build a bot that can have an open dialogue with the user, having three important conversational properties: informativity, coherence and ease of answering.

This project solves the problem by creating a learning system of two agents already trained with supervised seq2seq model and each talking with one another with a goal of maximizing long term rewards, such as increasing the length of conversation and metrics mentioned above.

The main components of the project would be:

- **Training Engine**: An implementation of a LSTM based attention model for this project.
- **Reinforcement Engine**: An implementation of policy gradient method for training the dataset.
- **Chat App**: A web application to have conversations with the chatbot.

2.4 Dataset Description

The original paper uses the OpenSubtitles dataset. For this project we will be using the Cornell Movie Dialogs Corpus instead, partly because it has meta-data and it is of smaller size.

This corpus contains a large metadata-rich collection of fictional conversations extracted from raw movie scripts:

- 220,579 conversational exchanges between 10,292 pairs of movie characters
- involves 9,035 characters from 617 movies

- in total 304,713 utterances
- movie metadata included:
    - genres
    - release year
    - IMDB rating
    - number of IMDB votes
    - IMDB rating
- character metadata included:
    - gender (for 3,774 characters)
    - position on movie credits (3,321 characters)

Dataset Location: https://www.cs.cornell.edu/~cristian/Cornell_Movie-Dialogs_Corpus.html

2.5 References

[1] Deep Reinforcement Learning for Dialogue Generation, Li et al

[2] Sequence to Sequence Learning with Neural Networks - Sutskever et al.

CHAPTER

THREE

DATA WRANGLING

3.1 Data Description

The Cornell Movie Dialog corpus contains a metadata-rich collection of fictional conversations extracted from raw movie scripts:

- 220,579 conversational exchanges between 10,292 pairs of movie characters
- involves 9,035 characters from 617 movies
- in total 304,713 utterances
- movie metadata included:
  - genres
  - release year
  - IMDB rating
  - number of IMDB votes
  - IMDB rating
- character metadata included:
  - gender (for 3,774 characters)
  - position on movie credits (3,321 characters)

B) Files description:

In all files the field separator is " +++$+++ "

- movie_titles_metadata.txt
  - contains information about each movie title
  - fields:
    - movieID,
    - movie title,
    - movie year,
    - IMDB rating,
    - no. IMDB votes,
    - genres in the format ['genre1','genre2',..,'genreN']
- movie_characters_metadata.txt

  - contains information about each movie character
  - fields:
    - characterID
    - character name
    - movieID
    - movie title
    - gender ("?" for unlabeled cases)
    - position in credits ("?" for unlabeled cases)
- movie_lines.txt
  - contains the actual text of each utterance
  - fields:
    - lineID
    - characterID (who uttered this phrase)
    - movieID
    - character name
    - text of the utterance
- movie_conversations.txt
  - the structure of the conversations
  - fields
    - characterID of the first character involved in the conversation
    - characterID of the second character involved in the conversation
    - movieID of the movie in which the conversation occurred
    - list of the utterances that make the conversation, in chronological order: ['lineID1','lineID2',..,'lineIDN'] has to be matched with movie_lines.txt to reconstruct the actual content
- raw_script_urls.txt
  - the urls from which the raw sources were retrieved

3.2 Load Data

We are primarily going to use movie_conversations.txt and movie_lines.txt. First load the movie_lines.txt and extract the lines.

```
# Python imports
import numpy as np
import pandas as pd
import re
```

```
# Load the data
lines = open('../src/data/interim/movie_lines.txt', encoding='utf-8', errors='ignore
↪').read().split('\n')
conv_lines = open('../src/data/interim/movie_conversations.txt', encoding='utf-8',␣
↪errors='ignore').read().split('\n')
```

```
# The sentences that we will be using to train our model.
lines[:10]
```

```
['L1045 +++$+++ u0 +++$+++ m0 +++$+++ BIANCA +++$+++ They do not!',
 'L1044 +++$+++ u2 +++$+++ m0 +++$+++ CAMERON +++$+++ They do to!',
 'L985 +++$+++ u0 +++$+++ m0 +++$+++ BIANCA +++$+++ I hope so.',
 'L984 +++$+++ u2 +++$+++ m0 +++$+++ CAMERON +++$+++ She okay?',
 "L925 +++$+++ u0 +++$+++ m0 +++$+++ BIANCA +++$+++ Let's go.",
 'L924 +++$+++ u2 +++$+++ m0 +++$+++ CAMERON +++$+++ Wow',
 "L872 +++$+++ u0 +++$+++ m0 +++$+++ BIANCA +++$+++ Okay -- you're gonna need␣
↪to learn how to lie.",
 'L871 +++$+++ u2 +++$+++ m0 +++$+++ CAMERON +++$+++ No',
 'L870 +++$+++ u0 +++$+++ m0 +++$+++ BIANCA +++$+++ I\'m kidding.  You know␣
↪how sometimes you just become this "persona"?  And you don\'t know how to␣
↪quit?',
 'L869 +++$+++ u0 +++$+++ m0 +++$+++ BIANCA +++$+++ Like my fear of wearing␣
↪pastels?']
```

```
# The sentences' ids, which will be processed to become our input and target data.
conv_lines[:10]
```

```
["u0 +++$+++ u2 +++$+++ m0 +++$+++ ['L194', 'L195', 'L196', 'L197']",
 "u0 +++$+++ u2 +++$+++ m0 +++$+++ ['L198', 'L199']",
 "u0 +++$+++ u2 +++$+++ m0 +++$+++ ['L200', 'L201', 'L202', 'L203']",
 "u0 +++$+++ u2 +++$+++ m0 +++$+++ ['L204', 'L205', 'L206']",
 "u0 +++$+++ u2 +++$+++ m0 +++$+++ ['L207', 'L208']",
 "u0 +++$+++ u2 +++$+++ m0 +++$+++ ['L271', 'L272', 'L273', 'L274', 'L275']",
 "u0 +++$+++ u2 +++$+++ m0 +++$+++ ['L276', 'L277']",
 "u0 +++$+++ u2 +++$+++ m0 +++$+++ ['L280', 'L281']",
 "u0 +++$+++ u2 +++$+++ m0 +++$+++ ['L363', 'L364']",
 "u0 +++$+++ u2 +++$+++ m0 +++$+++ ['L365', 'L366']"]
```

```
# Create a dictionary to map each line's id with its text
id2line = {}
for line in lines:
    _line = line.split(' +++$+++ ')
    if len(_line) == 5:
        id2line[_line[0]] = _line[4]
```

```
# Add the sentence end marker
id2line['L0'] = '<EOC>'
```

```
# Create a list of all of the conversations' lines' ids.
convs = [ ]
for line in conv_lines[:-1]:
    _line = line.split(' +++$+++ ')[-1][1:-1].replace("'","").replace(" ","")
    convs.append(_line.split(','))
```

```
convs[:10]
```

```
[['L194', 'L195', 'L196', 'L197'],
 ['L198', 'L199'],
 ['L200', 'L201', 'L202', 'L203'],
 ['L204', 'L205', 'L206'],
 ['L207', 'L208'],
 ['L271', 'L272', 'L273', 'L274', 'L275'],
 ['L276', 'L277'],
 ['L280', 'L281'],
 ['L363', 'L364'],
 ['L365', 'L366']]
```

```
# Sort the sentences into questions (inputs) and answers (targets)
questions = []
answers = []

for conv in convs:
    for i in range(len(conv)-1):
        questions.append(id2line[conv[i]])
        answers.append(id2line[conv[i+1]])

    # Add a conversation end marker
    questions.append(id2line[conv[len(conv)-1]])
    answers.append(id2line['L0'])
```

```
# Check if we have loaded the data correctly
limit = 0
for i in range(limit, limit+5):
    print(questions[i])
    print(answers[i])
    print()
```

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

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

Not the hacking and gagging and spitting part.  Please.
Okay... then how 'bout we try out some French cuisine.  Saturday?  Night?

Okay... then how 'bout we try out some French cuisine.  Saturday?  Night?
<EOC>

You're asking me out.  That's so cute. What's your name again?
Forget it.
```

3.3 Data Cleaning

Next we work on cleaning data by expanding English contractions such as “don’t” for “do not”.

```
contractions_dict = {
    "ain't": "am not ",
    "aren't": "are not",
    "'bout": "about",
    "can't": "cannot",
    "can't've": "cannot have",
    "'cause": "because",
    "could've": "could have",
    "couldn't": "could not",
    "couldn't've": "could not have",
    "didn't": "did not",
    "doesn't": "does not",
    "don't": "do not",
    "hadn't": "had not",
    "hadn't've": "had not have",
    "hasn't": "has not",
    "haven't": "have not",
    "he'd": "he would",
    "he'd've": "he would have",
    "he'll": "he will",
    "he'll've": "he will have",
    "he's": "he is",
    "how'd": "how did",
    "how'd'y": "how do you",
    "how'll": "how will",
    "how's": "how is",
    "i'd": "I had",
    "i'd've": "I would have",
    "i'll": "I will",
    "i'll've": "I will have",
    "i'm": "I am",
    "i've": "I have",
    "isn't": "is not",
    "it'd": "it had",
    "ot'd've": "it would have",
    "it'll": "it will",
    "it'll've": "it will have",
    "it's": "it is",
    "let's": "let us",
    "ma'am": "madam",
    "mayn't": "may not",
    "might've": "might have",
    "mightn't": "might not",
    "mightn't've": "might not have",
    "must've": "must have",
    "mustn't": "must not",
    "mustn't've": "must not have",
    "needn't": "need not",
    "needn't've": "need not have",
    "o'clock": "of the clock",
    "oughtn't": "ought not",
    "oughtn't've": "ought not have",
    "shan't": "shall not",
    "sha'n't": "shall not",
    "shan't've": "shall not have",
    "she'd": "she would",
    "she'd've": "she would have",
```

(continues on next page)

(continued from previous page)

```
    "she'll": "she will",
    "she'll've": "she will have",
    "she's": "she is",
    "should've": "should have",
    "shouldn't": "should not",
    "shouldn't've": "should not have",
    "so've": "so have",
    "so's": "so is",
    "that'd": "that had",
    "that'd've": "that would have",
    "that's": "that is",
    "there'd": "there would",
    "there'd've": "there would have",
    "there's": "there is",
    "they'd": "they would",
    "they'd've": "they would have",
    "they'll": "they will",
    "they'll've": "they will have",
    "they're": "they are",
    "they've": "they have",
    "to've": "to have",
    "wasn't": "was not",
    "we'd": "we would",
    "we'd've": "we would have",
    "we'll": "we will",
    "we'll've": "we will have",
    "we're": "we are",
    "we've": "we have",
    "weren't": "were not",
    "what'll": "what will",
    "what'll've": "what have",
    "what're": "what are",
    "what's": "what is",
    "what've": "what have",
    "when's": "when is",
    "when've": "when have",
    "where'd": "where did",
    "where's": "where is",
    "where've": "where have",
    "who'll": "who will",
    "who'll've": "who will have",
    "who's": "who is",
    "who've": "who have",
    "why's": "why is",
    "why've": "why have",
    "will've": "will have",
    "won't": "will not",
    "won't've": "will not have",
    "would've": "would have",
    "wouldn't": "would not",
    "wouldn't've": "would not have",
    "y'all": "you all",
    "y'all'd": "you all would",
    "y'all'd've": "you all would have",
    "y'all're": "you all are",
    "y'all've": "you all have",
    "you'd": "you had",
```

(continues on next page)

(continued from previous page)

```
    "you'd've": "you would have",
    "you'll": "you shall",
    "you'll've": "you shall have",
    "you're": "you are",
    "you've": "you have"
}

contractions_re = re.compile('(%s)' % '|'.join(contractions_dict.keys()), re.
↪IGNORECASE)

def expand_contractions(s, contractions_dict=contractions_dict):
    def replace(match):
        return contractions_dict[match.group(0).lower()]
    text = contractions_re.sub(replace, s)
    return re.sub(r"[-()\"#/@;:<>{}`+=~|.!?,]", "", text)


expand_contractions('you\'re great!')
```

```
'you are great'
```

```
# Clean the data
clean_questions = []
for question in questions:
    clean_questions.append(expand_contractions(question))

clean_answers = []
for answer in answers:
    if answer != '<EOC>':
        clean_answers.append(expand_contractions(answer))
    else:
        clean_answers.append(answer)
```

```
# Take a look at some of the data to ensure that it has been cleaned well.
limit = 0
for i in range(limit, limit+5):
    print(clean_questions[i])
    print(clean_answers[i])
    print()
```

```
Can we make this quick  Roxanne Korrine and Andrew Barrett are having an incredibly␣
↪horrendous public break up on the quad  Again
Well I thought we would start with pronunciation if that is okay with you

Well I thought we would start with pronunciation if that is okay with you
Not the hacking and gagging and spitting part  Please

Not the hacking and gagging and spitting part  Please
Okay then how about we try out some French cuisine  Saturday  Night

Okay then how about we try out some French cuisine  Saturday  Night
<EOC>

you are asking me out  that is so cute what is your name again
Forget it
```

We now have a clean Question and Answer datasets that are used for further processing and gathering further insights.

```
# Save the files
conv_final = pd.DataFrame({'question': clean_questions, 'answer': clean_answers})
conv_final.to_csv('../src/data/processed/movie_qa.txt')
```

```
conv_final.head()
```

CHAPTER

FOUR

DATA INSIGHTS

Load the already cleaned question answer pairs generated.

```
# Python Imports
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

```
# Load the QA dataset
qa = pd.read_csv('../src/data/processed/movie_qa.txt', index_col=0)
qa.head()
```

4.1 Response Length

We first look at the how long a typical response is. We start by filtering out End of conversation markers 'EOC'

```
# Filter out EOC
qa_no_eoc = qa[qa.answer != '<EOC>']['answer']

# Get lengths for each value
resp_len = qa_no_eoc.apply(lambda x: len(str(x).split()))

resp_len.describe()
```

```
count    221616.000000
mean         11.083054
std          12.630780
min           0.000000
25%           4.000000
50%           7.000000
75%          14.000000
max         556.000000
Name: answer, dtype: float64
```

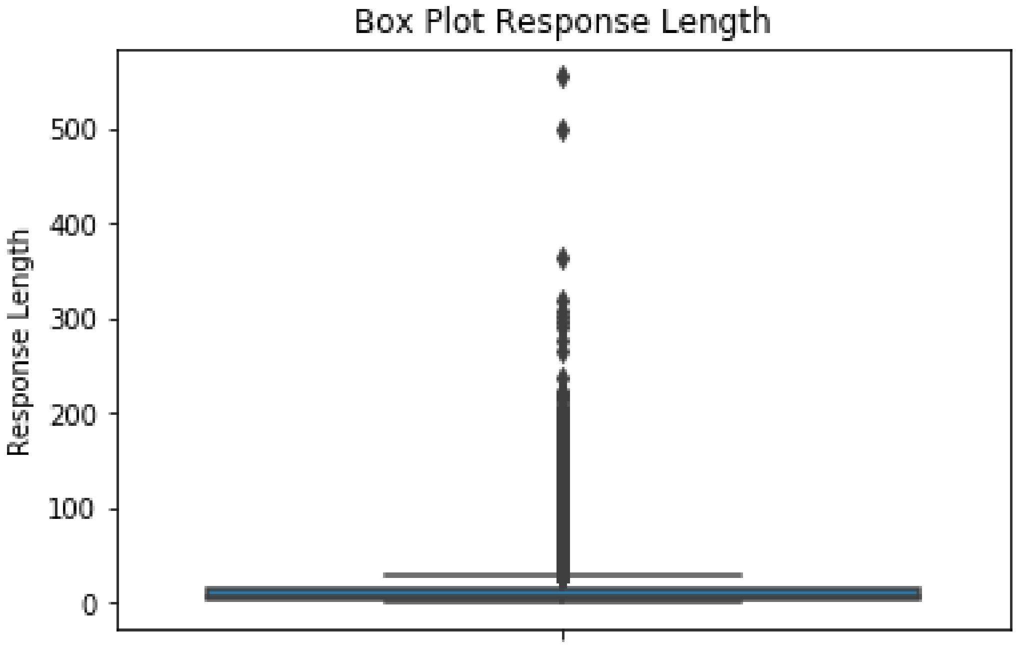
We see that the median response length is 7 words, but the distribution is skewed by presence of extreme outliers. Lets look at the box plot for verification.

```
# Box Plot
_ = sns.boxplot(resp_len, orient='v')
_ = plt.ylabel('Response Length')
```

(continues on next page)

(continued from previous page)

```
_ = plt.xlabel('')
_ = plt.title('Box Plot Response Length')
```



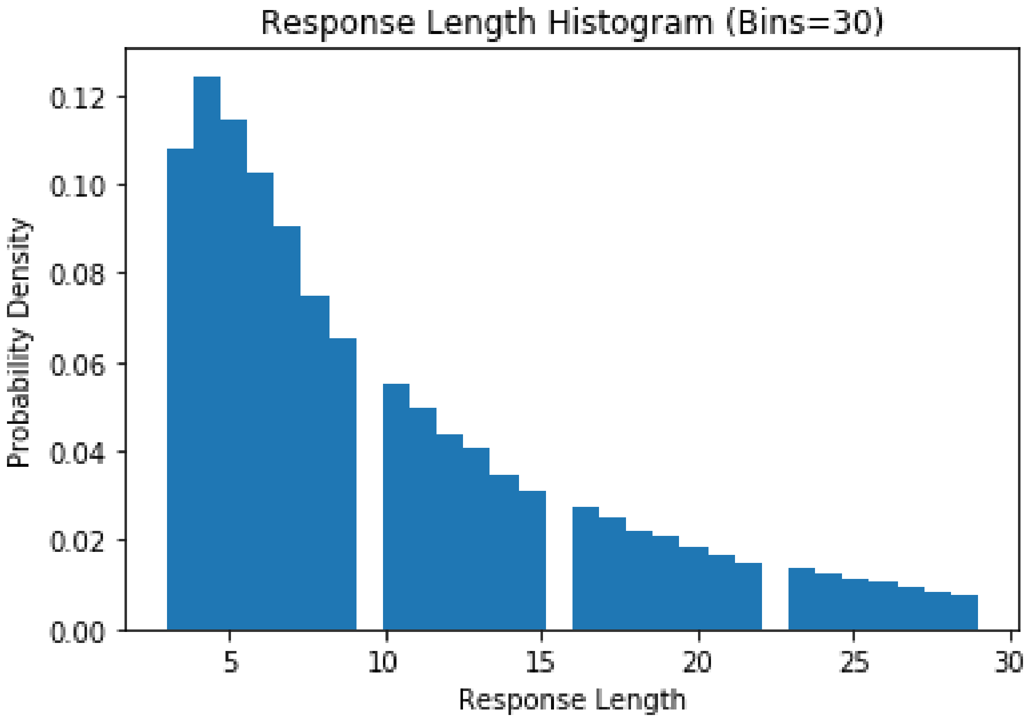
We remove the outliers by using the 1.5 times IQR thumb rule, so we filter out values outside range (2, 30)

```
filt = np.logical_and(resp_len > 2, resp_len < 30)
resp_len_filt = resp_len[filt]

resp_len_filt.describe()
```

```
count    173926.000000
mean          9.990921
std           6.379758
min           3.000000
25%           5.000000
50%           8.000000
75%          13.000000
max          29.000000
Name: answer, dtype: float64
```

```
_ = plt.hist(resp_len_filt, density=True, bins=30)
_ = plt.xlabel('Response Length')
_ = plt.ylabel('Probability Density')
_ = plt.title('Response Length Histogram (Bins=30)')
```



We see that response length goes down exponentially. People prefer brevity in responses over longer sentences. This is an important insight for training batches, as the longer input sequences lead to padding in shorter sequences increasing cpu/memory usage.

4.2 Conversation Length

Next we look at how long do conversations last. This can be easily inferred by looking at the eoc_markers.

```
eoc_marks = sorted(list(set(qa.answer.index.values) - set(qa_no_eoc.index.values)))

conv_len_arr = [eoc_marks[0]]
for i in range(len(eoc_marks)-1):
    conv_len_arr.append(eoc_marks[i+1] - eoc_marks[i] - 1)

conv_len = pd.Series(conv_len_arr)
conv_len.describe()
```

```
count    83097.000000
mean         2.666955
std          2.891798
min          1.000000
25%          1.000000
50%          2.000000
75%          3.000000
max         88.000000
dtype: float64
```

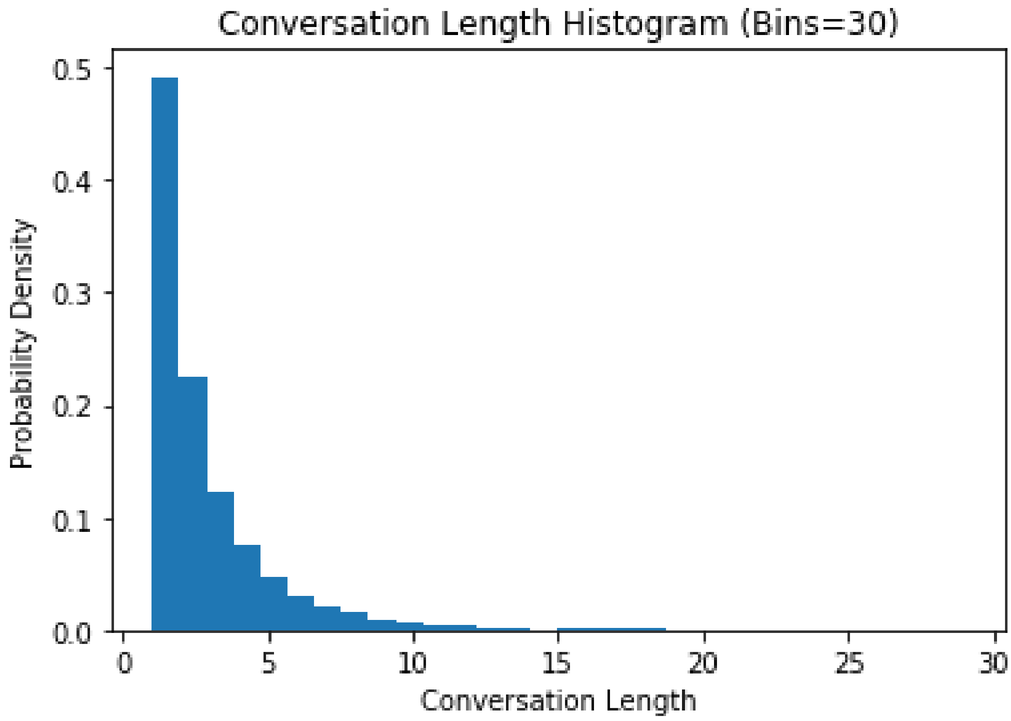
We again see extreme values in this distribution. 75% of the conversations are less than 3 turns. We again filter for extreme values in range (2,30) and check how many conversations are dropped.

```
filt = np.logical_and(conv_len > 0, conv_len < 30)
conv_len_filt = conv_len[filt]

conv_len_filt.describe()
print('% kept: {0:.2%}'.format(conv_len_filt.count() / conv_len.count()))
```

```
% kept: 99.92%
```

```
_ = plt.hist(conv_len_filt, density=True, bins=30)
_ = plt.xlabel('Conversation Length')
_ = plt.ylabel('Probability Density')
_ = plt.title('Conversation Length Histogram (Bins=30)')
```



We see that in naturally occuring conversations, most conversations end in 2-3 turns. This gives us an insight that we can limit the number of turns to some small number, such as 5, for evaluation of the model.

4.3 Most common Responses

Next we look at the most common responses generated in the data.

```
from collections import Counter

resp_cnt = Counter(qa_no_eoc)

resp_cnt.most_common(10)
```

```
[('What', 1532),
 ('Yes', 1484),
```

(continues on next page)

(continued from previous page)

```
 ('No', 1423),
 ('Yeah', 1117),
 ('Why', 502),
 ('I do not know', 397),
 ('Okay', 357),
 ('Sure', 277),
 ('Oh', 266),
 ('Thank you', 246)]
```

We see that most common responses are responses to questions and most act as conversation stoppers (with the exception of ‘Why?’). We expect them to see again in the list of dull responses next.

```
resp_dist = pd.Series([i for i in resp_cnt.values()])
resp_dist.describe()
```

```
count    189270.000000
mean          1.170899
std           7.122922
min           1.000000
25%           1.000000
50%           1.000000
75%           1.000000
max        1532.000000
dtype: float64
```

We see that most responses occur in the corpus only once. This should also result in model predicting unique responses.

4.4 Dull responses

Identifying dull responses is an important functionality for deciding on rewards in Reinforcement learning module. We look at all the questions that result in end of conversation.

```
# Questions generating a end of conversation response
qa_dull = qa[qa.answer == '<EOC>']['question']

dull_cnt = Counter(qa_dull)

dull_cnt.most_common(10)
```

```
[('No', 360),
 ('Yes', 356),
 ('Yeah', 307),
 ('What', 234),
 ('Okay', 211),
 ('Sure', 154),
 ('I do not know', 145),
 ('Thank you', 136),
 ('Oh', 120),
 ('Right', 104)]
```

This is again expected as these responses are stop responses that typically end a conversation.

CHAPTER

FIVE

MODEL DESCRIPTION

5.1 Recurrent Neural Networks

Recurrent Neural Networks (RNN) are a particular kind of artificial neural networks that are specialized in extracting information from sequences. Unlike simple feedforward neural networks, a neuron's activation is also dependent from its previous activations. It allows the model to capture correlations between the different inputs in the sequence.In this project, we implement a sequence to sequence (Seq2Seq) model using two independent RNNs, an encoder, and a decoder.

However, this type of architecture can be very challenging to train, partly because it is much more prone to exploding and vanishing gradients. These problems can be overcome by choosing more stable activation functions like ReLU or tanh and by using more sophisticated cells like LSTM and GRU, which involve more parameters and computations than the vanilla RNN cell but are designed to avoid vanishing gradients and capture long range dependencies. In this project, we make use of the LSTM cell and gradient clipping to solve these problems.

In a regular RNN, at time-step t, the cell state h_t is computed based on its own input and the cell state h_{t-1} that encapsulates some information from the precedent inputs: > h_t = $f(W^{hx}x_t + W^{hh}h_{t-1})$

In this case f is the activation function. The cell output for this time-step is then computed using a third series of parameters W^{hy}: > y_t = $W^{hy}h_t$

With LSTM or GRU cells, the core principle is the same, but these type of cells additionally use additional sigmoid units called gates that allow to forget information or expose only particular part of the cell state to the next step with a specified probability.

5.2 Sequence to Sequence Models

The principle of Seq2Seq architecture is to encode information on an input sequence x_1 to x_T and to use this condensed representation known as context vector to generate a new sequence y_1 to $y_{T'}$. Each output y_t is based on previous outputs and the context vector : > $p(A|Q)$ = $p(y_1, y_2, ..., y_{T'}|x_1, x_2, ..., x_T)$ = $\prod_{t=1}^{T'} a_i \; p(y_t|c_t, y_1, ..., y_{t-1})$

The loss l is the negative log-probability of the answer sequence $A = [y_1, ..., y_T]$ given the question sequence $Q = [x_1, ..., x_T]$, averaged over the number N of (Q, A) pairs in a minibatch : > l = $-\frac{1}{N}\sum_{(Q,A)\in N} p(A|Q)$

Multiple layers of RNN cells can be stacked over each other to increase the model capacity like in a regular feedforward neural network.

The following figure presents a Seq2Seq model with a two layers encoder and a two layers decoder:

This project uses a 512 cell RNN cell with two layers. Different weights are used for encoder and decoder cells: > $W^{h*}_{encoder} \neq W^{h*}_{decoder}$

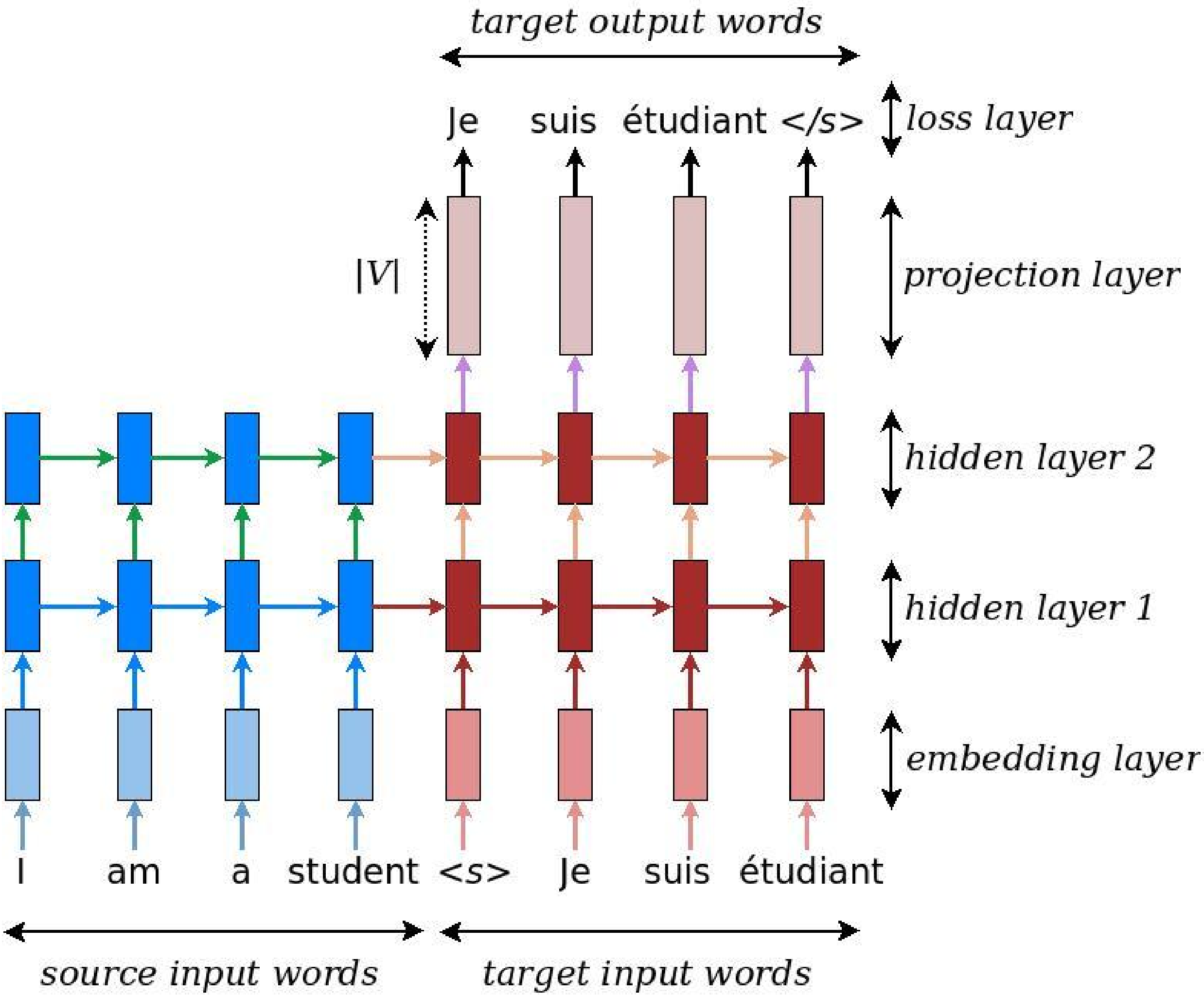


Fig. 1: alt text

At the bottom layer, the encoder and decoder RNNs receive as input the following: * first, the source sentence, * then a boundary marker '<GO>' which indicates the transition from the encoding to the decoding mode, and the target sentence.

For training, we will feed the system with the following tensors, which are in time-major format and contain word indices: * encoder_inputs [max_encoder_sequence_length, batch_size]: source input words. The input word array is reversed before feeding to the network. * decoder_inputs [max_decoder_sequence_length, batch_size]: target input words. * decoder_outputs [max_decoder_sequence_length, batch_size]: target output words, these are decoder_inputs shifted to the left by one time step with an end-of-sentence tag '<S>' appended on the right.

Here for efficiency, we train with multiple sentences (batch_size = 30) at one go. This is a hyper-parameter that is tuned while training.

5.2.1 Embeddings

Given the categorical nature of words, the model must first look up the source and target embeddings to retrieve the corresponding word representations. For this embedding layer to work, a vocabulary is first chosen for each language. Usually, a vocabulary size V is selected, and only the most frequent V words are treated as unique. All other words are converted to an "unknown" token and all get the same embedding. The embedding weights, one set per language, are usually learned during training.

Note that one can choose to initialize embedding weights with pretrained word representations such as word2vec or Glove vectors. In general, given a large amount of training data we can learn these embeddings from scratch, as we do in this project.

5.2.2 Encoder-Decoder

Note that sentences have different lengths to avoid wasting computation, we tell dynamic_rnn the max source sentence length for the batch through sequence_length param and use padding to pad smaller sentences in the batch. The encoder is trained with questions as the input while the decoder is trained with answers as the input. The decoder also needs to have access to the last encoder state information, and one simple way to achieve that is to initialize it with the last hidden state of the encoder.

5.2.3 Loss

The output of the decoder layer is the unnormalized probabilities called 'logits'. The loss is the cross entropy loss of the predicted output with a expected output of class 1 of N. This sum of all logits is divided by batch size to make hyper parameters invariant to batch size.

5.2.4 Optimization

One of the important steps in training RNNs is gradient clipping. Here, we clip by the global norm. The max value, max_gradient_norm, is often set to a value like 5 or 1. We select Adam optimizer and a starting learning rate in the range 0.0001 to 0.001; which decreases as training progresses.

5.3 Attention Mechanism

The first drawback with the previous model is that all the relevant information to decode is fed through a fixed size vector. When the encoding sequence is long, it often fails to capture the complex semantic relations between words entirely. On the other hand, if the fixed size vector is too large for the encoding sequence, this may cause overfitting problem forcing the encoder to learn some noise.

Furthermore, words occurring at the beginning of the encoding sentence may contain information for predicting the first decoding outputs. It is often complex for the model to capture long term dependencies, and there is no guarantee that it will learn to handle these correctly. This problem can be partially solved by using a Bi-Directional RNN (BD-RNN) as the encoder. A BD-RNN will encode its input twice b y passing over the input in both directions. We use a *Bidirectional Encoder* in this project. This model allows for hidden representations at each timestep to capture both future and past context information from the input sequence.

The basic idea of attention is that instead of attempting to learn a single vector representation for each sentence, we keep around vectors for every word in the input sentence, and reference these vectors at each decoding step.

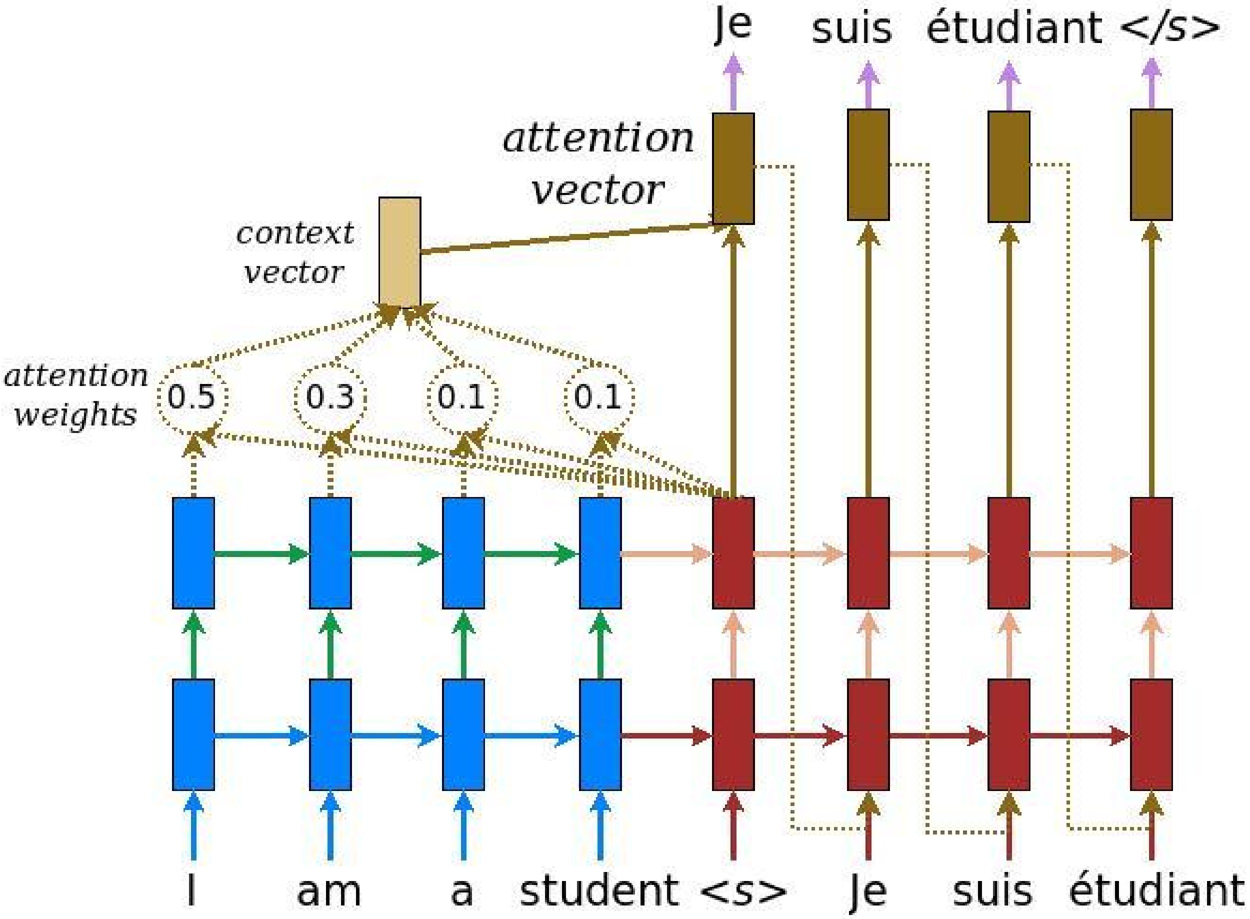


Fig. 2: alt text

It consists of the following stages: * The current target hidden state is compared with all source states to derive attention weights. * Based on the attention weights we compute a context vector as the weighted average of the source states. * Combine the context vector with the current target hidden state to yield the final attention vector * The attention vector is fed as an input to the next time step *(input feeding)*.

The first three steps can be summarized by the equations below: > $\alpha_{ts} = \frac{exp(score(h_t,\overline{h_s}))}{\sum_{r=1}^{S} exp(score(h_t,\overline{h_r}))}$

$c_t = \sum_s \alpha_{ts}\overline{h_s}$

$a_t = f(c_t, h_t) = \tanh(W_c[c_t; h_t])$

Here, the function score is used to compared the target hidden state h_t with each of the source hidden states $\overline{h}_s$, and the result is normalized to produced attention weights (a distribution over source positions). There are various choices of the scoring function; popular scoring functions include the multiplicative and additive forms. We use the additive form (Bahdanau) using *tanh*. > $score(h_t, \overline{h}_s)$ = $v_a^T tanh(W_1 h_t + W_2 \overline{h}_s)$

Once computed, the attention vector a_t is used to derive the softmax logit and loss. This is similar to the target hidden state at the top layer of a vanilla seq2seq model.

We also use a **dropout** with a keep probability of 0.75%

This model is used to train word embeddings and then further used to train the reinforcement learning model using policy gradient method.

5.4 References

- Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015.Neural machine translation by jointly learning to align and translate. ICLR.
- Minh-Thang Luong, Hieu Pham, and Christopher D Manning. 2015.Effective approaches to attention-based neural machine translation. EMNLP.
- Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014.Sequence to sequence learning with neural networks. NIPS.

CHAPTER

SIX

REINFORCEMENT LEARNING MODEL

Sequence to Sequence Models are trained based on Maximum Likelihood Principle, which gives the most probable answer. The model lacks some belief concerning the appearance of a token in the vocabulary, hence a model trained with maximum likelihood will tend to output short general answers that are very common in the vocabulary, like yes, no, maybe, which is a huge problem for conversation diversity. This lack of diversity leads to repetive and short-sighted 'stop' answers that end the conversation.

Reinforcement learning model is one non maximum likelihood based approach to solve this problem of dull, repetitive answers.

6.1 Model Setup

The learning system consists of two agents. We use p to denote sentences generated from the first agent and q to denote sentences from the second. The two agents take turns talking with each other.

A dialogue can be represented as an alternating sequence of sentences generated by the two agents: $p_1, q_1, p_2, q_2, ..., p_i, q_i$. We view the generated sentences as actions that are taken according to a policy defined by an encoder-decoder recurrent neural network language model. The parameters of the network are optimized to maximize the expected future reward using policy gradient search method.

Policy gradient methods are more appropriate for this scenario than Q-learning, because the encoder-decoder RNN is initialized using MLE parameters that already produce plausible responses, before changing the objective and tuning towards a policy that maximizes long-term reward.

Q-learning, on the other hand, directly estimates the future expected reward of each action, which can differ from the MLE objective by orders of magnitude, thus making MLE parameters inappropriate for initialization.

The components (states, actions, reward, etc.) of this sequential decision problem are summarized as follows:

6.2 State

The state is denoted by the previous dialog turn, either p_i or q_i. A vector representation of this dialog is retrieved by feeding it into the LSTM Encoder part of the Seq to Seq model.

6.3 Action

An action a is the dialogue utterance to generate. The action space is infinite since arbitrary-length sequences can be generated.

6.4 Policy

A policy takes the form of an LSTM encoder-decoder > pRL(pi+1|qi)) and is defined by its parameters.

Note that a stochastic representation of the policy (a probability distribution over actions given states) is used. A deterministic policy would result in a discontinuous objective that is difficult to optimize using gradient-based methods.

6.5 Reward

r denotes the reward obtained for each action. In this subsection, we discuss major factors that contribute to the success of a dialogue and describe how approximations to these factors can be operationalized in computable reward functions.

6.5.1 Ease of Answering

A turn generated by a machine should be easy to respond to. This aspect of a turn is related to its forward-looking function: the constraints a turn places on the next turn.

A proposed measure for the ease of answering for a generated turn is using the negative log likelihood of responding to that utterance with a dull response. A list S of dull responses is manually constructed a list of dull responses consisting turns such as > "I don't know what you are talking about",

"I have no idea",

that have bee found occur very frequently in SEQ2SEQ models of conversations.

The reward function is given as follows: > $r_1 = -\frac{1}{N_S} \sum_{s \in S} \frac{1}{N_s} log p_{seq2seq}(s|a)$

where: > N_S is the cardinality of the turn in S

N_s is the cardinality of tokens in common with $s \in S$

$p_{seq2seq}$ represents the MLE output of the Seq2Seq model

6.5.2 Information flow

We want each agent to contribute new information at each turn to keep the dialogue moving and avoid repetitive sequences. We therefore penalize semantic similarity between consecutive turns from the same agent.

Let h_{p_i} and $h_{p_{i+1}}$ denote representations obtained from the encoder for two consecutive turns p_i and p_{i+1}.

The reward is given by the negative log of the cosine similarity between them: > r_2 = - log cos(h_{p_i}, $h_{p_{i+1}}$)

6.5.3 Semantic Coherance

We also need to measure the adequacy of responses to avoid situations in which the generated replies are highly rewarded but are ungrammatical or not coherent. We therefore consider the mutual information between the action a and previous turns in the history to ensure the generated responses are coherent and appropriate: > $r3$ = $\frac{1}{N_a} log p_{seq2seq}(a|q_i)$ + $\frac{1}{N_{q_i}} log p^{backward}_{seq2seq}(q_i|a)$

where: > $log p_{seq2seq}(a|q_i)$ is the probability of generating a given q_i

$log p^{backward}_{seq2seq}(q_i|a)$ is the probability of q_i being the preceding char given that a follows.

The second probability is the output of the backward cell of the Bidirectional RNN used for training. Both the above probabilities are scaled with respect to their length.

The final reward is a weighted sum of the above probabilities with weights (0.25, 0.25, 0.5).

6.6 Curriculum Learning Strategy

A curriculum Learning strategy is employed in which we begin by simulating the dialogue for 2 turns, and gradually increase the number of simulated turns. We generate 5 turns at most, as the number of candidates to examine grows exponentially in the size of candidate list. Five candidate responses are generated at each step of the simulation.

We use Beam Search Decoder to generate a candidate list of responses/actions a and track rewards across the conversation.

6.7 Policy Gradient Search

The key idea underlying policy gradients is to push up the probabilities of actions that lead to higher return, and push down the probabilities of actions that lead to lower return, until you arrive at the optimal policy.

The goal of training is to find the parameters of the agent that maximize the expected reward. We define our loss L given parameters θ as the negative expected reward: > $L_\theta = \sum_{w_1^g,...,w_T^g} p_\theta(w_1^g, ..., w_T^g) r(w_1^g, ..., w_T^g)$

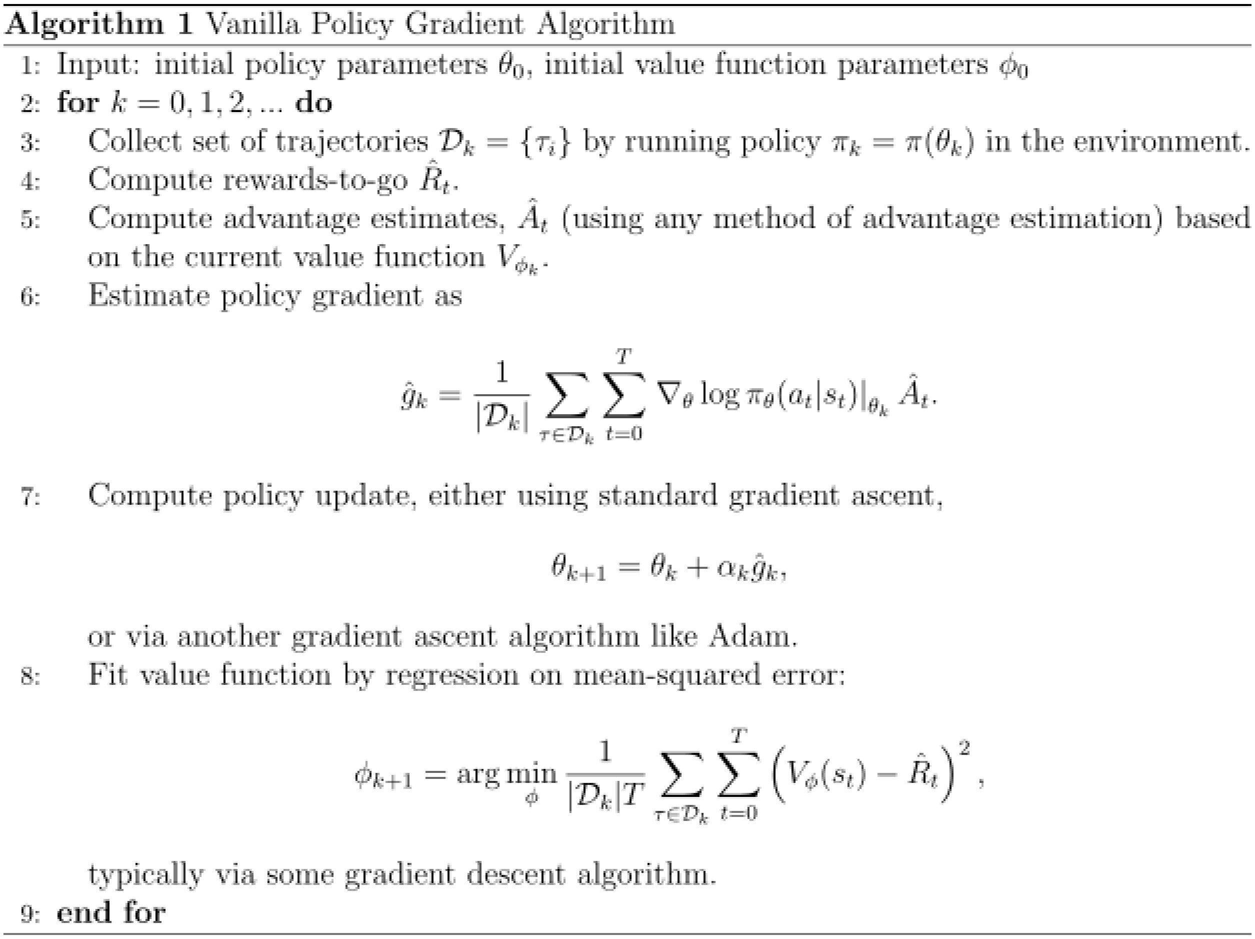
which is simple sum product over probability of picking an action with its associated reward.

The partial derivative of this loss function over the outputs of the softmax classifier is the same as that of the multiclass logistic regression classifier, which is the difference between the prediction and the actual 1-of-N representation of the target word.

$$\frac{\partial L_\theta}{\partial o_t} = \sum_{w_1^g,...,w_T^g} (p_\theta(w_1^g, ..., w_T^g) - 1)(r(w_1^g, ..., w_T^g) - \bar{r}_{t+1})$$

where $\bar{r}_{t+1}$ is the expected reward.

The general algorithm for policy gradient is given as:



Algorithm 1 Vanilla Policy Gradient Algorithm

```
Input: initial policy parameters $\theta_0$, initial value function parameters $\phi_0$
for $k = 0, 1, 2, ...$ do
    Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
    Compute rewards-to-go $\hat{R}_t$.
    Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function $V_{\phi_k}$.
    Estimate policy gradient as
        $\hat{g}_k = \frac{1}{|\mathcal{D}_k|} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)|_{\theta_k} \hat{A}_t.$
    Compute policy update, either using standard gradient ascent,
        $\theta_{k+1} = \theta_k + \alpha_k \hat{g}_k,$
    or via another gradient ascent algorithm like Adam.
    Fit value function by regression on mean-squared error:
        $\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left( V_\phi(s_t) - \hat{R}_t \right)^2,$
    typically via some gradient descent algorithm.
end for
```

Policy Gradient Method trains a stochastic policy in an on-policy way. This means that it explores by sampling actions according to the latest version of its stochastic policy. The amount of randomness in action selection depends on both initial conditions and the training procedure. Over the course of training, the policy typically becomes progressively less random, as the update rule encourages it to exploit rewards that it has already found. This may cause the policy to get trapped in local optima.