Abusive and Hate Speech Tweets Detection with Text Generation

Abhishek Nalamothu
Wright State University

Follow this and additional works at: https://corescholar.libraries.wright.edu/etd_all

Part of the Computer Engineering Commons, and the Computer Sciences Commons

Repository Citation

This Thesis is brought to you for free and open access by the Theses and Dissertations at CORE Scholar. It has been accepted for inclusion in Browse all Theses and Dissertations by an authorized administrator of CORE Scholar. For more information, please contact library-corescholar@wright.edu.
ABUSIVE AND HATE SPEECH TWEETS DETECTION WITH TEXT GENERATION

A Thesis submitted in partial fulfillment of the requirements for the degree of Master of Science

by

ABHISHEK NALAMOTHU
B.E., Saveetha School of Engineering, Saveetha University, India, 2016

2019
Wright State University
I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPERVISION BY Abhishek Nalamothu ENTITLED Abusive and Hate Speech Tweets Detection with Text Generation BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF Master of Science.

____________________________
Amit Sheth, Ph.D.
Thesis Director

____________________________
Mateen M. Rizki, Ph.D.
Chair, Department of Computer Science and Engineering

Committee on Final Examination

____________________________
Amit Sheth, Ph.D.

____________________________
Keke Chen, Ph.D.

____________________________
Valerie L. Shalin, Ph.D.

____________________________
Barry Milligan, Ph.D.
Interim Dean of the Graduate School
ABSTRACT

Nalamothu, Abhishek. M.S., Department of Computer Science and Engineering, Wright State University, 2019. Abusive and Hate Speech Tweets Detection with Text Generation.

According to a Pew Research study, 41% of Americans have personally experienced online harassment and two-thirds of Americans have witnessed harassment in 2017. Hence, online harassment detection is vital for securing and sustaining the popularity and viability of online social networks. Machine learning techniques play a crucial role in automatic harassment detection. One of the challenges of using supervised approaches is training data imbalance. Existing text generation techniques can help augment the training data, but they are still inadequate and ineffective. This research explores the role of domain-specific knowledge to complement the limited training data available for training a text generator.

We conduct domain-specific text generation by combining inverse reinforcement learning (IRL) with domain-specific knowledge. Our approach includes two adversarial nets, a text generator and a Reward Approximator (RA). The objective of the text generator is to generate domain-specific text that is hard to discriminate from real-world domain-specific text. The objective of the reward approximator is to discriminate the generated domain-specific text from real-world text. During adversarial training, the generator and the RA play a mini-max game and try to arrive at a win-win state. Ultimately, augmenting diversified and semantically meaningful, generated domain-specific data to the existing dataset improves detection of domain-specific text. In addition to developing the Generative Adversarial Network-based framework, we also present a novel evaluation that uses variants of the BLEU metric to measure the diversity of generated text; uses perplexity and cosine similarity to measure the quality of the generated text. Experimental results show that the proposed framework outperforms a previous baseline (IRL without domain knowledge) on harassment (i.e., Abusive and Hate speech) tweet generation. Additionally, the generated tweets effectively augment the training data for online abusive and hate speech detection.
(tweet classification) resulting in a 9% accuracy improvement in classification using the augmented training set compared to the existing training set.
# Contents

1 Introduction 1

2 Preliminaries 5

2.1 Word2vec .................................................. 5
   2.1.1 Skip-Gram model Learning Objective ................ 7
   2.1.2 Skip-Gram model architecture ...................... 7

2.2 Language Modeling (LM) .................................. 11

2.3 Reinforcement Learning (RL) .............................. 13
   2.3.1 Exploration and exploitation trade off .......... 15
   2.3.2 Approaches to Reinforcement Learning .......... 16

2.4 Inverse Reinforcement Learning (IRL) ................... 20

2.5 Support Vector Machine (SVM) Classification .......... 21

3 Related Work 23

3.1 Recurrent Neural Networks ............................... 24
   3.1.1 Recurrent Neural Networks Training .............. 27
   3.1.2 Problems with RNN Training ..................... 33

3.2 Long Short Term Memory (LSTM) ........................ 41

3.3 Text Generation using Auto regressive models ........ 48
   3.3.1 Exposure bias problem .......................... 50

3.4 Generative Adversarial networks ......................... 50

3.5 Reinforcement learning based algorithms ............... 54
   3.5.1 SeqGAN ........................................ 54
   3.5.2 Rank GAN ....................................... 59

4 Research Contribution to Text Generation 64

4.1 Text Generation with IRL ................................ 64
   4.1.1 Reward Approximator ............................. 65
   4.1.2 Text Generator ................................ 68

5 Data & Experimental setting 74

5.1 Abusive and Hate Speech Tweets ......................... 74
5.2 Experimental setting .................................................. 76

6 Evaluation ................................................................. 79
  6.1 New Evaluation Measures ........................................... 79
     6.1.1 BiLingual Evaluation Understudy (BLEU) ................. 79
     6.1.2 Normalized perplexity ....................................... 81
     6.1.3 Max of Cosine similarity .................................. 82

7 Results & Conclusion .................................................. 84
  7.1 Experimental results ............................................... 84
  7.2 Conclusion & Future Work ....................................... 87

Bibliography ............................................................. 89
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Example of a size 5 window</td>
<td>7</td>
</tr>
<tr>
<td>2.2</td>
<td>The Skip-Gram model architecture</td>
<td>8</td>
</tr>
<tr>
<td>2.3</td>
<td>Mapping word embeddings with Word2vec</td>
<td>10</td>
</tr>
<tr>
<td>2.4</td>
<td>The basic idea of reinforcement learning and elements involved in it</td>
<td>13</td>
</tr>
<tr>
<td>2.5</td>
<td>Reinforcement learning demonstration with Maze Game Environment</td>
<td>15</td>
</tr>
<tr>
<td>2.6</td>
<td>Feeding pong game frames to a policy network</td>
<td>18</td>
</tr>
<tr>
<td>2.7</td>
<td>Value based RL approach on Grid game</td>
<td>19</td>
</tr>
<tr>
<td>2.8</td>
<td>Sparse Reward failure of RL algorithm</td>
<td>20</td>
</tr>
<tr>
<td>2.9</td>
<td>Decision boundary of an SVM classifier with separating hyper-planes on both sides</td>
<td>22</td>
</tr>
<tr>
<td>3.1</td>
<td>State of the art Text generation models</td>
<td>24</td>
</tr>
<tr>
<td>3.2</td>
<td>Recurrent neural network-folded form</td>
<td>25</td>
</tr>
<tr>
<td>3.3</td>
<td>Recurrent neural network - unfolded form</td>
<td>26</td>
</tr>
<tr>
<td>3.4</td>
<td>The left plot represents Cross entropy vs Cost function. The right plot represents the objective i.e., Minimizing the cost.</td>
<td>29</td>
</tr>
<tr>
<td>3.5</td>
<td>Recurrent neural network - training with example</td>
<td>30</td>
</tr>
<tr>
<td>3.6</td>
<td>A recurrent neural network with one hot encoded targets</td>
<td>30</td>
</tr>
<tr>
<td>3.7</td>
<td>A recurrent neural network - Back Propagation through time</td>
<td>33</td>
</tr>
<tr>
<td>3.8</td>
<td>A recurrent neural network - Long term dependency problem.</td>
<td>34</td>
</tr>
<tr>
<td>3.9</td>
<td>Sigmoid f(x) and it’s derivative $\frac{\partial f}{\partial x}$</td>
<td>37</td>
</tr>
<tr>
<td>3.10</td>
<td>Tanh f(x) and it’s derivative $\frac{\partial f}{\partial x}$</td>
<td>38</td>
</tr>
<tr>
<td>3.11</td>
<td>Recurrent Neural network - Vanishing Gradient problem</td>
<td>39</td>
</tr>
<tr>
<td>3.12</td>
<td>Recurrent Neural network - Exploding Gradient problem</td>
<td>40</td>
</tr>
<tr>
<td>3.13</td>
<td>Gradient clipping</td>
<td>41</td>
</tr>
<tr>
<td>3.14</td>
<td>ReLU activation function $f(x)$ and it’s derivative $\frac{\partial f}{\partial x}$</td>
<td>42</td>
</tr>
<tr>
<td>3.15</td>
<td>Standard Vanilla RNNs.</td>
<td>43</td>
</tr>
<tr>
<td>3.16</td>
<td>LSTM</td>
<td>43</td>
</tr>
<tr>
<td>3.17</td>
<td>Notations used in the LSTM.</td>
<td>44</td>
</tr>
<tr>
<td>3.18</td>
<td>LSTM - Memory (cell) value propagation</td>
<td>44</td>
</tr>
<tr>
<td>3.19</td>
<td>Working Mechanism of LSTM with example.</td>
<td>45</td>
</tr>
<tr>
<td>3.20</td>
<td>LSTM - Forget fate ($f^t$)</td>
<td>46</td>
</tr>
<tr>
<td>3.21</td>
<td>LSTM - input gate ($i^t$)</td>
<td>46</td>
</tr>
</tbody>
</table>
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Passing of memory value through the chain of repeated layers.</td>
<td>44</td>
</tr>
<tr>
<td>3.2</td>
<td>Objective of the generator $G$ and the discriminator $D$</td>
<td>51</td>
</tr>
<tr>
<td>5.1</td>
<td>Example tweets from each class</td>
<td>75</td>
</tr>
<tr>
<td>5.2</td>
<td>Hyper-parameters configurations</td>
<td>77</td>
</tr>
<tr>
<td>6.1</td>
<td>BLEU example</td>
<td>79</td>
</tr>
<tr>
<td>6.2</td>
<td>BLEU score calculation for candidate w.r.t reference sentence (example from the above table)</td>
<td>80</td>
</tr>
<tr>
<td>6.3</td>
<td>Configuration of $BLEU_{div}$ and $BLEU_s$</td>
<td>80</td>
</tr>
<tr>
<td>6.4</td>
<td>Experimental results between IRL (previous base line) and $IRL_{cs}$ (our model)</td>
<td>82</td>
</tr>
<tr>
<td>6.5</td>
<td>Experimental results of MCS</td>
<td>83</td>
</tr>
<tr>
<td>7.1</td>
<td>Examples from training data, and generated data from different techniques</td>
<td>84</td>
</tr>
<tr>
<td>7.2</td>
<td>SVM classification results on different techniques</td>
<td>86</td>
</tr>
<tr>
<td>7.3</td>
<td>Diversity of generated sentences</td>
<td>86</td>
</tr>
</tbody>
</table>
Acknowledgment

Guru (Teacher) = "GU" means darkness, "RU" means dispeller. Thanks to all my Gurus (Advisor and committee).

Firstly, I would like to express my sincere gratitude to my thesis advisor Dr. Amit P. Sheth for his continuous support, patience, motivation, and immense knowledge. His guidance helped me in all the time of research. I could not have imagined having a better advisor for my thesis study.

Also, I would like to convey my sincere gratitude to the rest of my thesis committee: Dr. Keke Chen, Dr. Valerie L. Shalin, and Dr. Krishnaprasad Thirunarayan, for their insightful comments and encouragement, and for the hard question which incented me to widen my research from various perspectives.

My sincere thanks also goes to my mentor Dr. Shreyansh Bhatt for believing in me and always giving me the right push.

I would also like to thank everyone at Kno.e.sis to make my time here enjoyable and memorable.

Finally, I must express my very profound gratitude to my parents and my family for letting me pursue my dreams. I could not have asked for anything better. I wouldn’t be here without your love and support.
Dedicated to my father Sri Hari Nalamothu (1963-2008). Thank you for always watching over me and your guiding hand will forever be on my shoulder.
Online social media such as Twitter and Facebook are vital communication and networking mediums. According to a Pew Research study, approximately 76% of the American population used social media in 2017, compared to just 7% in 2005 [24]. With the popularity of social network among school and college students [6], it is important to detect attempts of potential online harassment.

Machine learning techniques play a crucial role in automatic harassment detection. Based on our analysis of online content, we found that the amount of truly harassing social media content relative to normal content is small, but it can have serious consequences on a victim. So while this is a significant problem, it is also challenging to develop automatic harassment detection system based on supervised learning. Therefore, augmenting training with an improved harassment training data can improve harassment detection. Existing text generation techniques based on Generative Adversarial Networks (GANs) provide a novel approach to augmenting the training data. In this thesis, we focused on detecting a particular form of online harassment i.e., abusive and hate speech detection.

Coherent and semantically meaningful text generation that mimics the real world text is a crucial task in NLP (Natural Language Processing). Recent studies have shown that auto-regressive models such as recurrent neural networks (RNNs) with the Long Short
Term Memory networks (LSTMs) [10, 29, 31] can achieve excellent text generation performance. However, they suffer from an exposure bias problem [1], i.e., generator can see the previous token while training but do not have an access to it while generating the text samples. Current proposed techniques to solve this problem include Gibbs sampling [28], scheduled sampling [1], and adversarial methods, such as SeqGAN [32], LeakGAN [11], MaliGAN [4], and RankGAN [17]. GAN techniques have been the most popular techniques as they do not require to define an explicit probability density function[9]. In GAN, a discriminator helps adversarial text generation models to evaluate whether a given text is real or not. Then a generator aims to generate text that is hard to discriminate from the real-world text, which results in a maximized reward signal from discriminator via reinforcement learning (RL). The entire text sequence generation of these adversarial techniques can alleviate the problem of exposure bias. In spite of their success, these adversarial models still face two challenges such as reward sparsity and mode collapse.

Diverse text generation by employing inverse reinforcement learning (IRL) [27] is proposed to tackle these two challenges. This model assigns instant rewards to a generation policy that generates text sequence by sampling one word at a time, thus providing dense reward signals. Using an entropy regularized policy gradient [7] as an optimization policy results in a more diversified text generator. Both dense rewards and entropy regularized policy gradient can alleviate reward sparsity and mode collapse (It happens when the generator learns how to produce samples from a few modes of the data distribution still misses many other modes i.e., the generator generates a limited diversity of samples, or even the same sample, regardless of the input). However, this model suffers with a small training data set because deep adversarial neural nets have high model capacity and depend on the availability of large quantities of the training data to learn a nonlinear function that generalizes the distribution.

To address the data scarcity challenge, we propose a framework that incorporates domain-specific knowledge into the IRL-based text generation model. Using IRL with
domain-specific knowledge, we show that maximization of cosine similarity between generated sentence Word2Vec (W2V) embedding and training data W2V in an optimization policy will lead to better text generation with less training data. Here, W2V embeddings are modeled using domain-specific knowledge (tweets). In addition to the framework, we present a new evaluation that uses metrics including variants of BLEU [22] to measure the diversity of generated text; and perplexity and cosine similarity to measure the quality of the generated text. Variants of BLEU such as Diverse BLEU and Self BLEU measure diversity between generated data and training data, and the generated data to itself respectively. For any given text generation model, perplexity measures how well a model predicts a sample text. Cosine similarity between generated sentence W2V and training W2V measures the quality of the generated text. Experimental results on these metrics show that the proposed framework outperforms the previous baseline (IRL without domain knowledge) on abusive and hate speech tweet generation. Ultimately, the generated tweets effectively augment the training data for online abusive and hate speech detection (tweet classification) resulting in a 9% improvement in classification accuracy with the augmented training set compared to the existing training set.

The contributions of this work are summarized as follows:

1. We extend the optimization of IRL-based GAN to incorporate a domain knowledge term that preserves meaning so as to generate diverse and meaningful text given a small training data set.

2. We propose three new evaluation measures based on the BLEU score, perplexity, and cosine similarity to better evaluate the quality and diversity of generated texts.

3. We enhanced the detection of abusive and hate speech tweets by providing generated data to augment the small training dataset.
Chapter Overview

The rest of the thesis is organized as follows.

Chapter 2: Preliminaries  Contains essential definitions (that are required to understand our work) with examples to provide an overview of Word2Vec, Language Modeling (LM), Reinforcement Learning (RL), Inverse Reinforcement Learning (IRL), and Support Vector Machine (SVM) classification.

Chapter 3: Related Work  Reviews earlier attempts to generate text. In this chapter, we talk about problems with existing algorithms.

Chapter 4: Research Contribution to Text Generation  Explains intuition behind how adding a domain knowledge term to the latest text generation optimization function produce better sentences given a small training dataset.

Chapter 5: Data & Experimental setting  Describes the abusive and hate speech tweet dataset. Also, it presents the optimal hyper-parameter setting and pipeline for our experiment.

Chapter 6: Evaluation  Illustrate the necessity for new evaluation metrics. Explains how proposed new measures capture the diversity and quality of generated sentences.

Chapter 7: Results & Conclusion  Compares examples of generated sentences and the classification results of our technique with others. Also, we talk about future improvements and concludes our thesis work.


2 Preliminaries

This chapter helps the reader to understand the background knowledge that is expected to study the thesis work presented in the following chapters. We discuss the following topics:

1. The Skip-Gram (SG) word2vec model
2. Language Modelling (LM)
3. Reinforcement Learning (RL)
4. Inverse Reinforcement Learning (IRL)
5. The Support Vector Machine (SVM) binary classifier

2.1 Word2vec

The combination of Natural Language Processing (NLP), and Artificial intelligence (AI) allows machines to process human language and, in some scenarios, even repeat it. The end goal of NLP is to understand and interpret meaning in a way that is helpful for a human user. But, understanding is one of the primary challenges of NLP. This is because machines use binary, unlike text and speech used by humans. NLP needs a process that transforms
text and speech to numbers. This process is called word embedding, with Word2Vec as one of the more well-known embedding models.

Word2Vec (a prediction based model) was introduced by Tomas Mikolov et al., 2013 [20] (a Czech computer scientist working in the field of machine learning) at Google. It is a semantic learning framework that utilizes a two-layer neural network to learn the vector representation of words or phrases in a particular corpus. The Word2Vec model takes text as input and outputs feature vectors that represent the vocabulary of a corpus. This model utilizes predictive analysis to guess a word based on its neighboring words, unlike frequency-based models. The 2 variants of the Word2Vec model are:

1. **Continuous Bag of Words (CBOW)**: This model learns to maximize the probability of a target word based on its neighboring words (a window of words around the current word). Thus the model learns to guess the words by looking at its context. Therefore, it learns to generalize the way the word can be used in all distinct contexts. This type of learning gives less attention to rare words because the objective is to guess the most probable words.

   For example, **it is really a ... day to go to a picnic**. In this instance, the model tries to predict “beautiful” rather than “delightful” because of word probabilities. This makes the model unable to learn infrequent words.

2. **Skip-Gram (SG)**: This model learns to predict the neighboring words (context) based on the current word. So, in the example case, the model takes “delightful” and guesses its context i.e., **it is really a ... day to go to a picnic** with high probability. This model gives equal attention to both infrequent and frequent words. Here, the combination of “delightful” and its neighboring words is treated as a new observation. Therefore, this model needs sufficient samples for each context.

   According to Mikolov, SG works well with a small training corpus, and even learns and represents rare words effectively. So, we choose this Skip-Gram model to train word
embedding because we also have limited training data. Let us now look into the details of the Skip-Gram model.

2.1.1 Skip-Gram model Learning Objective

The objective of the SG model is to create a word embedding that is effective for predicting the neighboring words given a current word. Specifically, the model tries to maximize average log-likelihood throughout the entire corpus.

$$\arg\max_{\theta} \frac{1}{T} \sum_{t=1}^{T} \sum_{j \in n, j \neq 0} \log p(w_{t+j}|w_t; \theta)$$

(2.1)

This equation states that there is some likelihood $p$ to observe a specific word within a size $n$ window of the present word $w_t$. The probability of the neighboring words is conditioned on $w_t$ and the setting of parameter $\theta$. For example, the probabilities of the words “he”, “is”, “that”, “retard” depends on the current word “not” as shown in Figure 2.1 and a model parameter $\theta$. The model optimizes $\theta$ so that the conditional probability of neighboring words is maximized throughout the entire corpus.

![Figure 2.1: Example of a size 5 window](image)

In Figure 2.1, the current word $w_t$ is “not”, and neighboring words are “He”, “is”, “that”, “retard”

2.1.2 Skip-Gram model architecture

The Skip-Gram model architecture is a two-layer neural network. It uses a linear activation function in hidden layers and the Softmax activation function in the output layer. It is not
possible to directly feed the text to a neural network. One of the effective ways to feed text input is to convert all of the words from a corpus vocabulary into one-hot encoding vectors. A one-hot vector is a $1 \times N$ matrix (vector) used in natural language processing to differentiate each word in vocabulary from any other word in the vocabulary. The vector comprises of zeros in all positions with the exception of a single 1 in a position used uniquely to identify the word. For example, “he” is the $i^{th}$ word in the corpus vocabulary. So, a one-hot encoding vector that represents “he” contains 1 at $i^{th}$ position and 0 at all other positions.

Figure 2.2: The Skip-Gram model architecture
Figure 2.2 shows the Skip-Gram model two-layer neural network architecture. It uses a linear activation function in the hidden layers and Softmax activation function in the output layer. \( q \) and \( q' \) are the weights which are represented by the parameter \( \phi \). \( j \) is the context.

**Hidden layer**

A hidden layer utilizes linear activation functions. A weight matrix with \( k \) rows (where \( k \) is the size of the vocabulary) and \( l \) columns (where \( l \) is size of the features that SG model extracts from each sample) represents the hidden layer. The hidden layer is computed by doing matrix multiplication between a \( k \)-dimensional one-hot encoded vector and a \( k \times l \) weight matrix \( q \). The matrix row corresponding to “1” (in the one-hot encoded vector) will be selected as shown in the following matrix multiplication.

\[
\begin{bmatrix}
0 & 0 & 1 & \ldots & 0
\end{bmatrix} \times \begin{bmatrix}
z_{11} & z_{12} & z_{13} & \ldots & z_{1l} \\
z_{21} & z_{22} & z_{23} & \ldots & z_{2l} \\
z_{31} & z_{32} & z_{33} & \ldots & z_{3l} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
z_{k1} & z_{k2} & z_{k3} & \ldots & z_{kl}
\end{bmatrix} = \begin{bmatrix}
z_{31} & z_{32} & z_{33} & \ldots & z_{3l}
\end{bmatrix}
\]

This implies that the hidden layer only works as a look-up table. The resulting hidden layer output represents a word vector for a given word input.

**Output layer**

The resulting word vector of dimension \( 1 \times l \) from matrix multiplication is given to the output layer. The output layer utilizes a Softmax activation function. The output layer consists of \( n \) number (window size) \( k \)-dimensional vectors as shown in Figure 2.2. The outputs (probabilities) of these each \( k \)-dimensional vector vary from 0 to 1 and their sum is equal to 1. For example, consider the architecture as shown in Figure 2.2 with vocabulary
size as 5.

\[
\text{output vector } w(t+1) = [\text{He, is, not, that, retard}]
\]

\[
= [0.16, 0.12, 0.02, 0.55, 0.15] = \text{that}
\]

Similarly, the model applies softmax on the remaining three 5-dimensional vectors, where it ideally aims to output “He”, “is”, and “retard”.

**Intuition**

If two distinct words have similar neighboring words (context), then the model outputs very similar word embeddings for these two words. Thus, given current words, the model outputs very similar neighboring words. For example, the model outputs similar neighboring words for the input words “Boy” and “girl”. This is because their word embedding vectors are closer to each other. Consequently, word embeddings help machines to capture the semantics of natural human language.

![Figure 2.3: Mapping word embeddings with Word2vec](image)

Figure 2.3: Mapping word embeddings with Word2vec

In Figure 2.3, the distance between “boy” and “girl” is less than the distance between
“boy” and “Rome” because the likeness of boy and girl appearing in a similar context is high.

2.2 Language Modeling (LM)

Language modeling (LM) informs many kinds of Natural language processing (NLP) applications, such as Sentiment analysis, Question answering, Machine translation, Text generation, speech recognition, and summarization, etc. The responsibility of LM is to represent the text in an accessible form for a computer. The problem with language modeling is that natural language vocabulary introduces ambiguity that human beings readily resolve. Using conventional grammar and structure, linguists attempt to define language with brittle success. Learning from the data is an alternative approach to model language. There are two approaches to model the language from data.

1. **Statistical Language Models**: A statistical language model is a probability distribution over word sequences. It assigns probability \( p(w_1, w_2, ..., w_T) \) to a sequence of tokens i.e., from a sentence in a language. In other words, it is the likelihood of seeing a sentence in the corpus. For example:

\[
p_1 = P(\text{“Tomorrow is going to be sunny”}) = 0.01
\]

\[
p_2 = P(\text{“Tomorrow is not good for a rocket launch”}) = 0.00001
\]

While both examples are valid sequences, probability of sequence 1 is high because, for instance, the LM can better model general weather discussions rather than the favorable weather for a rocket launch given a weather corpus. In addition to assigning a probability to each sequence, LM also assigns the probability of a given word following the sequence. That is, LM learns to predict the next word given the true sequence.
2. **Neural Language Models**: The *curse of dimensionality* is a basic issue that makes language modeling and other learning issues hard. The dimensionality curse occurs when a vast amount of distinct word combinations from vocabulary need to be discriminated, and the learning algorithm requires at least one instance per appropriate word combination. The number of necessary examples may expand exponentially as the amount of vocabulary rises. In language modeling, the issue arises from the enormous amount of possible word sequences. For example, with a series of 10 words taken from a $10^4$ vocabulary, $10^{50}$ possible sequences are available.

Neural Language Modeling (NLM) exploits an ability to generalize distributed representations to diminish the impact of the dimensionality curse. The NLM learns to map word embedding vectors (features vectors) to a prediction of interest, such as computing the probability distribution of a target word given the true sequence. A word embedding vector (Word2Vec) enables words with a similar meaning to have closer representations based on their use. When two word embeddings are closer to each other, then they are functionally similar (semantic and grammatical similarity) and can be replaced in the same context by one another, helping the neural network to model a function that makes good predictions about the training set (that is, the set of word sequences from a corpus used to train the model).

Bengio et al., (2003) [2] described NLM with the following 3 model properties:

(a) Maps each word in the vocabulary to a word embedding (a vector which captures features of words)

(b) Expresses the joint probability function of word sequences in terms of their word embedding.

(c) Learns the word embedding and the probability model parameters at the same time.

The above 3 properties constitute a rather simple model that learns a word embedding
and probability function that represents a model from the training corpus.

The neural-network-based language modeling approaches have recently begun to consistently outperform the classical statistical modeling methods, both as standalone applications and as a part of more difficult NLP applications.

Initially, feed forward neural networks are used to model language. Later, Recurrent Neural Networks (RNNs) and Long Short Term Memory (LSTM), a special kind of RNNs, are introduced to model the language for longer sequences (Next chapter 3 discusses details of RNNs and LSTM.).

### 2.3 Reinforcement Learning (RL)

Reinforcement learning (RL) is a form of machine learning. It allows an agent to make a sequence of decisions using feedback from its actions and experiences through trial and error in an interactive environment to maximize some notion of collective reward. RL is utilized by various machines and software to discover an optimized path or behavior for a specific scenario.

![Figure 2.4: The basic idea of reinforcement learning and elements involved in it](image-url)
Key terminology of Reinforcement learning:

In Figure 2.4, the agent performs action $a_t$ given state $s_t$, and reward $r_t$ (for the previous action). According to the action $a_t$, the environment provides reward $r_{t+1}$ and state $s_{t+1}$ to the agent. This is an iterative process until the agent reaches the final state.

1. **Environment:** Physical/Digital world where the agent works.

2. **State:** The agent’s current situation in the Environment.

3. **Reward:** The environment’s feedback.

4. **Policy:** Model(probability function) that maps the state of the agent to its actions.

5. **Value:** Future reward that an agent would collect by executing an action in a specific state.

Comparison of RL with supervised and unsupervised learning

Although both RL and supervised learning use mapping between input and output, they differ. Unlike supervised learning, which uses target label to provide a correct set of actions to the agent as feedback for performing a task, RL utilizes rewards and penalty as signals of beneficial and negative behavior. In other words, training data in supervised learning comes with an answer key, so that the model is trained with right answers. While there is no answer key in reinforcement learning, reinforcement agents choose what actions to perform to accomplish the given task. It is bound to learn from its experience despite the lack of a training dataset. RL can best be clarified through games, such as the maze below.

In supervised learning training data appear as follows:

1. l,r,u,d,l,d,u,r,l→ w

2. l,l,r,u,d,r,l,r,u,d→ lo
where l, r, u, d, w, lo corresponds to left, right, up, down, win, and lose respectively. Both the data samples came with labels (win/lose), which informs the agent by following this sequence of actions, it will either win/lose. But, a reinforcement agent looks at the scoreboard (reward) given by the maze game environment as feedback to optimize its learning.

Reinforcement learning is distinct in terms of objectives compared to unsupervised learning. While the goal of unsupervised learning is to find similarities and differences between data points, the goal of reinforcement learning is to find an appropriate model of action that maximizes the agent’s total collective reward.

### 2.3.1 Exploration and exploitation trade off

In reinforcement learning, there is a significant notion of exploration and exploitation trade-off. Exploration is about discovering more information about the environment, while exploitation is profits from already known information to maximize rewards.

![Maze game Environment](image)

Figure 2.5: Reinforcement learning demonstration with Maze Game Environment

In Figure 2.5, a robot-mouse seeks a final reward, either cheese (+1000 points), or...
water (a smaller reward worth +10 points) on its path. In the meantime, the robot mouse wants to avoid lightening spot (electric shock punishment worth -10 points) on its path.

The maze game example illustrates the exploration and exploitation trade-off. In this example, the robot-mouse wants to reach the endpoint where it gets cheese as the final reward. In this game, after some exploration, the mouse could discover the three sources of water grouped near the entrance and may waste all of its time exploiting this discovery of mini-paradise. By continuously exploiting this discovery, the mouse earns small rewards and never proceeds into the maze to seek a bigger reward. This scenario illustrates the significance of the trade-off between exploration and exploitation. A simple approach for the mouse would be executing the best-known actions (which surely provides rewards) for most (say 75-80\%) of the times, while exploring a fresh, randomly chosen path in the remaining (say 20-25\%) times, although wandering away from the known prize.

This approach is called an \textit{epsilon-greedy} approach, where epsilon decides the percentage of time an agent explores a randomly chosen path instead of performing the best-known action to maximize the rewards. In practice, people start the RL model with a high epsilon value and reduce this value after a delay. The agent (mouse) thus explores and learns about the majority of the environment over the time, discouraging further exploitation of known environment to encourage further exploration. Nevertheless, reward is not always immediate. In the maze game, the robot-mouse may traverse a lengthy stretch of the maze passing several decisions points before reaching the final reward.

\subsection{2.3.2 Approaches to Reinforcement Learning}

Following are the two major approaches to solve reinforcement learning problem.

1. \textbf{Policy-based approach}: Mathematically, we represent policy as follows:

   \[ a = \pi(s) \]  
   \hspace{1cm} (2.2)
Where $\pi$ is a policy function that maps each state $s$ to the best corresponding action $a$ (from the available actions) at that state. The policy essentially defines the behavior of the agent. $\pi$ functions as a conduct policy: “the best thing to do when I observe state $s$ is to take action $a$”. For example, consider an automatic driving car policy that includes something like: “If car sees a yellow light and it is more than 100 feet from the junction, then it should apply the brake. Otherwise, continue to move forward”. The objective of the policy function is to maximize expected reward (in case of automatic driving car minimizing the accidents) by choosing the best action at a given state.

We can further divide policies into two different kinds:

(a) A stochastic policy $\pi_s$, formally represented as follows:

$$
\pi_s = p(A = a | S = s)
$$

This gives a probability distribution over different actions given a state.

(b) A deterministic policy $\pi_d$ always returns the same action $a$ regardless of given state $s$

Consider an example where we teach an RL agent to play pong [16]. Here, we feed gaming frames (with the scoreboard on top) as input to the RL algorithm. The scoreboard in the feeding frames acts as a reward or feedback to the RL agent. Whenever the agent scores +1, the model realizes that the action taken by the policy at that state was good. So, the algorithm learns the direction in which an agent has to move depends on the scoreboard. Here we utilize deep neural networks to learn the policy function. Thus, we call these a policy network (PN). PN takes frames as input and gives rewards back to the network aiming to optimize the policy using a policy gradient.
The RL agent learns to play pong in the following way. Initially, game frames are fed to the PN, letting the algorithm choose the movements of the agent. At first, movements (actions) chosen by the algorithm are bad but eventually, the agent makes a point. A set of lucky points helps the policy to understand what action to take at the given state to maximize the reward. Thus, in time, the RL agent is likely to choose those actions that give a reward compared to an action that may not. Intuitively, the RL agent is learning to play pong by optimizing the policy.

2. **Value Based approach**: The objective of the agent in a value-based RL approach is to optimize the value function $V_\phi(s)$. $V_\phi(s)$ gives the maximum expected future reward that an agent gets at each state.

Each state’s value is defined as the expected total amount of reward to be collected by the agent over the future, from a specific state.

\[
V_\phi(S) = \mathbb{E}_\phi [R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots | S_t = s] \tag{2.4}
\]

Where $R_{t+1}, R_{t+2}, R_{t+3}, \ldots$ are the rewards associated with the future states. Agent always choose next state $s$ with high expected reward according to $V_\phi(s)$.

For the value-grid game example, the objective of the agent is to choose a state that
Figure 2.7: Value based RL approach on Grid game.

gives the maximum expected total reward. In Figure 2.7, at each state, the value-based approach takes the largest value to achieve its goal i.e., maximum points. So, the agent follows the 6 $\rightarrow$ 9 $\rightarrow$ 45 $\rightarrow$ 45 $\rightarrow$ 27 $\rightarrow$ 27 path from start to end to achieve maximum rewards.

**Limitation**

During the training of an agent, if the agent loses points because of choosing a particular sequence of actions, then the policy model tries to discard or decrease the likelihood of choosing the same sequence of actions. This approach has a potential “sparse reward” problem as the policy model does not consider the fact that the partial sequence was useful in the training process, i.e., giving rewards at the end of the game sequence instead of giving them at each state.

In Figure 2.8, except for the last 2 states, the remaining states (a good partial sequence) of sequence 3 are similar to winning sequence 1. Considering the last 2 states, the policy decreases the likelihood of a first 6 sequence of actions which may have the potential to form a winning sequence with the change of 2 last states.

The sparse reward setting of RL makes the algorithm very sample-inefficient. This requires numerous, varied training examples to train the agent. The complexity of the
Figure 2.8: Sparse Reward failure of RL algorithm

environment also contributes to the failure of sparse reward setting in many situations.

Reward shaping is proposed to solve this sparse reward problem but it is limited due to the need for reward function customization for every game.

2.4 Inverse Reinforcement Learning (IRL)

In this setting, we will provide real-world data that acts as an expert agent policy or behavior. The objective of IRL is to learn the reward function that explains expert behavior. Intuitively, we assume that training data generated from the human world has the right (best) sequence of actions. Thus, learning a reward function that explains this expert’s behavior helps the model to pick the best possible action in the future. For example, in the text generation process, estimating a reward function that explains the human written sentences supports modeling a good text generator.

Why are we interested in finding the reward function of a given problem?

In many RL tasks, there is no source for the reward signal. For example, the human-driven data feed to automate the vehicle does not have any rewards attached to it. Similarly, a human-written sentence feed to the text generation process does not have any rewards at-
tached to it. So, the data feed must be carefully hand-crafted to precisely represent the task. Usually, researchers tweak the RL agent’s reward function manually until the observation of desired behavior is reached, which is a tedious process. Therefore, IRL searches a well-suited reward function for some task by observing expert (human) performance from the training data of that task, and then automatically retrieves the respective reward function from these training data observations. Learning a reward function and then approximating the policy w.r.t a reward function is significant compared to the other way around. Also, learning a reward function captures salient features of the task whereas a policy contains many irrelevant steps to approach the task.

So, IRL better models the real-world problems given data without labeled rewards.

2.5 Support Vector Machine (SVM) Classification

SVM is one of the available classification algorithms suitable for text. It does not require much training data to begin providing accurate results. However, it needs more computational, though affordable resources.

The objective of the SVM classifier is to maximize the decision boundary i.e., maximizing the margin between two classes of training dataset as shown in the figure 2.9 such that the decision boundary is furthest away from any data point. The separating hyper-plane can be expressed in terms of the data points (red and green from each class in the figure, called support vectors) that are closest to the boundary hyper-planes.
Figure 2.9: Decision boundary of an SVM classifier with separating hyper-planes on both sides.
3

Related Work

Coherent and semantically meaningful text generation that mimics the real world text is a crucial task in NLP. Recently, studies have shown that auto-regressive models such as recurrent neural networks (RNNs) with long short term memory networks (LSTMs) [10, 29, 31] can achieve excellent performance. However, they suffer from the exposure bias problem [1]. Current techniques proposed to solve this problem, include Gibbs sampling [28], scheduled sampling [1], and adversarial methods, such as SeqGAN [32], LeakGAN [11], MaliGAN [4], RankGAN [17]. Following the framework of generative adversarial networks (GAN) [9], a discriminator helps the adversarial text generation models to evaluate whether a given text is real or not. Then a generator aims to generate text that is hard to discriminate from the real-world text, which results in a maximized reward signal from discriminator via reinforcement learning (RL). The entire text sequence generation of these adversarial techniques can alleviate the problem of exposure bias. Nevertheless, despite their success, these adversarial models still face two challenges such as reward sparsity and mode collapse.

Diverse text generation by employing inverse reinforcement learning (IRL) [27] is proposed to tackle these two challenges. This model assigns instant rewards to the generation policy that generates text sequence by sampling one word at a time, thus allocating
more dense reward signals. Using an entropy regularized policy gradient [7] as an optimization policy results in a more diversified text generator. Dense rewards and an “entropy regularized” policy gradient can alleviate reward sparsity and mode collapse. However, this model requires relatively large number of training samples that are not available for our domain-of-interest.

![Figure 3.1: State of the art Text generation models](image)

Studying auto-regressive models such as RNN, LSTM (a special kind of RNN), and RL based GANs such as Sequential GANs, Rank GANs, and IRL based GANs provides a better understanding of the disadvantages of current state-of-the-art text generation models.

### 3.1 Recurrent Neural Networks

Recurrent neural networks (RNNs) [19] are a family of artificial neural networks that are applicable for processing sequential data. The importance of these models that take sequences as inputs and predict sequences as outputs is illustrated by machine translation, language modeling like text generation, named-entity recognition, speech recognition, etc. The typical RNNs that maps sequence to sequence is parameterized by two bias vectors \(b_h,\)
three weight matrices $[U, W, V]$, and a random initial hidden state $[h^{(i)}]$ as shown in Figure 3.2. By using recurrent connections, RNNs remembers a high-dimensional sequence in the form of a hidden state by representing it with a fixed dimensionality. The current predictions $[o^{(t)}]$ made by RNNs are influenced by current hidden state $[h^{(t)}]$ (which represents the history of sequence). So, RNNs combines two sources of inputs, current input $[x^{(t)}]$ (present) and the current hidden state $[h^{(t)}]$ (recent past) to determine the current prediction $[o^{(t)}]$, mimicking humans reliance on both past and present information to make decisions.

In Figure 3.2, the hidden layers of the RNNs have the same weights and bias. So, all hidden layers can be folded in together as a single recurrent layer, called an RNN-folded form.

RNN include the following type of models: Many to Many, One to Many, and Many to One. In this related work, our interest is to study Many to One RNNs (as a generative model) for modeling sequential data. Henceforth, RNNs refer to the Many to One model. The objective of the RNN is to maximize the likelihood of a true token in the training sequence given the previously observed tokens. We used the simplest possible version of
recurrent neural networks, to help the reader apprehend sequential modeling. The recurrent neural network has an input layer $x$, a hidden layer $h$ (also called a state or context layer) and output layer $o$ as shown in Figure 3.3. The input vector $x^{(t)}$ at time $t$ is a Word2vec (W2V) embedding that represents the current word. The output vector (typically of dimensions corresponding to the vocabulary size that we are using to train the model) is denoted by $o^{(t)}$, with the hidden layer as $h^{(t)}$ (state of the network). Hidden and output layers are computed as follows:

$$
h^{t} = \sigma \left( U \ast x^{t} + W \ast h^{(t-1)} + b_{h} \right)
$$ (3.1)

$$
o^{t} = \text{Softmax} \left( b_{o} + V \ast h^{t} \right)
$$ (3.2)

where $\sigma$ = Activation function

![Figure 3.3: Recurrent neural network - unfolded form](image)
3.1.1 Recurrent Neural Networks Training

RNNs training involves two steps, feed-forward propagation (forward pass) and backward propagation through time (BPTT). Unlike feed-forward neural networks (Multi-Layer Perceptron), hidden layer values in RNNs are computed using current input and the previous hidden state (sequence history).

Forward Pass

Consider a input sequence \( x^{1:T} = x^1, x^2, \ldots, x^t, \ldots, x^T \) of length \( T \) given to an RNN. Let \( x^t \) be the word2vec embedding input at time \( t \), and \( h^{(t-1)} \) and \( h^{(t)} \) are previous hidden state and current hidden state respectively as shown in the figure 3.3.

The hidden state value at time \( t \) can be calculated by combining the previous hidden state \( h^{(t-1)} \) and current input \( x^t \) as shown in equation 3.1. Intuitively, the current hidden state represents both present \( x^{(t)} \) and recent past information \( x^{1:t-1} \). Usually, the initial hidden state \( h^{(i)} \) is set to zero. But several research studies showed that RNN performance and stability improved by using nonzero initial hidden state values [35].

Current output \( o^t \) can be computed by using the current hidden state \( h^t \) as shown in equation 3.2. Intuitively, current output \( o^t \) is predicted based on the sequence \( x^{(1:t)} \). The output vector \( o^t \) represents the probability distribution of the next token given the previous token \( x^{(t)} \) in the sequence and context \( x^{1:t-1} \) (hidden state). Softmax assures that this probability distribution is valid, i.e., \( 1 > o^t_m > 0 \) (the probability of each token in the output vector is greater than zero and less than one) for any token \( m \) and \( \sum_{m=1}^{k} o^t_m = 1 \) (the sum of the probabilities of all token in the output vector is equal to 1), where \( k \) is size of the vocabulary.

The objective of the RNN is to maximize the likelihood of a true token in the training sequence given the previously observed tokens. So, we consider cross-entropy to calculate an error vector (loss or cost vector) at every single training step as shown in equation 3.3, and weights are updated with the BPTT algorithm. To compute a cost vector we could also
use negative log-likelihood and maximum likelihood estimation. The relation between the three cost functions: maximize likelihood estimation $\rightarrow$ minimize negative log-likelihood $\rightarrow$ minimize cross-entropy) is as follows.

$$J^t = - \sum_k y^t \log o^t$$  \hspace{1cm} (3.3)

where $J^t = \text{Cross-entropy}$, $y^t$ is the target one-hot encoded vector (which the actual token in the sequence i.e., $x^{(t+1)}$).

The target $y^t$ is a one-hot encoded vector where position of token $x^{(t+1)}$ (that follows the training sequence $x^{1:t}$) is set to 1. So, the equation 3.3 can be rewritten as follows:

$$J^t = - \sum_k 1 \log o^t$$  \hspace{1cm} (3.4)

The overall cost is equal to the sum of the errors (according to cross entropy) of the entire sequence of the length $T$. The objective is to minimize the overall cost as shown in Figure 3.4, which maximizes the likelihood of the predicting the true token in the training sequence given the previously observed tokens.

$$L_\theta = \frac{1}{T} \sum_{t=1}^T J^t$$  \hspace{1cm} (3.5)

where $L_\theta = \text{Cost function (optimization or objective function) with parameter } \theta$.

**RNN forward pass training with example**

Let us look at the RNN forward pass training with the following example tweet: **He is not that retard**. The feeding of tokens (he, is, not, that, retard) in the sequence occurs as shown in Figure 3.5.
The current hidden state $h^{(t)}$ can be calculated by considering current input $x^{(t)} = \text{“not”}$, and the sequence of tokens until time $t$ i.e., $h^{(t-1)}$.

$$\text{Current hidden state } h^{(t)} = (x^t + h^{(t-1)})$$

$$= (\text{not} + (\text{He, is}))$$

$$= (\text{current token} + \text{sequence of tokens } x^{1:t-1})$$

The output vector $o^t$ represents the probability distribution of the next token given the previous token $x^{(t)}$ in the sequence and context $x^{1:t-1}$ (hidden state). Here, token “is” has a high probability but the token “that” is the true token in the sequence.

$$\text{probability of each token } o^t = [0.20, 0.30, 0.15, 0.22, 0.13]$$

$$= [\text{He, is, not, that, retard}]$$

We compute error according to cross entropy. Cross entropy is calculated between
target one-hot encoding vector and output vector as shown in Figure 3.6. Here 0.657 is the error (as shown in the example cost calculation 3.6). The objective of training is to minimize this 0.657 to 0. The cost function $L_\theta$ takes the average of the errors calculated at each step and updates the parameter of the model $\theta$ with the BPTT algorithm.

In Figure 3.6, the partially shaded circles in output vector have a high probability but
the completely shaded circle is the true token in sequence. Multiplication of the target one-hot encoded vector and the output vector ensures maximizing the prediction of the true token during generation.

\[ \text{Target } y^t = [0, 0, 0, 1, 0] = \text{that} \]

Cross entropy \( J^t = H(\text{target one-hot encoded vector, output vector}) \)
\[
= H([0, 0, 0, 1, 0], [0.20, 0.30, 0.15, 0.22, 0.13]) \\
= [0, 0, 0, 1, 0] \odot [\log(0.20), \log(0.30), \log(0.15), \log(0.22), \log(0.13)] \\
= -(1 \times \log(0.22)) \\
= -(-0.657) \\
= 0.657
\]

\[(3.6)\]

**Back Propagation Through Time (BPTT)**

During the training of an RNN, an error can be propagated further in time (multiple layers) to remember longer training sequences. Generally, we consider the full sequence (tweet) as one example during training, so total loss is just the average of the errors at each time step.

Recall that our objective is to compute the gradients of the errors concerning the weights \( U, V, W \) and then optimize the parameter \( \theta \) using gradient descent. Just like the losses that we add at each time step, we also add the gradients at each time step for each training sequence during training:

\[
\frac{\partial J}{\partial W} = \sum_t \frac{\partial J^t}{\partial W}
\]

We use the chain rule of differentiation to compute the gradients. We apply the back-propagation algorithm starting from the error at the current token (when time \( t = T \)) and propagates this backward all the way (time \( t = 1 \)) to the initial token. We provide \( J^t \) as an example.
\[
\frac{\partial J^t}{\partial V} = \frac{\partial J^t}{\partial o^t} \frac{\partial o^t}{\partial V} = \frac{\partial J^t}{\partial o^t} \frac{\partial Z^t}{\partial V} = (y^t - o^t) \otimes h^t
\]  \hspace{1cm} (3.7)

where \(Z^t = V h^t\), and \(\otimes\) is the outer product between two vectors. From the derivation, we can say that computing gradients for \(V\) only depends upon \(o^t, y^t, h^t\) (simple matrix multiplications between them).

But computing the derivations of \(W, U\) \(\frac{\partial J^t}{\partial W}, \text{and} \frac{\partial J^t}{\partial U}\) are different. Following the equation on the chain rule of differentiation for \(W\) provides a justification:

\[
\frac{\partial J^t}{\partial W} = \frac{\partial J^t}{\partial o^t} \frac{\partial o^t}{\partial h^t} \frac{\partial h^t}{\partial W}
\]  \hspace{1cm} (3.8)

Now, note that \(h^t = \sigma (U \ast x^t + W \ast h^{(t-1)})\) depends on \(h^{(t-1)}\), which again depends on \(W\) and \(h^{(t-2)}\), and so on as shown in the figure 3.7. So during derivation we can not consider \(h^{(t-1)}\) as constant. We must reapply the chain rule, resulting in the following equation:

\[
\frac{\partial J^t}{\partial W} = \sum_{t_s=1}^{T} \frac{\partial J^t}{\partial o^t} \frac{\partial h^t}{\partial h^{t_s}} \frac{\partial h^{t_s}}{\partial W}
\]  \hspace{1cm} (3.9)

Where \(t_s\) is a time step. As the same weight \(W\) is used in each time step until the final time step (i.e., from time \(t_s = 1\) to \(t_s = t\)) that we are concerned about, we must back propagate the gradients from time \(t_s = t\) through the recurrent network layers all the way to \(t_s = 1\).

BPTT is very similar to standard back-propagation that we apply in deep neural feed-forward networks. The major difference in BPTT is that we add gradients for weight parameter \(W\) at each time step. Vanilla neural networks do not share weights parameters (or other parameters) across all the layers, hence we do not need to add these gradients. Similar to standard back-propagation, BPTT seems to be an elegant name for regular back-
propagation on an unfolded RNN. We could therefore define a delta vector and propagates backward. Example: 
\[
\delta^{(t)}_{(t-1)} = \frac{\partial J^t}{\partial h^{(t-1)}} = \frac{\partial J^t}{\partial h^t} \frac{\partial h^t}{\partial h^{(t-1)}} \frac{\partial h^{(t-1)}}{\partial h^{(t-2)}}.
\]
We can apply same equations to other layers as well.

Figure 3.7: A recurrent neural network - Back Propagation through time

As sequences (tweets) getting longer (20 tokens or more), gradients from the end of the sequence can not reach to the beginning of the sequence. So, BPTT can not update the parameters at the beginning layers and that makes it difficult to train vanilla (standard) RNN. In practice, many people reduce back-propagation to a few layers.

3.1.2 Problems with RNN Training

Long Term Dependencies

The major idea behind RNNs is that they can connect previous and present information to predict the future. Sometimes, RNN fails to connect and utilize historical information to predict future tasks. The success of a standard RNN is determined by the distance between the relevant information and its need in future time step to predict the output.

At times, having recent information is sufficient to predict the present task. For example, consider a language model that is trying to predict the next token with maximum
probability given the previous sequence of tokens. If the model is trying to predict the next token in “Airplane is ready to take ...”, then the apparent next token, in this case, is “off”. The model does not require further context. In this kind of scenarios, RNN can learn past information and utilize it effectively because the distance between the relevant information and the place it is required to predict the next token is small.

However, there are also instances where the model needs more context. Consider an example of a long sequence “I grew up in Mexico...I speak Spanish fluently” (as shown in the figure 3.8). In this example, the part “I speak ” predicts that the next token probably would be the name of a language, but to predict which language, the model needs the context of Mexico (from the beginning of the sequence). In this example, the distance between the relevant information and the place it is needed is long. Unfortunately, as the sequences grow longer, it is entirely possible that the distance between the relevant information increases, which makes Vanilla RNN language models ineffective for learning longer-term dependencies (connecting relevant information in longer sequences).

Figure 3.8: A recurrent neural network - Long term dependency problem.
In Figure 3.8, the distance between relevant information “Mexico” and the place it is needed to predict the name of the language “Spanish” is long. Here $h_t$ is initial hidden state, $O_t$ is output at $t^{th}$ time step. Also, every hidden state is computed using recent history and current input.

Theoretically, standard RNNs have the absolute capability of handling sequences with long term dependencies. A researcher should carefully select parameters for the RNN language model to address this problem. In practice, RNNs fail to learn long term dependencies. The problem was deeply explored by Bengio, et al., (1994) and Hochreiter (1991), German [13, 3]. RNNs are unable to learn long-term dependencies for two reasons: Vanishing gradients, and exploding gradients.

Vanishing Gradients

Learning longer sequences requires RNN with deep hidden layers. Commonly, including more hidden layers makes the RNN model able to learn complex arbitrary functions to effectively predict future tasks. For example, if we are training an RNN model to learn and generate 32 length sequences, then the RNN model requires 32 hidden layers (each for one token). Applying gradient-based optimization techniques (like BPTT) on these RNNs with deep hidden layers cause the vanishing gradient problem while training.

Loss (error) gradient is the direction and value computed during the training of network that is utilized to optimize the network parameters (weights and biases) in the desired direction (gradient descent decreases parameter values and gradient ascent increases parameter values) by right magnitude. Generally, researches apply gradient descent ($\nabla$) in their work. So, in this related work also, gradient descent is applied during optimization. Now, when we execute Back Propagation Through Time, i.e., passing gradients of error(loss) with respect to weights in the backward direction (from time $t_0=T$ to $t_0=1$), the gradients become increasingly while propagating backward in the network as shown in Figure 3.11. This slows the learning ability of the neurons in the initial layers of the RNN.
That makes them hard to train. As the initial layers of the deep RNN are very important to identify the core elements of the input sequence, this limitation can lead to inaccurate sequences. This also causes learning inability to remember and retrieve long-term dependencies at a later point in time to predict the next token.

Let us take a closer look at the gradient derivation 3.9 in the BPTT section to understand what causes these gradients to become smaller and smaller as they propagate to initial layers.

\[
\frac{\partial J}{\partial W} = \sum_{t=1}^{T} \frac{\partial J}{\partial o_t} \frac{\partial o_t}{\partial \hat{h}_t} \frac{\partial \hat{h}_t}{\partial \hat{h}_{ts}}
\]

Here \(\frac{\partial h_t}{\partial h_{ts}}\) is a chain rule in itself! For example, \(\frac{\partial h_t}{\partial h_{(t-2)}} = \frac{\partial h_t}{\partial h_{(t-1)}} \frac{\partial h_{(t-1)}}{\partial h_{(t-2)}}\). Taking the derivative of the vector function with respect to a vector results in a Jacobian matrix (containing a first-order partial derivative for a vector function). We can represent the above gradient as follows:

\[
\frac{\partial J}{\partial W} = \sum_{t=1}^{T} \frac{\partial J}{\partial o_t} \frac{\partial o_t}{\partial \hat{h}_t} \left( \prod_{t=1}^{T} \frac{\partial \hat{h}_t}{\partial h_{(t+1)}} \right) \frac{\partial \hat{h}_{ts}}{\partial W} \tag{3.10}
\]

According to Pascanu, et al., (2013) [23], the upper bound of the L2-norm (think of it as absolute value) of the above Jacobian matrix is 1. This is because the sigmoid and tanh activation functions map all values into a range in between 1 to -1. The derivation of the sigmoid and tanh bounded by 0.25 and 1 respectively are shown in Figure 3.9, 3.10.

As in Figures 3.9, 3.10, the derivatives of sigmoid and tanh become flat and approach zero at both ends. When the derivatives approach these flat ends, the related neurons reach a saturated state. These zero gradients make the other gradients in the previous layers approach zero. This is caused by multiple matrix multiplications with the Jacobian matrix with small values (which are close to zero). Thus, the gradient values shrink exponentially fast and eventually vanish after certain time steps as shown in the figure 3.11. So, the
gradient contribution from distant layers does not reach the layers at the beginning. This makes RNN model fail to learn the relation between the “Mexico” and “Spanish” because the gradients (error) from “Spanish” can not reach (does not show impact) to “Mexico” (the beginning layer). Model end up not learning long-term dependencies. This Vanishing gradient problem is not exclusive to RNN. Even vanilla neural networks that use stochastic gradient descent for optimization have the same problem. It is just that RNNs tend to have deeper hidden layers (generally according to the size of the sequence). We expect this problem more often.

In figure 3.11, the size of the ∇ (gradient descent) indicates the gradient value.

**Exploding Gradients**

Depending on the activation functions and network parameters that are applied to RNNs, the model could end up with Jacobian matrices with large values that cause exploding gradients instead of vanishing gradients. Hence this is called the exploding gradient problem.
Vanishing gradients have received more attention than exploding gradients for two reasons. First, identification of exploding gradients is easy because gradients will become Not-A-Number (NAN) as shown in Figure 3.12 and crash the model. Second, there is a simple clipping solution available i.e., by carefully pre-defining the threshold values to clip the gradients, one can reduce the exploding gradient problem as shown in Figure 3.13. Vanishing gradients are harder to solve because it is not evident when they happen or how to handle them.

In figure 3.12, the size of the \( \nabla \) (gradient descent) indicates the gradient value. Error gradients can add up during optimization and cause very large gradients. Consecutively, this results in large updates to model parameters and causes an unstable neural network. In the worst case, the magnitude of the parameters can become so large as to overflow and cause in Not-A-Number (NAN) value. Here, gradient contribution from distant layers did not reach the layers at the beginning. This makes RNN model fail to learn the relation between the Mexico and Spanish because the gradients (error) from Spanish can not reach
The left side of the figure 3.13 is gradient without clipping and the gradients explode. The right side figure 3.13 is gradient clipping that confines the gradient values within the threshold limits. Here, $J(w, b)$ model based on the weight $w$ and biases $b$.

**Solution to Vanishing and Exploding gradient problems**

As we discussed earlier, the vanishing gradient problem is hard to solve compared to exploding gradient problem. However, there are a few ways to counter the vanishing gradient problem. Careful initialization of weight matrices (particularly matrix $W$) and regularization can lower the effects the vanishing gradients. Generally, researchers prefer the Rectified Linear Unit (ReLU) activation function as an alternative to sigmoid or tanh to combat the
vanishing gradient problem. This is because the ReLU derivative is a either 0 or 1 as shown in Figure 3.14. So, it is not as expected to suffer from the vanishing gradient problem. A more preferred solution to use is to use Long-short Term Memory (LSTM) or Gated Recurrent Units (GRU) units in an RNN architecture. Most widely used LSTMs models in Natural Language Processing (NLP) community were proposed in 1997 [14]. GRUs, initially proposed in 2014 [5], are a simplified variant of LSTMs. Both of these (LSTMs, and GRUs - special kind of RNN) architectures were especially created to handle these vanishing and exploding gradient problems and effectively learn the long-term dependencies. Researchers tailor different variations of LSTMs according to the problem they are solving. So, we will discuss LSTMs in detail in the next section.
3.2 Long Short Term Memory (LSTM)

LSTMs are especially created to effectively solve long-range dependencies. Memorizing information about longer sequences and utilizing this information in a later point of time is their default behavior (and LSTMs do not struggle to learn long term dependencies). Standard Vanilla RNNs have the form of a chain of redundant modules of a neural network (with simple activation layers like single Tanh) as shown in the figure 3.15. LSTMs also have a similar chain-like structure as in the standard vanilla RNNs, but interestingly the redundant modules have four neural network layers interact in a special way as shown in Figure 3.16.

In Figure 3.16, \( f^t \) is a forget gate (a neural network with sigmoid), \( i^t \) is an input gate (a neural network with Sigmoid), \( \hat{C}^t \) is a temporary cell state or candidate layer (a neural network with Tanh), \( op^t \) output gate (a neural network with sigmoid), \( h^t \) hidden stat (a vector), \( c^t \) current memory or cell state (a vector).

In Figure 3.17, each line in the LSTM architecture carries an entire vector from one node to another node. Point-wise operations like vector multiplication and addition are rep-
Figure 3.14: ReLU activation function $f(x)$ and its derivative $\frac{df}{dx}$ represented by pink circles. Neural networks layers are represented by yellow rectangles. Concatenation denoted by merging lines, and forking lines indicate information being copied and passed to a different location.

The memory (cell) state (horizontal line passing at the top of architecture as shown in Figure 3.18) plays a main role in LSTM. This cell value runs over the chain of redundant modules of LSTM architecture carrying memory. In this way, the memory value has only a few linear interactions. It is very easy for the model to pass a memory value unchanged.

The LSTMs can update (by adding new information or discarding current information) a memory value carrying by cell state with the help of carefully regulated structures called gates. Gates can control the information that goes to the cell state. These input ($i^t$), forget ($f^t$), and output gates ($o^t$) are built out of sigmoid networks and a point-wise multiplication operation.

The sigmoid activation function maps the inputs to the range between 0 and 1. Intuitively, sigmoid describes the amount of information from each component it should let
through. Zero means it will not allow any information to the cell state and whereas one means it will allow all information. So, sigmoid activation functions at all three gates decide the amount (between 1 and 0) of information they will allow to the cell state.

**LSTM Working Mechanism with example**

In figure 3.19, LSTM remembers long term dependency between “Mexico” and “Spanish” with the help of cell state.
In Table 3.1, the cell state remembers a country name by setting its memory as 1 and carries this information to a later point in time (“speak”) to predict the name of the language (“Spanish”). After the utilization of this information, the cell state discards this memory and sets its memory to 0.

**Forget Gate**

Initially, the model must decide what information it should discard from the memory that is carrying by cell state. A forget layer (sigmoid activation function) takes this decision. According to current input at time step $t$ ($x^t$) and previous hidden state ($h^{(t-1)}$), forget gates output a number between 1 and 0 as shown in the figure 3.20. Point-wise multiplication between the previous cell state $c^{(t-1)}$ and resultant sigmoid values (between 0 and 1) from

<table>
<thead>
<tr>
<th>Memory value</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input token</td>
<td>I</td>
<td>grew</td>
<td>up</td>
<td>in</td>
<td>Mexico</td>
<td>...</td>
<td>I</td>
<td>speak</td>
<td>Spanish</td>
<td>fluently</td>
</tr>
</tbody>
</table>

Table 3.1: Passing of memory value through the chain of repeated layers.
the forget gate decides how much information the model will throw away.

\[ f^t = \sigma \left( U \ast x^t + W \ast h^{(t-1)} \right) \]  
(3.11)

Let us look into the example as shown in the figure 3.20. In this example, the cell state remembers that “Mexico” predicts “Spanish”. Suppose, if the model detects a new country “French” in the training sequence, then the model must forget the old information about the country to predict the new language “French”. So, the forget gate discards memory about the name of the country after it is utilized in the model to detect the name of the language.

**Input Gate & Candidate layer**

After the forget gate step, the model must decide what new information it is going to add to the memory carrying by cell state. This process consists of two parts. The first part is the input gate \( \tilde{i}^t \) layer that helps the model decide which values it will update. The second part is the tanh layer that outputs a vector of new candidate values \( \tilde{C}^t \).

Figure 3.19: Working Mechanism of LSTM with example.
these two parts to update the memory cell.

\[
\hat{c}^t = \text{tanh} \left( U \ast x^t + W \ast h^{(t-1)} \right) \tag{3.12}
\]

\[
i^t = \sigma \left( U \ast x^t + W \ast h^{(t-1)} \right) \tag{3.13}
\]

Now, it is time for the model to update the memory value \(c^{(t-1)}\) carrying by cell state...
into new memory value \(c^t\). The resultant vector of new candidate values from point-wise multiplication between \(i^t\) and \(\hat{C}^t\) decides the scale at which the memory values will update. The model just needed to do a point-wise linear operation between \(c^{(t-1)}\) and resultant vector of new candidate values to make this update happen as shown in Figure 3.22.

\[
c^t = f^t \ast c^{t-1} + i^t \ast \hat{C}^t
\]  

(3.14)

Figure 3.22: LSTM - updating its cell state

As we have seen in the above example, this is where the model will actually update its memory value when it looks at the name of the country “Mexico” as input.

**Output Gate**

Ultimately, the model must decide what it is going to output \(o^t\). The output is just a filtered version of memory value carrying by cell state. In this step, initially the model executes a sigmoid layer that will decide what parts of cell state it will output. Then the model will execute point-wise multiplication between the cell state that passed through the tanh layer (to change the range of values between -1 to 1) and output from the sigmoid layer. This multiplication gives the current hidden state \((h^t)\) which contains only those parts of the cell state that the model decided to output.
\[ op^t = \sigma \left( U * x^t + W * h^{(t-1)} \right) \]  
\[ h^t = op^t * \tanh(c^t) \]  

In figure 3.23, output gate leaks some information from the cell state to hidden state to learn the long-range dependencies along with recent past information.

Overall, The combination of a hidden state \((h^t)\) and cell state \((c^t)\) helps the LSTMs to learn the recent past information along with long-range dependencies.

### 3.3 Text Generation using Auto regressive models

Text generation for both vanilla RNN and LSTM is the same. In simple terms, text generation is a process of selecting a token based on the sequence of tokens selected so far, and including this token to the sequence. This process repeats until either \(<eos>\) (end of the sequence) or a standard length (which set by the researcher according to his/her interest).

Example: P(that || He, is, not).

There are few strategies to choose the next token given the sequence of tokens:
**Sampling**: Sampling is the process of selecting a better fit token from the conditional token probability distribution. For example, a trained simple RNN selects the token “bus” with high probability compared to “plane” given the token “school”.

**Greedy**: A greedy approach is the process of always selecting the token with highest probability (using ArgMax). This process always selects token “bus” given the token “school”.

**Beam search**: Greedy search does not always generate a final sequence with the overall highest probability. A beam search maintains the track of various probable alternative tokens at each time step to prevent misleading local maxima. For example Beam search selects “bus” and “van” given token “school”, and reexamine which token allows the longer sequence with high probability during generation. One can set the number of variants the model can select at each time step.

Figure 3.24: A recurrent neural network - Text generation

One can select any of the above approaches for selecting the next token given the sequence of tokens during the text generation process. Even though auto-regressive models excelled in text generation, they suffer from the exposure bias problem.
3.3.1 Exposure bias problem

Auto-regressive models are trained to predict the next token based on the previous token in the training sequence. However, during text generation, prediction of the next token is based on the previously predicted token instead of ground truth token from the training sequence as shown in figure 3.24. Exposing the model only to the training data distribution instead of its own learned model predictions causes the exposure bias problem [1, 26, 25, 18, 33, 12]. In other words, model training happens from the words that are drawn from the training data distribution, in contrast with the words that are drawn from the learned model distribution. Divergence between training data distribution and learned model distribution outputs errors that accumulate fast along the generation process, known as error propagation.

Bengio et al., 2015 [1] published scheduled sampling (SS) strategy to solve this exposure bias problem. Later Huszr et al., 2015 [15] pointed out incorrectness of the objective function under the SS method and gave a theoretical explanation of why GANs inclined to generate more natural-looking text.

3.4 Generative Adversarial networks

Goodfellow, et al.,(2014)[9] proposed General Adversarial Net (GAN), a promising framework for mitigating the exposure bias problems (mentioned in the above section) with auto-regressive models. Particularly, in GAN, a $\phi$-parameterized discriminator $D_\phi$ aims to differentiate whether a specified data instance is real or not, and a $\theta$-parameterized generator $G_\theta$ aims to confuse $D_\phi$ by producing high-quality data which is hard to discriminate from the real-world data. This strategy has been effective and used to generate samples of natural images in computer vision tasks.

The GAN framework is easy to apply when the both $G_\theta$, $D_\phi$ are Multi-Layer Perceptrons (MLP). Generator’s distribution $d_g$ over data $w$ can be learned by defining a prior on
input noise variables $d_z(z)$, then provide a mapping to data space as $G_\theta(z)$, where $z$ is the noise vector. $D_\phi(w)$ denotes the probability that $w$ drawn from the real data rather than $d_g$.

The objective of the discriminator $D_\phi(w)$ is to maximize the probability of allocating the right labels to both real word training data and samples drawn from $G_\theta$. Concurrently, we train the $G_\theta$ to minimize $\log(1 - D_\phi(G_\theta(z)))$.

<table>
<thead>
<tr>
<th>At Discriminator $D$</th>
<th>At Generator $G$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D(x) \rightarrow$ should be maximized</td>
<td>$D(G(z)) \rightarrow$ should be minimized</td>
</tr>
<tr>
<td>$D(G(z)) \rightarrow$ should be minimized</td>
<td>$D(G(z)) \rightarrow$ should be maximized</td>
</tr>
</tbody>
</table>

Table 3.2: Objective of the generator $G$ and the discriminator $D$

![Diagram](image.png)

Figure 3.25: Generative Adversarial networks - image generation
In short, $D_\phi$ and $G_\theta$ play the mini max game with value function $\mathcal{L}(G_\theta, D_\phi)$:

$$\min_{G_\theta} \max_{D_\phi} \mathcal{L}(G_\theta, D_\phi) = \mathbb{E}_{w \sim h_{data}(w)}[\log D_\phi(w)] + \mathbb{E}_{z \sim d_z(z)}[\log(1 - D_\phi(G_\theta(z)))] \quad (3.17)$$

Where $h_{data(w)}$ is real-world data.

Unfortunately, employing adversarial nets to text generation has the following problem. GAN framework is configured for producing continuous real-valued data but has problems in directly producing sequences of discrete tokens such as text sequences. The optimization of the generator $G_\theta$ parameters is guided by gradient of the loss from $D_\phi$ with reference to the outputs by Generator. Loss gradient from $D_\phi$ slightly change the parameter $\theta$ such that generator produce more realistic values. However, if the $G_\theta$ has generate discrete tokens, then “slight change” does not make sense because there is no equivalent token to the slight change in the training dictionary space. Since Vanilla GAN does not back-propagate the gradients to update the generator several techniques have been suggested to mitigate this problem, such as SeqGAN[32], LeakGAN[11], RankGAN[17] etc.

SeqGAN[32] addresses the back-propagation failure (differentiation) problem by employing RL methods, but still suffers from the reward sparsity problem. It is important to balance how good (potential to generate realistic sequence) is the partially generated sequence as it is now and the future score of the completely generated sequence. LeakGAN[11], RankGAN[17] managed the reward sparsity problem using Hierarchical RL and ranking methods respectively, on the other hand, they failed to generate diverse text due to mode collapse issue of GANs. GANs who employed IRL framework effectively alleviated sparse reward and mode collapse problem, yet they are not effective in small training data scenarios. IRL based GAN failed to generate meaningful sentences because there is not sufficient data to balance the trade-off between reward maximization (meaning) and entropy regularization (diverse). In our work, we address this data scarcity problem by extending the optimization of IRL based GAN to incorporate a term (which captures domain
knowledge) that preserves the meaning so as to generate diverse and meaningful text. Our method generated better meaningful and diverse text with the small training dataset.

**Mode collapse Problem of GANs**

Mode collapse occurs when the generator produces a limited diversity of samples, or even the same sample, irrespective of the input. Remember, generator G’s objective is to fool the discriminator by making it assign the highest probability to the generated samples. Mathematically, G tries to minimize \( \mathbb{E}_{z \sim \mathcal{D}_z}[\log(1 - D_{\phi}(G_{\theta}(z)))] \), or in other words, to generate the point \( x^* = G(z) \) such that \( x^* = \arg\max_x D(x) \). This means that the generator will converge to find the optimal data point \( x^* \) (realistic images in case of image generation, realistic text incase of text generation) to fool the discriminator. During the mode collapse, the generator produce limited samples regardless of the input \( z \). So, to achieve generator’s goal, it produces these limited data points which are not notably diverse from the training data so that discriminator gives them the highest probability.

Broadly we have reinforcement learning algorithms and inverse reinforcement learning algorithm:

1. **Reinforcement learning based algorithms**
   
   (a) Sequence Generative Adversarial Nets with Policy Gradient (SeqGAN)
   
   (b) Rank GAN

2. **Inverse reinforcement learning algorithm**

   (a) IRL based GAN - In chapter 4, we discussed this latest state of the art text generation model (baseline model for our work) and how we addressed the incapability of this model in handling small training data set.
3.5 Reinforcement learning based algorithms

3.5.1 SeqGAN

Yu et al., 2017 [32] proposed Sequential GAN (SeqGAN) that uses the probability score (this score tells whether a sequence is real/fake) as a reward from discriminator to model the generator. In SeqGAN, the generator’s $G_\theta$ objective is to get the high reward from discriminator $D_\phi$ (which means generating a realistic sequence $O_{1:T} \sim G_\theta$). Whereas discriminator $D_\phi$ wants to give lower reward to the generated sequence $O_{1:T}$ and high rewards to the human written sequence $I_{1:T}$.

The sequence generation problem is described as follows. The generative model $G$ is parameterized by $\theta$. So, we denote generative model as $G_\theta$. The generated sequence is denoted by $O_{1:T} = (o_1, ..., o_t, ..., o_T)$, $o_t \in \mathcal{V}$, $\mathcal{V}$ is the vocabulary of the training corpus. Candidate tokens from this vocabulary act as elements of the generated sequence. At time step $t$, the current state $s_t$ is considered as sequence generated until that point of time $(o_1, ..., o_{t-1})$ and the action $a_t$ is choosing the next token $o_t$. Policy function in SeqGAN $G_\theta(o_t|O_{1:t-1})$ is probabilistic, while the transition between the state is definitive after an chosen action i.e., if the action $a_t = o_t$ and the current state $s_t = O_{1:t-1}$ then $\psi^a_{s,s+} = 1$; for all other next states $s^{++}$, $\psi^a_{s,s^{++}} = 0$. The Generative model $G$ is parameterized by $\phi$. So, we denote generative model as $D_\phi$. $D_\phi(O_{1:T})$ is a probability value which represents what are the chance of a sequence $O_{1:T}$ is from human written sequence or not.

**Policy Gradient**

The generator model’s objective (which is equivalent to policy in reinforcement learning) $G_\theta(o_t|O_{1:t-1})$ is to produce a sequence from the beginning state $s_o$ to optimize the expected
reward at the end. Formally, the objective is denoted as follows:

$$
\mathcal{L}(\theta) = \mathbb{E}[R_T | s_0, \theta] = \sum_{o_1 \in \mathcal{V}} G_\theta(o_1 | s_0) \cdot V^{G_\theta}_{D_\phi}(s_0, o_1)
$$

(3.18)

Where $R_T$ is the complete sequence reward from the discriminator $D_\phi$.

The action-value function (or simply value function) $V^{G_\theta}_{D_\phi}(s_t, a_t)$ of a sequence is the expected collective reward beginning from current state $s_t$, by taking action $a_t$ according to the policy $G_\theta$. The discriminator $D_\phi(O^{m}_{1:T})$ (which is equivalent to rewarder in reinforcement learning) gives probability values as the reward indicating the possibility of sequence being real. Mathematically, we can denote action-value function as follows:

$$
V^{G_\theta}_{D_\phi}(a = o_T, s = O_{1:T-1}) = D_\phi(O_{1:T})
$$

(3.19)

Action-value of an intermediate state can be evaluated by employing Monte Carlo search method with a a roll-out policy $G_\delta$. This policy can be utilized to sample the last $T - t$ unknown tokens. Formally, $M$-time Monte Carlo search represented as follows:

$$
\{O^1_{1:T}, ..., O^M_{1:T}\} = MC^{G_\delta}(O_{1:t}; M)
$$

where $O^m_{1:t} = (o_1, ..., o_t)$ and $O^m_{t+1:T}$is sampled based on the current state $s_t$ and the roll-out policy $G_\delta$. In practise, people set $G_\theta$ as the roll-out policy.

We execute roll-out policy beginning from $s_t$ till the end of the sequence to get $M$ number of output samples. Formally, we have:

$$
V^{G_\theta}_{D_\phi}(a = o_T, s = O_{1:T-1}) = \begin{cases} 
\frac{1}{M} \sum_{m=1}^{M} D_\phi(O^m_{1:T}), O^m_{1:T} \in MC^{G_\delta}(O_{1:t}; M) \text{ for } t < T \\
D_\phi(O_{1:t}) \text{ for } t = T 
\end{cases}
$$

(3.20)
If we take a look at the equation, there is no intermediate reward, the action-value function is defined in an iterative manner as the value of the next-state starting from state $s^+ = O_{1:t}$ and rolling out till the end.

We can re-train the discriminator function, once after we have a set of more realistic generated sentences as follows:

$$
\min_{\phi} \mathbb{E}_{O \sim p_{data}} [\log D_{\phi}(O)] - \mathbb{E}_{O \sim G_\theta} [\log (1 - D_{\phi}(O))] 
$$

(3.21)

The gradient of the generators (parameter $\theta$) objective function $\mathcal{L}(\theta)$ can be derived as follows:

$$
\nabla_{\theta} \mathcal{L}(\theta) = \sum_{t=1}^{T} \mathbb{E}_{O_{1:t-1} \sim G_\theta} \left[ \sum_{o_t \in \mathcal{V}} \nabla_{\theta} G_\theta(o_t \mid O_{1:t-1}) \cdot V_{D_{\phi}}^{G_\theta}(O_{1:t-1}, o_t) \right] 
$$

(3.22)

Subsequently, the generators parameter $\theta$ updated as follows:

$$
\theta \leftarrow \theta + \alpha_j \nabla_{\theta} \mathcal{L}(\theta)
$$

(3.23)

where $\alpha_j \in \mathbb{R}^+$ represents the associated learning rate at $j$-th step. Also, we can adopt Adam and RMS prop (advanced gradient algorithms).

**Generator Architecture**

We employ LSTM as our generator model. An LSTM(a special kind of RNN) associates the input embedding vectors $x_1, \ldots, x_T$ of the sequence $i_1, \ldots, i_T$ into a hidden states sequence $h_1, \ldots, h_T$ according to function $j$ which the update iteratively.

$$
\begin{align*}
\nabla_{\theta} \mathcal{L}(\theta) &= \sum_{t=1}^{T} \mathbb{E}_{O_{1:t-1} \sim G_\theta} \left[ \sum_{o_t \in \mathcal{V}} \nabla_{\theta} G_\theta(o_t \mid O_{1:t-1}) \cdot V_{D_{\phi}}^{G_\theta}(O_{1:t-1}, o_t) \right] \\
\theta &\leftarrow \theta + \alpha_j \nabla_{\theta} \mathcal{L}(\theta) \\

\end{align*}
$$

(3.22)

Subsequently, the generators parameter $\theta$ updated as follows:

$$
\begin{align*}
\theta &\leftarrow \theta + \alpha_j \nabla_{\theta} \mathcal{L}(\theta) \\

\end{align*}
$$

(3.23)

where $\alpha_j \in \mathbb{R}^+$ represents the associated learning rate at $j$-th step. Also, we can adopt Adam and RMS prop (advanced gradient algorithms).

**Generator Architecture**

We employ LSTM as our generator model. An LSTM(a special kind of RNN) associates the input embedding vectors $x_1, \ldots, x_T$ of the sequence $i_1, \ldots, i_T$ into a hidden states sequence $h_1, \ldots, h_T$ according to function $j$ which the update iteratively.

$$
\begin{align*}
\nabla_{\theta} \mathcal{L}(\theta) &= \sum_{t=1}^{T} \mathbb{E}_{O_{1:t-1} \sim G_\theta} \left[ \sum_{o_t \in \mathcal{V}} \nabla_{\theta} G_\theta(o_t \mid O_{1:t-1}) \cdot V_{D_{\phi}}^{G_\theta}(O_{1:t-1}, o_t) \right] \\
\theta &\leftarrow \theta + \alpha_j \nabla_{\theta} \mathcal{L}(\theta) \\

\end{align*}
$$

(3.22)

Subsequently, the generators parameter $\theta$ updated as follows:

$$
\begin{align*}
\theta &\leftarrow \theta + \alpha_j \nabla_{\theta} \mathcal{L}(\theta) \\

\end{align*}
$$

(3.23)

where $\alpha_j \in \mathbb{R}^+$ represents the associated learning rate at $j$-th step. Also, we can adopt Adam and RMS prop (advanced gradient algorithms).
Algorithm 1 Sequence Generative Adversarial Nets

**Require:** generator policy $G_θ$; discriminator $D_φ$; roll-out policy $G_δ$; a sequence training dataset $DS = \{I_1:T\}$

1: Init $G_θ$, $D_φ$ with random weights $θ$, $φ$.
2: Use MLE to Pre-train $G_θ$ on $DS$
3: $δ \leftarrow θ$
4: Use $G_θ$ to generate negative samples for training $D_φ$
5: Utilize minimizing the cross entropy to pre-train $D_φ$
6: repeat
7:   for $g$-steps do
8:     Generate a sequence $O_{1:T} = (o_1, …, o_T) \sim G_θ$
9:     for $t$ in $1 : T$ do
10:       Calculate $V(a = a_t; s = O_{1:t-1})$ by Eq.
11:     end for
12:   end for
13:   for $d$-steps do
14:     Generate negative examples and combine with given positive examples $DS$ using current $G_θ$
15:     Train discriminator $D_φ$ for $l$ epochs by Eq
16:   end for
17: $δ \leftarrow θ$
18: until SeqGAN converges
Besides, hidden states $h_t$ are mapped to output token distribution by a softmax output layer $f$,

$$p(o_t|i_1,...,i_t) = f(h_t) = \text{softmax}(b + wh_t) \quad (3.25)$$

A weight matrix $w$ and a bias vector $b$ are parameters in this equation.

**Discriminator architecture**

We can utilize convolutional architecture (CNN) to model discriminator. In this model, we denote an input sequence $i_1,...,i_T$ as:

$$\mathcal{U}_{1:T} = i_1 \oplus i_2 \oplus ... \oplus i_T,$$

where $i_t \in \mathbb{R}_e$ is the $e$-dimensional embedding of a token and $\oplus$ is the concatenation operator to create the matrix $\mathcal{U}_{1:T} \in \mathbb{R}^{T \times e}$. Then a kernel $w \in \mathbb{R}^{c \times e}$ employs a convolutional operation to a $c$ window size words to generate a new feature map:

$$l_i = \sigma(w \odot \mathcal{E}_{i:i+c-1} + b)$$
where ⊗ operator is the summation of element wise production, σ is a non-linear function and b is a bias. we employ a max-over-time pooling method over the feature maps \( \tilde{l} = \max \{ l_1, ..., l_{T-c+1} \} \). Ultimately, we implement a fully connected sigmoid activation layer to output the probability which indicates likeliness of input sequence being real.

3.5.2 Rank GAN

Lin et al., 2018 [17] proposed Adversarial Ranking for Language Generation (RankGAN). Even though the framework of SeqGAN and RankGAN is same, there is a major difference between the objective of their discriminator. Unlike SeqGAN, where the objective of the discriminator is to give rewards, in RankGAN, discriminator learns to give lower rank to the generated sequence compared to the real sequence.

\( \theta \) and \( \phi \) are the variable parameters in \( G_\theta \) and \( R_\phi \) respectively. In general, mini max game with the objective function \( \mathcal{C} \) is played by \( G_\theta \) and \( R_\phi \) represented as follows:

\[
\min_\theta \max_\phi \mathcal{C}(G_\theta, R_\phi) = \mathbb{E}_{Y \sim h_{data}}[\log R_\phi(Y|J, L^-)] + \mathbb{E}_{Y \sim G_\theta}[\log(1 - R_\phi(Y|J, L^+))] \tag{3.26}
\]

Where \( h_{data} \) is the human-written text sequence (real data). \( Y \sim G_\theta \) and \( Y \sim h_{data} \) represents that \( Y \) is sampling from generated sentences and human-written sentences respectively. The relative ranks are estimated by using the reference set \( J \). When \( Y \) is sampled from \( h_{data} \), then model considers \( L^- \) comparison set which contains synthesized data pre-sampled from \( G_\theta \). Similarly, if \( Y \) is sampled from \( G_\theta \), then model considers \( L^+ \), which contains human written data.

**Generator Architecture**

We employ LSTM to model our generator. An LSTM can take word embedding of the current token \( i_t \), previous cell state \( c_{t-1} \) and hidden state \( h_{t-1} \) to predict the current hidden state \( h_t \) and cell state \( c_t \). Taking the advantage of long term dependency learning of LSTM,
our generator produces more sensible sequences $Y = (i_0, i_1, i_2, ..., i_T)$ with length $T$.

**Ranker Architecture**

We can utilize convolutional architecture (CNN) to model Ranker (discriminator). Using a series of non linear functions $F$, the proposed ranker $R_\phi$ maps concatenated sequence matrices into the vectors of embedded features $e_Y = F(Y)$. Subsequently, the reference feature $e_J$ extracted by $R_\phi$ in advance are employed to compute the ranking score of the sequence features $e_Y$.

Figure 3.27: Rank GAN
Ranking

Ranking in terms of a relevance score of the input sequence \( Y \) given a reference \( J \) can be formulated as follows:

\[
\psi(Y|J) = \cosine(e_Y, e_J) = \frac{e_y \cdot e_J}{\|e_y\|\|e_J\|}
\] (3.27)

Where, the \( e_y \) and \( e_J \) are the embedded feature vectors of the input and reference sequence respectively. Norm operator is denoted by \( \| \cdot \| \). Ranking score for a particular sequence given a comparison set \( L \) is calculated by a softmax-like formula:

\[
P(Y|J, L) = \frac{\exp(\gamma \psi(Y|J))}{\sum_{Y' \in L'} \exp(\gamma \psi(Y'|J))}
\] (3.28)

Using trial and error, people set the parameter \( \gamma \) during experiments. The idea behind this parameter is very similar to the Boltzmann exploration method in RL. Higher \( \gamma \) increase the biases in favor of the greater score sentence, where as lower \( \gamma \) treat all sentences with equal probability. The set of input sequence to be ranked is denoted by \( L' = L \cup \{Y\} \).

The input sentence \( y \) expected ranking score is computed as follows:

\[
R_\phi(y|J, L) = \mathbb{E}_{j \in J}[P(y|J, L)]
\] (3.29)

Where, \( y \) indicates input sequence. It is either synthesized or human-written sentence. \( j \) is sampled from reference set \( J \). Model can be able to compute the relative ranks for the complete sentences given comparison set \( L \) and the reference set \( J \). The objective functions of \( G_\theta \) and \( R_\phi \) utilizes ranking scores for optimization.
Training

Partial sequence’s expected future reward $H$ can be calculated by:

$$H_{\theta,\phi}(y_{1:t-1}, J) = \mathbb{E}_{y_0 \sim G_{\theta}}[R_{\phi}(y_0|J, L^+, y_{1:t-1})]$$ \hspace{1cm} (3.30)

Here, the complete sentence $y_0$ sampled utilizing roll-out policy (similar like SeqGAN) given starter sequence $y_{1:t-1}$. More precisely, the starting sequence $y_{1:t-1} = (y_0, y_1, y_2, ..., y_{t-1})$ is fixed and roll-out policy (which is same as $G_{\theta}$) successively samples the rest of the sequence until the last token $y_T$. This is called as a “path”. We can set number of paths $M$ and sample them. The average score of all these $M$ samples generated by roll-out policy represents the approximate expected future reward of the $y_{1:t-1}$ (current partial sequence).

Gradient of the generator $G_{\theta}$ objective can be designed as:

$$\nabla_{\theta} \mathcal{L}(s_0) = \mathbb{E}_{y_{1:T} \sim G_{\theta}} \left[ \sum_{t=1}^{T} \sum_{y_t \in H} \nabla_{\theta} \pi_{\theta}(y_t|y_{1:t-1}) H_{\theta,\phi}(y_{1:t}, J) \right]$$ \hspace{1cm} (3.31)

Here, $\nabla_{\theta}$ denotes partial derivative operator. $y_0$ is the first produced token and $s_0$ is the initial state. Average of all the sampled complete sequences from the current generator model $G_{\theta}$ with in one minibatch is represented by $\mathbb{E}_{y_{1:T} \sim G_{\theta}}$. $\pi_{\theta}$ represents policy which equals to $p(y_t|y_{1:t-1})$. While calculating partial derivative of $G_{\theta}$, we fix the $R_{\phi}$.

Likewise, while optimizing the ranker’s $R_{\phi}$ objective function, we fix the $G_{\theta}$. It has been found in practise that model learns better by maximizing $\log(1 - R_{\phi}(Y|J, L^+))$, instead of minimizing the $\log(R_{\phi}(Y|J, L^+))$, here $Y$ is sampled from $G_{\theta}$ ($Y \sim G_{\theta}$). Therefore, we optimize the following objective of the ranker.

$$\mathcal{L}_{\phi} = \mathbb{E}_{Y \sim h_{data}}[\log R_{\phi}(Y|J, L^-)] - \mathbb{E}_{Y \sim G_{\theta}}[\log R_{\phi}(Y|J, L^+)]$$ \hspace{1cm} (3.32)
It is important to keep in mind that when estimating data that belongs to generated sequences, then model use $L^+$ as comparison set which consists of human-written sentences; In contrast, if human-written data is evaluating by the model, the $L^-$ comparison set which consists of synthesized data sampled from $G_\theta$.

In brief, SeqGAN suffers from the reward sparsity problem. Even though RankGAN[17] managed the reward sparsity problem using ranking methods, they failed to generate diverse text due to mode collapse issue of GANs. IRL based GANs addressed these problems. In the coming chapter, we discussed this latest state of the art text generation model (base-line model for our work) and how we addressed the incapability of this model in handling small training data set.
4

Research Contribution to Text Generation

4.1 Text Generation with IRL

Text generation is formulated as generating a text sequence \( w_{1:T} = w_1, w_2, \ldots, w_T \) according to the \( \theta \)-parameterized autoregressive (RNN/LSTM) probabilistic model \( g_\theta(w) \), here \( w_t \in \mathcal{V} \), i.e., the token \( w_t \) belongs to the given vocabulary \( \mathcal{V} \). The training of generative model \( g_\theta \) happens on the given dataset \( DS = \{w(m)\}_{m=1}^M \) with \( m \) real world samples and an underlying generating distribution \( q_{data} \). In this work, we model text generation as inverse reinforcement learning (IRL) problem.

Firstly, text generation process can be seen as Markov decision process (MDP). The generation of the token \( w_t \) at each time step \( t \) depends on the policy \( \pi_\theta(a_t|s_t) \) of the model, where \( a_t \) is the action to choose the next token \( w_{t+1} \) and \( s_t = w_{1:t} \) is the current state (represents partial sequence of length \( t \)). In simple terms, policy \( \pi_\theta \) is just a conditional probability of choosing an action \( a \) of selecting a token from set of actions (vocabulary of given dataset) given state \( s \) of partially generated sequence. A text sequence \( w_{1:T} = w_1, w_2, \ldots, w_T \) can be using the MDP’s trajectory \( \Gamma = \{s_1, a_1, s_2, a_2, \ldots, s_T, a_T\} \) (a sequence
of states and actions - selecting the next token from set of vocabulary given the partial sequence). Hence, the probability of \( w_{1:T} \) is formulated as follows:

\[
g_\theta(w_{1:T}) = g_\theta(\Gamma) = \prod_{t=1}^{T} \pi_\theta(a_t = w_{t+1}|s_t = w_{1:t})
\] (4.1)

Here, the state transition is ignored because its deterministic: \( p(s_{t+1} = w_{1:t+1}|s_t = w_{1:t}, a_t = w_{t+1}) = 1 \).

Secondly, reward function for the text generation is not explicitly given. Every text sequence \( w_{1:T} = w_1, w_2, ..., w_T \) in the training data set \( \mathcal{D}_{\text{train}} \) is formulated by an expert’s trajectory \( \Gamma \) with underlying distribution \( q(\Gamma) \). Here, model learns to estimate a reward function which explains expert behaviour.

Precisely, framework of IRL composes of two steps:

1. Underlying reward function estimation (which explains expert behaviour) from the training dataset.

2. Learning an optimal policy on a small training dataset to generate realistic text sequence, which aims to achieve maximum expected rewards.

These two steps are carried out alternately until they converge.

### 4.1.1 Reward Approximator

According to the framework of maximum entropy IRL [34], we assume that the training data text sequences are sampled from the distribution \( q_\phi(\Gamma) \),

\[
q_\phi(\Gamma) = \frac{1}{Z} \exp(R_\phi(\Gamma))
\] (4.2)

Where, \( R_\phi(\Gamma) \) is \( \phi \)-parameterized unknown reward function and assumed to be the
summation of rewards at each step \( r_\phi(s_t, a_t) \): \( R_\phi(\Gamma) = \sum_t r_\phi(s_t, a_t) \), and \( Z = \int_\Gamma \exp(R_\phi(\Gamma)) d\Gamma \) denotes the partition function.

**How can an IRL framework solve the Sparse reward problem?**

IRL formulates reward of an entire sequence \( R_\phi(\Gamma) \) as the sum of the rewards \( r_\phi(s_t, a_t) \) that are given to each token (action-state pair) while generating the sequence. Unlike past models where they give a single reward at the end of the entire sequence which discourages the partial sequences which have potential to generate a realistic sentence by the end of text generation, this method encourages the model to follow a potential partial sequence.

**Objective of Reward Approximator**

The reward approximator aims to maximize the log-likelihood of the samples in the training dataset:

\[
\mathcal{L}_r(\phi) = \frac{1}{M} \sum_{m=1}^{M} \log q_\phi(\Gamma_m) = \frac{1}{M} \sum_{m=1}^{M} R_\phi(\Gamma_m) - \log Z \tag{4.3}
\]

where \( \Gamma_m \) represents the \( m^{th} \) sample in the training set \( \mathcal{D}_{\text{train}} \).

Therefore, the gradient \( \nabla_\phi \) of the reward approximate objective \( \mathcal{L}_r(\phi) \) is:

\[
\nabla_\phi \mathcal{L}_r(\phi) = \frac{1}{M} \sum_m \nabla_\phi R_\phi(\Gamma_m) - \frac{1}{Z} \int_\Gamma \exp(R_\phi(\Gamma)) \nabla_\phi R_\phi(\Gamma) d\Gamma \tag{4.4}
\]

\[
= \mathbb{E}_{\Gamma \sim q_{\text{data}}} \nabla_\phi R_\phi(\Gamma) - \mathbb{E}_{\Gamma \sim q_\phi(\Gamma)} \nabla_\phi R_\phi(\Gamma)
\]

Intuitively, the reward approximator intends to decrease the rewards of the trajectories \( \Gamma \) that are sampled from the distribution \( q_\phi(\Gamma) \) and increase rewards for the real texts. There by, \( q_\phi(\Gamma) \) approximates \( q_{\text{data}} \).

**Importance sampling**

In practice, it is actually inefficient to sample \( \Gamma \sim q_\phi(\Gamma) \). This is because initially \( q_\phi(\Gamma) \)
does not know the underlying distribution of given training data. So, there are lots of possible actions given a state. Hence this leads to so many trajectories during sampling. That is why, we use importance sampling to directly sample the trajectories according to the roll-out policy $g_\delta$.

Choosing a best action (an action that leads to realistic sentence during generation) given an intermediate state can be evaluated by employing Markov chain Monte Carlo (MCMC) search method with a roll-out policy $g_\delta$. This policy can be utilized to sample the last $T - t$ unknown tokens. Formally, $K$-time Monte Carlo search represented as follows:

$$\{W_{1:T}^1, \ldots, W_{1:T}^K\} = MCMC^{g_\delta}(W_{1:t}; K)$$ (4.5)

where $W_{1:t}^k = (w_1, \ldots, w_t)$, and $W_{t+1:T}^k$ is sampled based on the current state $s_t$ and the roll-out policy $g_\delta$. In practise, people set $g_\delta$ same as the generator $g_\theta$.

Figure 4.1: MCMC sampling at each state for computing the expected total reward using Reward approximator R

We execute roll-out policy beginning from $s_t$ ($s_3$ in fig 4.1) till the end of the sequence.

67
to get $K$ number of output samples as shown in the figure 4.1. Formally, we have:

$$V_{R_{\phi}}^{g}(a = w_t, s = W_{1:t-1}) = \begin{cases} \frac{1}{K} \sum_{k=1}^{K} R_{\phi}(W_{1:T}^m), W_{1:T}^m \in M C M C_{g}^{\theta}(W_{1:T}; K) & \text{for } t < T \\ R_{\phi}(W_{1:t}) & \text{for } t = T \end{cases}$$

(4.6)

If we take a look at the equation 4.6, the action-value function $V_{R_{\phi}}^{g}$ is defined in an iterative manner as the value of the next-state starting from state $s = W_{1:t}$ and rolling out till the end.

Now, we can concretely formalized the equation 4.4 as:

$$\nabla_{\phi}\Sigma_{\tau}(\phi) \simeq \frac{1}{M} \sum_{i=1}^{M} \nabla_{\phi}R_{\phi}(\Gamma) - \frac{1}{\sum_{j=1}^{K}} \sum_{j=1}^{K} \Upsilon_{j} \nabla_{\phi}R_{\phi}(\Gamma')$$

(4.7)

where $\Upsilon_{j} \propto \exp\left(\frac{R_{\phi}(\Gamma_j)}{g_{\theta}(\Gamma_j)}\right)$. For each batch, we draw $M$ texts from the train set and $K$ texts from $g_{\theta}$.

### 4.1.2 Text Generator

**Text generator architecture**

The text generator predicts the next word in the sequence according to the policy $\pi_{\theta}(a|s)$. We use LSTM neural network to learn the $\pi_{\theta}$ function. LSTM neural networks models the current state $s_t$ as follows:

For $\Gamma = \{s_1, a_1, s_2, a_2, \ldots, s_T, a_T\}$,

$$s_t = \text{LSTM}(s_{t-1}, v_{a_{t-1}})$$

$$\pi_{\theta}(a_t|s_t) = \text{softmax}(Ws_t + b)$$

Where current action $a_t$ (choosing next token from the vocabulary - learning the un-
derlying distribution), and \( s_t \) is the vector representation of the hidden state \( h_t \); \( a_{t-1} \) word embedding is denoted by \( v_{a_{t-1}} \); All trainable parameters of LSTM (generator) including weights \( W \), biases \( b \) are denoted by \( \theta \).

**Objective of the baseline Text Generator**

Text generator follows the “entropy regularized” policy gradient [30, 21] and its goal is to maximize the expected reward and regularization of entropy.

Recently, Shi, Zhan, et al. (2018) [27] implemented IRL framework for diverse text generation. Following is the objective of the text generator in this implementation:

\[
\mathcal{L}_g(\theta) = \mathbb{E}_{\Gamma \sim g_\theta(\Gamma)}[R_\phi(\Gamma)] + H(g_\theta(\Gamma))
\] (4.8)

Where \( H(g_\theta(\Gamma)) \) is an entropy term:

\[
H(g_\theta(\Gamma)) = -\mathbb{E}_{g_\theta(\Gamma)}[\log g_\theta(\Gamma)]
\]

and this corresponds to exploration (This controls the exploration part of the model in the exploration-exploitation trade-off). It avoids premature entropy collapse and encourage the generator (policy) to produce more diverse texts.

Maximizing entropy and reward can get us a better text generator. However, this objective function (eq 4.8) is not effective given a small training data (as shown in the results section). This is because the objective function needs more data to learn be better text generator in this kind of setting. If we do not feed good enough data then it becomes hard for a generator to learn how to balance the reward maximization and exploration. Hence, it can result in the less meaningful text as the generator puts more weight on the entropy or it can result in similar text generated if we put less weight on the exploration. Hence, we need an external data-source or knowledge-source to preserve meaning so that even after putting more weight on the exploration part (entropy), we do not compromise on the meaning of the text.

In this work, we extend the optimization to incorporate a term that preserves the mean-
ing so as to generate diverse and meaningful text.

**Objective of the proposed Text Generator**

\[ \mathcal{L}_g(\theta) = \mathbb{E}_{\Gamma \sim g_\theta(\Gamma)}[R_\phi(\Gamma)] + \text{Max}(CS_{\Gamma \sim g_\theta(\Gamma)}(E(\Gamma), \mathcal{D}_{\text{train}})) + H(g_\theta(\Gamma)) \]  \hspace{1cm} (4.9)

where \( \text{Max}(CS_{\Gamma \sim g_\theta(\Gamma)}(E(\Gamma), \mathcal{D}_{\text{train}})) \) is the maximum of the cosine similarities between generated sentence embedding and training dataset, and cosine similarity \( CS_{\Gamma \sim g_\theta(\Gamma)}(E(\Gamma), \mathcal{D}_{\text{train}}) \) is defined as follows:

\[ CS_{\Gamma \sim g_\theta(\Gamma)}(E(\Gamma), \mathcal{D}_{\text{train}}) = \left\{ CS^{E(\Gamma)}_{E(\mathcal{D}_{\text{train}}^1)}(E(\Gamma)), CS^{E(\Gamma)}_{E(\mathcal{D}_{\text{train}}^2)}(E(\Gamma)), \ldots, CS^{E(\Gamma)}_{E(\mathcal{D}_{\text{train}}^M)}(E(\Gamma)) \right\} \]  \hspace{1cm} (4.10)

Here, \( CS^{E(\Gamma)}_{E(\mathcal{D}_{\text{train}}^m)} = \frac{E(\Gamma) \cdot E(\mathcal{D}_{\text{train}}^m)}{||E(\Gamma)|| \cdot ||E(\mathcal{D}_{\text{train}}^m)||} \) is the cosine similarity between sentence embedding of sequence drawn from the generator and \( m^{th} \) sample embedding from the given training data set. Sentence level embedding is the average of word embedding of tokens \( e(w_t) \) in the sentence, \( E(w_{(1:T)}) = \frac{1}{T} \sum_{t=1}^{T} e(w_1) + e(w_2) + \ldots + e(w_T) \)

**How can Max of cosine similarity alleviate data scarcity problem?**

Since we have limited training data, an increase in exploration (high entropy value) leads to the generation of a less meaningful sentence. So, adding cosine similarity term, which captures domain knowledge in the form of word embeddings, to the objective function (eq 4.8) compensates the limited training data problem. Here, the generation of a token’s word embedding \( e_t \) using skip-gram Word2vec model on a domain-specific data source captures the domain knowledge in \( e_t \). Max of cosine similarity term in the objective aims to maximize the similarity between generated sentence embedding \( E(w_{(1:T)}) \) (average of tokens embedding \( \sum_{t=1}^{T} e_t \) and given training dataset by approximating the generated sentence embedding to the closest sentence embedding among the training data samples. Intuitively, this process ensures the exploration (entropy) to choose diverse tokens such that average
of those token embeddings must be closer to any of the sentence embedding among the training data samples while reward maximization makes sure the combination of those diverse tokens make sense as a sentence. Thus, adding cosine similarity term in the objective brings balance in the entropy-reward maximization trade-off. Where entropy try to generate diverse text (diversity) and maximization of reward assures the generation of a realistic sentence (meaning). For example, consider the following two sentences:

1. Training sentence: I am tired of feminist bitches bc they are just disgusting

2. Generated sentence: I feel disgusting feminist bitches are annoying.

Here, cosine similarity guides the entropy and reward maximization of the generator to produce the above sequence of tokens because cosine similarity between average generated tokens embedding and average tokens embedding of the training sample is high.

Intuitively, we can rewrite the entropy regularized expected reward (eq 4.9) as follows:

$$\mathcal{L}_g(\theta) = -KL(g_\theta(\Gamma) \| q_\phi(\Gamma)) + \log Z$$  \hspace{1cm} (4.11)

where $Z = \int \exp(R_\phi(\Gamma)) d\Gamma$ represents the partition function and can be seen as a constant regardless of the $\theta$. Thus, the objective is also to minimize the KL divergence between the generator distribution $g_\theta(\Gamma)$ and the assumed underlying distribution $q_\phi(\Gamma)$.

Therefore, the derivative of $\mathcal{L}_g(\theta)$ is

$$\nabla_\theta \mathcal{L}_g(\theta) = \sum_t \mathbb{E}_{\pi_\theta(a_t|s_t)} \nabla_\theta \log \pi_\theta(a_t|s_t)$$ \hspace{1cm} (4.12)

The figure 4.2 illustrates our work architecture. Left part of the figure: Reward approximator $R$ is trained over the generated data by Generator $G$ and the given training data. Right part of the figure: $G$ is trained by policy gradient where cosine similarity signal is provided by Max of cosine similarity module, and the final reward signal is provided
by R are propagate back to intermediate action $S_3$ value using Markov chain Monte Carlo search.

$$\left[R_\phi(\Gamma_{t:T}) + V_{{s_{t+1}}}^{CS} - \log_\pi(\alpha_t|s_t) - 1\right]$$

is the gradient loss $\nabla_\theta$ that propagates back from current state $s_t$ to all the way to the beginning state $s_1$ and updates the parameter $\theta$ of the model w.r.t to $s_t$. Where $R_\phi(\Gamma_{t:T})$ represents the partial trajectory $\Gamma_t, \ldots, \Gamma_T$ reward. Here, $V_{{s_{t+1}}}^{CS}$ is expected cosine similarity at $s_{t+1}$, you can find the definition at figure 4.2.

For receiving lower variance, $R(\Gamma_{t:T})$ can be approximately calculated by

$$R_\phi(\Gamma_{t:T}) \simeq r_\phi(s_t, a_t) + U(s_{t+1})$$ \hspace{1cm} (4.13)

where $U(s_{t+1})$ represents the total reward expected at state $s_{t+1}$ and can be approximately calculated by Markov chain Monte Carlo (MCMC) as shown in the figure 4.2.

**Why Can IRL Mitigate the Mode Collapse?**

Mode collapse in GANs is partially caused by Jensen-Shannon (JS) divergence which is reverse KL divergence $KL(q_\theta(\Gamma)||q_{data})$. KL divergence approximate the $g_\theta(\Gamma)$ to $q_{data}$. This approximation encourages $g_\theta(\Gamma)$ to generate safe sequences and prevent producing
samples where training data does not occur. Unlike other GANs, IRL based GANs objective is $KL(g_\theta(\Gamma)\|q_\phi(\Gamma))$. Instead of $q_{data}$ IRL framework use $q_\phi(\Gamma)$. Due to the assumption $q_\phi(\Gamma)$ never equals to 0, mode collapse problem can be alleviated by IRL based GANs.

Training

The training process contains two steps:

1. Updating reward approximator (r-step)
2. Updating text generator (g-step)

These two steps are applied repeatedly as outlined in the below Algorithm. At first, we have randomly parameterized $r_\phi$ and $\pi_\theta$ with pre-trained parameters by MLE on $D_{S_{train}}$.

The aim of the reward approximator (r-step) is to update $r_\phi$ with fixed $\pi_\theta$. The aim of the text generator (g-step) is to update $\pi_\theta$ with fixed $r_\phi$. These two steps applied iteratively as shown in the algorithm 2.

Algorithm 2 IRL for Text Generation

1: repeat
2: Pretrain $\pi_\theta$ on $D_{S_{train}}$ with Maximum Likelihood Estimation (MLE)
3: for $l_r$ epochs in r-step do
4: Sample $\Gamma_1(1), \Gamma_2(2), \ldots, \Gamma_i(i), \ldots, \Gamma_M(M) \sim q_{data}$
5: Sample $\Gamma'_1(1), \Gamma'_2(2), \ldots, \Gamma'_i(i), \ldots, \Gamma'_M(M) \sim g_\theta$
6: Update $\phi \leftarrow \phi + \alpha \nabla_\phi L_r(\phi)$
7: end for
8: for $l_g$ batches in g-step do
9: Sample $\Gamma_1(1), \Gamma_2(2), \ldots, \Gamma_i(i), \ldots, \Gamma_M(M) \sim g_\theta$
10: Computed expected reward $R_\phi(\Gamma_i:T)$ by MCMC
11: Computed expected cosine similarity $V^C_{\text{CS}}$ by MCMC
12: Update $\theta \leftarrow \theta + \beta \nabla_\theta L_g(\theta)$
13: end for
14: until Converges
5

Data & Experimental setting

5.1 Abusive and Hate Speech Tweets

We have 43,100 tweets which are labeled as “abusive”, “hate speech”, and “normal”. Founta, Antigoni Maria, et al. (2018) [8] published definitions of above-mentioned labels based on all the descriptions that are found in the related literature. According to their publication, any tweet that contains strongly impolite, rude or hurtful language that uses profanity and show degradation of someone or something, or show intense emotion are labeled as “abusive”, and any tweet which contains language that is used to convey hatred towards a targeted person or group, or designed to be derogatory, humiliate, or insult group members based on characteristics such as race, religion, ethnicity, sexual orientation, disability, or gender are labeled as “hate speech”. All tweets that are not included in any of the previous categories are labeled as “normal”. Even though they are labeled as normal they still contain offensive that does not hurt or abuse anyone.

The length of the tweets ranges from 5 to 40 words. The average length is 18.08 words. The vocabulary size of these 43K tweets is 9,388 after removing low-frequency words (less than 5 times). Since we have only 3,500 tweets which are labeled as “abusive” and “hate speech”. Considering these 3500 tweets as two different classes are not effective to train
Table 5.1: Example tweets from each class

<table>
<thead>
<tr>
<th>Examples of Abusive tweets</th>
</tr>
</thead>
<tbody>
<tr>
<td>can someone sum this up before i call this guy retarded &lt; hyperlink &gt;</td>
</tr>
<tr>
<td>RT @User: SICK OF BITCHES ON THE INTERNET &lt; emojis &gt; &lt; hyperlink &gt; @Users #fbloggers #fbchat</td>
</tr>
<tr>
<td>RT @User: I hate them hoe ass braids &lt; emojis &gt; &lt; hyperlink &gt;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Examples of Hate speech tweets</th>
</tr>
</thead>
<tbody>
<tr>
<td>RT @User: bruh i fucking hate people like this &lt; emoji &gt; &lt; hyperlink &gt;</td>
</tr>
<tr>
<td>@User @User Yes because u idiots never run out of nonsense</td>
</tr>
<tr>
<td>RT @User: I hate it when I’m trying to board a bus and there’s already an asshole on it. &lt; hyperlink &gt;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Examples of Normal tweets</th>
</tr>
</thead>
<tbody>
<tr>
<td>@User bro i cant fucking wait for the Video Screensharing!!! &lt; emoji &gt;</td>
</tr>
<tr>
<td>Forgetting to pack a sports bra for the gym is the fucking worst &lt; emoji &gt;</td>
</tr>
<tr>
<td>RT @User My face be oily as hell when I wake up n I hate it.</td>
</tr>
</tbody>
</table>

the classification model. So, we combined and labeled them as “offensive tweets” and treat them as a positive class while training the model. Finally, we have 39,600 normal tweets (negative class) and 3,500 offensive tweets. The word length of “offensive tweets” ranges from 5 to 16 after removing twitter keywords like re-tweets (RT), emojis, and hyperlinks as a part of data processing as shown in the figure 5.3. The average word length is 10.8. The vocabulary ($V$) size of these 3.5K tweets is 6,280. Since we have very limited positive class tweets, We split them into 90-10 train to test ratio (We randomly choose 3,150 tweets as train data set and 350 as test data set from 3.5K tweets).

**Domain Specific data:**

For this experiment, the closest dataset we could find for the harassment domain is Abusive and Hate Speech Tweets dataset. In this dataset along with 3,500 abusive and hate speech tweets, the normal labeled tweets also have the offensive language as you have seen in the table 5.1. So, for now, we consider this dataset (43K tweets) to generate word embedding for the offensive tweets vocabulary as you see in figure 5.4.
5.2 Experimental setting

We use 2 LSTM neural networks with hyper-parameters that are shown in the table 5.2 as text generator and reward approximator to setup this text generation experiment. Then we
use them to generate 640 sequences of length 16.

<table>
<thead>
<tr>
<th>Hyper-Parameter</th>
<th>Abusive and Hate Speech Tweets dataset</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Text generator</strong></td>
<td></td>
</tr>
<tr>
<td>Embedding dimension</td>
<td>128</td>
</tr>
<tr>
<td>Hidden layer dimension</td>
<td>128</td>
</tr>
<tr>
<td>Batch size</td>
<td>64</td>
</tr>
<tr>
<td>Optimizer &amp; learning-rate</td>
<td>Adam, 0.0025</td>
</tr>
<tr>
<td><strong>Reward Approximator</strong></td>
<td></td>
</tr>
<tr>
<td>Embedding dimension</td>
<td>256</td>
</tr>
<tr>
<td>Hidden layer dimension</td>
<td>512</td>
</tr>
<tr>
<td>Batch size</td>
<td>64</td>
</tr>
<tr>
<td>Drop out</td>
<td>0.75</td>
</tr>
<tr>
<td>Optimizer &amp; learning-rate</td>
<td>Adam, 0.0005</td>
</tr>
</tbody>
</table>

Table 5.2: Hyper-parameters configurations.

**Training strategy**

In experiments, we find that our framework’s stability and efficiency depends on the strategy of training. The text generation model is effected by number of pretraining epochs. Our model generates meaningful and diverse text of length 16 with 40 epochs of Maximum
Likelihood Estimation (MLE) pretraining. The proportion of number of epochs $l_r : l_g$ between generator and reward approximator impacts the convergence and final performance. It means that adequate training on the approximator in each iteration will result in better outcomes and convergence. Therefore, we designed our model with $l_r : l_g = 5 : 1$ as our final training setting.

Figure 5.4: Pipeline of our framework
Evaluation

6.1 New Evaluation Measures

6.1.1 BiLingual Evaluation Understudy (BLEU)

BLEU [22] algorithm is used for measuring the quality of the text that has been machine-translated from one natural language to another. To compare a candidate translation with various reference translations, BLEU utilizes a modified type of precision.

The important features of a good generation model are generating diverse (w.r.t each other and w.r.t the training data) and meaningful (quality) sentences. So, we need evaluation metrics which measures quality and diversity of the generated text. In literature, people used BLEU to evaluate the similarity between generated sentences and training sentences. This computed similarity between n-grams (n=2,3,4,5) of generated sentences and training sentences is considered as quality of the generated sentence. Since our model objective

<table>
<thead>
<tr>
<th>Candidate sentence</th>
<th>the</th>
<th>the</th>
<th>cat</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference sentence</td>
<td>the</td>
<td>cat</td>
<td>is</td>
</tr>
</tbody>
</table>

Table 6.1: BLEU example
Table 6.2: BLEU score calculation for candidate w.r.t reference sentence (example from the above table)

<table>
<thead>
<tr>
<th>Model</th>
<th>set of grams</th>
<th>score</th>
</tr>
</thead>
<tbody>
<tr>
<td>unigram</td>
<td>“the”, “the”, “cat”</td>
<td>$\frac{1 + 1 + 1}{3} = 1$</td>
</tr>
<tr>
<td>Bigram</td>
<td>“the the”, “the cat”</td>
<td>$\frac{0 + 1}{2} = \frac{1}{2}$</td>
</tr>
</tbody>
</table>

is to generate sentences which are less similar i.e., diverse from train data, we proposed a different variation of BLEU called Diverse BLEU to measure this diversity. Also, we used Self BLEU [27] to measure diversity between generated sentences. We employed perplexity and cosine similarity in a novel way to measure quality of generated sentences. This section discusses about new evaluation measures to calculate diversity using variants of BLEU, and quality using perplexity and cosine similarity.

**Diversity:** We use the following variants of BLEU score to evaluate the diversity of the generated sentences.

1. **Diverse BLEU** ($BLEU_{div}$): It evaluates BLEU score for each generated sentence by considering the training set as reference.

2. **Self BLEU** ($BLEU_s$): It evaluates BLEU score for each generated sentence by considering other generated sentences as reference.

Intuitively, the objective of $BLEU_{div}$ is to compute the diversity between generated set and training set, while $BLEU_s$ is to compute the diversity between generates sentences as shown in the table 6.3. Here, lower values of these both metrics indicate more diversity.

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Evaluated Set</th>
<th>Reference Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>$BLEU_{div}$</td>
<td>generated set $G_s$</td>
<td>train set</td>
</tr>
<tr>
<td>$BLEU_s$</td>
<td>generated sentence $G_s$</td>
<td>$G_s - G_s$</td>
</tr>
</tbody>
</table>

Table 6.3: Configuration of $BLEU_{div}$ and $BLEU_s$

**$BLEU_{div}$:** For computing $BLEU_{div}$, we sample 640 generated sentence from the generator as evaluated set, and reference set consists of whole train data set. We list the $BLEU_{div}$
scores of base line and our model in the table 6.4. Apparently, the lower the $BLEU_{div}$ score is, the more diversity the generator gets compared to the train set. From table 6.4, Our method out performs base line model.

**BLEU$_s$** : For $BLEU_s$, we measure diversity of each generated sentence by considering other 639 generated sentences as reference set ($G_s - G_s$). We get $BLEU_s$, by Averaging these diversity scores. Intuitively, the lower the $BLEU_s$ score is, the more diverse the generated sentence from each other. From table 6.4, base line model is fall behind our method.

However, these evaluation measures only diversity and dont examine the quality of generated sentences in any way.

**Quality:**

We utilize Perplexity and cosine similarity to measure the quality of the generated texts.

### 6.1.2 Normalized perplexity

Perplexity is a measure of how closely a text generation model generates a sample [18] that resembles the real-world text. Lower perplexity value indicates that the model generates better quality sentences. If we consider the quality part of our model, then it aims to generate more realistic sentences that relate to the chosen domain. That is why we consider Domain-specific dataset $\varphi$ as a reference set to measure perplexity instead of training data set. We modeled this normalized perplexity such that given a generation model it gives value that ranges between 0 and 1. Here 0 means it generates gibberish and 1 means it generates very realistic domain-specific sentences. To normalize perplexity, we compute perplexity of 43K randomly generated tweets (set) $\rho$ of length 18 (average length of domain-specific data) where each word is sampled from the vocabulary of $\varphi$. n-gram
Normalized perplexity \( P_{nmz}^n \) of a generated data set is defined below.

\[
P_{nmz}^n = \left( \frac{G_S P_{nmz}^n - \varphi P^n}{\rho P^n - \varphi P^n} \right) - 1
\] (6.1)

Where \( \varphi P^n \) is average perplexity of an underlying model that generated domain-specific data and it is set to 1, and \( \rho P^n \) is average perplexity of a model that generated gibberish data and it is set to 0. Intuitively, this measure acts as a scale where 1 indicates the real-world model and 0 indicates model which generates gibberish. Apparently, the higher the generator model’s normalized perplexity \( G_S P_{nmz}^n \) score is, the more realistic the generated sentence gets to the real-world domain-specific data. From table 6.4, we outperform the previous baseline model.

<table>
<thead>
<tr>
<th>Grams</th>
<th>BLEUdiv</th>
<th>Self-BLEU</th>
<th>Perplexitynmz</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IRL</td>
<td>IRLcs</td>
<td>IRL</td>
</tr>
<tr>
<td>2-grams</td>
<td>0.471</td>
<td><strong>0.393</strong></td>
<td>0.672</td>
</tr>
<tr>
<td>3-grams</td>
<td>0.261</td>
<td><strong>0.189</strong></td>
<td>0.518</td>
</tr>
<tr>
<td>4-grams</td>
<td>0.185</td>
<td><strong>0.113</strong></td>
<td>0.441</td>
</tr>
<tr>
<td>5-grams</td>
<td>0.154</td>
<td><strong>0.087</strong></td>
<td>0.396</td>
</tr>
</tbody>
</table>

Table 6.4: Experimental results between IRL (previous baseline) and IRLcs (our model)

### 6.1.3 Max of Cosine similarity

Max of cosine similarity evaluates similarity between generator and the training data in terms of the average of tokens embeddings (We can call this as “embedding similarity”). This metric outputs the average maximum of cosine similarities (MCS) between generated sentence embeddings (sentence embedding = average of token embeddings of that sentence) and train dataset embeddings. Thus, MCS is defined as following:

\[
MCS = \frac{1}{l} \sum_{i=1}^{l} \max(CS_{\epsilon_1}^i, CS_{\epsilon_2}^i, ..., CS_{\epsilon_M}^i)
\] (6.2)
where $l$ is size of generated dataset, $e_i$ is the $i^{th}$ generated sentence embedding, and $E^m$ is the $m^{th}$ training sentence embedding.

<table>
<thead>
<tr>
<th>Metric</th>
<th>IRL</th>
<th>IRLcs</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCS</td>
<td>0.92</td>
<td>0.99</td>
</tr>
</tbody>
</table>

Table 6.5: Experimental results of MCS

Here, higher the MCS, better the quality of generated text. From table 6.5, we outperform the previous base line model.

As a whole, all four of these metrics ensure that generated sentences are diverse from the training dataset and also from each other while they are closer (meaningful) to the real-world domain specific data and training data (in terms of sentence level embeddings). Consider the following example:

1. Training sentence: I am tired of feminist bitches bc they are just disgusting

2. Generated sentence: I feel disgusting feminist bitches are annoying.

Here, $\text{BLEU}_{div}$ tells us how diverse generated sentence from training set and normalized perplexity gives us its closeness to the real-world domain specific data and MCS informs us its sentence level embedding closeness to the training data.
7

Results & Conclusion

7.1 Experimental results

<table>
<thead>
<tr>
<th>Training data for text generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>can someone sum this up before i call this guy retarded</td>
</tr>
<tr>
<td>sick of the bitches on the internet</td>
</tr>
<tr>
<td>i hate them hoe ass braids</td>
</tr>
<tr>
<td>bruh i fucking hate people like this</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Generated data using synonyms</th>
</tr>
</thead>
<tbody>
<tr>
<td>can anyone sum up this before i call this man back</td>
</tr>
<tr>
<td>disgusted of gripe on the internet</td>
</tr>
<tr>
<td>one hate them hoe ass braid</td>
</tr>
<tr>
<td>bruh iodine bally hate people like this</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Generated data from base model</th>
</tr>
</thead>
<tbody>
<tr>
<td>(IRL with out domain knowledge)</td>
</tr>
<tr>
<td>yass i hate that a nigga or a shit</td>
</tr>
<tr>
<td>establish a red good girl chance you bad be gone</td>
</tr>
<tr>
<td>your so annoying on your side protest attendance</td>
</tr>
<tr>
<td>them hoes gone fucking hate</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Generated data from our model</th>
</tr>
</thead>
<tbody>
<tr>
<td>two beautiful bitches and shit in your fucking retirement world</td>
</tr>
<tr>
<td>akademiks is fucking disgusting</td>
</tr>
<tr>
<td>fuck yes i would have slapped the fuck out of her</td>
</tr>
<tr>
<td>he was just a puppet idiot on a string</td>
</tr>
</tbody>
</table>

Table 7.1: Examples from training data, and generated data from different techniques
Considering we have small training dataset, we followed the k-fold cross (k=10) validation to limit overfitting. In this process, initially we randomly split given training set without augmented generated data into 10 batches, and each time when we execute a separate SVM classifier learning experiment, we reserve one batch for testing and train the model on the combined remaining 9 batches and generated data. We execute 10 separate SVM classifiers learning experiments and took an average of the various classification performance metric results as the final measures to compare different techniques as shown in the table 7.2. Augmentation of generated sentences from our model to the given training data certainly improve the detection of offensive tweets compared to the other techniques.

**Generating data using synonyms:**

Here, we generated each new sentence by replacing each possible original sentence token with different synonyms.

**High precision, low recall:**

We can trust classification judgments made by algorithms with high precision. From the table 7.2, the algorithm has 95.75\% testing precision given training data with augmented diverse generated data using our model. However, our technique has a relatively low rate of recall (71.72\%) compared to precision, meaning that some of the offensive tweets in the original corpus are not yet identified. Intuitively, there is a high chance that our technique does not classify normal tweets as offensive but it may classify some of the offensive tweets as normal.

In table 7.3, examples that explain diversity of generated sentences. Here, generator explores the vocabulary and combines the tokens in a such a diverse way that they are small parts of the different training sentences but combined them together they form a meaningful sentence.
Table 7.2: SVM classification results on different techniques

<table>
<thead>
<tr>
<th></th>
<th>Training data without augmentation</th>
<th>Training data with synonyms generated data</th>
<th>Training data with diverse generated data</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Average Accuracy</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Training = 80.93 ± 0.68</td>
<td>Training = 92.14 ± 0.43</td>
<td>Training = 96.80 ± 0.44</td>
<td></td>
</tr>
<tr>
<td>Testing = 70.30 ± 0.16</td>
<td>Testing = 72.25 ± 0.37</td>
<td>Testing = 78.97 ± 0.30 = ~ 79</td>
<td></td>
</tr>
<tr>
<td><strong>Average Recall or average sensitivity</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Training = 82.65 ± 0.86</td>
<td>Training = 86.80 ± 0.60</td>
<td>Training = 94.06 ± 0.76</td>
<td></td>
</tr>
<tr>
<td>Testing = 64.25 ± 0.12</td>
<td>Testing = 65.81 ± 0.29</td>
<td>Testing = 71.66 ± 0.28</td>
<td></td>
</tr>
<tr>
<td><strong>Average Precision</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Training = 98.82 ± 0.14</td>
<td>Training = 99.42 ± 0.06</td>
<td>Training = 99.92 ± 0.02</td>
<td></td>
</tr>
<tr>
<td>Testing = 91.54 ± 0.14</td>
<td>Training = 92.60 ± 0.23</td>
<td>Testing = 95.85 ± 0.17</td>
<td></td>
</tr>
<tr>
<td><strong>Average F1-score</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Training = 90.01 ± 0.57</td>
<td>Training = 92.68 ± 0.37</td>
<td>Training = 96.90 ± 0.41</td>
<td></td>
</tr>
<tr>
<td>Testing = 75.50 ± 0.12</td>
<td>Testing = 76.94 ± 0.28</td>
<td>Testing = 82.01 ± 0.23</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.3: Diversity of generated sentences

<table>
<thead>
<tr>
<th></th>
<th>Reference from training data</th>
<th>Generated sentence</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Example 1</strong></td>
<td>i hate bitches that do not know how to mind they business</td>
<td>let this bitch say i hate bitches</td>
</tr>
<tr>
<td><em>f**uck sake do not</em> let this slip you retard i hate when a manly looking bitch say she want a nigga like shorty</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Example 2</strong></td>
<td>i can not wait to be disgusting bitches</td>
<td>done with these ugly ass niggah</td>
</tr>
<tr>
<td><em>i</em> be on my ugly ass niggah line</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In plot 7.1, Y-axis: It represents average rewards w.r.t to meaning (green) plots considered, and average entropy w.r.t to diversity (red) plots considered. At the end of iterations: our model generated relatively better meaningful sentences than baseline model and also improved balance between diversity and meaning; Diversity score of baseline model exceeded reward score, which means it is generating meaningless diverse sentences.

In plot 7.2, generated sentence from our model got relatively better rewards. In other words, our model generating better meaningful sentences than base-line model.
7.2 Conclusion & Future Work

In this work, we proposed a method for the detection of abusive and hate speech tweets given training data with a small set of positive samples. This method augments more positive samples using text generation. Since existing text generation methods are not effective with small training data, we proposed a new method by incorporating domain knowledge in inverse reinforcement learning (IRL) framework. This method alleviates the problem of small training data. Also, we propose three new evaluation measures based on BLEU score, perplexity, cosine similarity to better evaluate the quality and diversity of generated texts.

In the future, we would like to improvise the way of incorporating domain knowledge in the IRL framework. Also, we would like to extend this work to generate style-based offensive tweets to detect multiple classes of online harassment.
Figure 7.2: Rewarder objective plot
Bibliography


[31] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google’s


