THE COOPER UNION
ALBERT NERKEN SCHOOL OF ENGINEERING

Crafting Adversarial Text Samples for Recurrent Neural Networks Using Windowed Inference and Search

by
Cory Nezin

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Engineering

May 2018

Professor Fred L. Fontaine, Advisor

THE COOPER UNION FOR
THE ADVANCEMENT OF SCIENCE AND ART

ALBERT NERKEN SCHOOL OF ENGINEERING

This thesis was prepared under the direction of the Candidate's Thesis Advisor and has received approval. It was submitted to the Dean of the School of Engineering and the full Faculty, and was approved as partial fulfillment of the requirements for the degree of Master of Engineering.

Richard Stock, Dean of Engineering Date

Professor Fred L. Fontaine Date

Acknowledgments

First and foremost I thank my advisor, Professor Fred Fontaine. Professor Fontaine has served as an excellent adviser in all my time as an electrical engineering student. He has shown fierce dedication to his field, The Cooper Union, and his students. His signal processing course allowed me to discover a vast and interesting field, and he inspired me to pursue it. This thesis would not be possible without his advisement and education which he provided me over the past two years. I will not soon forget what I learned in his courses, nor the opportunities afforded as a result.

I would like to thank all of my peers, the faculty, the alumni, the administration, and The Cooper Union as a whole. This is a weird, beautiful, and sometimes pain inducing institution. I feel that the sense of community is perhaps greater than any other place. I hope that I will continue to be a part of the community for a long time so I can give back what The Cooper Union has given to me. I give particular thanks to Brenda So, Ross Kaplan, and Gordon Macshane, some of the closest friends I made here. They have helped me in ways I cannot even describe.

Finally, I thank my parents. They have always provided me support, and allowed me to grow to be my own person. My intense work has reduced the time that I can spend with them, but I know they understand that this sacrifice is worth it.

Abstract

Neural networks have recently been found vulnerable to "attacks" which cause them to misclassify samples that were given a very small disturbance. Attacks based on iterative gradient methods have been largely studied for numerically valued data like images. In this work two new algorithms are described which extend the same kind of results to text. One algorithm, "window search", does not require a continuous or differentiable model. The other algorithm, "gradient assisted window search", is a hybrid algorithm which exploits word2vec based gradients for fast search. The hybrid algorithm uses the gradient as a guide for candidate word replacements, and then performs an exponential search to determine the minimum number of replacements required to alter the classification of a text sample. The algorithm is tested under white box, gray box, and black box scenarios.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem Statement

Automatic text analysis and classification have become important issues with the rise of the internet and digital text. Every minute, millions of emails and texts are sent, and at least thousands of news articles are published. [2, 3]. With this huge rise in information, tools have been deployed to protect people from malicious actors.

Spam emails are not only annoying but dangerous. They often contain phishing scams which attempt to obtain identifying credentials from targets, or links to malicious software. In 2002, a very effective email spam filter was developed and outperformed competitors by a large margin. [4] The success of the algorithm was afforded by the implementation of Bayesian filtering. The inventor of the algorithm, Paul Graham, applied basic rules of probability to the problem of classifying an email as spam or not. Both spam filters and spam sources have of course evolved since then, both constantly attempting to overcome the other.

The term "fake news," which has recently come into existence, refers to a news article which is intentionally created to misguide the readers with false statements. With the new-found ease of creating a website that looks like that of a legitimate

newspaper, it can be difficult to tell whether a news article is real or not. The authors of [5] suggested a machine learning solution, where an algorithm automatically separates fake news from true.

The huge rise of digital communication has also led to the expansion of forensic linguistics: automatic web scraping and search engine reporting can help investigative authorities identify criminals and terrorists on the internet. Because of the huge amount of data to sift through, it is unlikely that the vast majority of text is analyzed by humans. While organizations like the NSA, FBI, and CIA are rightfully secretive about their methods, it is likely that they use some kind of pattern recognition or machine learning to identify suspicious individuals and organizations.

Recently, machine learning algorithms have been found vulnerable to adversarial attacks which cause them to misclassify samples after only minor alterations. [6] While these adversarial attacks originally targeted image classifiers, they have been extended, to some extent, to the domain of language. [7] These attacks stand to compromise the methods of defense just previously described. Neural networks in particular have been found extremely useful in those defenses, but also very vulnerable to adversarial attacks. This thesis takes the perspective of the red team, that is, we attempt to find efficient and effective methods of attack so that defenses may be correspondingly developed.

1.2 Our Contribution

This thesis contributes two algorithms for causing misclassification in recurrent neural networks acting on text. The method found in [7] was an iterative method completely based on the gradient of the network with respect to the inputs. This thesis finds the information provided by the gradient to be unreliable, and describes search based methods instead. One method, window search, does not even require a differentiable

model. The other method, gradient assisted window search, is a hybrid model which uses the gradient to search more quickly, though with degraded performance.

This work measures the performance of adversarial algorithms by counting the number of word replacements that are required to cause a misclassification. Time to obtain a misclassification was also measured. Both window search and gradient assisted window search show a very large improvement over the algorithm in [7] in a white box scenario, and an even larger improvement in a black box scenario.

1.3 Overview

Chapter 2 provides a background on machine learning, especially the algorithms employed in our experiments. These algorithms include recurrent neural networks, word2vec, and gradient based training algorithms like gradient descent. Chapter 3 describes the problem statement in more detail and develops the terminologies and definitions to discuss it. Chapter 4 briefly discusses the results obtained from training recurrent neural networks and a word embedding for sentiment classification. It also provides motivation for reducing the impact of the gradient on decision making in the algorithms. Chapter 5 describes the developed algorithms in detail, and provides a time and space complexity analysis. Chapter 6 gives a discussion of the results under several different scenarios. Chapter 7 concludes the thesis with a summary of results and methods, as well as a discussion of future work.

Chapter 2

Machine Learning for Text

2.1 Machine Learning

Machine learning is the general task of finding patterns given a set of data. The methods by which these tasks are accomplished range from the simple linear regression to more complex neural networks. Machine learning problems fall into primarily two categories: supervised learning and unsupervised learning. [8]

In the case of supervised learning, we are given a set of inputs, $\{x\}_{i=1}^{N} \in X$ and outputs, $\{y\}_{i=1}^{N} \in Y$ of some unknown function, f. These sets are often referred to as "features" and "labels" respectively. The goal is then to determine what that function is. This is usually not feasible because the model for the function is either not complex enough, or is too complex. These issues are known as underfitting and overfitting respectively. The objective is therefore simplified to finding the function, $g \in M$ where M is some set of model functions, and where g minimizes some loss function, $L(g, x, y)$. That is:

$$\arg\min_{g \in M} L(g, x, y) \tag{2.1}$$

The loss function penalizes mismatches between $g(x_i)$ and y_i so that minimizing

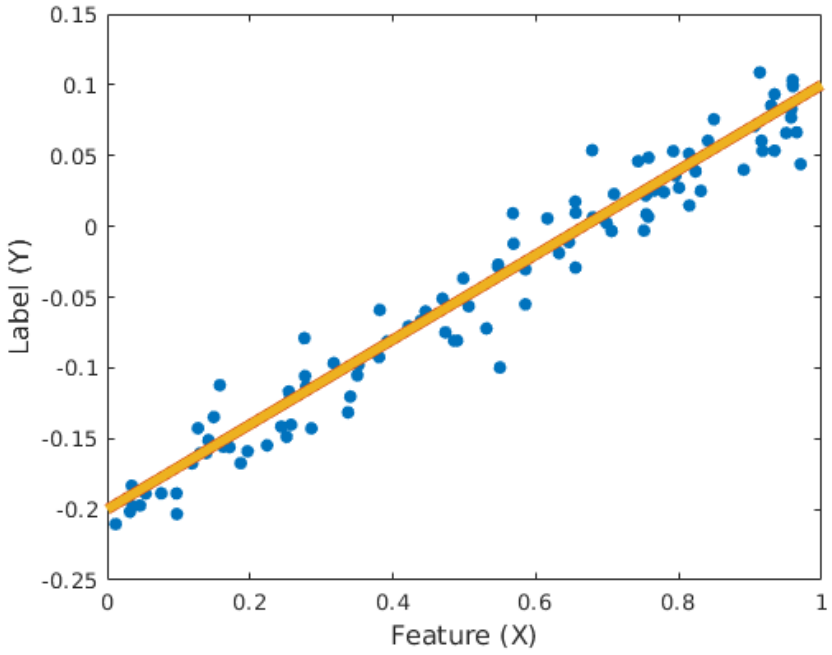


Figure 2.1: Linear regression for a noisy line. A blue dot represents the position of a feature/label pair. The yellow line represents the approximate affine approximation function.

yields, conceptually, $g \approx f$. One simple case of this is linear regression. In linear regression, M is the set of linear (or in most cases affine) functions that map X to Y. and the loss function is given by:

$$L(g, x, y) = \frac{1}{N} \sum_{i=1}^{N} ||g(x_i) - y_i||^2 \tag{2.2}$$

It turns out that under these constraints, an exact solution can be found. An illustration where $X = Y = \mathbb{R}$ is shown in Figure 2.1.

In the case of unsupervised learning, we are given only some set of features, $\{x\}_{i=1}^{N}$ and asked to find some pattern in the data. Finding a patterm can consist of finding clusters of data points which are "close" together by some measure, or finding some lower dimensional representation for the data, these are essentially problems of lossy compression. Usually algorithms work by minimizing some loss function so that techniques can be borrowed from supervised learning, though these are not necessarily

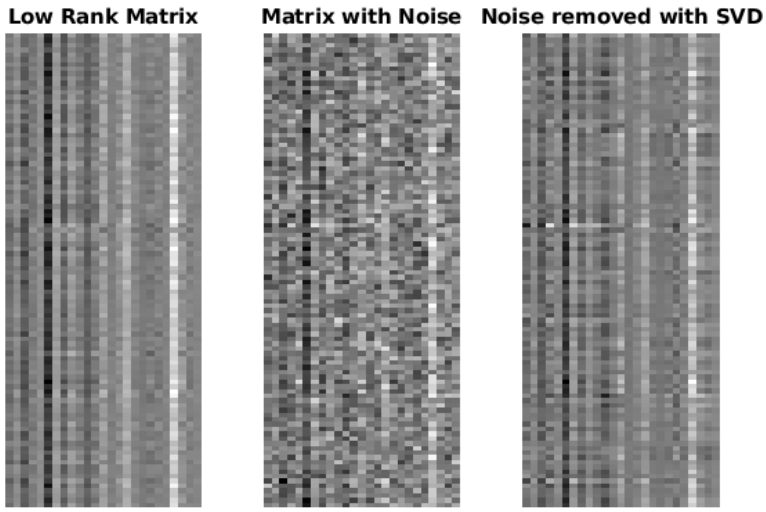


Figure 2.2: Illustration of using SVD to recover an underlying signal. This is an example of unsupervsied learning.

measures of the algorithm's success.

The well known examples of unsupervised learning are principal component analysis and its cousin singular value decomposition. Given some features in the form of a matrix X, singular value decomposition finds a matrix, $\hat{X}$ of rank $r < rank(X)$ such that $||X - \hat{X}||_F$ is minimzed. This has the effect of finding a lower dimensional representation of X which contains as much information as possible and therefore tells us which dimensions are "important." One example is shown in Figure 2.2, which uses singular value decomposition to remove noise from an assumed low rank matrix. In Figure 2.3, a plot shows the relative contribution of automatically detected signal components.

2.1.1 Hyperparameters

Hyperparameters are parameters of a function which do not change during the "learning process", that is, they are set by the user at the beginning, and the regular parameters of the function are determined with the hyperparameters held constant. In

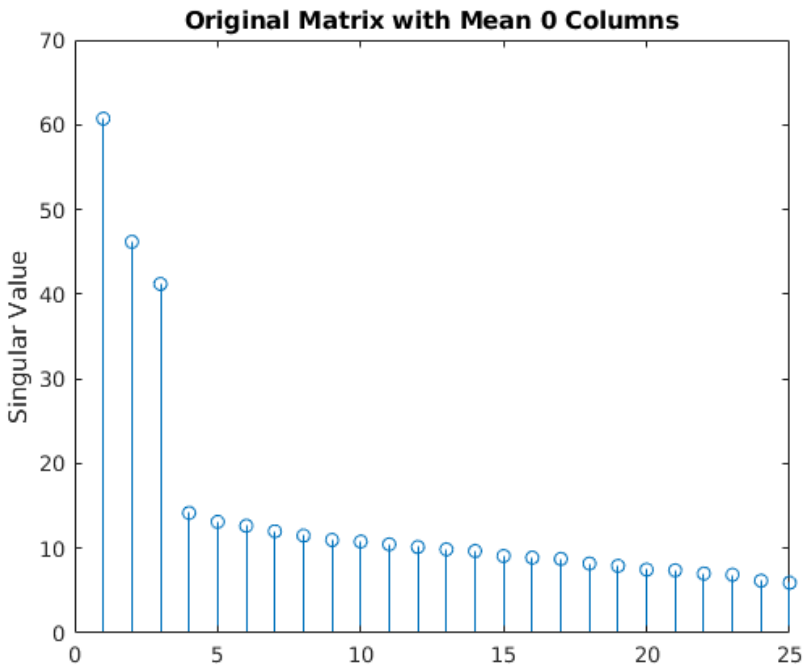


Figure 2.3: Set of 25 singular value for the low rank matrix in figure 2.2. There are hree distinct which tells us that the origin rank was very likely three.

the example of using singular value decomposition to denoise a low rank matrix, the hyperparameter would be the desired rank of the resulting matrix.

It can be seen from Figure 2.3 that this hyperparameter could be calculated, or at least guessed from the number of dominant singular values. In most modern machine learning models, it is very difficult to efficiently determine optimal hyperparameters, often requiring a brute force search over several combinations. This technique is known as grid search, which searches over n hyperparameters on an n dimensional grid and chooses the hyperparameters that minimize some loss function, not necessarily the same as the model's loss function. The process of choosing optimal hyperparameters is known as hyperparameter tuning.

2.1.2 Overfitting

If one allows the model, or set of permissible functions, to be more complex, one may represent more complicated functions. Figure 2.4 shows an example of both a quadratic fit and a 20^{th} order polynomial fit to data points with a more complicated pattern. These two images represent the issue of overfitting: while the underlying curve is in fact quadratic, the higher order polynomial achieves a lower error. It is often true, especially in simpler cases, that increasing the model complexity will reduce the value of the loss function. However if the goal is to determine the actual underlying pattern of the data, one may want to choose something simpler at the cost of a worse loss value.

Regularization is the technique of introducing information to a machine learning problem to reduce overfitting. One of the most common forms of regularization is called Tikhanov regularization. [9] It is also known by the names of ridge regression in statistics and weight decay in machine learning. In linear regression, Tikhanov regularization penalizes a function according to the magnitude squared of each coefficient. This can be applied to any linear transform as follows. By the Riesz representation theorem, for any real-valued linear function, $g : \mathbb{R}^m \to \mathbb{R}$, there exists a vector $v \in \mathbb{R}^m$ such that $g(x) = \langle x, v \rangle$. Suppose v_g is the vector corresponding to linear function g in this manner, then for linear regression the loss function in equation 2.2 can be modified as:

$$L(g, x, y) = \lambda ||v_g||_2^2 + \frac{1}{N} \sum_{i=1}^{N} ||g(x_i) - y_i||^2 \tag{2.3}$$

where λ is a real scalar hyperparameter that may be tuned with grid search, for example. The hyperparameter λ achieves a tradeoff between matching the data and reducing the size of the coefficients in the function g. While there is no efficient method for computing the optimal value for λ in general, it has a simple value given

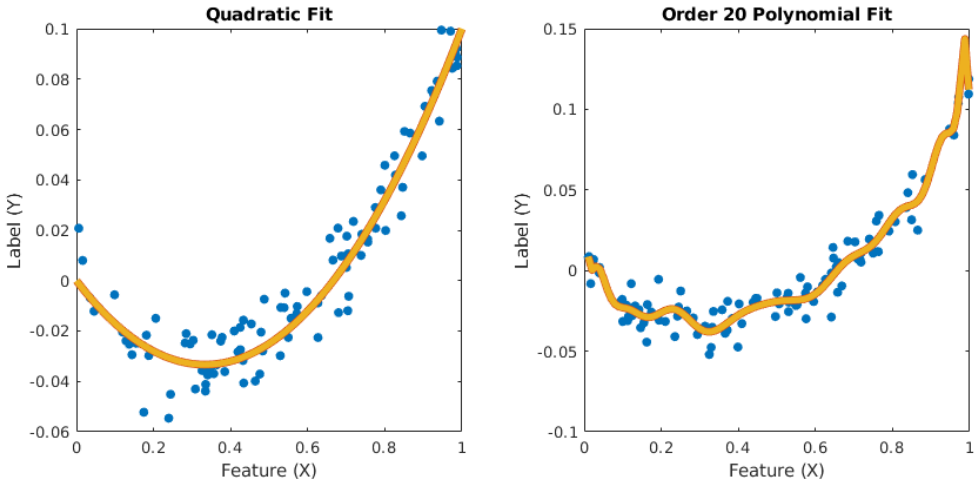


Figure 2.4: The figure on the left is the result of an optimization constrained to second order polynomials while the figure on the right is the result of a 20^{th} order constraint. The second order constraint achieves a result closer to the underlying curve.

a certain assumption. If one assumes a Gaussian distribution with 0-mean and λ^{-1} variance for the prior distribution of the vector v, then the result of the maximum a posteriori (MAP) estimation minimizes equation 2.3.

2.1.3 Training, Validation, and Testing

Using the loss function over all data points is not a suitable measure of the success of a machine learning algorithm. As discussed in section 2.1.2, models which fit the underlying function worse may easily achieve a better overall loss. In fact for any finite data set that could feasibly represent a function, it is easy to find a function in the form of a hash table that can exactly represent the mapping of inputs to outputs. Of course, this hash table would not "generalize" to other data points and would not represent the underlying function in a meaningful way.

To help obtain a true measure of success, it is standard to separate the data into three parts: training, validation, and testing. Essentially, training is used to tune regular parameters, validation is used to tune hyperparameters (e.g. with grid search), and testing is used only to evaluate the effectiveness of the algorithm. If

a model function is guilty of overfitting, it will be revealed by a low training loss and a high testing loss. Measuring the validation loss over time can also be used for a simple form of regularization called *early stopping*. In early stopping, training is stopped when the validation loss begins to increase. Of course validation loss may be noisy so usually the loss over time is smoothed and a human hand picks the time to stop.

2.2 Neural Networks

A neural network is a class of mathematical functions which uses non-linearities in order to increase representational power as compared to simple linear models. [8] They differ from other non-linear models like logistic regression by having multiple "layers" of nonlinearity between the input and the output. These non-linearities are called *activation functions*. The multilayer perceptron is a simple example of neural network which is covered in the following section.

2.2.1 Multilayer Perceptron

A multilayer perceptron is a function $f : \mathbb{R}^n \to \mathbb{R}^m$. Given some vector $x \in \mathbb{R}^n$ of inputs, the output vector $f(x) \in \mathbb{R}^m$ of a multilayer perceptron is given by:

$$f(x) = \vec{\sigma_p} \circ T_p \circ \cdots \circ \vec{\sigma_2} \circ T_2 \circ \vec{\sigma_1} \circ T_1 \circ x \tag{2.4}$$

where functions $T_1, \ldots, T_p$ are assumed to be affine, and often represented as matrix multiplications plus vector valued constants. The functions $\vec{\sigma_1}, \ldots, \vec{\sigma_p}$ are in general nonlinear, and are called the activation functions. The activiation functions are predetermined and the training of the nueral network involves selecting the coefficients in $\{T_i\}$. In order for equation 2.4 to make sense, one must assume that the dimensions of each function are compatible with one another. Activation functions are usually sim-

ple functions which operate on an element-by-element basis. Here the neural network described by (2.4) has $p+1$ layers. Each activation function adds a layer to the *input layer* which consists of x alone. Note that the final activation function $\vec{\sigma}_p$ is optional. It will usually be included if the neural network is performing a classification task since the output is discrete valued and excluded if performing a regression task where the output is continous.

In reference to the previous section, the loss function, L for a neural network binary classifier is typically given by binary cross entropy:

$$l(f, x, y) = \frac{1}{m} \sum_{i=1}^{m} y_i \log(f_i(x)) + (1 - y_i) \log(1 - f_i(x)) \tag{2.5}$$

$$L(f, X, Y) = \frac{1}{N} \sum_{i=1}^{N} l(f, X_i, Y_i) \tag{2.6}$$

and $\vec{\sigma_p}$ is typically given by the softmax function:

$$\vec{\sigma_p}(\vec{x}) = \frac{e^{x_i}}{\sum_{i=1}^{N} e^{x_i}} \tag{2.7}$$

2.2.2 Universal Approximation Theorem

The simplest multilayer perceptron has three layers: the input, hidden, and output layers. In this case, the network function is simply given by

$$f(x) = T_2 \circ \vec{\sigma} \circ T_1 \circ x \tag{2.8}$$

Part of what makes neural networks of the above form so attractive is that they are simple yet powerful. Given fairly weak constraints on the activation functions, it is possible to represent any univariate continuous function on a compact set arbitrarily well with some T_1, T_2 with finite dimensional codomains. [10] This has been proven for the cases of the L^1 norm, L^2 norm, and the L^∞ norm. To be specific, a univariate

sigmoidal function is defined as $\sigma : \mathbb{R} \to \mathbb{R}$ and any function satisfying

$$\sigma(t) \to \begin{cases} 1 \text{ as } t \to \infty \\ 0 \text{ as } t \to -\infty \end{cases} \tag{2.9}$$

Define $\vec{\sigma}$ as the vectorization of σ, where:

$$\vec{\sigma}(x)_i = \sigma(x_i) \tag{2.10}$$

Two common choices for such a function in neural networks are the logistic function (given by $\sigma(x) = \frac{1}{1-e^x}$) and the tanh function (given by $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$). The universal approximation theorem says that given a compact set, $I \subset \mathbb{R}^m$, $\epsilon > 0$, and this sigmoidal activation, one can pick T_1, T_2 such that any of the conditions in equations (2.11)-(2.13) could be met.

$$||f(x) - g(x)||_\infty < \epsilon \; \forall x \in I \tag{2.11}$$

$$||f(x) - g(x)||_1 < \epsilon \; \forall x \in I \tag{2.12}$$

$$||f(x) - g(x)||_2 < \epsilon \; \forall x \in I \tag{2.13}$$

for any function, $g(x)$ in $C(I)$, $L^1(I)$, or $L^2(I)$ respectively.

However, as the author of [10] writes, this theorem gives no upper bound on the dimensionality of the output of T_1 and posits that this value is likely very large. In addition, one is still left the task of determining the functions T_1 and T_2 which produce the desired results. Since they are affine and x is finite dimensional, they may be represented with matrices and so the task is to find the corresponding coefficients. This task is called training. This is usually done with an approximate algorithm, stochastic gradient descent, covered in section 2.3.2.

The two main issues of training is that the algorithms are not exact, and the size

of matrices required to represent the function may be too large to be computationally feasible. For these reasons, most neural network research is devoted to creating structures which facilitate learning or reduce the computational complexity of a model. Section 2.3 discusses several training strategies which help learn the correct parameters as well as avoid the problem of overfitting. Section 2.4 describes recurrent neural networks, a structure which is particularly efficient at learning features of time series.

2.3 Training Algorithms

Neural networks are highly nonlinear, non-convex functions and therefore very difficult to train efficiently. Colloquially, one can think of any kind of machine learning training as being in a large field and trying to find the lowest valley or the highest peak. Unfortunately, one cannot see the terrain nearby and only has local information about the current point. Stochastic gradient descent is the ubiquitous choice for almost every training scenario. Powerful extensions of stochastic gradient descent have also been invented, such as the Adam optimizer. This section discusses both.

2.3.1 Newton's Method

If weights are allowed to range over an open set (in fact they are usually over $\mathbb{R}$) then $\nabla L = 0$ is a necessary condition for the loss to be minimum. This condition also guarantees the function is at least at a global minimum or maximum. This equation is usually not directly solvable for nonlinear functions like neural networks. However, Newton's method can be used to successively approximate it. In reference to the loss function discussed in section 2.1, let $L(g, x, y) = L(w)$ where w is the vector of weights for the neural network. Applying Taylor's Formula to the gradient:

$$\nabla L(w + \Delta) \approx \nabla L(w) + HL(w)\Delta \tag{2.14}$$

where H is the Hessian matrix of L. To find the zero of ∇L, set $L(w+\Delta) = 0$ which results in:

$$\Delta = -[HL(w)]^{-1}\nabla L(w) \tag{2.15}$$

Which finally yields the update step

$$w \leftarrow w - [HL(w)]^{-1}\nabla L(w) \tag{2.16}$$

The issue with this method is in the computation of the Hessian and its inverse. Most modestly complex neural networks have at least thousands of parameters meaning that H will be a very large matrix (containing the number of parameters squared). Not only does this put a strain on memory resources, but the inversion of such a matrix is extremely time consuming. In addition, the loss is defined over every point in the data set meaning even computing the loss is very time consuming for large data sets, let alone its derivative with respect to all parameters. SGD mitigates these issues by avoiding the computation of both the loss and the Hessian.

2.3.2 Stochastic Gradient Descent

Stochastic gradient descent (SGD) is a very simple algorithm which pushes the parameters of a model in the direction associated with the steepest loss decrease. First, assume that the loss function $L(g, x, y)$ discussed in section 2.1 can be broken down as a sum of loss functions over each feature-label pair, that is:

$$L(g, x, y) = \sum_{i=1}^{N} l(g, x_i, y_i) \tag{2.17}$$

Given an initial set of weights (often initialized pseudo-randomly), SGD follows the update in equation 2.18.

$$w \leftarrow w - \eta \nabla l(g, x_i, y_i) \tag{2.18}$$

where η is a real valued hyperparameter called the learning rate.

Equation 2.18 says that SGD alters the weights in the direction of the negative gradient. Because the the gradient of a function normally points in the direction of maximum increase, this update points in the direction of maximum decrease. This type of learning strategy is called *hill climbing*. Note that the updates applied to the weights are entirely based on a local calculation, the gradient. The algorithm is stochastic in the sense that the updates depend only on the *randomly chosen* x_i, y_i pair. Using the function l instead of L is akin to stochastically approximating the true gradient of L with one of its components.

Supposing the algorithm halts since all gradients of $l(g, x_i, y_i)$ are zero, this would imply that the gradient of L is given by

$$\nabla L(g, x, y) = \nabla \sum l(g, x_i, y_i) = \sum \nabla l(g, x_i, y_i) = 0 \tag{2.19}$$

which is a necessary, but not sufficient condition for having a global minimum at that point. In practice this never occurs and the algorithm is said to converge if the changes become "small enough." It is obvious that for complicated functions like neural networks, this method in general does not produce optimum results. That is, the coefficients to which this algorithms converges do not minimize the loss function, L. However as mentioned with regard to overfitting, completely minimizing the loss function does not necessarily yield good results anyway. One may force convergence of the algorithm by applying an exponential decay to the learning rate. This method is known as exponential learning decay. Note that this may not actually force

convergence in scenarios where the gradient grows exponentially or faster.

SGD is an approximation of Newton's method in two senses. At each step, value $[HL(w)]^{-1}$ is approximated by ηI and $L(g, x, y)$ is approximated as $l(g, x_i, y_i)$. One obvious issue with using SGD is that one must choose the learning rate with little or no guidance from the data. Usually values between 10^{-3} and 1 work fairly well as initial learning rates. This arbitrary factor has caused others to develop more sophisticated techniques which essentially estimates the best learning rate at a given point in time during training.

2.3.3 Adam Optimizer

The Adam Optimizer algorithm [11] calculates how "reliable" each element of the gradient vector is and weighs the updates accordingly. It accomplishes this by estimating the first and second moments of the gradient vector with a moving average exponential filter, estimating the mean and variance of each element. The update weights are inversely proportional to the square root of the second moment, as described in algorithm 1.

Adam requires several more hyperparameters as compared to SGD, but the results tend to be less dependent on them. The parameters β_1 and β_2 represent filter coefficients for estimating the first and second moments of the gradient. Figure 2.5 shows the frequency response with beta ranging from 0.8 to 1.0 logarithmically. Observe that a lower β represents a bias for lower frequencies. It turns out that these estimates of m and v are biased, but can be fixed with a simple multiplicative correction (lines 9 and 10). If you consider the second moment to represent the noise power present in the signal, then the quantity $\hat{m} \oslash \hat{v}$ somewhat represents the signal to noise ratio. Using this interpretation, this update weighting is somewhat similar to a Wiener filter where the components of the signal which have the most noise are proportionally suppressed.

Algorithm 1 *Adam Optimizer.* $(A \circ B)_{ij} = A_{ij} \times B_{ij}$ is the Hadamard product of A and B. $(A \oslash B)_{ij} = A_{ij}/B_{ij}$ is Hadamard division. The hyperparameter α is similar to the learning rate, η in SGD. β_1 and β_2 are filter coefficients representing smoothing amount. The value ϵ is an arbitrary small number to avoid division by 0.

Require: β_1, β_2
Require: α
Require: w
Require: $l(w, x, y)$
$m \leftarrow \vec{0}$
$v \leftarrow \vec{0}$
$t \leftarrow 0$
while w not converged **do**
    $t \leftarrow t + 1$
    $G \leftarrow \nabla l(w, x_{t \bmod N}, y_{t \bmod N})$
    $m \leftarrow \beta_1 m + (1 - \beta_1)G$
    $v \leftarrow \beta_2 v + (1 - \beta_2)(G \circ G)$
    $\hat{m} \leftarrow m/(1 - \beta_1^t)$
    $\hat{v} \leftarrow m/(1 - \beta_2^t)$
    $w \leftarrow w - \alpha \times \hat{m} \oslash (\sqrt{\hat{v} + \epsilon})$
end while

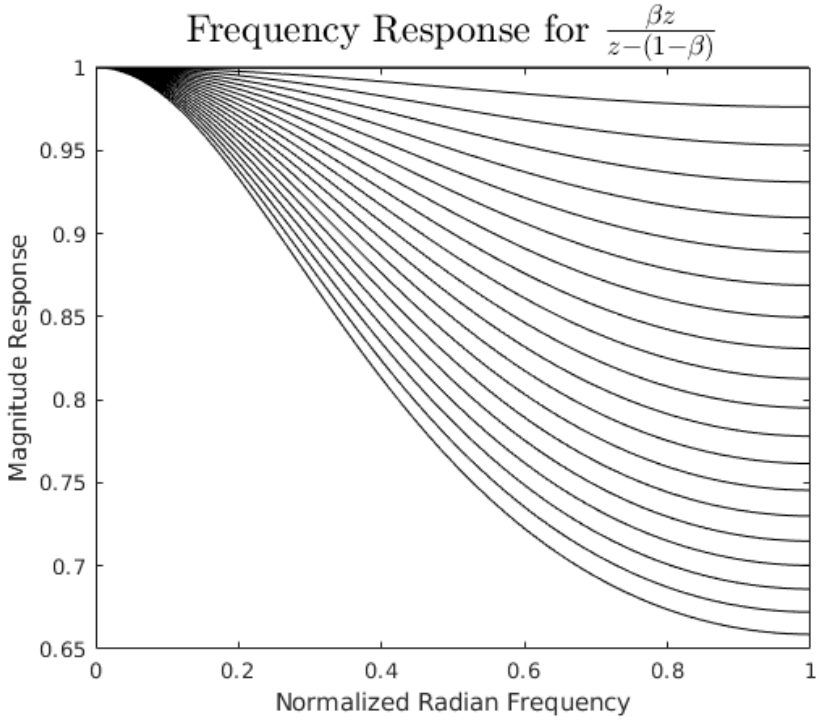


Figure 2.5: Frequency response for several moving average exponential filters. Lower lines are associated with lower values of β.

2.3.4 Automatic Differentiation

In all of the previous training algorithms, computation of the gradient of the loss, $\nabla l(w, x, y)$ has been ignored. This section briefly discuss automatic differentiation, a powerful dynamic programming algorithm for computing the gradient of an arbitrary differentiable function.

Consider some function which depends on multiple variables. In computing the function, it will likely be expressed as the compositions of other functions. An example function taken from [1] is given below.

$$f(x_1, x_2) = \log(x_1) + x_1 \times x_2 - \sin(x_2) \tag{2.20}$$

Equation 2.20 is a composition of log, $\times$, sin, $+$, and $-$. This composition can be captured in the form of a graph, as seen in figure 2.6. Starting at the output and working backwards, one can compute the output of the previous required computation. Automatic differentiation works in much the same way that a normal computation is performed except by applying the chain rule. For example, in order to compute the derivative with respect to x_1 at node v_7, representing the function $[\log(x_1) + x_1 \times x_2] - [\sin(x_1)]$, compute $D_{x_1}[\log(x_1) + x_1 \times x_2] - D_{x_1}[\sin(x_1)]$. In order to find the two component values, look at the nodes associated with those functions where the derivative will either already be computed, or where one will compute the derivative by climbing up the graph the same as in the previous step. Applying this algorithm for all inputs yields the gradient.

2.4 Recurrent Neural Networks

Recurrent neural networks are a subset of neural networks which have a key distinction from multilayer perceptrons: they can be applied to data of arbitrary length. To give

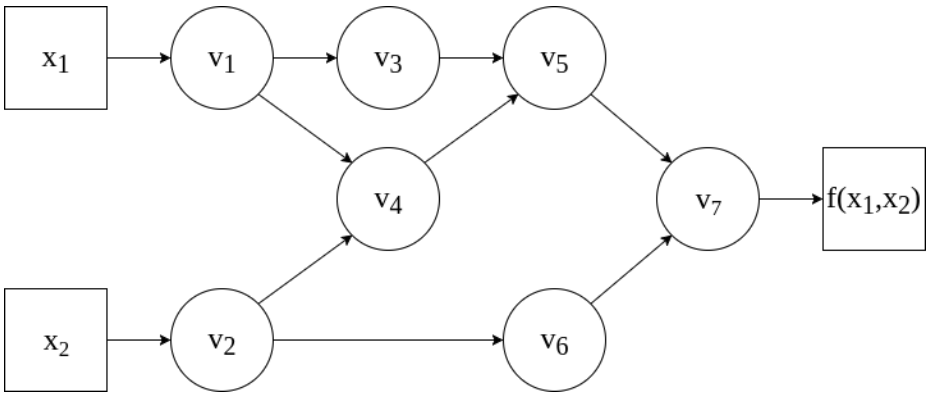


Figure 2.6: Graph reproduced from [1]. Each node represents a function of the incoming nodes.

more detail we introduce some notation. Let the set of all n-dimensional vector sequences (finite or otherwise) be given by

$$S_n = \{s : Z \to \mathbb{R}^n;\ \forall Z \subseteq \mathbb{Z}^+\} \tag{2.21}$$

While a multilayer perceptron is a function $f : \mathbb{R}^n \to \mathbb{R}^m$, a recurrent neural network is a function $R : S_n \to S_m$.

2.4.1 Overview

Of course, the multilayer perceptron could be extended to match this if one windows the sequence, computes the output, and then slides the input window. What truly separates recurrent neural networks from multilayer perceptrons is the fact that an RNN's output depends on its "state" at previous time steps. In particular, denoting the input sequence as $x = (x_1, x_2, \ldots, x_k)$, the states at time steps $1, \ldots, k$ are given by the following:

$$S_1 = s(x_1, S_0) \tag{2.22}$$

$$S_2 = s(x_2, s(x_1, S_0)) \tag{2.23}$$

$$S_3 = s(x_3, s(x_2, s(x_1, S_0))) \tag{2.24}$$

$$\vdots$$

$$S_k = r(x_k, S_{k-1}) \tag{2.25}$$

And the outputs are given by

$$R_1 = r(x_1, S_1) \tag{2.26}$$

$$R_2 = r(x_2, r(x_1, S_1)) \tag{2.27}$$

$$R_3 = r(x_3, r(x_2, r(x_1, S_1))) \tag{2.28}$$

$$\vdots$$

$$R_k = r(x_k, S_k) \tag{2.29}$$

If the output sequence is finite, the output of the RNN is the sequence $R(x) = (R_1, R_2, \ldots, R_k)$, else it is given by $R(x) = (R_1, R_2, R_3, \ldots)$. The function $s : \mathbb{R}^n \times \mathbb{R}^p \to \mathbb{R}^p$ is called the state transition function, and the function $r : \mathbb{R}^n \times \mathbb{R}^p \to \mathbb{R}^p$ is called the output function. It is assumed that S_0 is some arbitrary fixed value (usually zero), called the initial state. Note that this framework is all very similar to the state-space model, ubiquitous in control theory, except that nonlinearity is allowed in this case.

We can see that the adapted version of the MLP would not be able to represent functions which depend on inputs from time steps farther apart than the length of the window. In this regard, MLPs are to RNNs as FIR filters are to IIR filters. Unfortunately because of the non-linearities introduced in RNNs, there is no simple

method of analyzing stability, and so training of RNNs has proved difficult throughout their history. Recent advancements like the long short-term memory unit and dropout have made the training of large RNNs possible.

2.4.2 Early Networks

Some of the first known RNNs, now known as "simple recurrent networks" are the Elman [12] and Jordan [13] networks. The Elman update equations are given by the following equations:

$$S_k = \sigma_S(W_S x_k + U_S S_{k-1} + b_S) \tag{2.30}$$

$$R_k = \sigma_R(W_R S_k + b_R) \tag{2.31}$$

while the Jordan update equations are given by:

$$S_k = \sigma_S(W_S x_k + U_R R_{k-1} + b_S) \tag{2.32}$$

$$R_k = \sigma_R(W_R S_k + b_R) \tag{2.33}$$

The Elman equations are already in a form consistent with the given framework. Substituting the second Jordan equation into the first yields the following:

$$S_k = \sigma_S(W_S x_k + U_R \sigma_R(W_R S_{k-1} + b_R) + b_S) \tag{2.34}$$

$$R_k = \sigma_R(W_R S_k + b_R) \tag{2.35}$$

The functions σ_S and σ_R are sigmoidal activation functions. The parameters W_S,W_R, and U_R are matrices, b_S and b_R are vectors.

A result analogous to the universal approximation theorem was proved in [14]. Given mild conditions on r and s, similar to those of the universal approximation theorem, a recurrent neural net of sufficiently large size is capable of simulating a

universal Turing machine. This means that even a simple RNN (like a Jordan or Elman network) can compute any function, or perform any algorithm that can be performed by a regular computer. Of course just as with the universal approximation theorem, this says nothing about what is required to train such a network, how many resources it would require, or how efficiently it would operate.

2.4.3 Long Short-Term Memory

Consider computing the derivative of the output of a recurrent neural network with respect to one of its feedback weights, w. By the chain rule:

$$\frac{d}{dw}R_k(w, x_k, S_k) = r'(w, x_k, S_k) \times s'(w, x_{k-1}, S_{k-1}) \times$$
$$s'_{k-1}(w, x_{k-2}, S_{k-2}) \times \cdots \times s'(w, x_2, S_1) \times s'(w, x_1, S_0) \tag{2.36}$$
$$= R'_k(x_k, S_k) \prod_{i=1}^{k} s'(x_i, S_{i-1}) \tag{2.37}$$

The log derivative is given by

$$log(R'_k(x_k, S_k)) + \sum_{i=1}^{k} \log(s'(x_i, S_{i-1})) \tag{2.38}$$

Although it is hard to formally say anything about this value, intuitively since values are being accumulated over time, the log derivative acts as an unstable system. If the value goes to some very negative number, then the derivative will become extremely small. On the other hand if the value goes to some very large number, the derivative will become extremely large. These issues are known as the problems of vanishing and exploding gradients respectively.

The driving cause for exploding or vanishing gradients is that when the state vector transitions from one time step to another, it is multiplied by some matrix. The chain rule causes this matrix to be multiplied by itself repeatedly in the derivative

calculation which results in either exponential blow up or decay. The only way to maintain a relatively constant value is for that matrix to be identity, which is exactly the idea behind the long short-term memory (LSTM) unit. [15] The update equations for the original LSTM RNN are given by the following:

$$i_k = \sigma_i(W_i x_k + U_i R_{k-1} + b_i) \tag{2.39}$$

$$o_k = \sigma_o(W_o x_k + U_o R_{k-1} + b_o) \tag{2.40}$$

$$c_k = \sigma_c(W_c x_k + U_c R_{k-1} + b_c) \tag{2.41}$$

$$S_k = S_{k-1} + i_k \circ c_k \tag{2.42}$$

$$R_k = \sigma_R(S_k) \circ o_k \tag{2.43}$$

Where $\circ$ represents Hadamard (entry-wise) multiplication.

It is easy to see that these equations satisfy the update equations of a standard RNN, as defined previously. The direct "connection" between S_k and S_{k-1} helps mitigate the issue of vanishing or exploding gradient. It is important that i_k and c_k can have different signs so that the state is not stuck accumulating in one direction. For this reason, σ_i is usually tanh while σ_o is usually the logistic function.

The model was later extended by [16] with what they called a forget gate. The forget gate, described by the new update equations 2.44-2.45, allows the network to essentially discard information in the state and start again.

$$f_k = \sigma_f(W_f x_k + U_f R_{k-1} + b_f) \tag{2.44}$$

$$S_k = S_{k-1} \circ f_k + i_k \circ c_k \tag{2.45}$$

The same group responsible for the invention of forget gates also invented the peephole LSTM. [17] In this modification of the forget gated LSTM, all instances where the output of the neural network is fed back are replaced with the internal

state instead. For example equation 2.39 becomes

$$i_k = \sigma_i(W_i x_k + U_i S_{k-1}) \tag{2.46}$$

and so on.

2.4.4 Training RNN's

Recurrent neural networks are trained in much the same way that a standard neural network is trained with one key complication, the input is potentially unbounded. This means that the number of computations is potentially unbounded and so there is no single graph representing the network. To solve this issue in computing gradients, the neural network is *unfolded* for some finite amount of time steps. Any input which goes past the maximum time step is discarded in the computation.

Dropout is a regularization method that can be used to avoid overfitting when training neural networks. It as applicable to a wide range of neural net models, though its application to recurrent neural networks is slightly more complex. Essentially, dropout randomly zeros out certain values in the neural net at every time step. Dropout can be applied at the input, output, or internal state of an LSTM cell. It was found that applying dropout to state variables leads to poor performance, while applying dropout to the output of a cell leads to increased performance. [18] This type of RNN regularization simply means changing equation 2.43 to:

$$R_k = D(\sigma_R(S_k) \circ o_k) \tag{2.47}$$

where D zeros random elements of the vector according to independently and identically distributed Bernoulli distributions.

2.5 Numerical Representation of Language

Many modern machine learning techniques require a numerical representation of data and therefore cannot work with natural language directly. There are several different representations which have their own advantages and drawbacks. We will specifically look at vector space representations which are useful for classifying documents. This chapter will cover the simple bag-of-words model of documents and the more advanced *word2vec* model.

2.5.1 Bag-of-words

A bag-of-words model is a very simple numerical representation of a text document. The model assumes that a given document has already been tokenized, i.e. that the plain string of characters has been separated into a list words. In bag-of-words, a single document is represented as the multiset of all tokens in the document with no regard toward ordering. This means that the conversion is, in general, not invertible and so the original document cannot be retrieved from this representation.

Consider for example:

```
"This movie is not terrible, this movie is amazing!"
```

One possible bag-of-words representation for this document is

{this, movie, is, not, terrible, this, movie, is, amazing}

Notice that the above object is not a set, but a multiset since it contains one or more elements more than once. Also note that the following example is equivalent, since it contains the same words with the same frequency.

{this, movie, is, not, amazing, this, movie, is, terrible}

This object is conveniently represented as a hash Table mapping tokens directly to their frequency in the document, like the example in Table 2.1

this	2
movie	2
is	2
not	1
terrible	1
amazing	1

Table 2.1: Hash table representation of a multiset

In the context of comparing two documents, it is sometimes more useful to represent a document as a vector. In order to represent every possible document as a vector of the same length, then the number of elements must be equal to the total number of possible tokens, or the size of the vocabulary. For example, Table 2.1 might be represented as

$$x = \begin{bmatrix} 0 & \cdots & 2 & \cdots & 2 & \cdots & 2 & \cdots & 1 & \cdots & 1 & \cdots & 1 & \cdots & 0 \end{bmatrix}^{\mathsf{T}} \tag{2.48}$$

with dots representing some amount of zeros. With this representation, one simple measure of document similarity is given by the *cosine similarity*, $s_\theta(d_1, d_2)$, of two document vectors:

$$s_\theta(x_1, x_2) = \frac{x_1 \cdot x_2}{||x_1||_2||x_2||_2} \tag{2.49}$$

By the Cauchy–Schwarz inequality, this value has an upper bound of 1, which is achieved if and only if the two documents contain the same words with the same relative frequency.

The vector representation of documents also gives a hint as to how one might represent individual words numerically. If i is the index associated with the i^{th} word in a vocabulary, then one may associate that word with the standard basis vector e_i. Where every component of e_i is 0 except for the i^{th} which is 1. This is known as one-hot encoding. Let e_m be the standard basis vector associated with word m, then

the vector representation for a document multiset, M, would be given by

$$x = \sum_{m \in M} v(m) e_m \tag{2.50}$$

where v is the multiplicity of element m in M.

In most cases, an actual language processing system does not use the raw frequency, $n_{t,d}$, of a term t and document d, but rather some term frequency function, $tf(t,d)$, of the raw frequency and given document which results in what is called a *term weighting*. Common choices are [19]:

1. $tf(t,d) = \{1 \text{ if } t \text{ appears in } d\text{; else } 0\}$
2. $tf(t,d) = n_{t,d}/N$, where N is the length of the document
3. $tf(t,d) = \{1 + \log(n_{t,d}) \text{ if } n_{t,d} > 0\text{; else } 0\}$
4. $tf(t,d) = \frac{1}{2} + \frac{1}{2}\frac{n_{t,d}}{N}$, where N is largest raw frequency in the document.

The inverse document frequency of a term is given by

$$idf(t,D) = -\log d_t/D \tag{2.51}$$

where d is the number of documents containing the term t, and D is the number of documents in total. One of the most common term weightings is then given by the term frequency-inverse document frequency (tf-idf):

$$tfidf(t,d,D) = tf(n,d) \times idf(t,D) \tag{2.52}$$

The bag-of-words model has the advantage of extreme simplicity. However, it does not represent complex documents well since it has no regard for ordering. In addition, using this representation as a feature for machine learning is problematic

since the number of features is equal to the vocabulary size which should be a very large number. The following section discusses latent semantic analysis, originally a method for indexing documents, which represents words and documents as vectors.

2.5.2 Latent Semantic Analysis

Latent semantic analysis (LSA) is essentially a dimensionality reduction technique similar to principle component analysis. At its core is a truncated singular value decomposition (SVD) which is responsible for finding "key concepts" behind words. A set of word vectors for some vocabulary and set of documents can be found with the following steps:

1. Populate the term-document matrix, M: $M_{i,j} = tf(t_i, d_j)$
2. Using SVD, decompose M: $M = U\Sigma V^*$. U and V are square while Σ is diagonal.
3. Keep the largest k diagonal values in Σ, remove the other values' rows and columns and call the resulting matrix $\hat{\Sigma}$.
4. The vector corresponding to the i^{th} word is the transpose of the i^{th} row of $U\hat{\Sigma}$, a vector of k elements.

Using this method, words which appear in similar frequency across similar types of documents will have similar vectors, in the sense of cosine similarity. Sample noise is reduced by the dimensionality reduction achieved by truncating Σ.

2.5.3 Word2vec

Word2vec [20] is an extremely useful model which learns vector representations of individual words. Word vectors are learned in an unsupervised fashion, meaning that very large, unlabeled data sets can be leveraged during training. There are two

word2vec models which are complementary: Continuous Bag-of-Words (CBOW) and Continuous Skip-gram (Skip-gram). CBOW aims to create vectors which maximize accuracy of classifying a word given its surrounding words. Skip-gram aims to create vectors which maximize the accuracy of predicting surrounding words given the center word.

In the skip-gram model, the function used for word prediction is a simple hour-glass shaped log-linear model. Given a one-hot encoded word vector, as described in section 2.5.1, transpose it and right multiply it by some matrix, $A \in M_{V\times D}(\mathbb{R})$. The dimension V represents the size of the vocabulary and D represents the number of dimensions of the word embedding. In general, V should be much greater than D. Note that since the vector is one-hot encoded, this is the same as simply selecting a row from the matrix A. Now take the result and then right multiply some matrix, $B \in M_{D\times V}(\mathbb{R})$. Both of these multiplications combined are equivalent to right multiplying the one-hot encoded vector by some matrix $C \in M_{V\times V}(\mathbb{R})$, but of course much fewer values must be stored and C is constrained to being at most rank D. The resulting vector is then fed through either a softmax (given by equation (2.53)) or hierarchical softmax function (defined later in this section), yielding a probability vector which indicates the estimated probability of any word in the vocabulary being within some skip window, R of the center word.

$$\sigma(\vec{x}) = \frac{\exp(x_i)}{\sum_i \exp(x_i)} \tag{2.53}$$

In the CBOW model, the operation is very similar except that a multiple-hot encoded vector is used at the input. That is, instead of one element of the vector being 1 while the rest are 0, multiple elements are 1 or more indicating the words present within some skip window, R of the center word. Recalling section 2.5.1, this is the bag-of-words representation of the window surrounding the center word. As in the skip-gram

model, the algorithm right multiplies the transpose of this vector by some matrix, $A \in M_{V \times D}(\mathbb{R})$. In the case of skip-gram this amounted to selecting a row. Now that the vector is multiple-hot encoded, this amounts to adding several rows of the matrix together. Again multiply the resulting vector by some matrix, $B \in M_{D \times V}(\mathbb{R})$ and feed the result through a softmax function to obtain a probability vector indicating the likelihood that any word in the vocabulary is the center word.

In either case, all weights are randomly initialized, usually with a Gaussian random variable generator. The cross entropy loss is used as the loss function, and all weights are trained using gradient descent and automatic differentiation (see sections 2.3.2 and 2.3.4). The target vector represents some subset of all words within the skip window, though not necessarily the entire set. The number of skips represents how many times to sample words from the skip window to obtain the target vector. If the number of skips is double the skip window, the algorithm samples all words. Cross entropy is given by:

$$L(x, y, f) = \sum_{i}^{M} I_{y_i} \log(f(x_i)) \tag{2.54}$$

where $I_{y_i} = 1$ if y is the i^{th} class number, else $I_{y_i} = 0$

This method is similar to latent semantic analysis in that it is projecting high dimensional "term vectors" onto lower dimensional spaces. The algorithm differs in some key ways. First, the term-document matrix is essentially replaced with a term-term matrix where frequency is associated between terms and terms, rather between than terms and documents. This makes it possible to learn a *word2vec* representation with just one document or corpus. Second, the time complexity for computing the singular value decomposition of the term-term matrix is $\mathcal{O}(min(VT^2, V^2T)) = \mathcal{O}(V^2T)$ where V is the vocabulary size and T is the number of terms in the data set, including repeated terms. This value is very large for a large vocabulary. On the other hand,

the time complexity for training a *word2vec* representation is $E \times T \times C \times D \times \log V$, where E is the number of training epochs, C is the size of the maximum time difference of words, and D is the dimension of each word. Word2vec is then clearly a good candidate for scenarios with large vocabularies.

There are several extensions and improvements which improve the quality of the word vectors as well as the training speed. [21] Here we discuss hierarchical softmax, negative contrastive estimation, and subsampling of frequent words.

The typical softmax function with N inputs has N outputs. If one is classifying words from a vocabulary this value is typically very large, on the order of $10^4 - 10^7$. Hierarchical softmax is a computationally efficient approximation which reduces the computational load from N to about $\log_2(N)$. This optimization significantly changes the structure of the algorithm. Instead of computing the output as a matrix multiply followed by a softmax, the operation is performed in a hierarchical fashion. A given node, n in a binary tree is assigned a vector, v_n (as opposed to each row in a matrix). Suppose the path to a word u is given by the set of nodes p_u, including u. Then the hierarchical softmax of u and word w with vector v_w, is given by:

$$\hat{\sigma}(u, w) = \prod_{n \in p_u} \sigma(s(n) \times v_n^T v_w) \tag{2.55}$$

The sign function $s(n)$ arbitrarily maps nodes to a value of either 1 or -1 with the rule that two children of a given node cannot have the same sign. Assigning the edge connecting node n to its parent the value $\sigma(s(n) \times v_n^T v_w)$, consider each edge's value as being the conditional probability that a connecting child node is picked given that all of its ancestors have been picked. Since $\sigma(-x) + \sigma(x) = 1$, the sum of any two edge values connecting to children of a node is 1 meaning this is a valid conditional probability distribution. Now suppose that L nodes are in the path, p_u, before u. Denote them as $n_1, n_2, \ldots, n_L$. The probability of picking word u given the word w is

given by the probability of picking all nodes on the path from the root to u, inclusive:

$$P(u|w) = P(u, n_L, n_{L-1}, \ldots, n_1|w) \tag{2.56}$$

$$= P(u|n_L, \ldots, n_1, w) \times P(n_L|n_{L-1}, \ldots, n_1, w) \times \cdots \times P(n_1|w) \tag{2.57}$$

$$= \hat{\sigma}(u, w) \tag{2.58}$$

Equation (2.57) follows from (2.56) by the chain rule of probability. Since the largest length in the tree is typically not greater than $\log_2(V)$, this method gives exponential speedup compared to a linear calculation.

The alternative, which was used in this study, is negative sampling, a simplification of negative contrastive estimation. [22] Instead of a typical cross entropy loss, negative contrastive estimation is a modified loss with drastically reduced computational complexity. Let u be one of the target words, $\hat{v}_u$ is the column in the second layer matrix corresponding to word u, w is the center word, k is the number of negative samples, and w_i is drawn randomly from the distribtuion $P(w)$. If we define the objective G as:

$$G(u, w) = \log \sigma(\hat{v}_u^T v_w) + \sum_{i=1}^{k} \mathrm{E}\left[\log \sigma(-\hat{v}_{w_i}^T v_w)\right] \tag{2.59}$$

Then the objective increases as the w vector becomes more similar to the u vector while becoming less similar to "noise" vectors, v_{w_i}. Negative sampling (NEG) is defined by maximizing the objective in equation 2.59

This method reduces the computation of size V to one of size k, a small constant usually on the order of $5 - 20$. $P(w)$ is a probability mass function, a free parameter that must be chosen by the user. The authors of [21] found that the following

probability mass function works well:

$$P(w) = \frac{f(w)^{3/4}}{\sum_{i=1}^{V}(f(w_i)^{3/4})} \tag{2.60}$$

where $f(w)$ is the fraction of times the word w appears in the corpus.

Finally, performance can be improved by subsampling frequently occurring words like "in", "the" and "a". These words usually do not offer much information about their surrounding words. This is especially true in the skip-gram model which tries to classify surrounding words based on the center word. Given the word "the," it is nearly impossible to say anything about the words within a modest context size.

This effect can be avoided by randomly removing words from the training set, with higher probability if they are more common. The probability of discarding a word is given by $P(w)$ which, again, is a free parameter. The authors of [21] chose the following:

$$P(w) = \begin{cases} 0 & \text{if } f(w) < t \\ 1 - \sqrt{\frac{t}{f(w)}} & \text{else} \end{cases} \tag{2.61}$$

where the threshold t is another free parameter. The function was chosen since it is non-decreasing and therefore preserved frequency rank, and also because it "aggressively subsamples words whose frequency is greater than t." [21]

Chapter 3

Adversarial Derivations

Here we look at the problem of crafting adversarial derivations for text. Our goal is to alter a text sample in a small way which is not immediately noticeable to a reader, and yet causes a classifier to misclassify a particular sample. This problem was originally studied in the context of image classification with convolutional neural networks. This chapter considers a different kind of adversarial derivations/example.

3.1 Adversarial Examples

Adversarial examples were originally defined in the domain of image classification in the form of a constrained optimization problem. The following definition is similar to [6]. Take images to be vectors in $\mathbb{R}^m$, and S to be a finite set of classes, so that a classifier $f : \mathbb{R}^m \to S$ assigns each image a unique class.

Definition. An adversarial derivation, $x + r \in \mathbb{R}^m$, of an image, $x \in [0, 1]^m$ for a classifier, f, is a solution to the following optimization problem:

Minimize $||r||_2$ subject to:

1. $f(x + r) \neq f(x)$

2. $x + r \in [0, 1]^m$

The adversarial derivation is denoted as $x^* = x+r$. In words, an adversarial derivation is a sample with the minimum distance to a true sample such that it is classified as something different. A derivation also requires that every element stays in the interval $x_n + r_n \in [0, 1]$ because actual pixels must remain in some constant bounded interval. We generalize this concept.

Definition. A classifier, f, over a set, D, is a mapping, $f : D \to \{1, 2, \ldots, K\} = C$.

Clearly, this adversarial derivation definition does not translate well to the problem of natural language classification where the domain is not even numeric. A more general constrained optimization definition is given below, which is then adapted to the problem of creating adversarial derivations for text.

Definition. A distance function, d, over a set S is a mapping, $d : S \times S \to \mathbb{R}$ such that $\forall x, y, z \in S$:

1. $d(x, y) \geq 0$
2. $d(x, y) = 0 \iff x = y$
3. $d(x, y) \leq d(x, z) + d(z, y)$

One important example of a distance metric is the discrete metric, given by

$$\rho(v, v^*) = \begin{cases} 0 & \text{if } v = v^* \\ 1 & \text{if } v \neq v^* \end{cases}$$

It is also true that if S is a normed linear space, $d(x, y) = ||x - y||$ is a distance metric over that space.

Definition. Let d_D be a distance function over D, and d_C be a distance function over C. Then the point x^* is an adversarial derivation (AD) of $x \in D$ with tolerance $\epsilon > 0$ if it is a solution to the following constrained optimization problem:

$$\begin{aligned} \min_{x^* \in D(f)} \quad & d_D(x, x^*) \\ \text{subject to} \quad & d_C(f(x), f(x^*)) > \epsilon \end{aligned}$$

Note that an image classifier would have the domain $[0,1]^m$ and codomain $\{1, ...K\}$ where K is the number of classes. So the definition of an adversarial derivation for an image classifier is the same as the general definition, letting $\epsilon = 1/2$, $d_D(x, x^*) = ||x - x^*||^2$, $d_C(y, y^*) = |y - y^*|$.

As long as the function, f, has a non-singleton codomain, the constraint is satisfiable for small enough ϵ. In the case that the solution does not exist, $\delta = \inf\{d_D(x, x^*) \text{ s.t. } d_C(f(x), f($ $\epsilon\}$ exists, in which case one may find $\hat{x}$ that is arbitrarily close to δ. Satisfiability is true from the fact that for any distance metric, $d(y, y^*) > 0$ for $y \neq y^*$. That all being said, the distance between the sample and the derived sample may be so large that they are easily recognized to be different. A less relaxed definition follows.

Definition. An absolutely adversarial derivation (AAD) of x with similarity δ, difference ϵ, domain metric d_D, and codomain metric d_C is given by any solution to the following two constraints.

$$d_D(x, x^*) < \delta$$

$$d_C(f(x), f(x^*)) > \epsilon$$

In this less relaxed version of the problem, a solution may not exist for a given pair of δ, ϵ values. In fact, for a given continuous function, f, defined on an open domain, and tolerance, ϵ it is guaranteed that

$$\exists \delta > 0 \text{ s.t. } d_C(f(x), f^*(x)) < \epsilon$$

meaning no solution exists. However, it appeals to a more intuitive concept and allows for the possibility of a model immune to adversarial attacks. It says that the sample and its absolute adversarial derivation must be sufficiently close and the distance between the model outputs must be sufficiently far.

It is easy to see that if an AAD exists for a given sample, then any AD is also an AAD. This means one may solve the more relaxed problem to obtain a valid solution, and so we will work with the more practical objective of creating ADs.

3.2 Word Embeddings

Consider the common scenario of a text classifier which maps plain text files to one of several classes. It is common for the plain text to first be processed into a sequence of tokens, which are then each assigned an integer resulting in a sequence of integers.

Definition. Let s be a sequence of characters. Let $a_n \in \{0, 1, \ldots, V\} \forall n \in \{0, 1, \ldots, N\}$ and $E : s \to \{a_n\}_{n=1}^{N_s}$. Then E is called an encoder, V the encoder vocabulary size, and N_s the sample length with respect to E.

In plain words, an encoder maps a string to a finite sequence of bounded integers. The sequence length depends on both the encoder and the string. Assume a fixed encoder, and therefore vocabulary size, V. Since, after encoding, the distance between one word and another is arbitrary, the numerical representation can be further translate into a one-hot encoded vector. That is, the integer n is mapped to a vector where the n^{th} element is 1 and all others are 0. This ensures that all vectors representing words are unit norm and the distance between any two different words is the same. The set of one-hot encoded vectors of size V is denoted as 1_V.

This simple method of representing words as vectors results in a very high dimension representation of all words in the vocabulary, and thus even a very simple linear model would be very large and difficult to train. Using the word2vec model

discussed in the previous chapter, the dimension of this representation can be significantly reduced, while also encoding information about statistical semantic similarity about each word.

Definition. Let $f : 1_V \to \mathbb{R}^D$. the function f is called a word embedding and D is the size of f, or embedding size.

Let $W \in M_{D\times V}(\mathbb{R})$. Then clearly any word embedding, f, of size D may be represented as the matrix multiplication $Wv \ \forall v \in 1_V$. The matrix W is called the embedding matrix. This numerical representation of words is extremely useful since it allows one to apply more general and modern techniques to solving the problem of classification.

3.3 Adversarial Text Derivation

With clear definitions regarding the domain and numerical representation of words, we define an adversarial derivation of textual data in the context of a classification model, f. As per the definition of an adversarial derivation, one needs only to define the model tolerance, ϵ, as well as the domain metric, d_D and codomain metric, d_C. Recall ρ is the discrete metric with $\rho(a, b) = 0$ if $a = b$ and $\rho(a, b) = 1$ if $a = b$.

Definition. Let $\{v_i\}_{i=1}^N$ be the sequence of vectors obtained from a given word embedding and text sample. Then a discrete adversarial derivation is defined using the domain metric, $d_D(v, v^*) = \sum_i^N \rho(v_i, v_i^*)$, codomain metric $d_C(f(v), f(v^*)) = \rho(f(v), f(v^*))$, and tolerance $\epsilon = 1/2$.

That is, a discrete adversarial derivation $\{v_i^*\}$ of sample $\{v_i\}$ is the sample which changes the fewest number of words possible, while changing the classification.

3.3.1 Fast Gradient Sign Method

One existing technique for textual adversarial derivation is inspired by the fast gradient sign method for images (not discussed in this work). Through this paper, it will simply be refered to as the fast gradient sign method (FGSM). The pseudocode is given in Algorithm 2. [7] Essentially, it says to match the sign of the gradient to the sign of the difference between word vectors as well as possible at every step. This algorithm is the baseline against which this work's algorithms are tested. Its performance in the original paper and in our experiments are discussed in Chapter 6.

Algorithm 2 Fast Gradient Sign Method

Require: f ▷ Differentiable classifier model
Require: x ▷ Sequence of words forming a text sample
Require: W ▷ List of word vectors, a word embedding
1: $y \leftarrow f(x)$
2: $x^* \leftarrow x$
3: **while** $f(x^*) = y$ **do** ▷ Terminate when misclassification occurs
4: Randomly select a word index, i in the sequence x^*
5: $w \leftarrow \arg\min_{z \in W} ||sgn(x^*[i] - z) - sgn(\nabla_i f(x^*))||_1$
6: $x^*[i] \leftarrow w$
7: **end while**

Chapter 4

Preliminary Results

4.1 Word Embedding Training

In our example experiment, we used gradient descent to train a word2vec model with negative sampling. All gradients were calculated with the automatic differentiation framework, TensorFlow [23]. The hyperparameters chosen are shown in Table 4.1.

Learning rate	1.0
Batch size	128
Number of Batches	100,000
Embedding size	128
Number of skip windows	8
Size of skip windows	4
Vocabulary size	10,000

Table 4.1: Word embedding hyperparameters.

Our corpus was the concatenation of all preprocessed training samples from the training set in the "Large Movie Review Dataset." [24] Preprocessing consisted of the steps laid out in Table 4.2

The processing resulted in a corpus of 5,887,178 words totaling 31,985,233 characters. Since there are 25,000 training samples, each review is on average about 234 words, and 1279 characters. Of all 25,000 reviews training reviews, 27 had more

Start	The movie isn't {<br \> }good, I give it a 1
Convert to lower case	the movie isn't {<br \> }good, i give it a 1
Remove HTML	the movie isn't good, i give it a 1
Expand contractions	the movie is not good, i give it a 1
Remove punctuation	the movie is not good i give it a 1
Expand numbers	the movie is not good i give it a one
Remove extra whitespace	the movie is not good i give it a one

Table 4.2: Preprocessing Algorithm

than 1024 words. The word embedding converged to an average noise contrastive loss of about 5.07. Semantic difference between two words is measured by the angular distance between their embeddings, computed as

$$\frac{cos^{-1}\left(\frac{v^T u}{||v||_2||u||_2}\right)}{\pi}$$

The eight nearest neighbors for a few common words are shown in Table 4.3. The first few nearest neighbors are generally fairly high quality, and would usually make grammatical sense for replacement in a sentence. The quality of replacement falls off quickly after that however.

all	but	some	and	UNK	just	also	that	so
and	UNK	with	but	also	which	simpler	nerd	just
will	can	would	if	do	could	to	did	you
of	in	from	with	for	UNK	about	which	that
her	his	she	him	he	their	UNK	the	india
she	he	her	him	his	who	UNK	it	that
most	all	best	films	which	other	UNK	some	only
one	three	two	zero	five	only	nine	s	UNK
movie	film	show	story	it	but	really	that	just
film	movie	story	show	which	UNK	it	that	but

Table 4.3: Some examples of nearest neighbors in embedding space. UNK is a symbol for a word that is not in the vocabulary.

Number of Hidden Units, Learning Rate = 0.01						
2	4	8	16	32	64	128
0.778	0.784	0.812	0.850	0.878	0.904	0.961
Number of Hidden Units, Learning Rate = 0.1						
2	4	8	16	32	64	128
0.827	0.853	0.895	0.965	0.975	0.997	0.990

Table 4.4: Training Accuracy

4.2 RNN Training

This study focuses on one particular type of model which has proved to be very effective in text classification and prediction, a recurrent neural network utilizing long short-term memory (LSTM) units. Our base model has one layer of LSTM units where the output of each unit is averaged over time followed by a fully connected layer with two outputs, and a softmax function. The output represents the neural network's confidence in its prediction.

The model was trained for 50 epochs over the entire training set with a batch size of 1000 and a maximum unfolding length of 1024, meaning that 27 reviews would be clipped. The Adam optimizer with exponential step decay factor of 0.8 every 500 batches was used to minimize the model loss, softmax cross entropy. We used forget gates, peepholes, and output dropout for training.

We trained fourteen models, varying the number of hidden units and initial learning rates. The final training and testing accuracies are shown in tables 4.4 and 4.5 respectively. As expected, increasing the model size always increased the training accuracy, as did increasing the learning rate. The testing accuracy was less predictable. Most accuracies were in the neighborhood of 80%, 30% greater than the baseline of 50% random guessing.

The average training and test losses of each model are given in figures 4.2 and 4.1. The spacing on the training curve is finer because the value was sampled every epoch instead of every five epochs as in the case of the testing accuracy.

Number of Hidden Units, Learning Rate = 0.01						
2	4	8	16	32	64	128
0.753	0.755	0.771	0.809	0.816	0.829	0.834
Number of Hidden Units, Learning Rate = 0.1						
2	4	8	16	32	64	128
0.793	0.816	0.827	0.822	0.815	0.809	0.817

Table 4.5: Testing Accuracy

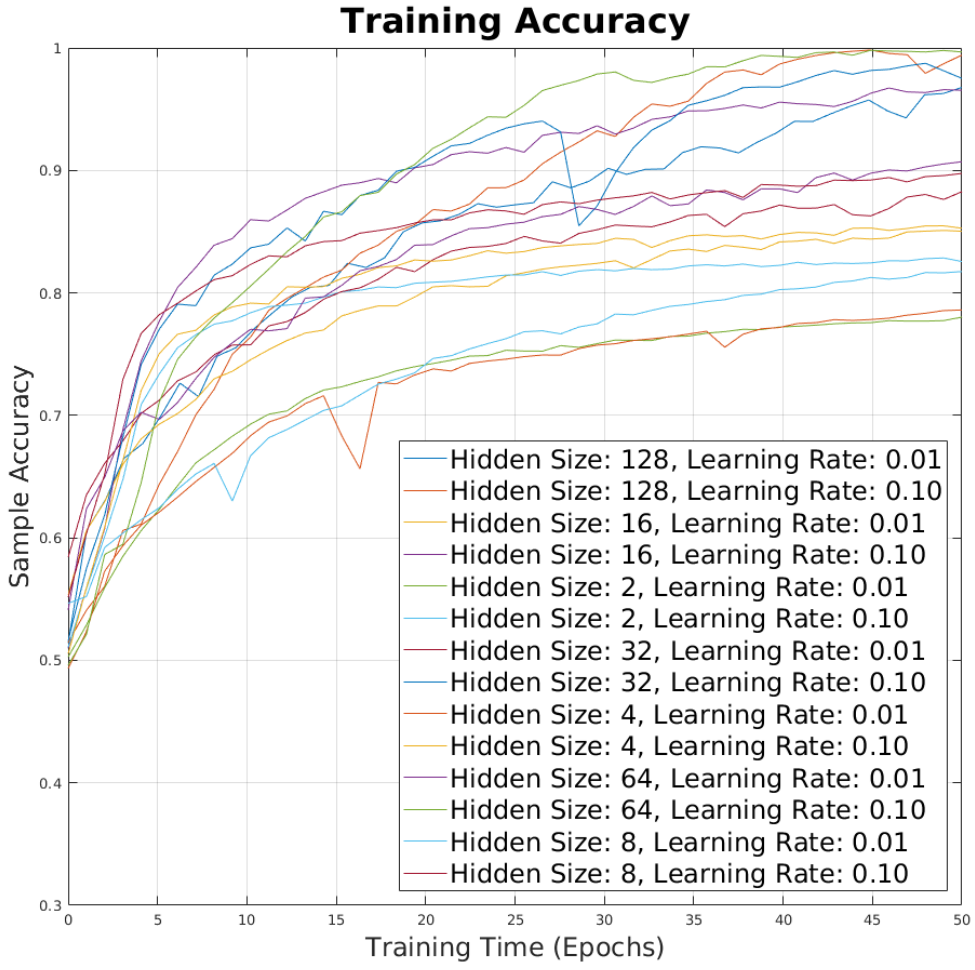


Figure 4.1: Training Accuracy of RNNs

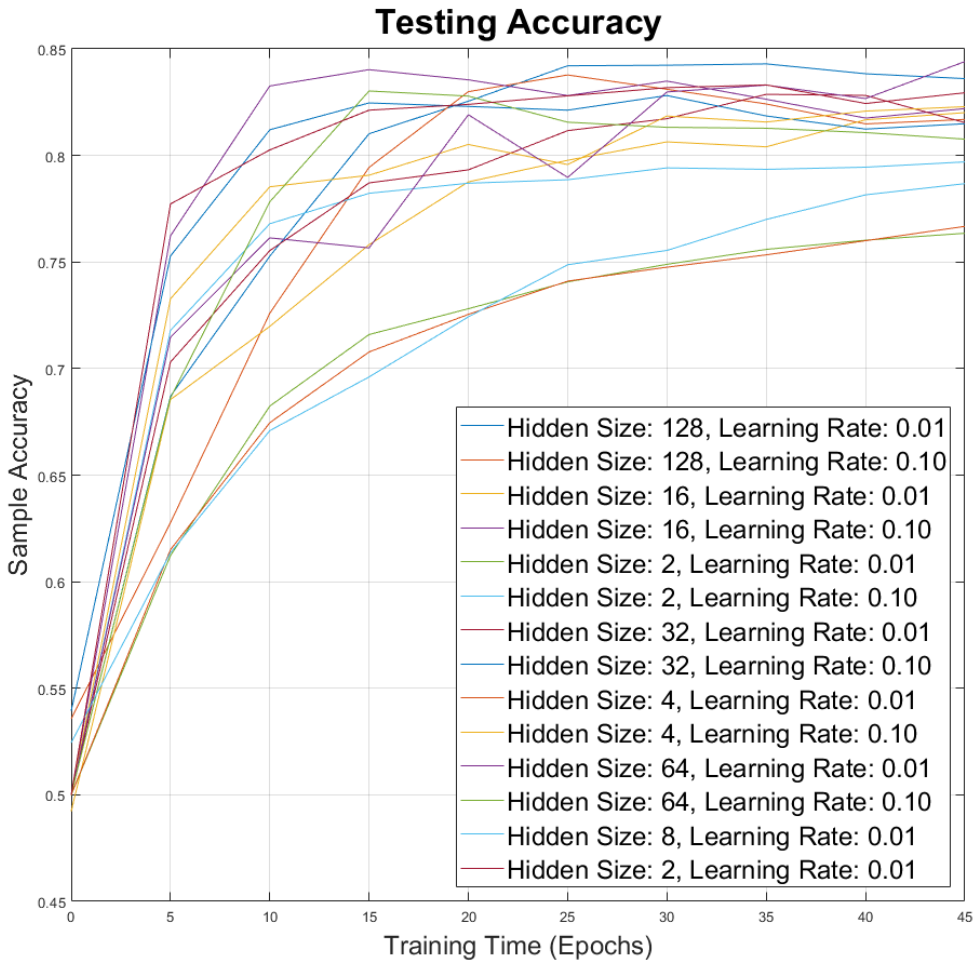


Figure 4.2: Testing Accuracy of RNNs

4.3 Stochastic Gradient Analysis

Although the set of possible inputs to our model is discrete, the model itself still has a continuous codomain. Therefore, we can still define the gradient of the function with respect to one of its inputs. We can do this because the neural network doesn't "know" that its input domain is discrete and in fact the same neural network could have been trained off of continuous data. Consider the j^{th} element of the gradient of the i^{th} output component of the model, with respect to some word, x. Fixing all other inputs consntant, this value is given by the derivative of the i^{th} output component with respect to the j^{th} embedding dimension of word x.

Definition. Let the gradient of a model output, f_i, with respect to an input vector, x be denoted by $g = \nabla f(x)_i$. The total gradient of the i^{th} output is given by the sum of the gradient components:

$$g_t = \sum_{i=1}^{D} \nabla f(x)_i \tag{4.1}$$

The gradient norm is given by

$$g_n = \sqrt{\sum_{i=1}^{D} |\nabla f(x)_i|^2} = ||\nabla f(x)||_2 \tag{4.2}$$

Both of these measures, along with the gradient itself, are shown for our model as well as a small excerpt of one of the text samples in Figure 4.3. The three words with the largest total gradients are "loved", "good", and "bad" which are all sentimental. First, a differentiable function, f, can be locally approximated by

$$f(w_j) \approx f(x_i) + (w_j - x_i)^T \nabla f(x_i) \tag{4.3}$$

$$= f(x_i) + w_j^T \nabla f(x_i) - x_i^T \nabla f(x_i) \tag{4.4}$$

$$= f(x_i) + w_j^T g - x_i^T g \tag{4.5}$$

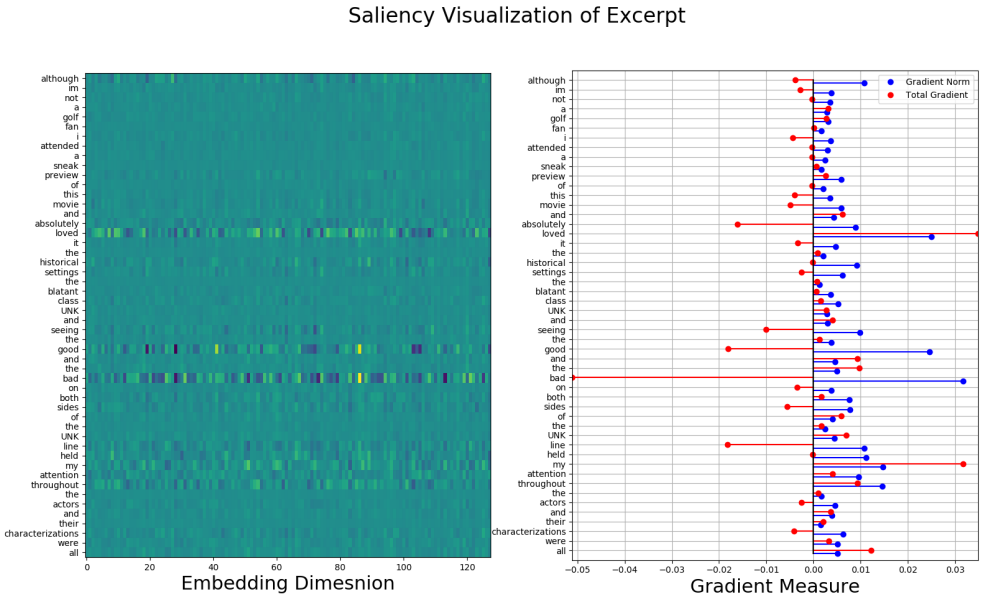


Figure 4.3: Different measures of word sentiment/importance

The following analysis considers specifically the affect of replacing the i^{th} input word vector, x_i with the j^{th} embedding word vector, w_j while all other input vectors remain constant.

Definition. With the above definitions of x_i, w_j, and $\nabla f(x_i)$, let

$$X = \Big[x_1, x_2, \ldots, x_N\Big] \in M_{D\times N}(\mathbb{R}) \tag{4.6}$$

$$W = \Big[w_1, w_2, \ldots, w_V\Big] \in M_{D\times V}(\mathbb{R}) \tag{4.7}$$

$$G = \Big[\nabla f(x_1), \nabla f(x_2), \ldots, \nabla f(x_N)\Big] \in M_{D\times N}(\mathbb{R}) \tag{4.8}$$

The actual perturbation that comes from replacing word w_j with x_i is given by the $(i,j)^{th}$ value of the delta matrix:

$$D_{i,j} = f(w_j) - f(x_i) \tag{4.9}$$

The approximated perturbation of replacing the i^{th} input with the j^{th} embedding

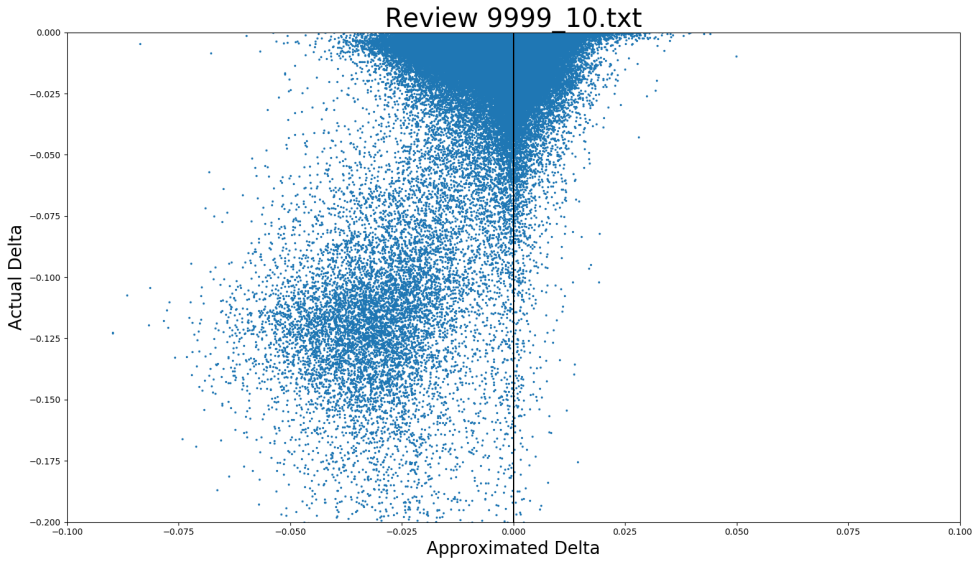


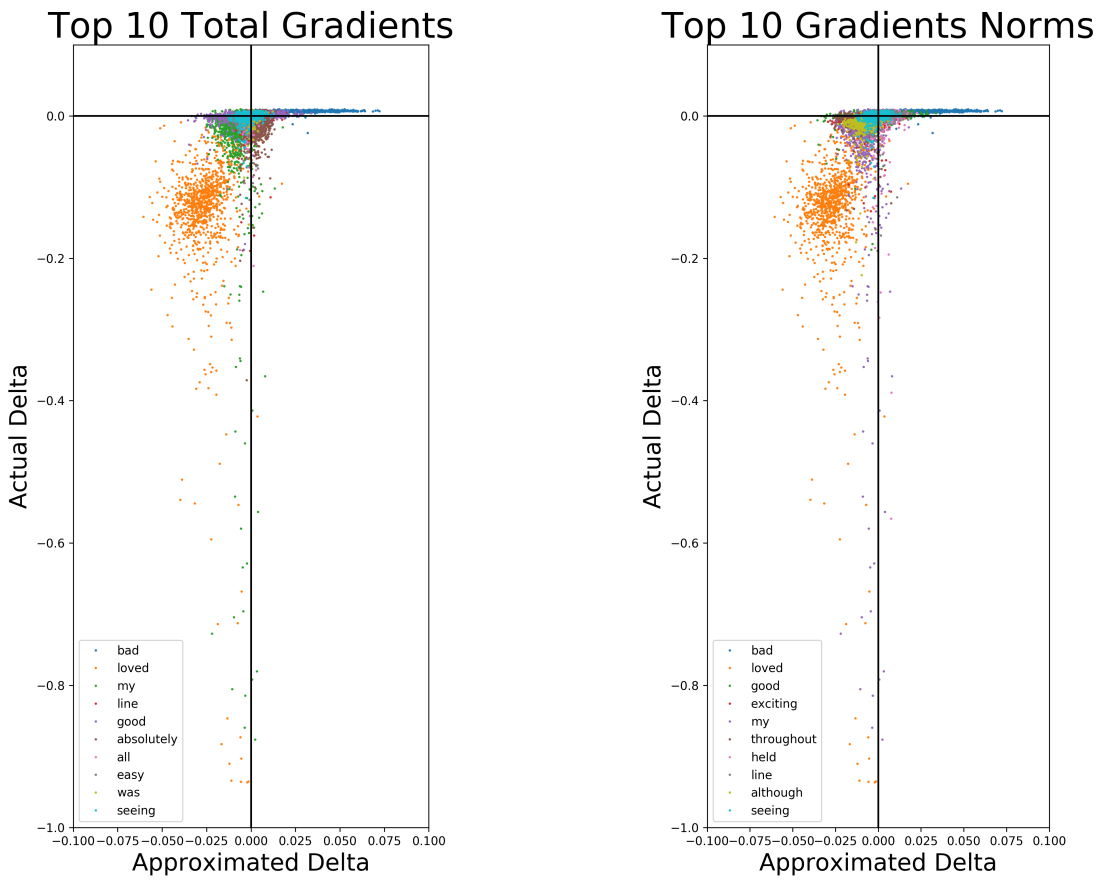
Figure 4.4: Predicted difference in prediction confidence vs. actual difference for sample file 9999_10.txt

word vector is given by the $(i, j)^{th}$ value of the approximate delta matrix:

$$\hat{D}_{i,j} = (G^T W)_{i,j} - (G^T X)_{i,i} \tag{4.10}$$

For a perfectly linear function, f, $D = \hat{D}$. For small output differences, the approximation worked fairly well in that a lesser approximate delta tended to produce a lesser actual delta, as seen in Figure 4.4. However, the approximation failed almost completely for large differences, as illustrated in Figure 4.5. This is not surprising given that the model is highly non-linear but it confirms that using the gradient alone is not a reliable technique for discovering adversarial derivations.

Individual $\hat{D}$ entries are not a good choice for determining which specific words to interchange, but measures of gradient may be useful in determining which words are susceptible to attack. Motivation is given with some simplified probabilistic analysis. Suppose that each of the embedding dimensions is distributed independently and identically across words, with some mean, μ and variance, σ^2. Let $g = \nabla f(x_i)$ for a



(a) The words with the top ten largest absolute total gradients are chosen

(b) The words with the top ten largest gradient norms are chosen

Figure 4.5: The predicted change in classifier confidence vs. the actual change. Only the most common 1,000 words are considered in replacement for this example. The words are listed in the legend in order of decreasing value.

given word vector, x_i and suppose it is replaced it with a random word vector, w_j. Recalling equation (4.5), the following analysis aims to estimate the term $w_i^T g = g^T w_i$ probabilstically since the term $x_i \nabla f(x_i)$ is fixed for a given word. Let $v = w_i$, then randomly replacing x_i with another word vector yields:

$$\mathrm{E}\left[g^T v\right] = \mathrm{E}\left[\sum_{i=1}^{D} g_i v_i\right] = \sum_{i=1}^{D} g_i \,\mathrm{E}\left[v_i\right] = \mu \sum_{i=1}^{D} g_i = \mu g_t \tag{4.11}$$

If the goal is purposefully altering classification, however, one might be more interested in the expected maximum value of $g^T v$, that is,

$$Z(g) = \mathrm{E}\left[\max_{0 \le n \le V} g^T v_n\right] \tag{4.12}$$

Unfortunately, there is no simple expression which captures this value. However assuming that $v_{n,i}$ is distributed normally, then $g^T v_n = p_n$ is also distributed normally with $p_n \sim \mathcal{N}(\mu g_t, \sigma^2 g_n^2)$. Then:

$$Z(g) = \mathrm{E}\left[\max_{0 \le n \le V} p_n\right] \tag{4.13}$$

There is still no closed form expression for this value, but there is a known [25] upper bound:

$$Z(g) \le \mu g_t + \sigma g_n \sqrt{2 \log V} \tag{4.14}$$

$$\mathrm{E}\left[\max \hat{D}_{i,*}\right] \le \mu g_t + \sigma g_n \sqrt{2 \log V} - u^T g \tag{4.15}$$

This inequality is intuitively satisfying. It says that the expected maximum perturbation grows with both the vocabulary size and the norm of the gradient. The total gradient also plays a role here, increasing or decreasing the expected maximum depending on the sign. Empirical study of our embedding and classifier show that the

term containing the standard deviation is usually much larger than the other two. It should be noted that the lower bound on the expected minimum, $Y(g)$, is simply given by a sign reversal of the second term:

$$Y(g) \geq \mu g_t - \sigma g_n \sqrt{2 \log V} \tag{4.16}$$

$$\mathrm{E}\left[\min \hat{D}_{i,*}\right] \geq \mu g_t - \sigma g_n \sqrt{2 \log V} - u^T g \tag{4.17}$$

In our word embedding, the average value of μ was -0.017, but there was significant variation across embedding dimension. The average standard deviation was fairly constant over embedding dimension, with a value of 0.55. The exact value of the result is not important however. What this tells us is that words associated with larger gradient norms have a proportionally larger chance of producing an outlier, at least according to the linear approximation.

Chapter 5

Exponential Windowed Searches

While gradients overall tend to be useful indicators for small perturbations, they are not entirely useful or accurate in predicting large differences. This makes sense given that gradient is only a local measure of change. Since adversarial derivation seeks to find large differences, an algorithm should not rely on the gradient alone. We discuss three improvements here, each of which builds on the last. Since it was found that in most cases a classification change could be achieved by replacing only one word, the base strategy focuses on this objective. If it turns out that more replacements are required, any of the algorithms presented can be extended with an exponential search.

In the case of replacing a single word, the entire search space for a given sample can be described as the Cartesian product of the set of vocabulary words and the set of all sample words. That is, if V ($|V| = N$) denotes the set of vocabulary words and S ($|S| = M$) denotes the set of sample words (along with their location in the sample), the search space is given by $V \times S$, with the size of the set being $|V \times S| = NM$. It is convenient to consider sequence equivalents of V and S: $v = (v_i)_{i=1}^{N}$ and $s = (s_i)_{i=1}^{M}$. Let a classification function for a sample be given by $r(s)$, and let the sequence $s^{i,j}$

Parallelization Tier	Time	Space
0	$\mathcal{O}(NM^2)$	$\mathcal{O}(W)$
1	$\mathcal{O}(NM)$	$\mathcal{O}(WM)$
2	$\mathcal{O}(M^2)$	$\mathcal{O}(WN)$
3	$\mathcal{O}(M)$	$\mathcal{O}(WNM)$

Table 5.1: Time and space complexities for full search with varying levels of parallelization.

be given by:

$$s^{i,j} = (s_1, s_2, \ldots, s_{i-1}, v_j, s_{i+1}, \ldots, s_M) \tag{5.1}$$

5.1 Full Search

For the objective of finding a single adversarial replacement, the brute force solution is simply a full search. Generate the full matrix:

$$M_{M,N}(\{0,1\}) \ni A_{i,j} = r(s^{i,j}) \tag{5.2}$$

and search for the symbol that corresponds to an adversarial example. This method of course has the advantage of achieving the upper bound for performance in generating adversarial derivations with one-word replacements. It has the disadvantage of being very time intensive. Assuming the classifier's time complexity is linear in the length of the sample, this algorithm's time complexity is $T = \mathcal{O}(NM^2)$ and its space complexity is $D = \mathcal{O}(W)$, where W is the number of weights stored in the model.

Of course with parallel operations, time complexity can be offloaded to memory complexity as illustrated in Table 5.1 Note that since the operation of a single recurrent neural network is not parallelizable, the time complexity will always have a factor of at least M no matter how much memory or computing power is available. With regard to the adversarial matrix A, the four complexities above correspond to com-

puting one entry at a time, one row at a time, one column at a time, and computing the entire matrix in parallel, respectively.

5.2 Window Search

As discussed in the previous section, a full search is not feasible for long samples. Depending on how the computation is organized, the algorithm will either run out of memory or take far too long to either find a solution or determine that there isn't one. Looking at the time complexities in Table 5.1, this is intuitive given that there is a quadratic dependence on M, the length of the sample. As mentioned in section 4.1, the average length of a review is about 234 words, meaning that typically $M^2 > N$ and sometimes $M^2 \gg N$. This paper considers two ways to deal with this quadratic growth.

The simplest solution is to cap M at some fixed value, C, and discard the rest of the sample. This method is called cap search. As discussed in section 2.4.4, this is in fact an efficient method used to train recurrent neural networks. Since V and W are also fixed for a given model, this has the advantage of fixing the memory usage in any of the above scenarios. This is desirable since available memory is a static resource which does not change from iteration to iteration, while time is more flexible. On the other hand, this method has the obvious drawback that it may either yield an example which is not truly adversarial, or it may fail to find an adversarial example although there is one. The time and space complexities are given in Table 5.2. This search generates the entries of a matrix A^{cs}, which we hope is approximately the same as the top C rows of A. Pseudocode is given in Algorithm 3.

It may be valuable to consider all words in a review for replacement, especially very negative or positive ones. In this case, the algorithm should still find those words and therefore should consider every word in the sample as a candidate for

Algorithm 3 Simple cutoff algorithm. Note that this algorithm does not require storage of a matrix as written, but it is convenient for generalization.

Require: s ▷ the sample
Require: r ▷ the classifier function
Require: c ▷ the class obtained
Require: C ▷ the cutoff value
Require: N ▷ the number of words in the vocabulary

```
S ← (s_1, s_2, ..., s_C)
for i = 1 to i = C do
    for j = 1 to j = N do
        A^cs_{i,j} ← r(S^{i,j})
        if A^cs_{i,j} ≠ c then
            if r(s^{i,j}) ≠ c then
                return s^{i,j}
            end if
        end if
    end for
end for
return None
```

Parallelization Tier	Time	Space
0	$\mathcal{O}(NC^2)$	$\mathcal{O}(W)$
1	$\mathcal{O}(NC)$	$\mathcal{O}(WC)$
2	$\mathcal{O}(C^2)$	$\mathcal{O}(WN)$
3	$\mathcal{O}(C)$	$\mathcal{O}(WNC)$

Table 5.2: Time and space complexities for full search with sample length capped at C.

Parallelization Tier	Time	Space
0	$\mathcal{O}(NMC)$	$\mathcal{O}(W)$
1	$\mathcal{O}(NC)$	$\mathcal{O}(WM)$
2	$\mathcal{O}(CM)$	$\mathcal{O}(WN)$
3	$\mathcal{O}(C)$	$\mathcal{O}(WNM)$

Table 5.3: Time and space complexities for window search with a window of size C.

replacement. Instead of simply taking the first C words of a sample, take C words surrounding our candidate word, like a sliding window, and infer for every word in the sample. The new complexities associated with this algorithm are in Table 5.3. Since C is strictly less than M, this algorithm, called window search (WS), is slower and more memory intensive than the previous algorithm, but should produce more accurate results. This search generates a matrix A^{ws} which should approximate A. Pseudocode for the algorithm is given in Algorithm 4.

5.3 Gradient Assisted Window Search

Finally, the complexity may be reduced even further by using the results of section 4.3. Because it was found that words associated with large gradients tend to be good words for replacement, the effective value of M can be reduced in the window search algorithm. We propose to only consider the top K words for replacement, as ordered by the gradient, and perform a search over those value. This method is called gradient assisted window search (GAWS). The complexities are given simply by replacing M by K in the window search algorithm, and can be found in Table 5.4. This search generates a matrix A^{gs} which should approximate K rows of A corresponding to the words in the original sample with the K largest gradients. Pseudocode is given in Algorithm 5.

Algorithm 4 Window Search algorithm

Require: s ▷ the sample
Require: r ▷ the classifier function
Require: c ▷ the class obtained
Require: C ▷ the window size
Require: N ▷ the number of words in the vocabulary

$M \leftarrow \text{LENGTH}(s)$
function GETWINDOWSUBSET(M,C,n,s)
    $C \leftarrow \text{MIN}(C, M)$
    $n1 \leftarrow n - C//2$
    $n2 \leftarrow n1 + C - 1$
    if $n1 \geq 1$ and $n2 \leq M$ **then**
        $i1 \leftarrow n1;\ i2 \leftarrow n2$
    else if $n1 < 1$ and $n2 \leq M$ **then**
        $i1 \leftarrow 1;\ i2 \leftarrow C$
    else if $n1 \geq 1$ and $n2 \geq M$ **then**
        $i1 \leftarrow M - C + 1;\ i2 \leftarrow M$
    else
        $i1 \leftarrow 1;\ i2 \leftarrow M$
    end if
return $(s_{i1}, s_{i1+1}, \ldots, s_{i2})$
end function
for $i = 1$ to $i = M$ **do**
    $S \leftarrow \text{GETWINDOWSUBSET}(M, C, i, s)$
    for $j = 1$ to $j = N$ **do**
        $A^{ws}_{i,j} \leftarrow r(S^{i,j})$
        if $A^{cs}_{i,j} \neq c$ **then**
            if $r(s^{i,j}) \neq c$ **then**
                return $s^{i,j}$
            end if
        end if
    end for
end for
return None

Parallelization Tier	Time	Space
0	$\mathcal{O}(NKC)$	$\mathcal{O}(W)$
1	$\mathcal{O}(NC)$	$\mathcal{O}(WK)$
2	$\mathcal{O}(CK)$	$\mathcal{O}(WN)$
3	$\mathcal{O}(C)$	$\mathcal{O}(WNK)$

Table 5.4: Time and space complexities for gradient assisted window search with a window of size C and taking the top K words.

Algorithm 5 Gradient Assisted Window Search (GAWS) algorithm.

Require: s ▷ the sample
Require: r ▷ the classifier function
Require: c ▷ the class obtained
Require: C ▷ the window size
Require: N ▷ the number of words in the vocabulary

$M \leftarrow \text{LENGTH}(s)$
for i in $(1, 2, \ldots, M)$ **do**
    $g_i \leftarrow ||\nabla r(s_i)||$ ▷ Get norm of the gradient with respect to the i^{th} input
end for
$I \leftarrow \text{TOPIND}(g, K)$ ▷ Get ordered indices for K largest gradients norms
for i in I **do**
    $S \leftarrow \text{GETWINDOWSUBSET}(M, C, i, s)$ ▷ See algorithm 4 for subroutine
    for $j = 1$ to $j = N$ **do**
        $A^{gs}_{i,j} \leftarrow r(S^{i,j})$
        if $A^{cs}_{i,j} \neq c$ **then**
            if $r(s^{i,j}) \neq c$ **then**
                return $s^{i,j}$
            end if
        end if
    end for
end for
return None

5.4 Multi-word Replacement

Up to this point, only searching for a one-word replacement adversarial derivation has been considered. While this is sufficient for about 70% of samples, there is still a need to produce derivations for the remaining 30%. Multi-word derivation can be achieved efficiently with two modifications to the original algorithms.

First, instead of dealing with the matrix A, one can utilize a more general matrix of classifier confidence levels, rather than just decisions. Suppose that the function $p(s)$ gives the probability that a sample s is class 0. Then in the same way the matrix A is generated along with all of its approximations, the matrix $P_{i,j} = p(s^{i,j})$ can be generated in the same way. Thresholding this matrix would yield A or its approximation. Second, if no derivations are detected, one can pick the entry of P with the highest or lowest value (depending on the classification target) and attempt using that substitution. From this point onward, this work assumes the second tier of parallelization because tier 3 was not feasible on the hardware used in this project.[1]

If replacing one word is not successful, the algorithm may select more words. Doing this naively would lead to combinatorial explosion. Trying every combination of L words in the full search would cost roughly $\prod_{i=0}^{L-1}(M-i)N = \frac{M!}{(M-L)!}N^L$ lookups, each lookup costing $\mathcal{O}(M)$ time. Since the time complexity is already a restricting factor, this is not feasible even for small L. We therefore implemented a fast greedy approach where the maximum is taken across all columns of P. This takes $\mathcal{O}(N)$ time and only $\mathcal{O}(M)$ space since the min/max is taken as soon as all elements are available. The top L entries of the result are chosen as candidate replacements. Sorting all values in the result requires $\mathcal{O}(M \log(M))$ time.

Now, some method of determining the smallest value of L which allows us to achieve a misclassification is still required. We assume that replacing more words than required will still lead to a misclassification, and therefore the class vs. L curve

[1] All algorithms are run with an AMD RYZEN processor, GTX 1080 Ti GPU, and 32GB of RAM.

Method	Time	Space
Full Search	$\mathcal{O}(M^2 + M\log(M) + M\log L)$	$\mathcal{O}(WN)$
Cap Search	$\mathcal{O}(C^2 + C\log(C) + M\log L)$	$\mathcal{O}(WN)$
Window Search	$\mathcal{O}(CM + M\log(M) + M\log L)$	$\mathcal{O}(WN)$
GAWS	$\mathcal{O}(CK + K\log(K) + M\log L)$	$\mathcal{O}(WN)$

Table 5.5: Overall time complexities for several search algorithms extended with an exponential search for multi-word replacement. These complexities assume the second tier of parallelization found in tables 5.1 through 5.4.

looks like a step function. Since it is monotonic, we can use an exponential search for the transition point which runs in $\mathcal{O}(M\log L)$ time and $\mathcal{O}(W + M)$ space given that all candidates have already been determined. This will usually be better than a binary search given that the number of words being replaced, L is often small compared to the size of the sample, M. Adding all steps together, the final time and space complexities of the given algorithms can be found in Table 5.5.

In order to obtain the modified pseudocode for Algorithm 4, replace lines 20 to 25 with $P^{ws}_{i,j} \leftarrow p(S^{i,j})$ and append Algorithm 6. Algorithm 5 can be extended in the same way, except by replacing lines 9 to 14.

Algorithm 6 Exponential Search Algorithm for Multi-word Replacement

Require: s ▷ the sample
Require: r ▷ the classifier function
Require: c ▷ the class obtained
Require: P ▷ probability matrix

$L \leftarrow \text{LENGTH}(s)$
for $i = 1$ to $i = L$ **do**
    if c $= 0$ **then**
        $p_i \leftarrow \text{MIN}(P_i)$; $I_i \leftarrow \text{ARGMIN}(P_i)$ ▷ P_i is the i^{th} row of P
    else if c $= 1$ **then**
        $p_i \leftarrow \text{MAX}(P_i)$; $I_i \leftarrow \text{ARGMAX}(P_i)$
    end if
    $J_i \leftarrow i$
end for

$L \leftarrow 0$; $R \leftarrow 1$
while $L < R$ **do**
    if $LimitFound = False$ **then**
        $R \leftarrow 2 \times R$
    end if
    $m \leftarrow (L + R)//2$
    if $c = 1$ **then**
        $args \leftarrow \text{TOPIND}(p, m)$ ▷ get arguments of largest probabilities
    else if $c = 1$ **then**
        $args \leftarrow \text{BOTIND}(p, m)$ ▷ get arguments of smallest probabilities
    end if
    $Is \leftarrow I[args]$; $Js \leftarrow J[args]$
    $S \leftarrow s$
    for $i = 1$ to m **do**
        $S_{Js_i} \leftarrow D_{Is_i}$ ▷ D is the dictionary of all words
    end for
    if $r(S) = c$ and $limitFound = True$ **then**
        $L \leftarrow m + 1$
    else if $r(S) \neq c$ **then**
        $limitFound \leftarrow True$
        $R \leftarrow m - 1$
        $minm \leftarrow \text{MIN}(minm, m)$
        if $m = minm$ **then**
            $bestS \leftarrow S$
        end if
    end if
end while

if $limitFound = True$ **then**
    return $bestS$
end if
return None

Chapter 6

Results

In order to find suitable choices for word substitution, the window search and GAWS algorithms, described in the previous sections were used, as well as FGSM which was described in section 3.3. Each of these algorithms is applied to a recurrent neural network with hidden size 16 and learning rate 0.1 as described in section 4.2. The model was trained for 15 epochs and achieved a testing accuracy of 0.822. 500 text samples which the RNN correctly classified were randomly chosen from the test set of the "Large Movie Review Dataset". The number of substitutions and time taken to achieve them was recorded.

6.1 White Box

This experiment assumes that all details of the model are known including the exact weights. We determine candidate word replacements based on inferring with those weights directly. These are the conditions under which the results in [7] where achieved. The graph in figure 6.1 shows the percentage of time a given number of words were required to change a classification. The graph in figure 6.2 shows the accumulation of the previously mentioned graph. These graphs can be interpreted as a probability mass function and a cumulative distribution function respectively.

	Mean	Median	Mode	Failures
FGSM	23.57	20	3	0
GAWS	2.24	1	1	0
WS	1.81	2	1	0

Table 6.1: Summary statistics for white box experiment. These numbers correspond to the full distributions visualized in figure 6.1.

Table 6.1 gives some summary statistics of the results.

Figure 6.1 shows shows that multi-word window search finds a single word replacement to change the classification about 70% of the time which is consistent with our findings for the single-word window search algorithm. Figure 6.2 shows that GAWS lags behind a fair amount, on average requiring about double the number of word replacements up until 11 word replacements where 100% of can be misclassified.

Overall the methods significantly outperform FGSM. From Table 6.1, it is clear that FGSM performed significantly worse in terms of average substitutions required as compared to [7]. This is likely explained by the fact the entire data set was included which yields a much longer average sample length: 228.49 words per sample compared to 71.06. This ratio of our average to theirs roughly matches the increase in substitutions required. Figure 6.3 shows that GAWS can complete searches in far less time than window search. This applies in both a mean sense and an extreme sense: the longest time GAWS took was 50 seconds while WS took almost 6 minutes on one occasion.

6.2 Gray Box

This experiment assumes that all hyperparameters of the model being attacked are known. In order to attack the model, we trained a second neural network with identical hyperparameters. Because stochastic gradient descent has random components, the models will likely have very different weights by the end of training. We infer

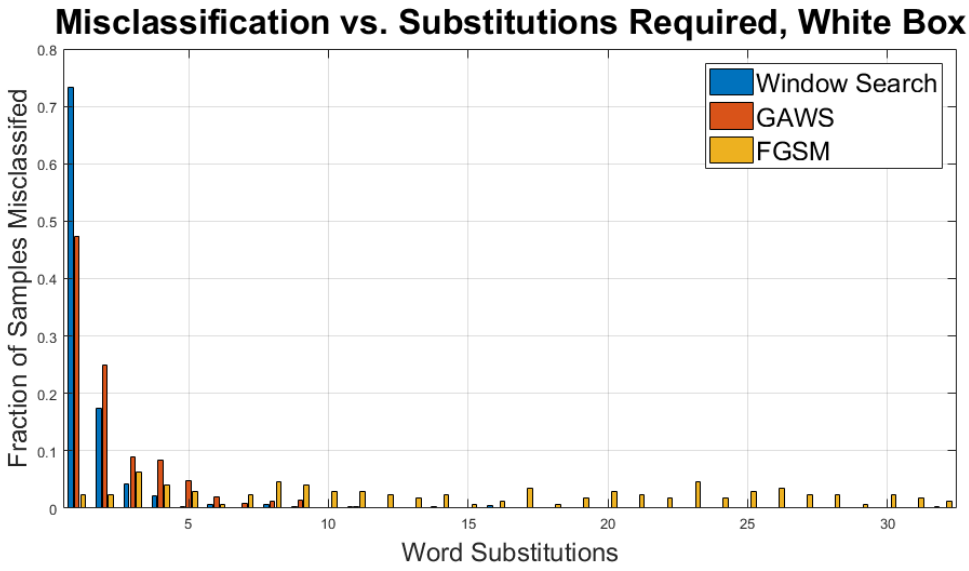


Figure 6.1: The x-axis represents the number of word substitutions used to change the fraction of samples represented by the y-axis.

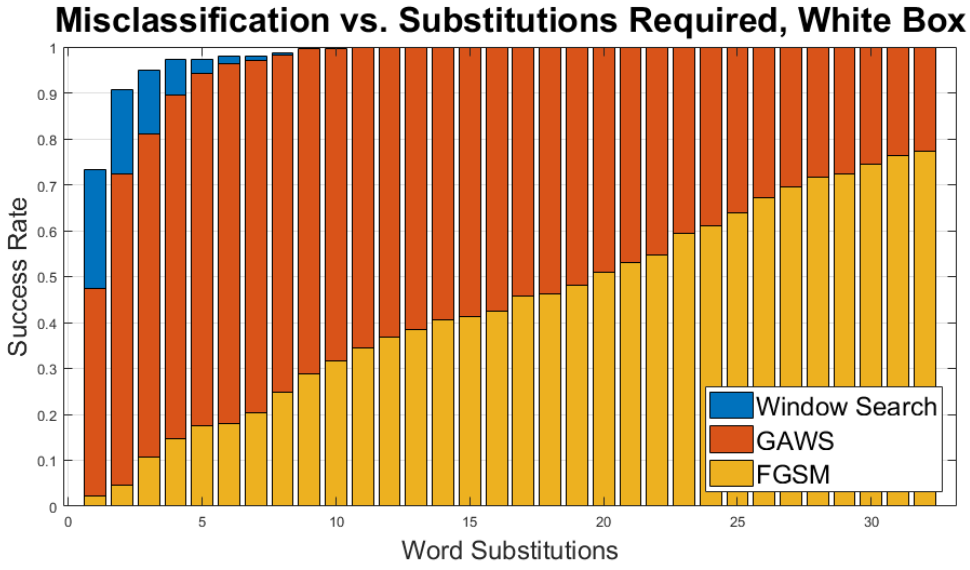


Figure 6.2: The y-axis represents the total fraction of samples can be successfully misclassified given the number of substitutions on the x-axis. This is the accumulation of the graph in figure 6.1

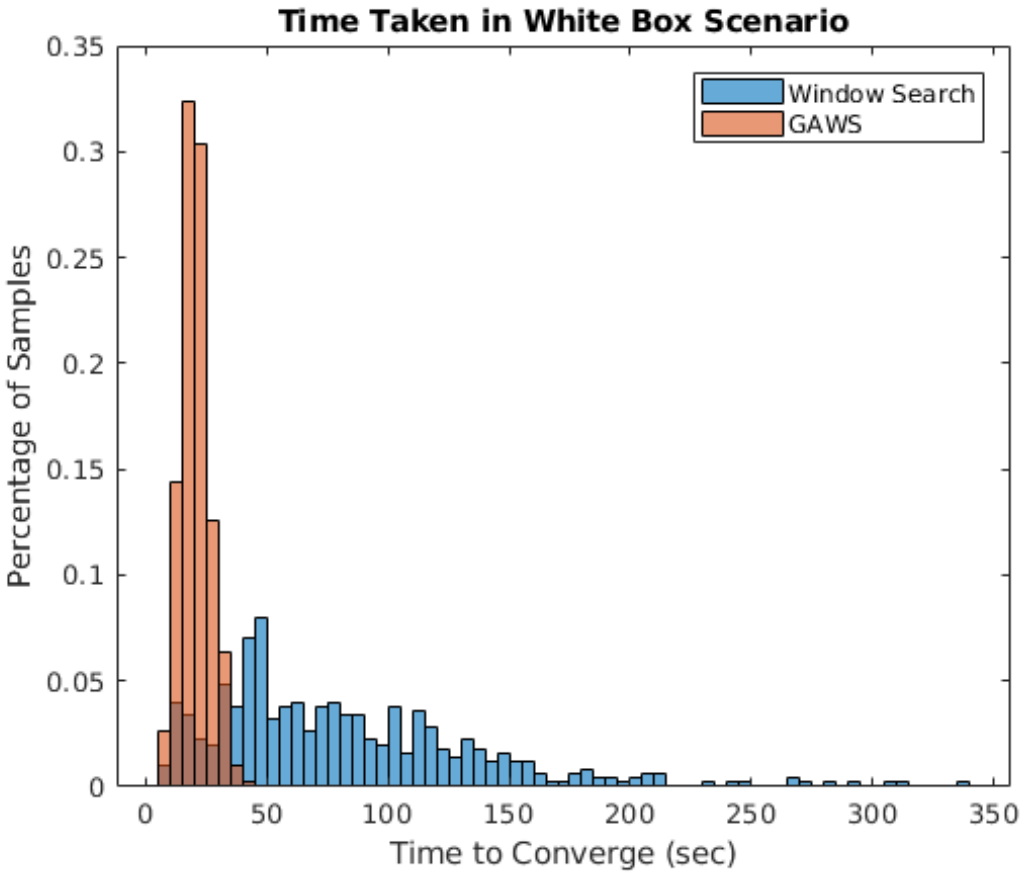


Figure 6.3: Time taken per sample for WS and GAWS algorithms.

	Mean	Median	Mode	Failures
FGSM	43.42	32	6	1
GAWS	10.06	3	1	1
WS	5.86	2	1	0

Table 6.2: Summary statistics for the gray box experiment. These numbers correspond to the full distributions visualized in figure 6.4. Failures are not counted toward mean, median, nor mode.

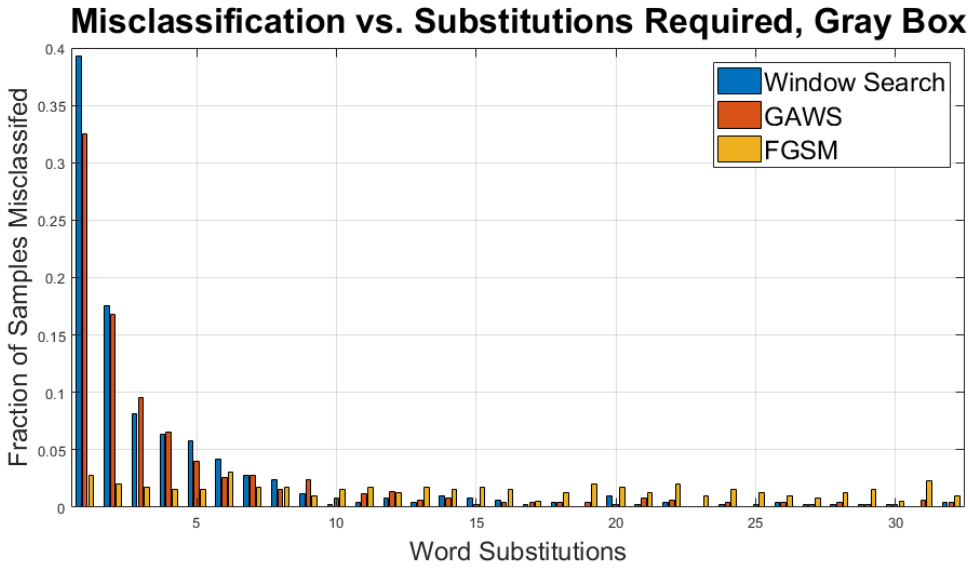


Figure 6.4: The x-axis represents the number of word substitutions used to change the fraction of samples represented by the y-axis.

and search on our stand-in model and then infer on the model under attack to check if it was successfully caused to misclassify the sample. If the model under attack misclassifies the text sample to begin with, it is not considered.

Table 6.2 shows that performance of the algorithms is significantly degraded if the exact weights are unknown. The average number of word replacements required to change classification increased by a factor of roughly 4.5 and 3.25 for GAWS and WS respectively. FGSM, on the other hand worsened by a factor less than 2. Even though GAWS and WS decreased in performance more than FGSM, they still far outperformed it, even if it is given the benefit of a white box attack.

Both Table 6.2 and figure 6.4 show, however, that most of the samples can be

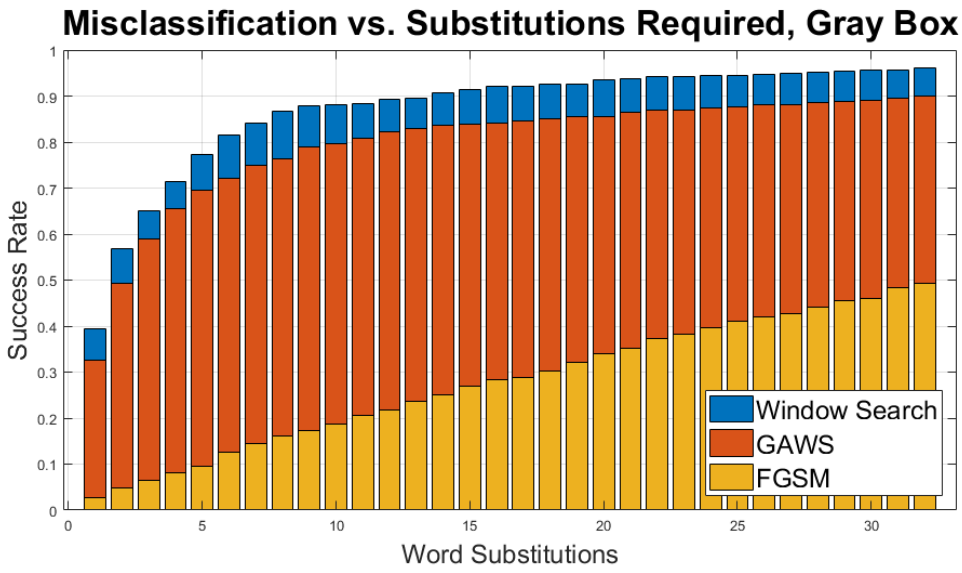


Figure 6.5: The y-axis represents the total fraction of samples can be successfully misclassified given the number of substitutions on the x-axis. This is the accumulation of the graph in figure 6.4

successfully attacked by changing just two or three words. These results indicate that the described algorithms do not exactly attack just one model with one set of weights, but exhibit some amount of transferability. Figure 6.5 shows that window search consistently outperforms GAWS in terms of the number of replacements required. The shape of the WS and GAWS sequences in 6.4 appear to be roughly exponential toward the start and flatten out further away. As in the white box scenario, FGSM seems to have a fairly constant distribution and therefore linear cumulative distribution.

6.3 Black Box

This experiment assumes that no details of the model are known, in particular the hyperparameters are unknown. In order to evaluate performance under this condition, we determine word substitutions using an RNN with a hidden layer of size 16 and attack an RNN with a hidden layer size of 8.

Table 6.3 shows that there are significantly more failures for the GAWS algorithm,

	Mean	Median	Mode	Failures
FGSM	43.42	32	6	0
GAWS	10.16	2	1	5
WS	7.16	2	1	0

Table 6.3: Summary statistics for the black box experiment. These numbers correspond to the full distributions visualized in figure 6.6. Failures are not counted toward mean, median, nor mode.

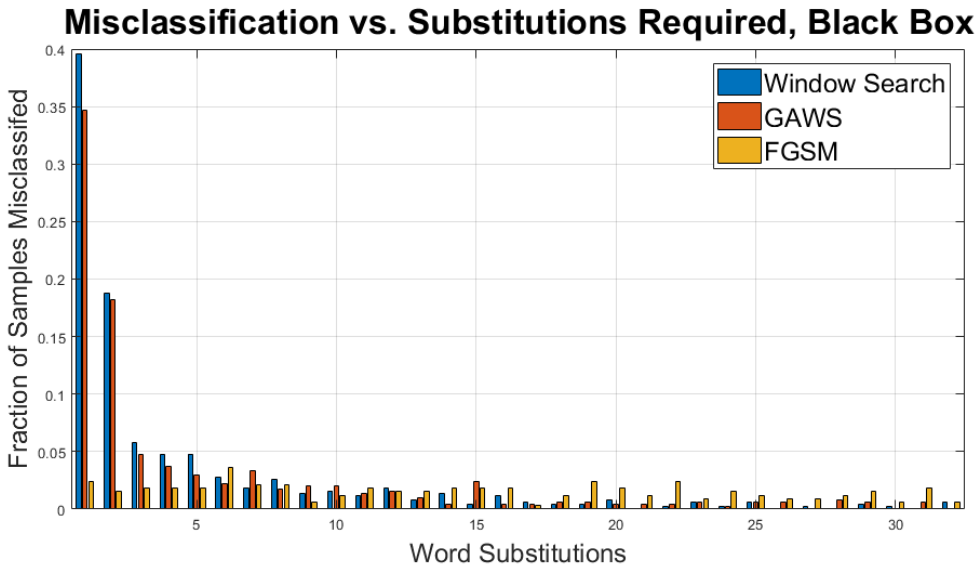


Figure 6.6: The y-axis represents the fraction of samples which were misclassified given the number of substitutions on the x-axis.

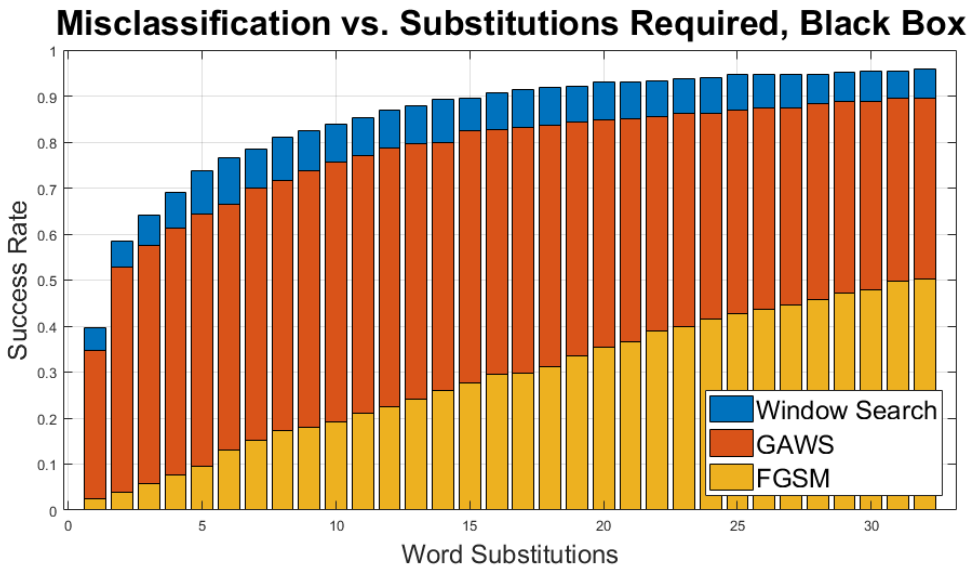


Figure 6.7: The y-axis represents the total fraction of samples can be successfully misclassified given the number of substitutions on the x-axis. This is the accumulation of the graph in figure 6.6

and slightly worse performance for window search algorithm. Otherwise the statistics and figures were mostly unaffected. None the less, this shows that our methods are viable even in black box scenarios, with window search achieving better performance than FGSM, even if it is afforded white box performance.

Note that while the cumulative distribution in figure 6.7 looks identical to the one in figure 6.5, it is only a representation of the first 32 values. In fact, large values far in the tail of the distribution contribute a significant amount to the sample mean. Because the sample size is a relatively small value of 500, it is not clear whether this is significant or just statistical noise.

Chapter 7

Conclusion

The goal of our study was to generate attacks capable of changing a correct text classification of a recurrent neural network into an incorrect one with as few word substitutions as possible. We implemented several new strategies for text based adversarial derivation, particularly targeting recurrent neural networks.

White box, gray box, and black box scenarios were used to test two new algorithms against a baseline algorithm, fast gradient sign method. The strategies employed offered significant improvement over previous methods in terms of average number of word substitutions. The gradient based algorithm significantly increased the algorithm speed, though with slightly worse performance. The new algorithms show some amount of transferability, still working across models with different hyperparameters, but at degraded performance.

These algorithms should be tested in producing similar results with more complex neural networks since only single layer RNN's were tested. It is not clear whether more complex neural networks will be more or less susceptible to attacks of this type or attacks in general. Some recommendations include adding more layers, making the layers larger, and changing the word embedding dimension.

In many cases words were replaced with nonsensical words which were either

associated with very negative or very positive reviews. While the number of words replaced might be small, it can be easily noticed, and a defense method might even work by detecting these common replacement words where they don't belong.

While other adversarial techniques have been shown to effective even for very large training sets, our training set is relatively small and therefore may be more vulnerable than something trained on more data. In particular, the word embedding, while trained on 31 million words, still contains words which are quite rare in the list of the most frequent 10,000.

During testing it was assumed that the word embedding was held constant across all models. This is not a totally unreasonable assumption since popular word embeddings are in fact published online and usually are not recreated for a new project. That being said, a defense technique may attempt to either retrain a word embedding or alter it in some way to create a less vulnerable model.

Ultimately the purpose of this work is to allow others to develop effective defenses. Any of the previous suggestions may aid in developing a defense, but there are plenty of defenses for other attacks already in place which may be suitable in this case. One simple defense might be achieved by running the attack on training samples and inserting the result into the training set.

Bibliography

[1] A. G. Baydin, B. A. Pearlmutter, and A. A. Radul, "Automatic differentiation in machine learning: a survey," *CoRR*, vol. abs/1502.05767, 2015. [Online]. Available: http://arxiv.org/abs/1502.05767

[2] J. James, "Data never sleep 5.0," https://www.domo.com/blog/data-never-sleeps-5/, 2017.

[3] M. Robinson, "How many stories do newspapers publish per day?" www.theatlantic.com/technology/archive/2016/05/how-many-stories-do-newspapers-publish-per-day/483845/, 2016.

[4] B. Livingston, "Paul graham provides stunning answer to spam e-mails," https://www.infoworld.com/article/2674702/technology-business/techology-business-paul-graham-provides-stunning-answer-to-spam-e-mails.html, 2002.

[5] N. Ruchansky, S. Seo, and Y. Liu, "CSI: A hybrid deep model for fake news," *CoRR*, vol. abs/1703.06959, 2017. [Online]. Available: http://arxiv.org/abs/1703.06959

[6] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, "Intriguing properties of neural networks," *ArXiv e-prints*, Dec. 2013.

[7] N. Papernot, P. McDaniel, A. Swami, and R. Harang, "Crafting Adversarial Input Sequences for Recurrent Neural Networks," *ArXiv e-prints*, Apr. 2016.

[8] C. M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag, 2006.

[9] G. H. Golub, P. C. Hansen, and D. P. O'Leary, "Tikhonov regularization and total least squares," *SIAM J. Matrix Anal. Appl.*, vol. 21, no. 1, pp. 185–194, Oct. 1999. [Online]. Available: http://dx.doi.org/10.1137/S0895479897326432

[10] G. Cybenko, "Approximation by superpositions of a sigmoidal fucntion," *Mathematics of control, signals and systems*, vol. 2, no. 4, pp. 303–314, 1989.

[11] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2014. [Online]. Available: http://arxiv.org/abs/1412.6980

[12] J. L. Elman, "Finding structure in time." *Cognitive science*, vol. 14(2), pp. 179–211, 1990.

[13] M. I. Jordan, "Serial order: A parallel distributed processing approach." *Technical Report 8604, Institute for Cognitive Science, University of Califor- nia, San Diego*, 1986.

[14] H. T. Siegelmann and E. D. Sontag, "Turing computability with neural nets." *Applied Mathematics Letters*, vol. 4(6), pp. 77–80, 1991.

[15] S. Hochreiter and J. Schmidhuber, "Long short-term memory." *Neural Computation*, vol. 9(8), pp. 1735–1780, 1997.

[16] F. A. Gers and J. Schmidhuber, "Learning to forget: Continual prediction with lstm." *Neural Computation*, vol. 12(10), pp. 2451–2471, 1997.

[17] ——, "Recurrent nets that time and count," Tech. Rep., 2000.

[18] W. Zaremba, I. Sutskever, and O. Vinyals, "Recurrent neural network regularization," *CoRR*, vol. abs/1409.2329, 2014. [Online]. Available: http://arxiv.org/abs/1409.2329

[19] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. New York, NY, USA: Cambridge University Press, 2008.

[20] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *CoRR*, vol. abs/1301.3781, 2013. [Online]. Available: http://arxiv.org/abs/1301.3781

[21] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed Representations of Words and Phrases and their Compositionality," *ArXiv e-prints*, Oct. 2013.

[22] M. Gutmann and A. Hyvärinen, "Noise-contrastive estimation of unnormalized statistical models, with applications to natural image statistics," *The Journal of Machine Learning Research*, vol. 13, pp. 307–361, 2012.

[23] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. J. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Józefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. G. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. A. Tucker, V. Vanhoucke, V. Vasudevan, F. B. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *CoRR*, vol. abs/1603.04467, 2016. [Online]. Available: http://arxiv.org/abs/1603.04467

[24] A. L. Maas, R. E. Daly, P. T. Pham, D. Huang, A. Y. Ng, and C. Potts, "Learning word vectors for sentiment analysis," in *Proceedings of*

the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies. Portland, Oregon, USA: Association for Computational Linguistics, June 2011, pp. 142–150. [Online]. Available: http://www.aclweb.org/anthology/P11-1015

[25] P. Massart, *Concentration inequalities and model selection*. Springer Verlag, 2007.

Appendix A

Code Appendix

This appendix includes a large sample of code used to obtain the results displayed in this thesis. Code should not be used without understanding, as certain programs may be used as a template rather than a standalone module.

```
# Name: Cory Nezin
# Date: 03/30/2018
# Task: Perform an exponential search window attack

import tensorflow as tf
import numpy as np
import review_proc as rp
import preprocess, rnn, word2vec, wa
import plotutil as putil
import argparse, os, sys, random, re
import time
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'

word_embedding = np.load('index_to_vector.npy')
word_to_embedding_index = np.load('word_to_index.npy').item()

m = word_to_embedding_index
embedding_index_to_word = dict(zip(m.values(),m.keys()))

k = 129
c = [0]*1024
t = []
f = 0
root_dir = './aclImdb/test/posneg/'
restore_name = '4786_9.txt'
flag = True
for file_name in os.listdir(root_dir):
    if file_name != restore_name and flag:
        print(file_name)
        continue
    elif flag:
        flag = False
        continue
    word_count = 0
    g = tf.Graph()
    print('Running attack on: ' + file_name)
    rvo = rp.review(root_dir + file_name)
    rvo.translate(rvo.length, word_to_embedding_index, embedding_index
_to_word)
    rvo.vec(word_embedding)
    # Actual Neural Network
    with g.as_default():
        global_step_tensor = tf.Variable(0, trainable=False, name='glo
bal_step')
        r = rnn.classifier(
            batch_size = 1,
            learning_rate = 0.0,
            hidden_size = 8,
            max_time = rvo.length,
            embeddings = word_embedding,
            global_step = global_step_tensor)
        with tf.Session() as sess:
            restore_name = './ckpts/gridckpt_8_10/imdb-rnn-e15.ckpt'
            tf.train.Saver().restore(sess,restore_name)
            decision, _ , _ = r.infer_dpg(sess,rvo)
    rnn_sentiment = 'pos' if not decision[0] else 'neg'
    if rnn_sentiment != rvo.sentiment:
```

```
        print('Neural net was wrong, continuing...')
        continue
    print('Starting clock...')
    t0 = time.clock()
    rv = rp.review(root_dir + file_name)
    rv.translate(rvo.length, word_to_embedding_index, embedding_index_
to_word)
    rv.vec(word_embedding)
    ii = 0;
    ind = np.random.permutation(rv.length)
    # infer stand in gradient
    print('Starting loop')
    print(rnn_sentiment,rvo.sentiment)
    while  rnn_sentiment == rvo.sentiment:
        print(ii,end=' ',flush=True)
        g = tf.Graph()
        with g.as_default():
            global_step_tensor = tf.Variable(0, trainable=False, name=
'global_step')
            r = rnn.classifier(
                batch_size = 1,
                learning_rate = 0.0,
                hidden_size = 16,
                max_time = rv.length,
                embeddings = word_embedding,
                global_step = global_step_tensor)
            with tf.Session() as sess:
                restore_name = './ckpts/gridckpt_16_10/imdb-rnn-e15.ck
pt'
                tf.train.Saver().restore(sess,restore_name)
                decision, probability, grad = r.infer_dpg(sess,rvo)
        grad = grad[0][0,:,:]
        # select a word i in the sequence x*
        if ii == 1024 or ii >= rvo.length:
            break
            f = f + 1
        i = ind[ii]
        ii = ii + 1
        # grad is rv.length by 128 (?)
        diff = np.sign(word_embedding[rv.index_vector[0,i],:] - word_e
mbedding)
        if rvo.sentiment == 'pos':
            jack = np.sign(grad[i,:])
        else:
            jack = -np.sign(grad[i,:])
        sgn_diff = diff-jack
        sgn_norm = np.linalg.norm(sgn_diff,axis=1,ord=0)
        j = np.argmin(sgn_norm)
        rv.index_vector[0,i] = j
        word_count = word_count + 1
        g = tf.Graph()
        # Infer decision
        with g.as_default():
            global_step_tensor = tf.Variable(0, trainable=False, name=
'global_step')
            r = rnn.classifier(
                batch_size = 1,
```

```
                learning_rate = 0.0,
                hidden_size = 8,
                max_time = rvo.length,
                embeddings = word_embedding,
                global_step = global_step_tensor)
            with tf.Session() as sess:
                restore_name = './ckpts/gridckpt_8_10/imdb-rnn-e15.ckp
t'
                tf.train.Saver().restore(sess,restore_name)
                decision, probability , _ = r.infer_dpg(sess,rv)
        rnn_sentiment = 'pos' if not decision[0] else 'neg'

    c[word_count] = c[word_count] + 1
    t1 = time.clock()
    print('')
    print('Time taken',t1-t0)
    t.append(t1-t0)
    np.save('./fgsm_black/t/'+file_name[:-4] + '.npy',np.array(t))
    np.save('./fgsm_black/c/'+file_name[:-4] + '.npy',np.array(c))
    np.save('./fgsm_black/f/'+file_name[:-4] + '.npy',np.array(f))
    print('histogram:',c)
    print('fails:',f)
    print('total:',sum(c))
```

```
# Name: Cory Nezin
# Date: 03/30/2018
# Task: Perform an exponential search window attack

import tensorflow as tf
import numpy as np
import review_proc as rp
import preprocess, rnn, word2vec, wa
import plotutil as putil
import argparse, os, sys, random, re
import time
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'

word_embedding = np.load('index_to_vector.npy')
word_to_embedding_index = np.load('word_to_index.npy').item()

m = word_to_embedding_index
embedding_index_to_word = dict(zip(m.values(),m.keys()))

k = 32
c = [0]*1024
t = []
f = 0
root_dir = './aclImdb/test/posneg/'
flag = True
for file_name in os.listdir(root_dir):
    #continue_name = '10173_1.txt'
    #print(file_name)
    #if file_name != continue_name and flag:
    #    continue
    #else:
    #    flag = False
    g = tf.Graph()
    print('Running attack on: ' + file_name)
    rvo = rp.review(root_dir + file_name)
    rvo.translate(rvo.length, word_to_embedding_index, embedding_index
_to_word)
    rvo.vec(word_embedding)
    # Actual Neural Network
    with g.as_default():
        global_step_tensor = tf.Variable(0, trainable=False, name='glo
bal_step')
        r = rnn.classifier(
            batch_size = 1,
            learning_rate = 0.0,
            hidden_size = 16,
            max_time = rvo.length,
            embeddings = word_embedding,
            global_step = global_step_tensor)
        with tf.Session() as sess:
            restore_name = './single_ckpt_16_10/imdb-rnn-e15.ckpt'
            tf.train.Saver().restore(sess,restore_name)
            decision, probability, grad = r.infer_dpg(sess,rvo)
    rnn_sentiment = 'pos' if not decision[0] else 'neg'
    if rnn_sentiment != rvo.sentiment:
        print('Neural net was wrong, continuing...')
        continue
```

```
    g = tf.Graph()
    t0 = time.clock()

    window_size = 200
    hidden_size = 16
    restore_name = './ckpts/gridckpt_16_10/imdb-rnn-e15.ckpt'
    #ii,jj,pp = wa.win_atk(rvo, window_size, word_embedding, hidden_si
ze, restore_name)
    ii,jj,pp = wa.win_atk(rvo, window_size, word_embedding, \
                        hidden_size, restore_name)
    if ii is None:
        continue
    if rvo.sentiment == 'pos':
        args = np.argsort(pp)
    else:
        args = np.flip(np.argsort(pp),axis=0)

    ii = ii[args]
    jj = jj[args]
    minm = float('inf')
    R = 1
    L = 0
    val_found = False
    print('Original review:')
    print(' '.join(rvo.tokens))
    while True:
        rv = rp.review(root_dir + file_name)
        if not val_found:
            R = R * 2
        m = (L + R) // 2 # m initialized as 1
        if m >= args.size:
            print('No derivation found, breaking...')
            f = f + 1
            break
        g = tf.Graph()
        ix = ii[:m+1]
        jx = jj[:m+1]
        w = [embedding_index_to_word[i] for i in ix]
        for n,j in enumerate(args[:m+1]):
            rv.tokens[j] = w[n]
        rv.translate(rv.length, word_to_embedding_index, embedding_ind
ex_to_word)
        rv.vec(word_embedding)
        with g.as_default():
            global_step_tensor = tf.Variable(0, trainable=False, name=
'global_step')
            r = rnn.classifier(
                batch_size = 1,
                learning_rate = 0.0,
                hidden_size = 16,
                max_time = rv.length,
                embeddings = word_embedding,
                global_step = global_step_tensor)
            with tf.Session() as sess:
                restore_name = './single_ckpt_16_10/imdb-rnn-e15.ckpt'
                tf.train.Saver().restore(sess,restore_name)
                decision, probability, batch_grad = r.infer_dpg(sess,r
```

```
v)
                rnn_sentiment = 'pos' if not decision[0] else 'neg'
                if rnn_sentiment == rv.sentiment and val_found:
                    L = m + 1
                elif rnn_sentiment != rv.sentiment:
                    print('New detected sentiment:')
                    print(rnn_sentiment)
                    print('Number of changed words:')
                    print(m+1)
                    print('ix:',ix)
                    print('jx:',jx)
                    print('Replaced',[rvo.tokens[args[n]] for n in ran
ge(m+1)]\
                          ,'with',[w[n] for n in range(m+1)])
                    np.save('./wa_gray_testing/ix/'+file_name[:-4]+'.n
py',ix)
                    np.save('./wa_gray_testing/jx/'+file_name[:-4]+'.n
py',jx)
                    minm = min(m+1,minm)
                    val_found = True
                    R = m - 1
                if L > R:
                    print('Search complete, minimum value is: ',minm)
                    c[minm] += 1
                    break
    t1 = time.clock()
    print('Time taken',t1-t0)
    t.append(t1-t0)
    np.save('./wa_gray_testing/ii/' + file_name[:-4] + '.npy',ii)
    np.save('./wa_gray_testing/jj/' + file_name[:-4] + '.npy',jj)
    np.save('./wa_gray_testing/pp/' + file_name[:-4] + '.npy',pp)
    np.save('./wa_gray_testing/t/'+file_name[:-4] + '.npy',np.array(t)
)
    np.save('./wa_gray_testing/c/'+file_name[:-4] + '.npy',np.array(c)
)
    if sum(c) % 50 == 0:
        print('histogram:',c)
        print('Fails:',f)
```

```
# Name: Cory Nezin
# Date: 03/30/2018
# Task: Perform an exponential search window attack

import tensorflow as tf
import numpy as np
import review_proc as rp
import preprocess, rnn, word2vec, wa
import plotutil as putil
import argparse, os, sys, random, re
import time
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'

word_embedding = np.load('index_to_vector.npy')
word_to_embedding_index = np.load('word_to_index.npy').item()

m = word_to_embedding_index
embedding_index_to_word = dict(zip(m.values(),m.keys()))

k = 32
c = [0]*1024
t = []
f = 0
root_dir = './aclImdb/test/posneg/'
for file_name in os.listdir(root_dir):
    t0 = time.clock()
    g = tf.Graph()
    print('Running attack on: ' + file_name)
    rvo = rp.review(root_dir + file_name)
    rvo.translate(rvo.length, word_to_embedding_index, embedding_index
_to_word)
    rvo.vec(word_embedding)
    # Generating the gradient
    with g.as_default():
        global_step_tensor = tf.Variable(0, trainable=False, name='glo
bal_step')
        r = rnn.classifier(
            batch_size = 1,
            learning_rate = 0.0,
            hidden_size = 16,
            max_time = rvo.length,
            embeddings = word_embedding,
            global_step = global_step_tensor)
        with tf.Session() as sess:
            restore_name = './ckpts/gridckpt_16_10/imdb-rnn-e15.ckpt'
            tf.train.Saver().restore(sess,restore_name)
            decision, probability, grad = r.infer_dpg(sess,rvo)
    grad = np.linalg.norm(grad[0][0,:,:],axis=1)
    window_size = 200
    hidden_size = 16
    restore_name = './ckpts/gridckpt_16_10/imdb-rnn-e15.ckpt'
    #ii,jj,pp = wa.win_atk(rvo, window_size, word_embedding, hidden_si
ze, restore_name)
    ii,jj,pp = wa.gaws(rvo, window_size, word_embedding, \
                       hidden_size, restore_name,grad,k,decision)
    if ii is None:
        continue
```

```
    if rvo.sentiment == 'pos':
        args = np.argsort(pp)
    else:
        args = np.flip(np.argsort(pp),axis=0)
    ii = ii[args]
    jj = jj[args]
    minm = float('inf')
    R = 1
    L = 0
    val_found = False
    print('Original review:')
    print(' '.join(rvo.tokens))
    while True:
        rv = rp.review(root_dir + file_name)
        if not val_found:
            R = R * 2
        m = (L + R) // 2 # m initialized as 1
        if m >= args.size:
            print('No derivation found.')
            f = f + 1
            break
        g = tf.Graph()
        ix = ii[:m+1]
        jx = jj[:m+1]
        w = [embedding_index_to_word[i] for i in ix]
        for n,j in enumerate(args[:m+1]):
            rv.tokens[j] = w[n]
        rv.translate(rv.length, word_to_embedding_index, embedding_ind
ex_to_word)
        rv.vec(word_embedding)
        with g.as_default():
            global_step_tensor = tf.Variable(0, trainable=False, name=
'global_step')
            r = rnn.classifier(
                batch_size = 1,
                learning_rate = 0.0,
                hidden_size = 16,
                max_time = rv.length,
                embeddings = word_embedding,
                global_step = global_step_tensor)
            with tf.Session() as sess:
                restore_name = './ckpts/gridckpt_16_10/imdb-rnn-e15.ck
pt'
                tf.train.Saver().restore(sess,restore_name)
                decision, probability, batch_grad = r.infer_dpg(sess,r
v)
                rnn_sentiment = 'pos' if not decision[0] else 'neg'
                if rnn_sentiment == rv.sentiment and val_found:
                    L = m + 1
                elif rnn_sentiment != rv.sentiment:
                    print('New detected sentiment:')
                    print(rnn_sentiment)
                    print('Number of changed words:')
                    print(m+1)
                    print('ix:',ix)
                    print('jx:',jx)
                    print('Replaced',[rvo.tokens[args[n]] for n in ran
```

```
ge(m+1)]\
                        ,'with',[w[n] for n in range(m+1)])
                    np.save('./gaws_testing/ix/'+file_name[-4:]+'.npy'
,ix)
                    np.save('./gaws_testing/jx/'+file_name[-4:]+'.npy'
,jx)
                    minm = min(m+1,minm)
                    val_found = True
                    R = m - 1
                if L > R:
                    print('Search complete, minimum value is: ',minm)
                    c[minm] += 1
                    break
    t1 = time.clock()
    print('Time taken',t1-t0)
    t.append(t1-t0)
    np.save('./gaws_testing/ii/' + file_name[:-4] + '.npy',ii)
    np.save('./gaws_testing/jj/' + file_name[:-4] + '.npy',jj)
    np.save('./gaws_testing/pp/' + file_name[:-4] + '.npy',pp)
    np.save('./gaws_testing/t/'+file_name[:-4] + '.npy',np.array(t))
    if sum(c) % 50 == 0:
        print('histogram:',c)
        print('fails:',f)
```

```
import os, re, sys
import tensorflow as tf
import numpy as np
import rnn
import preprocess
import random

# Yields (batch_size) examples, truncated at (max_time) words.
# Examples are randomly pulled from (input_directory) and
# Translated to integers using (emb_dict)
def dataset(input_directory, batch_size, emb_dict, max_time):
  file_gen = os.listdir(input_directory)
  random.shuffle(file_gen)
  batch_num = 0
  inputs = np.zeros((batch_size, max_time))
  targets = np.zeros((batch_size,2))
  sequence_length = np.zeros((batch_size))
  for name in file_gen:
    file_path = input_directory + '/' + name
    f = open(file_path)
    w = preprocess.tokenize(f.read())
    rating = int(re.sub('_|\.txt',' ',name).split()[1])
    targets[batch_num][0:2] = [0,1] if rating < 5 else [1,0]
    sequence_length[batch_num] = len(w)
    for time_num in range(min(max_time,len(w))):
      inputs[batch_num][time_num] = emb_dict.get(w[time_num],0)

    batch_num += 1
    if batch_num == batch_size:
      yield inputs,targets,sequence_length
      batch_num = 0
      inputs = np.zeros((batch_size, max_time))
      targets = np.zeros((batch_size,2))
      sequence_length = np.zeros((batch_size))

batch_size = 1000  # Number of reviews to consider at once
max_time = 1024   # Maximum number of words in a given review
# Load the word embedding
emb_dict = np.load('emb_dict.npy').item()
embeddings = np.load('final_embeddings.npy')

for hidden_size,lr in zip([2,4,8,16,32,64,128]*2,[0.01,0.001]*7):
  # Reset the TensorFlow graph
  g = tf.Graph()
  tf.reset_default_graph()
  with g.as_default():
    # Set global step to zero, for keeping track of training progress
    global_step_tensor = tf.Variable(0, trainable = False, name = 'glo
bal_step')
    # Make the RNN
    r = rnn.classifier(
      batch_size = batch_size,
      learning_rate = lr,
      hidden_size = hidden_size,
      max_time = max_time,
      embeddings = embeddings,
      global_step = global_step_tensor
```

```
    )

    # Training session
    with tf.Session() as sess:
      saver = tf.train.Saver(max_to_keep = 200)
      train_writer = tf.summary.FileWriter(sys.argv[1]+'/train_'+str(h
idden_size)+'_'+str(int(lr*1000)))
      test_writer = tf.summary.FileWriter(sys.argv[1]+'/test_'+str(hid
den_size)+'_'+str(int(lr*1000)))
      sess.run(tf.global_variables_initializer())
      for epoch in range(50):
        saver.save(sess, './'+sys.argv[2]+'_'+str(hidden_size)+'_'+str
(int(lr*1000))+"/imdb-rnn-e%d.ckpt"%epoch)
        print('epoch: ',epoch)
        # Testing #
        sess.run(tf.local_variables_initializer())
        if not (epoch % 5):
          for inputs,targets,sequence_length in dataset(\
              './aclImdb/test/posneg/',batch_size,emb_dict,max_time):
            accuracy,global_step,summary = sess.run([r.update_accuracy
, global_step_tensor, r.merged],
              feed_dict={r.inputs:inputs,r.targets:targets,r.sequence_
length:sequence_length,
                         r.keep_prob:1.0})
          test_writer.add_summary(summary, global_step)
        # Training #
        sess.run(tf.local_variables_initializer())
        for inputs,targets,sequence_length in dataset(\
            './aclImdb/train/posneg/',batch_size,emb_dict,max_time):
          loss,updates,embed,output,updated_accuracy,\
          mean,logits,prob,sequence_length,\
          probe,labels,global_step,learning_rate,summary = \
            sess.run([r.loss,r.updates,r.embed,r.output,r.update_accur
acy,\
                  r.mean,r.logits,r.probability,r.sequence_length,\
                  r.probe,r.targets,global_step_tensor,r.learning_rate
,r.merged],
              feed_dict={r.inputs:inputs,r.targets:targets,r.sequence_le
ngth:sequence_length,
                         r.keep_prob:0.5})
        train_writer.add_summary(summary, global_step)
```

```
import preprocess
import numpy as np
import re

class review:
    def __init__(self,review_filename):
        self.review_rating = int(re.search('_\d+',review_filename).gro
up()[1:])
        target_list = [0,1] if self.review_rating > 5  else [1,0]
        self.sentiment = 'pos' if self.review_rating > 5 else 'neg'
        review_file = open(review_filename).read()
        self.tokens = preprocess.tokenize(review_file)
        self.targets = np.array([target_list]);
        self.length = len(self.tokens)

    def translate(self, max_time, word_to_embedding_index, embedding_i
ndex_to_word):
        inputs = np.zeros((1,max_time),int)
        lookup = []
        self.unk_loc = []
        for index in range(min(max_time,self.length)):
            inputs[0][index] = word_to_embedding_index.get(self.tokens
[index],0)
            lookup.append( embedding_index_to_word.get(inputs[0][index
],'UNK') )
            if lookup[-1] == 'UNK':
                self.unk_loc.append(index)

        self.index_vector = inputs
        self.word_list = lookup

    def vec(self, word_embedding):
        self.vector_list = word_embedding[self.index_vector[0][0:self.
length],:]
```

```
import numpy as np
import matplotlib.pyplot as plt
import os, sys, re

root_dir = './ggs2_results/'
result_dir = 'diffs/'
prob_dir = 'probs/'
file_list = os.listdir(root_dir+result_dir)

n = 0; N = 0
p = 0; m = 0
P = 0; M = 0
init_probs = []
final_probs = []
delta = []

for file_name in file_list:
    rating = int(re.sub('_|\.npy',' ',file_name).split()[1])
    diff = np.load(root_dir+result_dir+file_name)
    prob = np.load(root_dir+prob_dir+file_name)
    if rating > 5:
        d = np.amin(diff)
    elif rating < 5:
        d = np.amax(diff)
    prob_positive = d + prob[0][0] # this gives p[:,0], probability of
 negative
    if (rating > 5 and prob_positive < 0.5) or (rating < 5 and prob_po
sitive > 0.5):
        n += 1
        m = m + 1 if rating > 5 else m
        p = p + 1 if rating < 5 else p
        print(rating,prob[0][0],d,prob[0][0]+d)
        #diff.shape = (10000*10,)
        #plt.plot(np.sort(diff),'.')
        #plt.show()
        init_probs.append(prob[0][0])
        final_probs.append(prob_positive)
        delta.append(d)
    P = P + 1 if rating < 5 else P
    M = M + 1 if rating > 5 else M
    N += 1

print('%d/%d = %f'%(n,N,n/N))
print('%d/%d = %f'%(m,M,m/M))
print('%d/%d = %f'%(p,P,p/P))
plt.hist(final_probs)
plt.show()
plt.hist(delta)
plt.show()
```

```
import numpy as np
import tensorflow as tf
import numpy as np
import review_proc as rp, preprocess, rnn, word2vec
import matplotlib.pyplot as plt
import plotutil as putil
import argparse, os, sys, random, re
from matplotlib.colors import LogNorm

parser = argparse.ArgumentParser( \
    description = 'Perform a greedy semantic attack on a recurrent neu
ral network')

parser.add_argument('-wi',
    help = 'Word to Index dictionary mapping string to integer',
    default = 'word_to_index.npy')

parser.add_argument('-iv',
    help = 'Index to Vector numpy array mapping integer to vector',
    default = 'index_to_vector.npy')

args = parser.parse_args()
word_embedding_filename = args.iv
word_to_embedding_index_filename = args.wi

try:
    word_embedding = np.load(word_embedding_filename)
    word_to_embedding_index = np.load(word_to_embedding_index_filename
).item()
except FileNotFoundError:
    print('Word embedding not found, running word2vec')
    word2vec.w2v(corpus_filename = './corpus/imdb_train_corpus.txt')

embedding_norm = np.linalg.norm(word_embedding,axis=1)
embedding_norm.shape = (10000,1)
normalized_word_embedding = word_embedding / embedding_norm
m = word_to_embedding_index
# Reverse dictionary to look up words from indices
embedding_index_to_word = dict(zip(m.values(), m.keys()))

g = tf.Graph()
rv = rp.review('./aclImdb/test/posneg/9999_10.txt')
with g.as_default():
    global_step_tensor = tf.Variable(0, trainable = False, name = 'glo
bal_step')
    # Create RNN graph
    r = rnn.classifier(
        batch_size = 1,
        learning_rate = 0.0,
        hidden_size = 16,
        max_time = 1024,
        embeddings = word_embedding,
        global_step = global_step_tensor
    )
    with tf.Session() as sess:
        tf.train.Saver().restore(sess, './ckpts/gridckpt_16_10/imdb-rn
n-e15.ckpt')
```

```
        print(rv.tokens)
        rv.translate(r.max_time,word_to_embedding_index,embedding_inde
x_to_word)
        rv.vec(word_embedding)
        decision, probability, batch_grad = r.infer_dpg(sess,rv)
        print(probability)
        rnn_sentiment = 'pos' if not decision[0] else 'neg'
        print('Neural Net Decision: ',rnn_sentiment,' Actual: ',rv.sen
timent)
        if rnn_sentiment != rv.sentiment:
            pass
        grad = batch_grad[0][0,0:rv.length,:]
        W = word_embedding; G = grad
        D = W @ (G.T)
        #print(np.amax(D),np.amin(D))
        c = np.sum(np.multiply(rv.vector_list,G),axis=1)
        d = D - c
        #np.save('predicted_diff.npy',d)
        actual_diff = np.load('actual_diff.npy')
        print(actual_diff.shape)
        print(d.shape)
        d.shape = (d.size,)
        actual_diff.shape = (d.size,)
        fig,ax = plt.subplots()
        fig.set_size_inches(5.5,10.5)
        #ax.plot(d,actual_diff,'.',ms=3)
        # Add colored dots for high norm words
        print(G.shape)
        d.shape = (10000,114)
        actual_diff.shape = (10000,114)
        n = np.linalg.norm(G,axis=1)
        i = np.argsort(n)[-10:]
        for index in reversed(i):
            print(index)
            ax.plot(d[0:1000,index],actual_diff[0:1000,index],'.',ms=2
,
                label=rv.word_list[index])
        #ax.plot([-0.2,0.2],[0.5,0.5],'k')
        plt.xlim((-0.1,0.1)); plt.ylim((-1.0,0.1))
        ax.axhline(y=0, color='k')
        ax.axvline(x=0, color='k')
        plt.title('Top 10 Gradients Norms',fontsize=30)
        plt.xlabel('Approximated Delta',fontsize=20)
        plt.ylabel('Actual Delta',fontsize=20)
        plt.tight_layout()
        plt.legend()
        fig.savefig('./writing/images/norm_color.png', dpi=300)
        plt.show()
        i = np.argmin(actual_diff)
        print(np.unravel_index(i,(10000,114)))
        print(rv.tokens[16])
        print(embedding_index_to_word[9050])
        rv.tokens[16] = embedding_index_to_word[9050]
        rv.translate(r.max_time,word_to_embedding_index,embedding_inde
x_to_word)
        decision, probability, batch_grad = r.infer_dpg(sess,rv)
        print(probability)
```

```
import expand
import re

from patterns import digits_dictionary, contractions_dictionary

def expand_number(m):
    word = []
    for char in m.group(0):
        word.append(digits_dictionary[char])
    return ' '.join(word) + ' '

def expand_contraction(m):
    return contractions_dictionary[m.group(0)]

def simplify(s):
    s = s.lower()
    s = re.sub('<br /><br />',' ',s)
    s = re.sub('(%s)' % '|'.join(contractions_dictionary.keys()),expan
d_contraction,s)
    s = re.sub(r'[^\w\s]','',s)
    s = re.sub(r'\d+',expand_number,s)
    s = re.sub(r'\s+',' ',s)
    return s

def tokenize(s):
    return simplify(s).split(' ')
```

```
import tensorflow as tf
import numpy as np

class classifier:
    def __init__(self,
           learning_rate,
           hidden_size,
           batch_size,
           max_time,
           embeddings,
           global_step):

      self.batch_size = batch_size
      self.learning_rate = tf.train.exponential_decay(
        learning_rate, global_step, 500, 0.8, staircase=True)
      self.hidden_size = hidden_size
      self.batch_size = batch_size
      self.max_time = max_time
      self.embeddings = tf.constant(embeddings, name = "emb")
      self.optimizer = tf.train.AdamOptimizer(self.learning_rate)
      self.global_step = global_step

      tf.summary.scalar('Learning Rate', self.learning_rate)

      self.make()

      self.saver = tf.train.Saver(max_to_keep = 200)

    def make(self):
      self.sequence_length = tf.placeholder(tf.float32, shape = (self.
batch_size))
      self.inputs = tf.placeholder(tf.int32, shape = (self.batch_size,
 self.max_time))
      self.embed = tf.nn.embedding_lookup(self.embeddings, self.inputs
)
      self.targets = tf.placeholder(tf.int64, shape = (self.batch_size
,2))
      self.keep_prob = tf.placeholder(tf.float32, shape = None)

      cell = tf.nn.rnn_cell.LSTMCell(
        num_units = self.hidden_size,
        use_peepholes=True)

      dropout_cell = tf.contrib.rnn.DropoutWrapper(
        cell = cell,
        output_keep_prob = self.keep_prob
        )

      self.output, state = tf.nn.dynamic_rnn(
        cell = dropout_cell,
        inputs = self.embed,
        sequence_length = self.sequence_length,
        initial_state = cell.zero_state(self.batch_size, tf.float32))

      self.sum = tf.reduce_sum(self.output, axis = 1)
      self.mean = tf.divide(self.sum, tf.expand_dims(self.sequence_len
gth,1))
```

```
      self.logits = tf.contrib.layers.fully_connected(self.mean,2,acti
vation_fn = None)
      #self.logits = self.mean
      self.probability = tf.nn.softmax(self.logits)
      self.decision = tf.argmax(self.probability,axis=1)
      self.actual = tf.argmax(self.targets,axis=1)
      self.probe = self.decision
      self.metric_accuracy,self.update_accuracy = \
          tf.metrics.accuracy(self.actual,self.decision)

      self.xent = tf.nn.softmax_cross_entropy_with_logits(
        labels = self.targets,
        logits = self.logits)

      self.loss = tf.reduce_mean(self.xent)
      # warning: changed logits to probability - may be numerically un
stable
      self.pos_grad = tf.gradients(self.probability[0][0], self.embed)
      self.neg_grad = tf.gradients(self.probability[0][1], self.embed)
      self.logit_grad = tf.gradients(self.logits[0][0], self.embed)
      self.updates = self.optimizer.minimize(self.loss, global_step =
self.global_step)

      tf.summary.scalar('Metric Accuracy', self.update_accuracy)
      tf.summary.scalar('Loss', self.loss)

      self.merged = tf.summary.merge_all()

    def infer_dpg(self,sess,rv):
        decision, probability, grad = \
            sess.run([self.decision,self.probability,self.pos_grad],
                feed_dict = \
                    {self.inputs:rv.index_vector,
                     self.targets:rv.targets,
                     self.sequence_length:[rv.length],
                     self.keep_prob:1.0})

        return decision, probability, grad

    def infer_rep_dpg(self,sess,rv,word_index):
        index_matrix = np.tile(rv.index_vector,(rv.length,1) )
        np.fill_diagonal(index_matrix,word_index)
        target_matrix = np.tile(rv.targets,(rv.length,1))

        decision, probability = \
            sess.run([self.decision,self.probability],
                feed_dict = \
                    {self.inputs:index_matrix,
                     self.targets:target_matrix,
                     self.sequence_length:[rv.length]*rv.length,
                     self.keep_prob:1.0})

        return decision, probability

    def infer_batched_prob(self,sess,rv,word_index,per_batch,top_idx):
        num_top = self.batch_size//per_batch
```

```
        index_matrix = np.tile(rv.index_vector,(self.batch_size,1))
        target_matrix = np.tile(rv.targets,(self.batch_size,1))
        n = 0
        for idx in top_idx:
            for i in range(per_batch):
                index_matrix[n,idx] = word_index + i
                n = n + 1

        decision, probability, grad = \
            sess.run([self.decision,self.probability,self.pos_grad],
                feed_dict = \
                    {self.inputs:index_matrix,
                     self.targets:target_matrix,
                     self.sequence_length:[rv.length]*self.batch_size,
                     self.keep_prob:1.0})

        return decision, probability, grad

    # inserts all words at index 'insert_location'
    def infer_insert(self,sess,rv,insert_location,divs,top_k):
        per = rv.length // divs
        end = per*divs
        new = per+1
        index_matrix = np.zeros((self.batch_size,new),'int')
        wim = np.reshape(rv.index_vector[0,0:end],(per,divs),'F')
        wim = np.insert(wim,insert_location,0,axis=0)
        wim = np.reshape(wim,(1,end+divs),'F')
        target_matrix = np.tile(rv.targets,(self.batch_size,1))
        for i in range(divs):
            start = i*top_k; end = (i+1)*top_k
            index_matrix[start:end,:] = np.tile(wim[0,new*i:new*(i+1)]
,(top_k,1))
        #index_matrix = np.tile(wim,(self.batch_size//divs,1))
        index_matrix[:,insert_location] = \
            np.arange(self.batch_size) % (self.batch_size//divs)
        d,p,g = sess.run( [self.decision, self.probability, self.pos_g
rad],
            feed_dict = \
                {
                  self.inputs: index_matrix,
                  self.targets: target_matrix,
                  self.sequence_length: [new] * self.batch_size,
                  self.keep_prob: 1.0
                })
        return d,p,g,index_matrix

    def infer_swap(self,sess,rv,swap_location,divs,top_k):
        per = rv.length // divs
        end = per*divs
        index_matrix = np.zeros((self.batch_size,per),'int')
        wim = np.copy(np.reshape(rv.index_vector[0,0:end],(per,divs),'
F'))
        wim[swap_location,:] = 0
        wim = np.reshape(wim,(1,end),'F')
        target_matrix = np.tile(rv.targets,(self.batch_size,1))
        replacement_vector = np.random.choice(np.arange(10000),size=se
lf.batch_size)
```

```
        for i in range(divs):
            start = i*top_k; end = (i+1)*top_k
            index_matrix[start:end,:] = np.tile(wim[0,per*i:per*(i+1)]
,(top_k,1))
        index_matrix[:,swap_location] = replacement_vector
            #np.arange(self.batch_size) % (self.batch_size//divs)
        d,p,g = sess.run( [self.decision, self.probability, self.pos_g
rad],
            feed_dict = \
                {
                  self.inputs: index_matrix,
                  self.targets: target_matrix,
                  self.sequence_length: [per] * self.batch_size,
                  self.keep_prob: 1.0
                })
        return d,p,g,index_matrix,replacement_vector

    def infer_window(self,sess,rv,word_index,window_size):
        w = window_size
        L = rv.length
        n = word_index
        n1 = n - w//2; n2 = n1 + w
        if n1 >= 0 and n2 <= L:
            i1 = n1; i2 = n2
        elif n1 < 0 and n2 <= L:
            i1 = 0; i2 = w
        elif n1 >=0 and n2 > L:
            i1 = L-w; i2 = L
        else:
            i1 = 0; i2 = rv.length
        index_matrix = np.tile(rv.index_vector[0,i1:i2],(10000,1))
        index_matrix[:,word_index-i1] = np.arange(10000)
        target_matrix = np.tile(rv.targets,(self.batch_size,1))

        d,p,g = sess.run( [self.decision, self.probability, self.pos_g
rad],
            feed_dict = \
                {
                  self.inputs: index_matrix,
                  self.targets: target_matrix,
                  self.sequence_length: [w] * self.batch_size,
                  self.keep_prob: 1.0
                })
        return d,p,g

    def infer_multi(self,sess,rv,ii,K):
        # K is the list of source word indices to be replaced
        # batch size is fixed at 10,000 with number of
        # destination words = 10,000 / len(K)
        N = self.batch_size // K.size
        index_matrix = np.tile(rv.index_vector[0,:],(self.batch_size,1
))
        c = 0
        for k in list(K):
            index_matrix[c*N:(c+1)*N,k] = ii[np.arange(N),k]
            c = c + 1
        target_matrix = np.tile(rv.targets,(self.batch_size,1))
```

```
        d,p,g = sess.run( [self.decision, self.probability, self.pos_g
rad],
            feed_dict = \
                {
                  self.inputs: index_matrix,
                  self.targets: target_matrix,
                  self.sequence_length: [rv.length] * self.batch_size,
                  self.keep_prob: 1.0
                })
        return d,p,g
```

```
import os, re, sys
import tensorflow as tf
import numpy as np
import rnn
import preprocess
import random

# Yields (batch_size) examples, truncated at (max_time) words.
# Examples are randomly pulled from (input_directory) and
# Translated to integers using (emb_dict)
def dataset(input_directory, batch_size, emb_dict, max_time):
  file_gen = os.listdir(input_directory)
  random.shuffle(file_gen)
  batch_num = 0
  inputs = np.zeros((batch_size, max_time))
  targets = np.zeros((batch_size,2))
  sequence_length = np.zeros((batch_size))
  for name in file_gen:
    file_path = input_directory + '/' + name
    f = open(file_path)
    w = preprocess.tokenize(f.read())
    rating = int(re.sub('_|\.txt',' ',name).split()[1])
    targets[batch_num][0:2] = [0,1] if rating < 5 else [1,0]
    sequence_length[batch_num] = len(w)
    for time_num in range(min(max_time,len(w))):
      inputs[batch_num][time_num] = emb_dict.get(w[time_num],0)

    batch_num += 1
    if batch_num == batch_size:
      yield inputs,targets,sequence_length
      batch_num = 0
      inputs = np.zeros((batch_size, max_time))
      targets = np.zeros((batch_size,2))
      sequence_length = np.zeros((batch_size))

batch_size = 1000  # Number of reviews to consider at once
max_time = 1024   # Maximum number of words in a given review
# Load the word embedding
emb_dict = np.load('emb_dict.npy').item()
embeddings = np.load('final_embeddings.npy')

for hidden_size,lr in zip([2,4,8,16,32,64,128]*2,[0.01,0.001]*7):
  # Reset the TensorFlow graph
  g = tf.Graph()
  tf.reset_default_graph()
  with g.as_default():
    # Set global step to zero, for keeping track of training progress
    global_step_tensor = tf.Variable(0, trainable = False, name = 'glo
bal_step')
    # Make the RNN
    r = rnn.classifier(
      batch_size = batch_size,
      learning_rate = lr,
      hidden_size = hidden_size,
      max_time = max_time,
      embeddings = embeddings,
      global_step = global_step_tensor
```

```
    )

    # Training session
    with tf.Session() as sess:
      saver = tf.train.Saver(max_to_keep = 200)
      train_writer = tf.summary.FileWriter(sys.argv[1]+'/train_'+str(h
idden_size)+'_'+str(int(lr*1000)))
      test_writer = tf.summary.FileWriter(sys.argv[1]+'/test_'+str(hid
den_size)+'_'+str(int(lr*1000)))
      sess.run(tf.global_variables_initializer())
      for epoch in range(50):
        saver.save(sess, './'+sys.argv[2]+'_'+str(hidden_size)+'_'+str
(int(lr*1000))+"/imdb-rnn-e%d.ckpt"%epoch)
        print('epoch: ',epoch)
        # Testing #
        sess.run(tf.local_variables_initializer())
        if not (epoch % 5):
          for inputs,targets,sequence_length in dataset(\
              './aclImdb/test/posneg/',batch_size,emb_dict,max_time):
            accuracy,global_step,summary = sess.run([r.update_accuracy
, global_step_tensor, r.merged],
              feed_dict={r.inputs:inputs,r.targets:targets,r.sequence_
length:sequence_length,
                         r.keep_prob:1.0})
          test_writer.add_summary(summary, global_step)
        # Training #
        sess.run(tf.local_variables_initializer())
        for inputs,targets,sequence_length in dataset(\
            './aclImdb/train/posneg/',batch_size,emb_dict,max_time):
          loss,updates,embed,output,updated_accuracy,\
          mean,logits,prob,sequence_length,\
          probe,labels,global_step,learning_rate,summary = \
            sess.run([r.loss,r.updates,r.embed,r.output,r.update_accur
acy,\
                  r.mean,r.logits,r.probability,r.sequence_length,\
                  r.probe,r.targets,global_step_tensor,r.learning_rate
,r.merged],
              feed_dict={r.inputs:inputs,r.targets:targets,r.sequence_le
ngth:sequence_length,
                         r.keep_prob:0.5})
        train_writer.add_summary(summary, global_step)
```