Abstract

Neural networks have recently been found vulnerable to "attacks" which cause them to misclassify samples given a very small disturbance. Attacks based on iterative gradient methods have been largely studied for continous domain samples like images. In this work we introduce a hybrid algorithm for the non-continous domain of text. Our algorithm uses 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. We test the algorithm under white box, gray box, and black box scenarios

Contents

Chapter 1

Background

1.1 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 unsepervised learning.

In the case of supervised learning, we are given a set of inputs, $\{x\}_{i=1}^{N} \in X$ and outputs, $\{y\}_{i=1}^{N} \in Y$ to 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 due our model for the function not being complex enough, or for being too complex (this is known as overfitting, discussed in section 1.1.2.) The objective is therefore simplified as 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:

$$\arg\min_{g \in M} L(g, x, y) \tag{1.1}$$

One simple case of this is linear regression. In linear regression, M is the set of linear (or in most cases affine) functions mapping X to Y. and the loss function is $L(g, x, y) = \frac{1}{N} \sum_{i=1}^{N} ||g(x_i) - y_i||^2$ It turns out that under these constraints, an

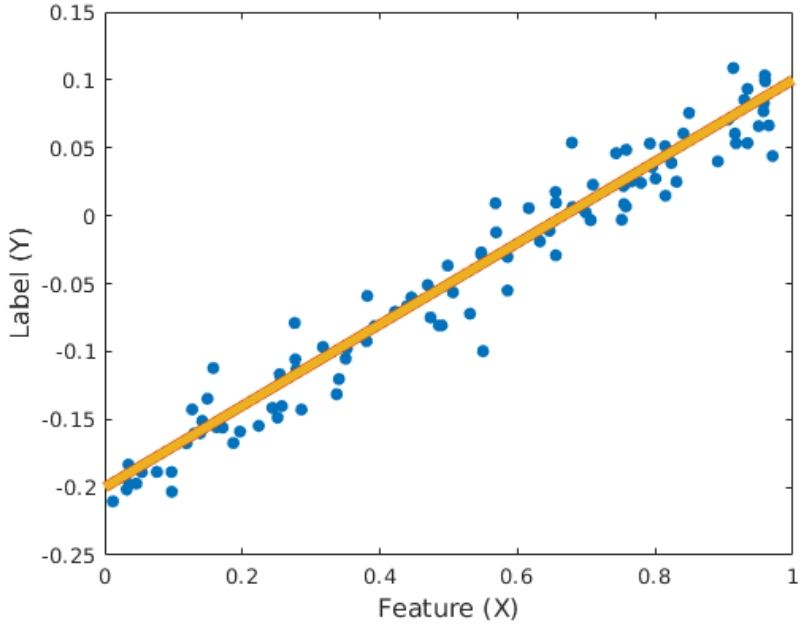


Figure 1.1: Linear regression for a noisy line. A blue dot represents the position of a feature/label pair. The yellow line represents the approximate affine approximation function.

exact solution can be found. An illustration where $X = Y = \mathbb{R}$ is shown in figure 1.1.

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, essentially a problem 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 measures of the algorithm's success.

One well known example of unsupervised learning is principal component analysis and its cousin singular value decomposition. Given some features in the form of a matrix X, singular value decomposition can find 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 1.2 uses singular value decomposition to remove noise from an

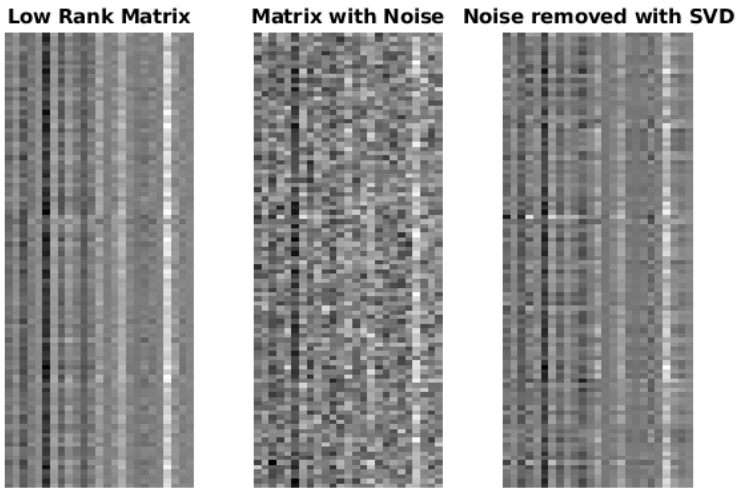


Figure 1.2: Illustration of using SVD to recover an underlying signal. This is an example of unsupervsied learning.

assumed low rank matrix. In figure 1.3, a plot shows the relative contribution of automatically detected signal components.

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

We can see from figure 1.4 that this hyperparameter could be calculated, or at least guessed from the number of dominant singular values. However, it may be impossible to know what the underlying rank was as in the case of figure 1.3 if the original matrix does not match our assumptions or if noise dominates the signal. 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 neccesarily the same as the model's loss

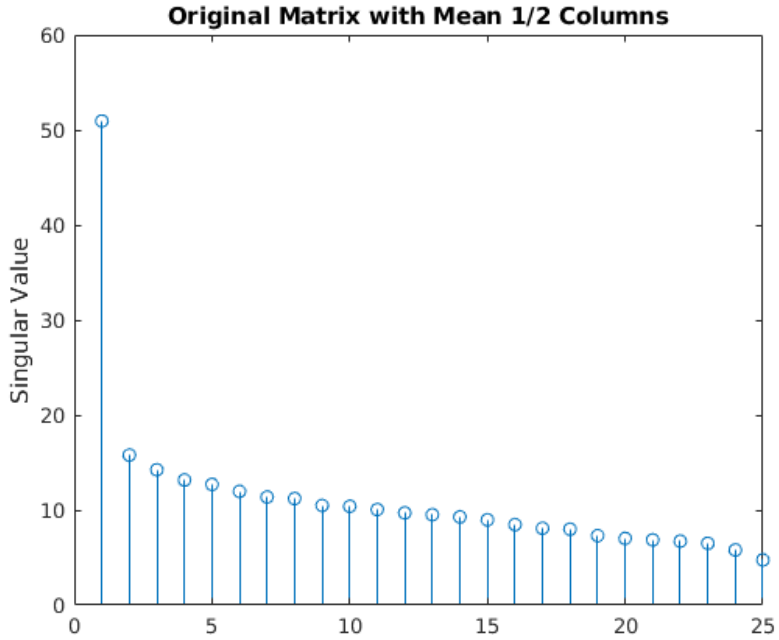


Figure 1.3: Set of 25 singular value for the low rank matrix with noise in figure 1.2. Each column of the original matrix has a mean of about 0.5. Each singular value is a measure of some signal component present in the matrix.

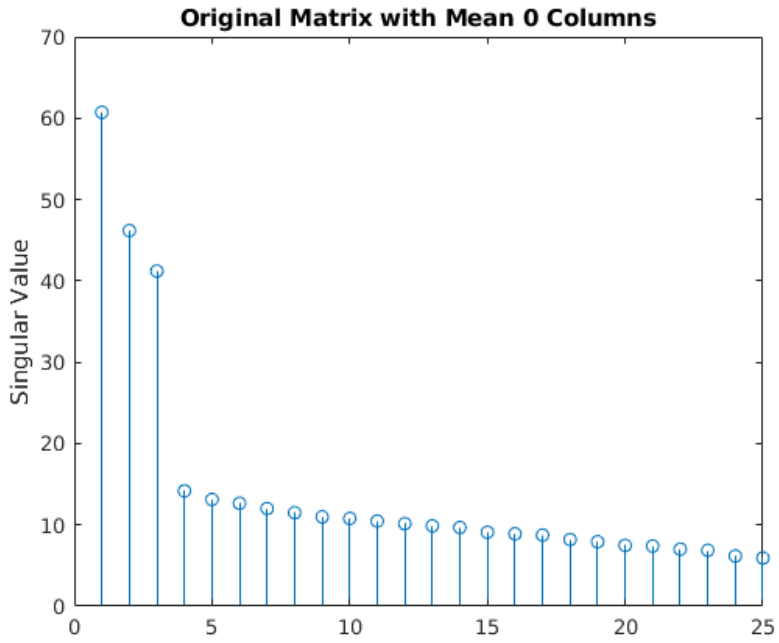


Figure 1.4: Set of 25 singular value for a low rank matrix, each column of the original matrix has mean 0. Here we see three distinct which tells us that the origin rank was very likely three.

function. The process of choosing optimal hyperparameters is known as hyperparameter tuning.

1.1.2 Overfitting

If we allow our model, or set of permissible functions, to be more complex, we can represent more complicated functions. Figure 1.5 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 in fact achieves a lower error. It is often true, especially in simpler cases, that increasing the model complexity will reduce the error of the objective. However if our goal is to determine the actual underlying pattern of the data, we may want to choose something simpler.

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. 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. By the Riesz representation theorem, for any linear function, $f : \mathbb{R}^m \to \mathbb{R}$, $\exists v \in \mathbb{R}^m$ s.t. $f(x) = \langle x, v \rangle$ for some vector $v \in \mathbb{R}^m$. Suppose v is the vector corresponding to linear function g in this manner, then for linear regression the new loss function is given in equation 1.2

$$L(g, x, y) = ||g(x_i) - y_i||^2 + \lambda ||v||_2^2 \tag{1.2}$$

where λ is a real scalar hyperparameter that may be tuned in the manner discussed in section 1.1.2. While there is no efficient method for computing the optimal value for λ in general, it has a simple value given a certain assumption. If we assume 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 is

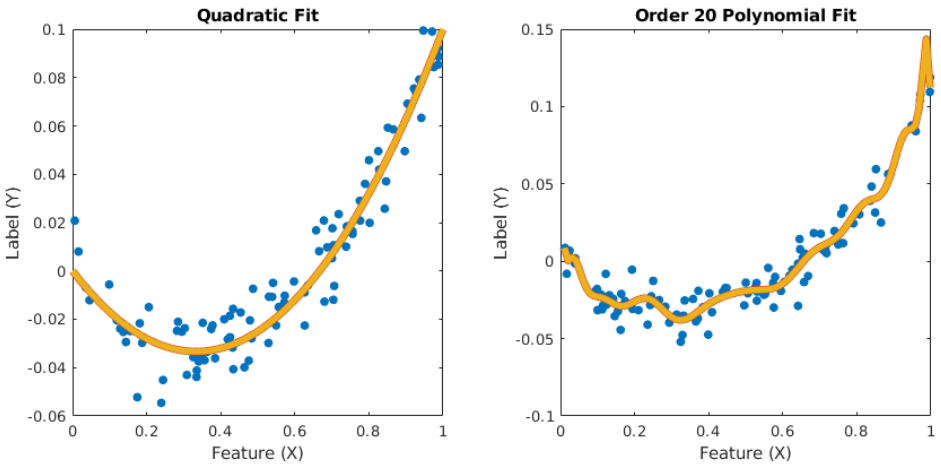


Figure 1.5: The figure on the left is the result of an optimization constrained to second order polynomials while the figure on the right is the result of a 20^{th} order constraint. The second order constraint achieves a result closer to the underlying curve.

the same as the solution to equation 1.2.

1.1.3 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 we discussed in section 1.1.2, models which fit the underlying function worse may easily achieve a better overall loss. In fact for any finite dataset that could feasible 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 wouldn't represent the underlying function in a meaningful way.

To help mitigate this issue, 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* which we discuss in section **??**.

1.2 Neural Networks

A neural network is a class of mathematical function which uses non-linearities in order to increase representational power as compared to simple linear models. 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 *activiation functions*. We now discuss one of the simplest neural networks to highlight and clarify some of these terms, the multilayer perceptron.

1.2.1 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 equation 1.3

$$f(x) = \vec{\sigma_p} \circ T_p \circ \cdots \circ \vec{\sigma_2} \circ T_2 \circ \vec{\sigma_1} \circ T_1 \circ x \tag{1.3}$$

The functions $T_1, \ldots, T_p$ are assumed to be affine and often represented as matrix multiplications. The functions $\vec{\sigma_1}, \ldots, \vec{\sigma_p}$ are assumed to be nonlinear, each of these is an activation function. In order for equation 1.3 to make sense, we must assume that the dimensions of each function are compatible with one another. As we will see in section **??**, activation functions are usually simple functions which operate on an element-by-element basis. Here we say the neural network described by 1.3 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 and exluded if performing a regression task.

1.2.2 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

$$f(x) = T_2 \circ \vec{\sigma} \circ T_1 \circ x \tag{1.4}$$

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 continous function on a compact set arbitrarily well with some T_1, T_2 with finite dimensional codomains. [1] This has been proved for the L^1 norm, L^2 norm, and the L^∞ norm. To be specific, if we define a univariate *sigmoidal* function as $\sigma : \mathbb{R} \to \mathbb{R}$ as any function satisfying

$$\sigma(t) \to \begin{cases} 1 \text{ as } t \to \infty \\ \\ 0 \text{ as } t \to -\infty \end{cases} \tag{1.5}$$

We define $\vec{\sigma}$ as the vectorization of σ. That is,

$$\vec{\sigma}(x)_i = \sigma(x_i) \tag{1.6}$$

(One common choice for such a function in neural networks is the logistic function given by $\sigma(x) = \frac{1}{1-\exp(x)}$.) The universal approximation theorem says that given a compact set, $I \subset \mathbb{R}^m$, $\epsilon > 0$, and this sigmoidal activation, we can pick T_1, T_2 such that any of the conditions in equations 1.7-1.9 could be met.

$$||f(x) - g(x)||_\infty < \epsilon \ \forall x \in I \tag{1.7}$$

$$||f(x) - g(x)||_1 < \epsilon \ \forall x \in I \tag{1.8}$$

$$||f(x) - g(x)||_2 < \epsilon \ \forall x \in I \tag{1.9}$$

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

However, as the author of [1] writes, this theorem gives no upper bound on the dimensionality of the output of T_1 and posits that this value be be very large. In addition, we are 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 our task is to find the corresponding coefficients, this task is called training. This is usually done with a very approximate algorithm, stochastic gradient descent, covered in section 1.3.2.

The two main issues of training is that the algorithms are not exact, and the size of matrices required to represent our function may be too large to be comuptationally feasible. For these reasons, most neural network research is devoted to creating structures which facilitate learning or reduce the computational complexity of a model. In section 1.4 we describe recurrent neural networks, a structure which is particularly efficient at learning features of time series. In section 1.3 we discuss several training strategies which help learn the correct parameters as well as avoid the problem of overfitting.

1.3 Training Algorithms

Neural networks are highly nonlinear, non-convex functions and therefore very difficult to train efficiently. We 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, we cannot see the terrain around us and only have local information about our 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.

1.3.1 Newton's Method

If our 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 we are at least at a global minimum or maximum. This equation is usually not directly solvable for nonlinear functions like neural networks. However, we can use Newton's method to successively approximate it. In reference to the loss function discussed in chapter **??**, 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, we have:

$$\nabla L(w + \Delta) \approx \nabla L(w) + HL(w)\Delta \tag{1.10}$$

If we want to find the zero of ∇L, we set $L(w + \Delta) = 0$ resulting in equation 1.11

$$\Delta = -[HL(w)]^{-1}\nabla L(w) \tag{1.11}$$

where H is the Hessian matrix of L. This yields the update step in equation 1.12

$$w \leftarrow w - [HL(w)]^{-1}\nabla L(w) \tag{1.12}$$

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 datasets, let alone its derivative with respect to all parameters. SGD mitigates these issues by avoiding the computation of both the loss or the Hessian.

1.3.2 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 we must assume that the loss function $L(g, x, y)$ discussed in chapter 1.1 can be broken down as a sum of loss functions over each feature-label pair, that is:

$$L(g, x, y) = \sum_{i=1}^{N} l(g, x_i, y_i) \tag{1.13}$$

Given an initial set of weights (often initialized pseudo-randomly), SGD follows the update in equation 1.14.

$$w \leftarrow w - \eta \nabla l(g, x_i, y_i) \tag{1.14}$$

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

Equation 1.14 says that SGD alters the weights in the direction of the negative gradient. Because the the gradient of a funciton 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

$$\nabla L(g, x, y) = \nabla \sum l(g, x_i, y_i) = \sum \nabla l(g, x_i, y_i) = 0 \tag{1.15}$$

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 we mentioned with regard to overfitting, you usually

do not want to completely minimize the loss function, so this is acceptable. 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.

We can think of SGD as approximating 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 good 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.

1.3.3 Adam Optimizer

The Adam Optimizer algorithm [2] essentially calculates how "reliable" each element of the gradient vector is and weights the updates accordingly. It accomplishes this by estimating the first and second moments of the gradient vector with a moving average exponential filter, essentially 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 1.6 shows the frequency response with beta ranging from 0.8 to 1.0 logarithmically. We see that 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 represent the signal to noise ratio. Using this interpretation, this update weighting is somewhat

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

similar to a Wiener filter where the components of the signal which have the most noise are proportionally supressed.

1.3.4 Automatic Differentiation

In all of the previous training algorithms, we have ignored the computation of the gradient of the loss, $\nabla l(w, x, y)$. Here we breifly 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 taken from [] is shown in equation 1.16.

$$f(x_1, x_2) = \log(x_1) + x_1 \times x_2 - \sin(x_2) \tag{1.16}$$

Equation 1.16 is a composition of log, $\times$, sin, $+$, and $-$. This composition can

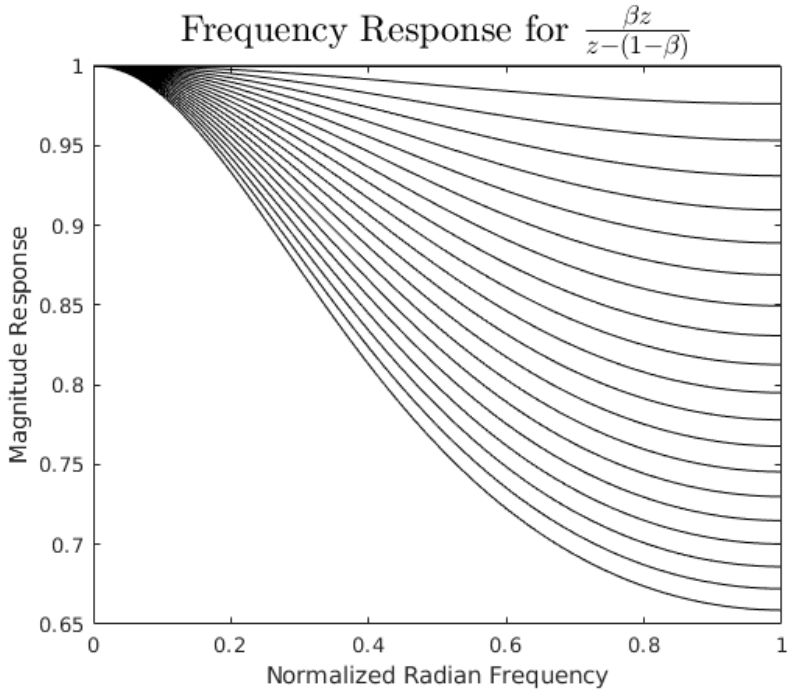


Figure 1.6: Frequency response for several moving average exponential filters. Lower lines are associated with lower values of β.

be captured in the form of a graph, as seen in figure 1.7. Starting at the output and working backwards, we 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 $[\log(x_1) + x_1 \times x_2] - [\sin(x_1)]$, we 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, we look at the nodes associated with those functions where the derivative will either already be computed, or where we 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.

1.4 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 more detail we introduce some notation. Let the set of all n-

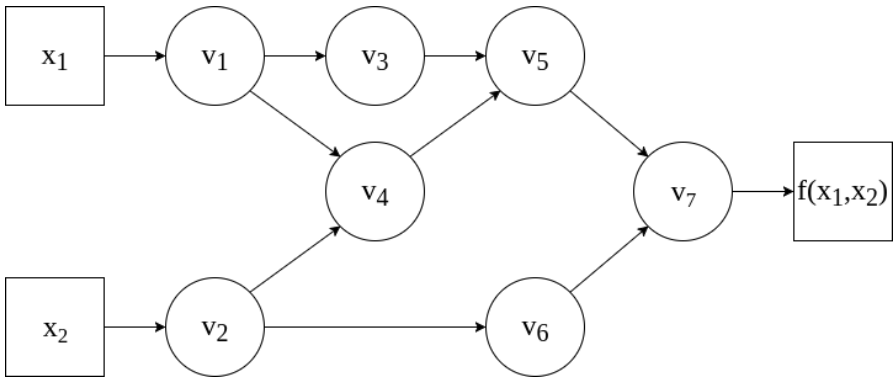


Figure 1.7: Graph reproduced from [3]. Each node represents a function of the incoming nodes.

dimensional vector sequences (finite or otherwise) be given by

$$S_n = \{s : Z \to \mathbb{R}^n;\ \forall Z \subseteq \mathbb{Z}^+\} \tag{1.17}$$

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

1.4.1 Overview

Of course, we could extend the multilayer perceptron match this if we window the sequence, compute the output, and then slide the input window. What truly separates RNNs from MLPs is the fact that an RNN's output depends on its "state" at previous time steps. In particular, if we denote the input sequence as $x = (x_1, x_2, \ldots, x_k)$ then the states at time steps $1, \ldots, k$ are given by equations 1.18-1.21.

$$S_1 = s(x_1, S_0) \tag{1.18}$$

$$S_2 = s(x_2, s(x_1, S_0)) \tag{1.19}$$

$$S_3 = s(x_3, s(x_2, s(x_1, S_0))) \tag{1.20}$$

$$\vdots$$

$$S_k = r(x_k, S_{k-1}) \tag{1.21}$$

And the ouputs are given by equations 1.22-1.25

$$R_1 = r(x_1, S_1) \tag{1.22}$$

$$R_2 = r(x_2, r(x_1, S_1)) \tag{1.23}$$

$$R_3 = r(x_3, r(x_2, r(x_1, S_1))) \tag{1.24}$$

$$\vdots$$

$$R_k = r(x_k, S_k) \tag{1.25}$$

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.

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 nonlinearities 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.

1.4.2 Early Networks

Some of the first known RNNs, now known as "simple recurrent networks" are the Elman [4] and Jordan [5] networks. The Elman update equations are given by equations 1.26 and 1.27.

$$S_k = \sigma_S(W_S x_k + U_S S_{k-1} + b_S) \tag{1.26}$$

$$R_k = \sigma_R(W_R S_k + b_R) \tag{1.27}$$

while the Jordan update equations are given by equations 1.30 and 1.31.

$$S_k = \sigma_S(W_S x_k + U_R R_{k-1} + b_S) \tag{1.28}$$

$$R_k = \sigma_R(W_R S_k + b_R) \tag{1.29}$$

The Elman equations are already in a form consistent with our framework. Substituting the second Jordan equation into the first yields equations 1.28-1.29

$$S_k = \sigma_S(W_S x_k + U_R \sigma_R(W_R S_{k-1} + b_R) + b_S) \tag{1.30}$$

$$R_k = \sigma_R(W_R S_k + b_R) \tag{1.31}$$

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 bias vectors.

A result analagous to the universal approximation theorem was proved in [6]. 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) could 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, or how many resources it would require.

1.4.3 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 we have equation 1.32

$$\frac{d}{dw}R_k(w, x_k, S_k) = r'(w, x_k, S_k) \times s'(w, x_{k-1}, S_{k-1}) \times$$
$$s'_{k-1}(w, x_{k-2}, S_{k-2}) \times \cdots \times s'(w, x_2, S_1) \times s'(w, x_1, S_0) \quad (1.32)$$
$$= R'_k(x_k, S_k) \prod_{i=1}^{k} s'(x_i, S_{i-1}) \quad (1.33)$$

The log derivative is given by

$$log(R'_k(x_k, S_k)) + \sum_{i=1}^{k} \log(s'(x_i, S_{i-1})) \quad (1.34)$$

Although it is hard to formally say anything about this value, intuitively we see since we are accumulating values 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 gradient is that when the state transitions from one time step to another, it is multiplied by some matrix followed by a nonlinearity. The chain rule causes this matrix to be multiplied by itself repeatedly in the derivative 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, that is exactly the diea behind the long short-term memory (LSTM) unit. [7] The update equations for the original LSTM RNN are given in equations

1.35-1.39

$$i_k = \sigma_i(W_i x_k + U_i R_{k-1} + b_i) \tag{1.35}$$

$$o_k = \sigma_o(W_o x_k + U_o R_{k-1} + b_o) \tag{1.36}$$

$$c_k = \sigma_c(W_c x_k + U_c R_{k-1} + b_c) \tag{1.37}$$

$$S_k = S_{k-1} + i_k \circ c_k \tag{1.38}$$

$$R_k = \sigma_R(S_k) \circ o_k \tag{1.39}$$

Where $\circ$ represents Hadamard (entrywise) multiplication.

It is easy to see that these equations satisfy the update equations of a standard RNN, as we defined. 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 and σ_o is usually the logistic function.

The model was later extended by [8] with what they called a forget gate. The forget gate, described by the new update equations 1.40-1.41, allows the network to essentially discard information in the state and start again.

$$f_k = \sigma_f(W_f x_k + U_f R_{k-1} + b_f) \tag{1.40}$$

$$S_k = S_{k-1} \circ f_k + i_k \circ c_k \tag{1.41}$$

The same group responsible for the invention of forget gates also invented the peephole LSTM. [9] 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 1.35 becomes equation 1.42 and so on.

$$i_k = \sigma_i(W_i x_k + U_i S_{k-1}) \tag{1.42}$$

1.4.4 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 nueral network is "unwrapped" for some finite amount of time steps. Any input which goes past the maximum time step is discarded in the computation.

1.5 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 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.

1.5.1 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. 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 the document

```
"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 not 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 1.1

this	3
movie	3
is	3
not	1
terrible	1
amazing	1

Table 1.1: 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. If we want 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 1.1 might be represented as

$$x = \begin{bmatrix} 0 & \cdots & 3 & \cdots & 3 & \cdots & 3 & \cdots & 1 & \cdots & 1 & \cdots & 1 & \cdots & 0 \end{bmatrix}^{\mathsf{T}} \tag{1.43}$$

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

$$s_\theta(x_1, x_2) = \frac{x_1 \cdot x_2}{||x_1||_2 ||x_2||_2} \tag{1.44}$$

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 we might represent individual words numerically. If i is the index associated with the i^{th} word in a vocabulary, then we 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. Let e_m be the standard basis vector associated with word m, then the vector represention for a document multiset, M, would be given by

$$x = \sum_{m \in M} v(m) e_m \tag{1.45}$$

where v is the multiplicity of element m in M.

In most cases, an actual language processing system will 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 [10]:

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

$$idf(t,D) = -\log d_t/D \tag{1.46}$$

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):

$$tfidf(t,d,D) = tf(n,d) \times idf(t,D) \tag{1.47}$$

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. We now look at latent semantic analysis, originally a method for indexing documents, which represents words and documents as vectors.

1.5.2 Latent Semantic Analysis

Latent semantic analysis (LSA) is essentially a dimensionality reduction technique similar to principle component analysis. At it's 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 unitary 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.

1.5.3 Word2vec

Word2vec [11] is an extremely useful model which learns vector representations of individual words. Word vectors are learned in an unsupervised fashion, mean-

ing that very large, unlabeled datasets can be leveraged during training. There are two word2vec models which are complementary: Continuous Bag-of-Words (CBOW) and Continous Skip-gram (Skip-gram). Continous Bag-of-Words 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 cener word.

In the skip-gram model, the function used for word prediction is a simple hourglass shaped log-linear model. Given a one-hot encoded word vector, as described in section 1.5.1, transpose it and right multiply it by some matrix, $A \in M_{V\times D}(\mathbb{R})$. Note that since the vector is one-hot, 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 or hierarchical softmax function, 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.

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 1.5.1, this is the bag-of-words representation of the window surrounding the center word. As in the skip-gram model, we right multiply the transpose of this vector by some matrix, $A \in M_{V\times D}(\mathbb{R})$. In the case of skip-gram this ammounted to selecting a row, now that the vector is multiple-hot encoded, this amounts to adding several rows of the matrix together. We 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 guassian random variable generator. The softmax cross entropy loss is used as the loss function, and all weights are trained using gradient descent and backpropagation (see sections 1.3.2 and **??**). The target vector represents a subset of all words within the skip window, though not necessarily the entire set. The number of skips represents how many times we sample the skip window to obtain a target vector. If the number of skips is double the skip window, we sample all words.

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 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 dataset, 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. [12] Here we discuss hierarchical softmax, negative contrastive estimation, and subsampling of frequent words.

The typical softmax function with N inputs has N outputs. If we are 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 equation 1.48

$$\hat{\sigma}(u, w) = \prod_{n \in p_u} \sigma(s(n) \times v_n^T v_w) \tag{1.48}$$

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. If we assign the edge connecting node n to its parent the value $\sigma(s(n) \times v_n^T v_w)$, we can think of 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:

$$\begin{aligned} P(u|w) &= P(u, n_L, n_{L-1}, \ldots, n_1|w) && (1.49)\\ &= P(u|n_L, \ldots, n_1, w) \times P(n_L|n_{L-1}, \ldots, n_1, w) \times \cdots \times P(n_1|w) && (1.50)\\ &= \hat{\sigma}(u, w) && (1.51) \end{aligned}$$

Equation 1.50 follows from 1.49 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 we used in our study, is negative sampling, an simplification of or negative contrastive estimation. [13] Instead of a typical softmax cross entropy loss, we can use a modified loss with drastically reduced computa-

tion complexity. Negative sampling (NEG) is defined by maximizing the objective in equation 1.52

$$G(u, w) = \log \sigma(\hat{v}_u^T v_w) + \sum_{i=1}^{k} \mathrm{E}\left[\log \sigma(-\hat{v}_{w_i}^T v_w)\right] \tag{1.52}$$

where u is 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 $w_i \sim P(w)$. The objective increases as the w vector becomes more similar to the u vector while becoming less similar to "noise" vectors, v_{w_i}. This 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 [12] found that the probability mass function given in equation 1.53 worked well.

$$P(w) = \frac{f(w)^{3/4}}{\sum_{i=1}^{V}(f(w_i)^{3/4})} \tag{1.53}$$

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 cetner word. Given the word "the" it is near impossible to say anything about the words within a modest context size.

This effect can be avoided by randomly removing words from the trainig 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 [12] chose

the function given in equation 1.54

$$P(w) = \begin{cases} 0 & \text{if } f(w) < t \\ 1 - \sqrt{\frac{t}{f(w)}} & \text{else} \end{cases} \tag{1.54}$$

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 "agressively subsamples words whose frequency is greater than t. [12]

Chapter 2

Problem Statement

2.1 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 [14]. We take images to be vectors in $\mathbb{R}$, and S is a finite set of classes, so that a classifier $f : \mathbb{R} \to S$ assigns each image a unique class.

Definition. An adversarial derivation, $x + r \in \mathbb{R}^m$, of an image, $x \in \mathbb{R}^m$ 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. $x + r \in [0, 1]^m$

We may also denote the adversarial derivation 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. We also require that every element stays in the interval $x_n + r_n \in [0, 1]$ because actual pixels must remain in some constant bounded interval.

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. We give a more general constrained optimization defintion below, which we will adapt 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 $\epsilon > 0$ given by

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

Note that 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 if we let $\epsilon = 1$, $d_D(x, x^*) = ||x - x^*||^2$, $d_C(y, y^*) = |y - y^*|$.

As long as the function, f, has a non-singleton codomain, the constraint is satisfiable for for small enough ϵ. In the case that the solution does not exist,

$\delta = \inf\{d_D(x, x^*) \text{ s.t. } d_C(f(x), f(x^*)) > \epsilon\}$ exists, in which case we 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. We therefore define a similar problem with very different properties

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. 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 we may solve the more relaxed problem to obtain a valid solution, and so we will work with the more practical objective of creating ADs.

2.2 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 we call E 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 sequence of bounded integers. The sequence is some length which depends on both the encoder and the string. We assume a fixed encoder, and therefore vocabulary size, V. Since, after encoding, the distance between one word and another is arbitrary, we 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. We denote the set of one-hot encoded vectors of size V 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 be difficult train. Using the word2vec model [12], the dimension of this representation can be significantly reduced, while also encoding information about statistical semantic similarity about each word.

Definition. Let $f : 1_V \to \mathbb{R}^D$. We call f a word embedding and we call D 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 we can now apply more general and modern techniques to solving the problem of classification.

2.3 Adversarial Text Derivation

Now that we have a clear idea of both the domain and numerical representation of words, we may define an adversarial derivation of textual data in the context

of a classification model, f. As per the definition of an adversarial derivation, we need only to define the model tolerance, ϵ, as well as the domain metric, d_D and codomain metric, d_C. We will consider primarily two definitions.

Definition. Let $\{v_i\}_{i=1}^N$ be the sequence of vectors obtained from a given word embedding and text sample, then a discrete adversrial derivation is defined has having 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 $\epsilon = 1$.

That is, a discrete adversarial derivation $\{v_i^*\}$ of sample $\{v_i\}$ is the sample which changes the least amount of words possible, while changing the classification. This definition is simple, though it may not yield very good results if solved. For example, a positive movie review, "This movie was good" could easily be changed to a negative review by changing just one word resulting in "This movie was bad". These two samples would obviously have different sentiments if read by a human.

Chapter 3

Preliminary Results

3.1 Word Embedding Training

We used gradient descent to train a word2vec model with noise contrastive estimation. The hyperparameters we chose are shown in table 3.1

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 3.1: Word embedding hyperparameters

Our corpus was the concatenation of all preprocessed training samples from the training set in the "Large Movie Review Datasets." [15] Preprocessing consisted of the steps laid out in table 3.2

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 3.2: Preprocessing algorithm

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 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, that is,

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

The eight nearest neighbors for a few common words are shown in table 3.3. We can see that the first few nearest neighbors are 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 3.3: Some examples of nearest neighbors in embedding space

3.2 RNN Training

Here we will focus on one particular type of model which has proven 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, corresponding to the logits of a positive and negative class. The network's confidence of a given samples class is given by the softmax of both

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 3.4: Training 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 3.5: Testing Accuracy

logits.

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 peepholes as well as 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 3.4 and 3.5 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.

3.3 Stochastic Gradient Analysis

Definition. Let the gradient of a model output, f_i, with respect to an input

vector, x be denoted by $g = \nabla f(x)$. The total gradient is given by

$$g_t = \sum_{i=1}^{D} \nabla f(x)_i \tag{3.1}$$

The gradient norm is given by

$$g_n = \sqrt{\sum_{i=1}^{D} |\nabla f(x)_i|^2} = ||\nabla f(x)||_2 \tag{3.2}$$

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 3.3. We see that 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

$$f(w_j) \approx f(x_i) + (w_j - x_i)^T \nabla f(x_i) \tag{3.3}$$

$$= f(x_i) + w_j^T \nabla f(x_i) - x_i \nabla f(x_i) \tag{3.4}$$

$$= f(x_i) + w_j^T g - x_i^T g \tag{3.5}$$

Here are considering specificially 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 defintions of x_i, w_j, and $\nabla f(x_i)$, let

$$X = \Big[x_1, x_2, \ldots, x_N \Big] \in M_{D \times N}(\mathbb{R}) \tag{3.6}$$

$$W = \Big[w_1, w_2, \ldots, w_V \Big] \in M_{D \times V}(\mathbb{R}) \tag{3.7}$$

$$G = \Big[\nabla f(x_1), \nabla f(x_2), \ldots, \nabla f(x_N) \Big] \in M_{D \times N}(\mathbb{R}) \tag{3.8}$$

The approximated perturbation, $f(w_j) - f(x_i)$, of replacing the i^{th} input with the

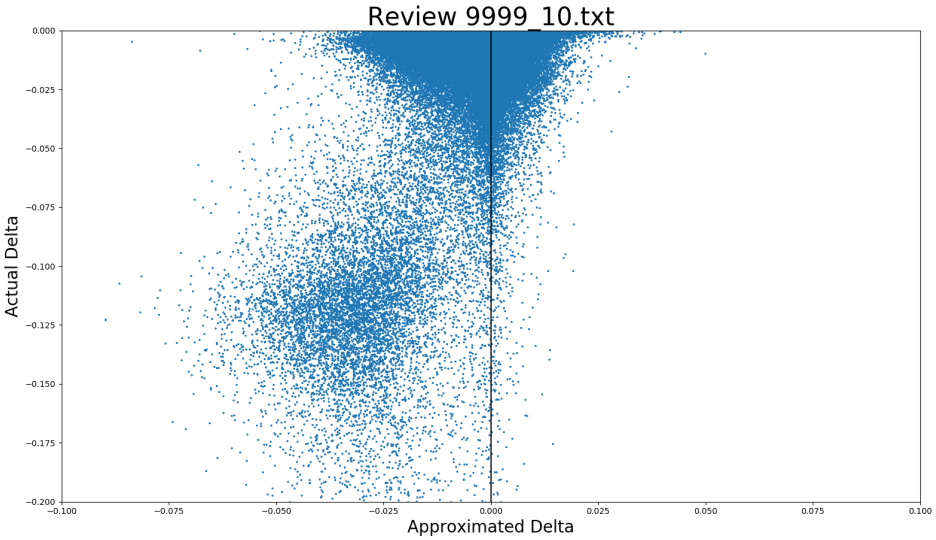


Figure 3.1: Predicted difference in prediction confidence vs. actual difference for sample file 9999_10.txt

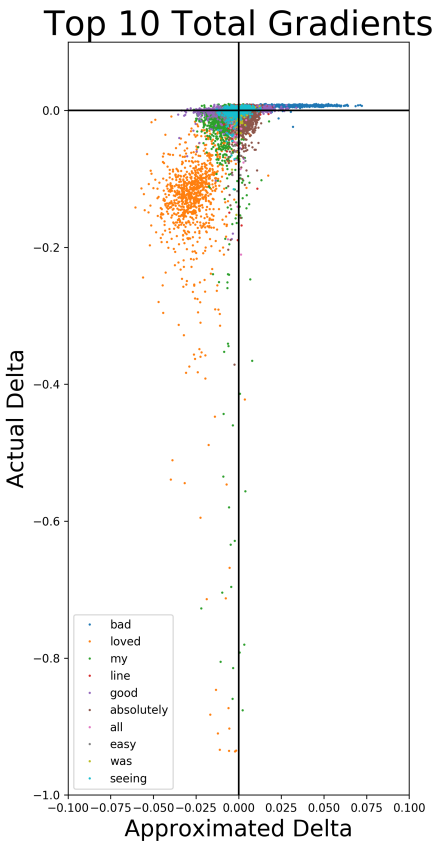
j^{th} embedding word vector is given by

$$D_{i,j} = (G^T W)_{i,j} - (G^T X)_{i,i} \tag{3.9}$$

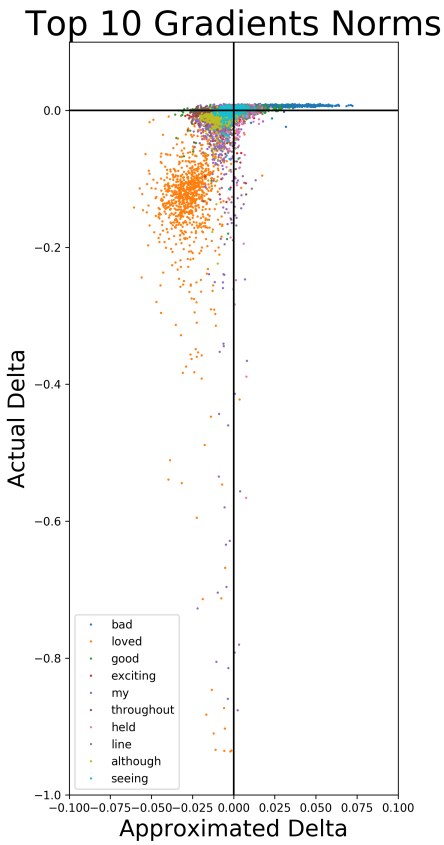
D is called the approximate delta matrix.

For a perfectly linear function, f, $D_{i,j} = f(w_j) - f(x_i)$. We found that 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 3.1. However, the approximation failed almost completely for large differences, as illustrated in figure 3.2. 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.

We see that individual $\hat{D}$ entries are not a good choice for determining which specific words to interchange, but that our measures of gradient may be useful in determining which words are susceptible to attack. We give motivation with some simplified probabilistic analysis. Suppose that each of the embedding dimensions is distributed independently and identically across words, with some mean, μ



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



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

Figure 3.2: 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.

and variance, σ^2. Let $g = \nabla f(x_i)$ for a given word vector, x_i and suppose we replace it with a random word vector, w_j. Recall equation 3.5, we are interested in estimating 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 $v = w_i$ and $u = x_i$, then randomly replacing u with another word vector, we have

$$\mathrm{E}\left[g^T v\right] = \mathrm{E}\left[\sum_{i=1}^{D} g_i v_i\right] = \sum_{i=1}^{D} g_i \,\mathrm{E}\left[v_i\right] = \mu \sum_{i=1}^{D} g_i = \mu g_t \tag{3.10}$$

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

$$Z(g) = \mathrm{E}\left[\max_{0 \le n \le V} g^T v_n\right] \tag{3.11}$$

Unfortunately, there is no simple expression which captures this value. However if we assume that $v_{n,i}$ is distributed normally, we have

$$Z(g) = \mathrm{E}\left[\max_{0 \le n \le V} g^T v_n\right] = \mathrm{E}\left[\max_{0 \le n \le V} \sum_{i=1}^{D} g_i v_{n,i}\right] = \mathrm{E}\left[\max_{0 \le n \le V} p_n\right] \tag{3.12}$$

where $p_n \sim \mathcal{N}(\mu g_t, \sigma^2 g_n^2)$ There is still no closed form expression for this value, but there is a known [16] upper bound:

$$Z(g) \le \mu g_t + \sigma g_n \sqrt{2 \log V} \tag{3.13}$$

$$\mathrm{E}\left[\max \hat{D}_{i,*}\right] \le \mu g_t + \sigma g_n \sqrt{2 \log V} - u^T g \tag{3.14}$$

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 including standard deviation is usually much larger. It should be noted that the lower bound on the expected minimum, $Y(g)$, is simply given

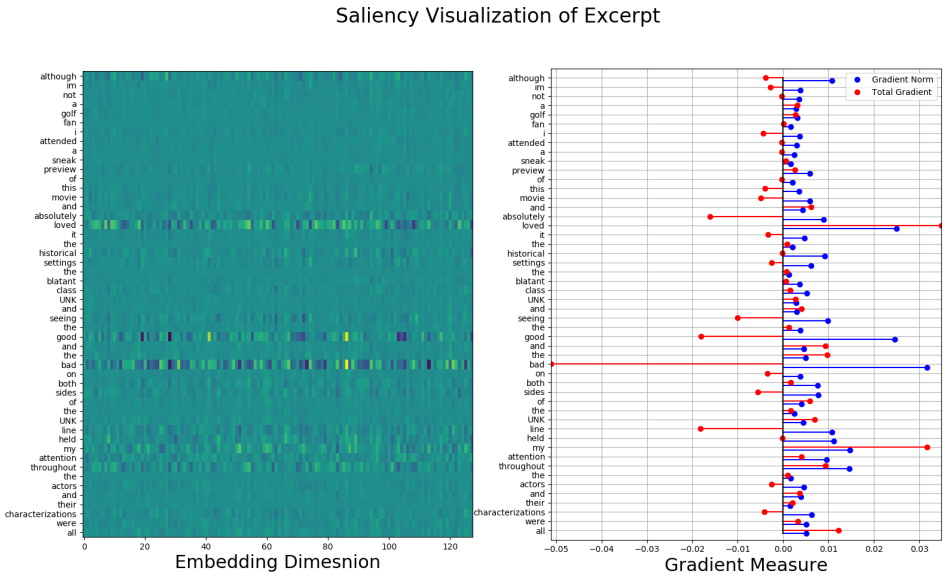


Figure 3.3: Different measures of word sentiment/importance

by a sign reversal of the second term:

$$Y(g) \geq \mu g_t - \sigma g_n \sqrt{2 \log V} \tag{3.15}$$

$$\mathrm{E}\left[\min \hat{D}_{i,*}\right] \geq \mu g_t - \sigma g_n \sqrt{2 \log V} - u^T g \tag{3.16}$$

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 4

Algorithm Description

While gradients tends to be overall useful indiciators for small perturbations, it is not entirely useful or accurate in predicting large differences. This makes sense given that gradient is only a local measure of change, however since we are attempting to find such large differences, we should not rely on the gradient alone. We discuss three solutions each of which builds on the last. Since we found that in most cases a classification change could be achieved by replacing only one word, we focus on this objective. If it turns out that more replacements are required, we can extend any of our approaches 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. In other words, if we let V ($|V| = N$) denote the set of vocabulary words and S ($|S| = M$) denote the set of sample words (along with their location in the sample), our 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 $s^{i,j}$ be given by equation 4.1

$$s^{i,j} = (s_1, s_2, \ldots, s_{i-1}, v_j, s_{i+1}, \ldots, s_M) \tag{4.1}$$

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

4.1 Full Search

For the objective of finding a single adversarial replacement, we begin with the brute force solution of a full search. We generate the matrix in equation 4.2 and search for the symbol which corresponds to an adversarial example.

$$M_{M,N}(\{0,1\}) \ni A_{i,j} = r(s^{i,j}) \tag{4.2}$$

This method of course has the advantage of providing 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, we can offload some time complexity to memory complexity and obtain any of the time complexities in table 4.1 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/computing power is available. With regard to the adversarial matrix A, the four complexities above correspond to computing one entry at a time, one row at a time, one column at a time, and computing the entire matrix in parallel, respectively.

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

4.2 Window Search

As we discuss in section **??**, a full search will not tend to work well for long samples. Depending on how the computation is organized, the algorithm will either run out of memory or take far too long to find a solution (or determine that there isn't one). Looking at the time complexities above, this is intuitive given that there is a quadratic dependence on M, the length of the sample. As mentioned in section 2.2, the average length of a review is about 234 words, meaning that typically $M^2 > N$ and sometimes $M^2 \gg N$. There are two ways we considered dealing with this quadratic growth.

The simplest solution is to cap M at some fixed value, C. and discard the rest of the sample, we call this cap search. As discussed in section **??**, 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 desireable 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 where there is one. The time and space complexities are given in table 4.2. This search generates a matrix A^{cs}, which we hope is approximately the same as the top C rows of A.

Consider that it may be valuable to consider all words in a review for replacement, especially very negative or positive ones. In this case, we would still like to find those words and therefore would like to consider every word in the sample

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

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

as a candidate for replacement. Instead of simply taking the first C words of a sample, we 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 4.3. Since C is strictly less than M, this is algorithm is slower and more memory intensive, and still not guaranteed to produce a correct result. This search generates a matrix A^{ws} which should approximate A.

4.3 Gradient Assisted Window Search

Finally, we may reduce complexity even further by using the results of section 3.3. Because we found that words associated with large gradients tend to be good words for replacement, we can reduce the effective value of M in the window search algorithm. Suppose we only consider the top K words for replacement, as ordered by the gradient and perform a search over those value, we call this method gradient assisted window search, or GAWS. The complexities are given simply by replacing M by K in the window search algorithm, and can be found in table **??**. This search generates a matrix A^{gs} which should approximate K rows of A corresponding to large gradients.

4.4 Multi-word Replacement

Up to this point, we have only considered searching for a one-word replacement adversarial derivation. While this is sufficient for about 70% of samples, there is still a need to produce derivations in the remaining 30%. If a given algorithm failed, it either didn't detect any adversarial derivations, or all of the adversarial derivations it detected were not true derivations. Both of these issues can be remedied with two extensions.

First, instead of dealing with the matrix A, we can deal with 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 1. Then in the same way we generate the matrix A and all of its approximations, we can also generate the matrix $P_{i,j} = p(s^{i,j})$. Thresholding this matrix would yield A or its approximation. Second, if no derivations are detected, we can pick the entry of P with the highest or lowest value (depending on what classification we are trying to cause) and attempt using that substitution. From this point onward, we assume the second tier of parallelization because we found that tier 3 was not feasible on our hardware.

If replacing one word is not successful, we can select more words. Doing this naively would lead to exponential blow up. 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 $O(M)$ 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 $O(N)$ time and only $O(M)$ space since the min/max is taken as soon as all elements are available)) and the top L entries of the result are chosen as candidate replacements. Sorting all values in the result requires $O(M \log(M))$ time.

Now, we still need some method of determining the smallest value of L which allows us to achieve a misclassification. We make the assumption that replacing more words than required will still lead to a misclassification, and therefore the

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

Table 4.5: 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 4.1 through 4.4.

class vs. L curve looks like a step function. Since it is monotonic, we can use an exponential search for the transistion point which runs in $O(M \log L)$ time and $O(W + M)$ 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 4.5

Chapter 5

Results

Here we list several different kinds of results for the window search method and the gradient assisted window search method. We present measures in the form of average number of word replacements per sample, time taken to converge, number of algorithm failures, as well as the transferability to different models with a similar architecture. In each case, we discard samples which were incorrectly classified in the first place, and use a single layer recurrent neural network with a hidden layer of size 16 as the base model. We measure the performace of each method over the same 500 randomly chosen samples.

5.1 White Box

Here we assume that all details of the model are known including the exact weights. We determine candidate word replacements based on inferring with those weights directly. figures 5.1 and 5.2 show the distribution of the number of words required to change a classification and the cumulative distribution respectively.

Gradient assisted window search achieved a mean of 2.25 words per misclassification, median of 2, and mode of 1. Figure 5.3 and 5.4 show the distribution of words required to change a classification and the cumulative distribution respectively. Figure 5.5 shows the distribution of time required to either achieve misclassification or recognize that the algorithm has failed to do so.

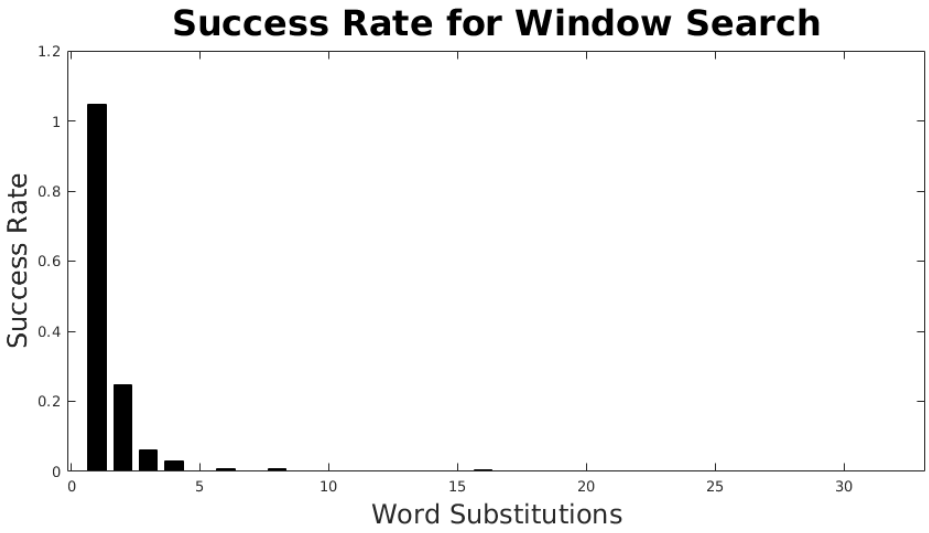


Figure 5.1: Success rate for window search vs. number of word substitutions required.

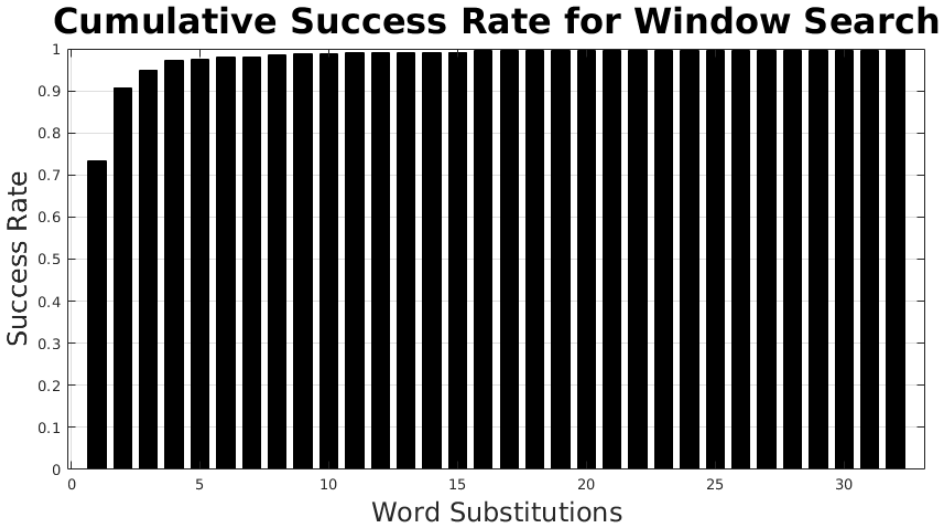


Figure 5.2: Cumulative success rate for window search vs. number of word substitutions required.

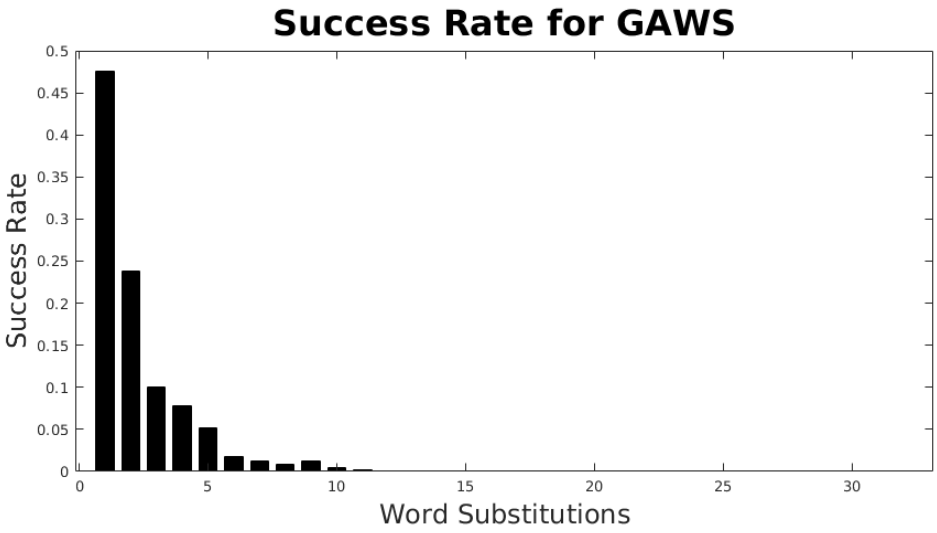


Figure 5.3: Success rate for window search vs. number of word substitutions required.

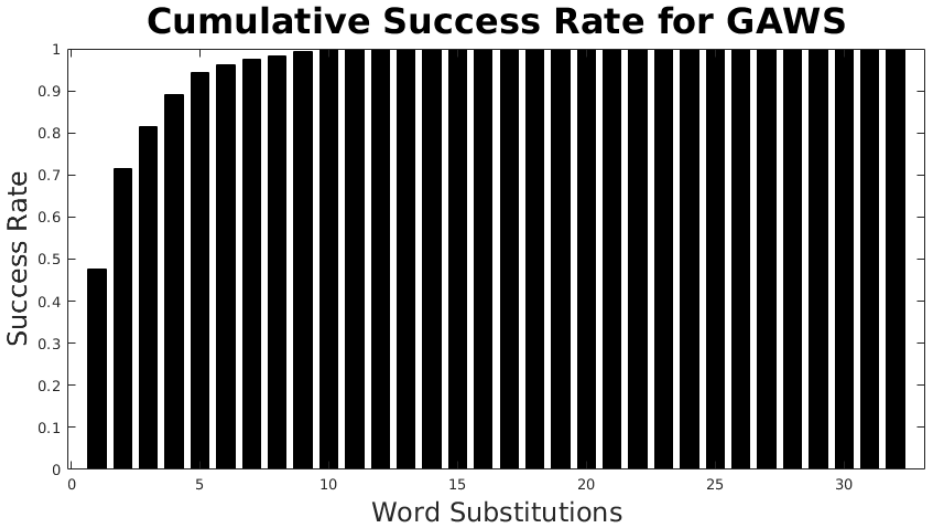


Figure 5.4: Cumulative success rate for window search vs. number of word substitutions required.

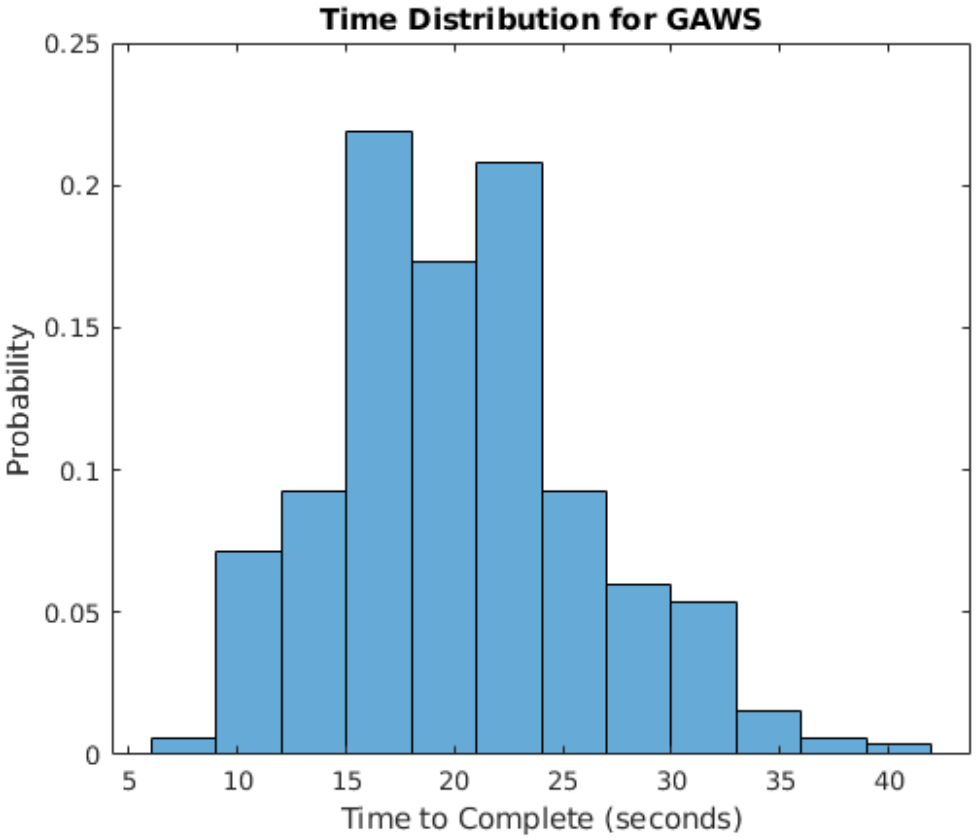


Figure 5.5: Distribution of time required for the GAWS algorithm

5.2 Gray Box

General structure of model known including hyperparameters algorithm run on stand-in model trained on same data, but not same model.

5.3 Black Box

Here we assume that no details of the model are known, in particular the hyperparameters are uknown. In order to evaluate performace 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.

Chapter 6

Conclusion

We implemented several new strategies for text based adversarial derivation, particularly targeting recurrent neural networks. The strategies we employed offered significant improvement over previous methods in terms of average number of word substitutions. The various optimizations we used offered a significant decrease in time spent per text sample, especially in larger cases, but ultimately degraded performance and caused failure for a significant amount of samples. Our methods show some amount of transferability, still working across models with different hyperparameters, but at significantly degraded performance.

6.1 Future Work

- Increased Complexity

  These algorithms should be tested in producing similar results with more complex neural networks since we tested only single layer RNN's. 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 recomendations include adding more layers, making the layers larger, and changing the word embedding dimension.

- Improving Semantic Similarity

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.

- Testing on Larger Datasets

  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 frequenct 10,000.

- Testing Across Different Word Embeddings

  In our testing we 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.

Bibliography

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

[2] 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

[3] 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

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

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

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

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

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

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

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

[11] 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

[12] 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.

[13] 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.

[14] 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.

[15] 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

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