Evaluating the Effects of Different Optimization Techniques on Word Embedding Performance

Joseph Bongo

12/08/2019

Introduction

Semantic Word Embeddings

Semantic word embeddings have revolutionized the field of natural language processing, allowing us to far more accurately represent text data in a quantitative way that can be used for a variety of modeling applications. Earlier methods for embedding text into numerical representations involved simply one hot encoding each word in a given corpus. For example, suppose we have a corpus that is a single document reading, "Hello, my friend". In this case, the embedding corresponding to the word "friend" would be the vector $[0, 0, 1]$. While this is can be somewhat useful, it does not consider the meaning of the actual word, but only the position where it occurs in a given corpus. The dimension of this vector is also completely dependent on the size of the vocabulary used in the corpus. The field of natural language processing was then revolutionized in 2013, Google published a paper about algorithms that their research scientists had developed called Word2Vec, which used transfer learning to produce embeddings related to the meaning of the word that perform much better in modeling applications than the traditional one hot encoded word vectors [Mikolov, Tomas, et al.]. One could also specify the size of their desired word vectors, meaning that Word2Vec also provides very useful dimensionality reduction for large bodies of text. Today, these embeddings are used in a variety of natural language processing applications such as recommendation engines, search, and chat bots. The model behind these embeddings is just a simple one hidden layer neural network. We will examine if the optimizer used in this neural network affects the quality of these embeddings, and if so, then which optimizer is most apt.

Word2Vec CBOW model

Word2Vec's Continuous Bag of Words model assumes that looking at the words that are used within the same context as a given word gives an idea about the word's semantics (or meaning) [Mikolov, Tomas, et al.]. Therefore, this allows the meaning of a word to be represented in a quantitative way more easily. Word2Vec can generate these embeddings by feeding the one-hot encoded vectors for the context words around a target word that we are trying to encode [Mikolov, Tomas, et al.]. We will call this word w_t. This context, which is typically the vectors for the two previous words and two words that occur next $(w_{t-2}, w_{t-1}, w_{t+1}, w_{t+2})$, is used as the input of a shallow neural network to classify what the target word is [Mikolov, Tomas, et al.]. While this is not particularly useful on its own, this is a form of transfer learning as the values of the hidden layer nodes of the neural network are now a representation of the different contexts of the output word w_t. The paper does not go into too much detail about how the model is trained, but the weights of neural networks are often optimized using gradient descent with a fixed learning rate. My original thought was that some of the algorithms that we have learned in S&DS 630 can do better, and while this is likely true, it became apparent very quickly that the reason that these simpler optimization algorithms are used is because they are much more computationally efficient. Nonetheless we have created the model above with four different optimizers. These optimizers are gradient descent (learning rate $\frac{1}{10t}$), gradient descent with a fixed learning rate, gradient descent using backtracking line search, and Adam.

Project, Algorithms, and Evaluation

The goal of this analysis is to see if different optimization techniques of the neural network weights and biases can produce better word embeddings, or if it perhaps does not matter which method is used and the weights converge/do not converge to the optimal values regardless. Evaluating the performance of word embeddings can be slightly subjective, but we have done this in two ways. The first way that we did this was to take the five other embeddings that have the highest cosine similarity to a given word and confirm that the words are synonyms. Comparing the cosine similarity to determine whether a particular pair of words to see how semantically related they are is a very common use of the word embeddings produced by this model. The second way to assess the quality of the word embeddings is to evaluate the accuracy of a simple classification task where the input is the embedding only. For this, a K-Nearest neighbors classifier was used to determine whether a given word is positive or negative. This type of task is referred to as sentiment analysis, and this is also a common application of Word2Vec [Mikolov, Tomas, et al.].

The algorithms will be evaluated not only on the performance of their corresponding embeddings, but also on factors such as total run time. Each optimization method will be used in the exact same type of neural network and will make the same number of passes through the dataset. We will also explore the computational efficiency and practicality of these four algorithms along with two others that could not be implemented due these types of problems. Those algorithms are Gradient Descent with exact line search as Newton's method. While this is a very applied project, going through some of the theoretical tenants of these algorithms will be very useful in determining which optimizer is the best to use for training Word2Vec.

Optimization Problem

Set-Up

Before we state the problem, we have to set up many of the variables that our model uses. Suppose we have a vocabulary V that is the set of all words in the corpus such that $|V| = v$ and a batch of size = x of inputs, we produce a $x \times v$ matrix X, whose rows are the sums of the one hot encoded input values of words $w_{t-2}, w_{t-1}, w_{t+1}, w_{t+2}$. We also define another X h of hidden layer values for a hidden layer of n nodes (i.e. h is a $x \times n$ matrix). Let ω_1 be the $v \times n$ dimensional matrix for the weights from the input layer X to the hidden layer h and let ω_2 be the $n \times v$ dimensional matrix for the weights from the hidden layer h to the softmax layer θ. Also, let b_2 be a $1 \times v$ vector of biases to be added to f which is then fed to the softmax function σ to produce output probabilities p. Additionally let b_1 be a $1 \times n$ vector of biases to be added to h .Therefore, for element-wise activation functions a_1, σ, we have $p = \sigma(f) = \sigma(h\omega_2 + b_2)$ and we also have $h = a_1(X\omega_1 + b_1)$ Typically for all sorts of language models, a_1 is simply the identity function meaning $h = X\omega_1 + b_1$. The output layer must be activated using the softmax function, as this produced the row vectors of probabilities that are the basis of neural network classification.

Problem Statment

We know that since the neural network will be a classifier, the loss function that we would like to minimize is the cross-entropy loss. Therefore, this is the objective function of our optimization problem and is stated below. It is important to reiterate that w_t is the one hot encoded vector of the target word, meaning that it is the indicator of correct classification. Therefore if we have a matrix $x \times v$ Y where each row corresponded the correct classification indicator for that given row of input p be the $x \times v$ vector of the output layer of our classification network, we have:

$$f_0 = -(1/v)\sum_{i=1}^{v} Y_i log(p_i) = -(1/v)\sum_{i=1}^{v} (Y \circ log(p))_i$$

We know that p is a function of $\omega_1, \omega_2, b_1, b_2$ and therefore we will rewrite it as $p(\omega_1, \omega_2, b_1, b_2)$. This gives us

$$f_0(\omega_1, \omega_2, b_1, b_2) = -(1/v)\sum_{i=1}^{v}(Y \circ log(p(\omega_1, \omega_2, b_1, b_2)))$$

Since p is the output of the sofmax function and intuitively p represents a matrix of row vectors of probabilities, it will by default have the following constraints.

$$\theta_i \leq 1, \theta_i \geq 0, ||\theta||_1 = 1$$

That being said, this does not mean that there is any contraint placed on the actual weights or biases. Therefore, our optimization problem is the unconstrained problem below:

minimize: $f_0(\omega_1, \omega_2, b_1, b_2) = -(1/v)\sum_{i=1}^{v}(Y \circ log(\theta(\omega_1, \omega_2, b_1, b_2)))$

Neural Network Set Up

It was important that the network is given a large corpus so that the words in question occur in as many contexts as possible, but too large of a corpus will lead to memory errors. Input data of too many words will also lead to very computationally expensive calculations, but also produce higher performing embeddings. After pondering what size of corpus to use, the models were trained on the first 50000 words of the Brown corpus, which was collected by faculty of Brown university in 1961 and contained news articles from Chicago Tribune: Society Reportage. I decided on the smaller side for a corpus because when evaluating the optimization techniques, the corpus need not be extremely large as all of the embeddings will be generated from that same corpus, meaning that all of the performance metrics will be relative. After some data cleaning was done to normalize the text (removing numbers, punctuation, etc.), the corpus had a final vocabulary of 7636 unique words. This text was processed in the input of our neural network via functions which I wrote in Python.

In order to configure the loss function and perform the actual optimization, I have written the code from scratch in Python that defines a Word2Vec class that trains the neural networks described in this section and creates the embeddings. This enabled full control over the architecture of the neural network as well as implementing any of the desired optimizers. Since our model input is a large amount of data (although maybe not for producing word embeddings), the networks were all trained using individual batches of the data of size 100 (i.e. stochastic gradient descent). Stochastic gradient descent is often used to train neural networks in practice and works by taking a subset (batch) of the total data and using the gradients relative to the batch to approximate the gradient of the entire data set. This introduces quite a bit of random error into the optimization, which impact the effectiveness of certain optimization techniques more than others.

Therefore, since our vocabulary is 7636 words, our neural networks all have 7636 input nodes, 50 hidden nodes, and 7636 output nodes. Since the network is fully connected this means the dimensions of our parameters $W_1, W_2, b_1 b_2$ are 7636×50, 50×7636, 1×50, and 1×7636 respectively. Typically in these types of large language models, the model actually only makes 1 or 2 passes through the data. Since our corpus is smaller than those typically used for these types of problems, I decided that 3 passes through the data was appropriate. Therefore, since our input data contains 43409 contexts and batch size is 100, it takes 435 iterations to pass through the data one time and therefore our model uses 1305 iterations. After the model is trained, the one-hot vector for each word makes a forward pass to the hidden layer of the model, giving us our embeddings.

Gradient Descent

When searching for the direction to go to minimize a particular loss function, the negative gradient is the most logical choice.[Boyd, Stephen, and Lieven Vandenberghe.] This is exploited with gradient descent, a

simple and computationally cheap yet effective optimization technique. We will go over four types of gradient descent, three of which were used in the neural networks produces. First, in order to understand gradient descent, the algorithm is as follows [Boyd, Stephen, and Lieven Vandenberghe.]:

Given a starting point $x \in domf_0$

for i in steps:

determine descent direction Δx (this is $-\nabla f_0(x)$)

perform line search to find the optimal learning rate t (if required)

update $x = x + t\Delta x$

For the first form of gradient descent that was used in the model, we used a fixed learning rate of $t = .01$, meaning that no line search was done, which saves the algorithm from much computational rigor. For our second form of gradient descent, we are using a learning rate of $t = \frac{1}{i}$ where i is the current iteration number. This also does not require any form of line search and updates the parameters by a smaller margin for each iteration, decreasing the size of the steps as we assume (hopefully) the algorithm approaches the optimal value. This assumption makes perfect sense when we consider all input data, however since we are implementing the gradient descent algorithm stochastically, it may not always be the case due to the randomness of the data that is actually considered at each iteration. The next two gradient descent methods that we will consider both use line search to determine the optimal learning rate. While this is more computationally rigorous, in theory it should cause the values to converge to the optimal values more quickly. The first of these two uses backtracking line search, which we were able to incorporate into the neural network. The backtracking line search algorithm uses two randomly initialized parameters $\alpha \in (0, 0.5)$ and $\beta \in (0, 1)$ to determine the optimal value t.

Given a descent direction Δx for f_0 and $x \in domf_0$

$t = 1$

while $f_0(x + t\Delta x) > f_0(x) + \alpha t \nabla f_0(x)^\intercal \Delta x$:

$t = \beta t$

One thing that we observe is that

$$\nabla f_0(x)^\intercal \Delta x = \nabla f_0(x)^\intercal (-\nabla f_0(x)) = -(\nabla f_0(x)^\intercal \nabla f_0(x)) = -||\nabla f_0(x)||_2^2$$

Therfore, we can substitute in the Frobenius norm and we get the stopping criterion to be

$$f_0(x + t\Delta x) \leq f_0(x) - \alpha t ||\nabla f_0(x)||_2^2$$

This algorithm is a very good way to approximate the optimal t, but in order to get that value for sure, one would have to use our final gradient descent method, which is gradient descent with exact line search. In exact line search, we find $t = argmin_{s \geq 0} f_0(x + s\Delta x)$ In this case, we see that we must s such that:

$$\frac{\partial}{\partial s}(f_0(x + s\Delta x)) = 0$$

We observe that this simplifies to

$$\frac{\partial}{\partial s}(f_0(x + s\Delta x)) = f_0'(x + s\Delta x)\Delta x = 0$$

which will yield the optimal value t.

Applying Gradient Descent

In order to attempt to solve this optimization problem using gradient descent, we must calculate the gradient of the objective function relative to each of our parameters. Let X be a matrix of inputs and Y be the one-hot embeddings for the corresponding target words for each context in X. We have $h = XW_1 + b1$, $f = hW_2 + b_2$, and $p = \sigma(f)$. We know that $\sigma(f)_k = \frac{exp(f_k)}{\sum_j exp(f_j)}$. Therefore, it is clear that for $f_0 = -\frac{1}{v}\sum_i Y_i log(p_i)$. In order to derive the 4 necessary gradients we will have to use repeated applications of the chain rule. First we will find $\frac{\partial f_0}{\partial W_2}$ which by the chain rule is equal to $\frac{\partial f_0}{\partial p}\frac{\partial p}{\partial f}\frac{\partial f}{\partial W_2}$. We see that $\frac{\partial f_0}{\partial p} = -\frac{1}{v}\sum_i \frac{Y_i}{p_i}$. From the quotient rule, we see that since $p = \sigma(f)$ where σ is the softmax function, we have:

$$\frac{\partial p}{\partial f} = \frac{\partial}{\partial f}\left(\frac{exp(f_k)}{\sum_j exp(f_j)}\right) = \frac{exp(f_k)\sum_j exp(f_j) - exp(f_k)exp(f_k)}{(\sum_j exp(f_j))^2} = \frac{exp(f_k)}{\sum_j exp(f_j)}\frac{\sum_j exp(f_j) - exp(f_k)}{\sum_j exp(f_j)} = p(1-p)$$

Then we see that $\frac{\partial f}{\partial W_2} = h$ and $\frac{\partial f}{\partial W_2} = I$. Therefore, we have

$$\frac{\partial f_0}{\partial W_2} = Y(1-p)h$$

and

$$\frac{\partial f_0}{\partial b_2} = Y(1-p)$$

In order to find $\frac{\partial f_0}{\partial W_1}, \frac{\partial f_0}{\partial b_1}$, we employ the same tactic. Through similar repeated applications of the chain rule, we have

$$\frac{\partial f_0}{\partial W_1} = \frac{\partial f_0}{\partial p}\frac{\partial p}{\partial f}\frac{\partial f}{\partial h}\frac{\partial h}{\partial W_1}$$

and

$$\frac{\partial f_0}{\partial b_1} = \frac{\partial f_0}{\partial p}\frac{\partial p}{\partial f}\frac{\partial f}{\partial h}\frac{\partial h}{\partial b_1}$$

We see that $\frac{\partial f}{\partial h} = W_2^\intercal$, $\frac{\partial h}{\partial W_1} = X$, and $\frac{\partial h}{\partial b_1} = I$. Therefore we have,

$$\frac{\partial f_0}{\partial W_1} = Y(1-p)W_2^\intercal X$$

and

$$\frac{\partial f_0}{\partial b_1} = Y(1-p)W_2^\intercal$$

This is how the gradients are calculated for all of the optimization techniques that we will explore, including Adam and Newton's method. Another line search technique that is frequently taught is exact line search. This method involves computing $t = argmin_{s\geq 0} f_0(x + s\Delta x)$ directly, which is often not practical in many circumstances, including training neural networks. Exact line search is typically used in one variable optimization problem when the cost of finding the minimum above is low relative to the cost of finding the direction. That description is nearly the antithesis of this problem as we are optimizing 4 variables and the computation of the descent direction is relatively straight forward compared to finding $argmin_{s\geq 0} f_0(x+s\Delta x)$. Therefore, it does not make sense to perform an exact line search. In fact exact line search is very rarely used in practice and backtracking lines search is often used as a approximation [Boyd, Stephen, and Lieven Vandenberghe.].

Overall, gradient descent is a simple optimization technique that is often the choice of optimizing neural networks due to its computational efficiency and relatively high performance. If line search is done, this will cause the algorithm to be more computationally intense and therefore have a longer run time but should yield higher performance. This added complexity is why it is not often used for training neural networks.

Adam

I wanted to include an optimization technique that was outside the scope of what was covered in this course. I knew the algorithm Adam has been used often to optimize the parameters of neural networks (and even seems to even be preferred over gradient descent). Adam goes about the optimization be first assuming that all parameters of the neural network as random variables [Kingma, Diederik P., and Jimmy Ba.]. That said, it is also expected that the objective function has random noise associated with it. Adam can take this into account taking the moments of these variables [Kingma, Diederik P., and Jimmy Ba.]. We know that the first moment of a random variable $E[X^1] = E[X]$ corresponds to the mean, while the second moment of a random variable $E[X^2]$ corresponds to the uncentered variance of that random variable. Therefore, Adam will take the first and second moments of the variable and add them to a sliding average for the value of that moment with decay by factors β_1, β_2 respectively [Kingma, Diederik P., and Jimmy Ba.]. These values are then divided by $(1-\beta_1^t), (1-\beta_2^t)$ where t is the iteration number, to remove any bias of being initialized at 0 [Kingma, Diederik P., and Jimmy Ba.]. The parameters are then updated by adding the gradient times a learning rate and $m_t\sqrt{v_t+\epsilon}$ where $\epsilon > 0$. As stated above, this simple technique works very well with loss functions that are prone to random error (like cross-entropy), and work well with stochastic gradient descent as a result, making them very popular for optimizing neural networks. The Adam algorithm, in a sense, is selecting an adaptive learning rate using only the first order derivatives and cheap computations, unlike backtracking line search. The pseudocode for the algorithm is written below. [Kingma, Diederik P., and Jimmy Ba.]

$\beta_1, \beta_2 \in [0, 1)$

initialize $m_0, v_0 = 0$

for t in steps:

$m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla f(x_{t-1})$

$v_t = \beta_2 v_{t-1} + (1-\beta_2)(\nabla f(x_{t-1}))^2$

$\hat{m}_t = m_t/(1-\beta_1^t)$

$\hat{v_t} = v_t/(1-\beta_2^t)$

$x_t = x_{t-1} - step.size * \frac{\hat{m_t}}{\sqrt{\hat{v_t}+\epsilon}}$

Newton's Method

Newtons method is a fast-converging algorithm for optimizing a given objective function, however it is a second order method, one needs the Hessian of the objective function along with its gradient to compute the descent direction. This is a much more computationally expensive calculation, which is why Newton's method is not often used in training neural networks.

In order to motivate why Newton's method converges so quickly, we know from Taylor's Theorem that the second order approximation of a given objective function f_0 is

$$f_0(x+t) \approx f_0(x) + f_0'(x)t + .5f_0''(x)t^2$$

The goal of this process is to find $x+t$ that minimizes $f_0(x+t)$. We see that when this is found

$$0 = \frac{\partial}{\partial t}(f_0(x+t)) = \frac{\partial}{\partial t}(f_0(x) + f_0'(x)t + f_0''(x)t^2) = f_0'(x) + f_0''(x)t$$

Therefore, the optimal value of t is $t = -\frac{f_0'(x)}{f_0''(x)}$. Here lies the difficult and computationally expensive part. Often, the Hessian of the objective function is difficult to determine and expensive to compute. This is

certainly the case for neural networks, as repeated applications of the chain rule get rather unwieldy as the order of the derivative increases. There are many methods called "Quasi-Newton" which all have different way of approximating the Hessian, but the algorithms that do this are often computationally expensive as well [Dennis, Jr, John E., and Jorge J. Moré.]. Therefore, Newton's method was not implemented to optimize the neural network. Had I included Newtons methods into the neural network, it would have worked as follows from the statement of the Algorithm in Boyd Vanderberghe:

Given a starting point $x \in dom f_0$

for i in steps:

determine the newton step $\Delta x_{nt} = -\nabla^2 f_0(x)^{-1} \nabla f_0(x)$

perform line search to find the optimal learning rate t (if required, this likely would not have been done if incorporated into the neural network)

update $x = x + \Delta x_{nt}$

Advantages and Disadvantages

When it comes to optimizing the parameters of a neural network, certain algorithms do a much better job than others. Newton's method and any method that uses exact line search seem to be completely impractical for this task due to their computational intensity, as well as their difficulty in evaluating the Hessian of the objective function and the optimal learning rate respectively. That being said, backtracking line search appears to also be a time intensive process when compared to simply using a learning rate that is fixed. Of the 4 models that I ran, gradient descent with backtracking line search took by far the longest with a total run time of 2511 seconds. Considering that all models were implemented stochastically and no other model took more than 2.5 minutes to train, this is exceptionally longer. This does allow us to approximate the optimal learning rate t, but it seems that this increase in precision is likely not worth the considerable increase in run time and complexity. That being said, I would also hypothesize the using stochastic gradient descent essentially kills the need of doing any sort of line search, as the optimal value of t is supposed to be calculated using the entire dataset, and not simply a batch because the gradient of a given batch is not necessarily the proper descent direction. Perhaps this will become apparent when the embeddings are evaluated, or it is possible the the gradient of the batch is a good enough approximation of the descent direction and line search does in fact help the model converge. I also felt the need to cap the number of iterations allowed in the backtracking line search. In theory, this is not something I would have done as I would want to get as close to the optimal learing rate as possible. However, common worries in Python are overflow error which is caused by numbers with too large of an absolute value, and underflow error which is caused by numbers that have an absolute value that is too close to (but not equal to 0). Since backtracking line search involves the multiplication of $t \leq 1$ by $\beta < 1$, this occasionally lead to underflow errors when the backtracking line search algorithm was left with no cap on the number of iterations.

The most exciting of the four optimizers used was definitely Adam. Adam took the next longest to run with a total run time of 143.87 seconds. This is likely because calculating the adaptive learning rate is a slightly more involved calculation than simply using a fixed learning rate or a learning rate that is a function of the iteration number. Regardless, this is clearly a very computationally efficient algorithm. Of the 4 algorithms, Adam showed the largest and most consistent decrease in loss over each iteration. Although this is not the best measure to use as implementing stochastic gradient descent causes much more randomness to be associated with the loss, it gives us the general idea that Adam might have performed the best optimization of all 4 optimization techniques. That said, the way that Adam probabilistically treats the parameters as random variables plays very well with our particular modeling application and loss function. While words that appear in the same context are often related, there is also going to be a lot of noise from unrelated words that appear near it for any number of reasons, or words that are incredibly common in any context (articles, pronouns, etc.) When one compounds this with the fact that each iteration of the neural network is only considering at most 100 observations, it seems like Adam is the most logical technique to use for optimizing these hyperparameters We will explore this further by evaluating the embeddings.

Finally the models that used gradient descent and gradient descent with a fixed learning rate trained in 120.96 and 122.57 seconds respectively. Again, this is quicker than Adam, likely due to nothaving to calculate an adaptive learning rate. The fact that not line search algorithm was implemented in either of these cases directly contributed to their remarkable efficiency. Each of these two techniques has a particular advantage. The advantage of gradient descent with a fixed learning rate is the fact that it is incredible interpretable and predictable, while still doing a good job of optimization. It is also good for stochastic gradient descent, because while a learning rate based on the iteration number causes certain parts of the dataset to not be considered as much in the descent direction calculation as others (unless otherwise specified), this never changes when the learning rate is fixed. That being said, the use of a learning rate that is proportional to the iteration also has its advantages. Specifically, this causes the change in the parameters to shrink as the number of observations increases. This is key as the values change less and less as an optimal value is approached, helping the algorithm converge. It is a common problem with fixed learning rate gradient descent that the learning rate given is too large and thus the algorithm diverges from optimal. In fact, this was occurring when I trained the neural network with a fixed learning rate of 1. I then changed this to a learning rate of .01 and then the algorithm began behaving more appropriately, implying that it indeed converges.

Evaluating The Embeddings

At the conclusion of the training of the models, four sets of embeddings were produced. The first was the set of those embedding as that were optimized with using gradient descent with a fixed learning rate of $t = .01$. The second set of embeddings consisted of those generated by the model that was optimized using gradient descent with a learning rate of $t = \frac{1}{10i}$, while the third consisted of the embeddings produced by the model that was optimized using gradient descent with backtracking line search. The final set of embeddings were those from the Word2Vec model that was optimized using Adam, with the recommended parameters of $\beta_1 = .9$, $\beta_2 = .999$, $\epsilon = 10^{-8}$. As described previously, the embeddings were each put through two experiments so that their quality could be evaluated, albeit somewhat subjectively.

Cosine Similarity of Synonyms

For the first evaluation task, the following 8 test words were chosen at random:

"bath", "weather", "offensive", "ambassador", "divorced", "innocent", "suburban", "grand"

The embeddings of these words then had the cosine similarity of them taken with the rest of the words in the vocabulary. It is common of high performing word embeddings that the embeddings of synonyms will often have a very high cosine similarity. Recall the the cosine similarity of two vectors ranges from 1 to -1 and is calculated below

$$sim(a, b) = \frac{a \cdot b}{||a|| * ||b||} = \frac{a^\intercal b}{||a|| * ||b||}$$

Below we have a table which shows for each embedding set, what the 5 words that have the highest cosine similarity to each of the test words are

Method	Word	1	2	3	4	5
Gradient Desent	bath	('cd', 0.769)	('survive', 0.737)	('tarzan', 0.734)	('stages', 0.731)	('immediate', 0.729)
Gradient Desent	weather	('unconstitutional', 0.788)	('pleads', 0.775)	('admitted', 0.767)	('rundown', 0.762)	('convey', 0.76)
Gradient Desent	offensive	('excise', 0.761)	('laos', 0.758)	('concentration', 0.757)	('brigantine', 0.743)	('itll', 0.739)
Gradient Desent	ambassador	('julian', 0.768)	('pennock', 0.759)	('smoky', 0.757)	('lived', 0.75)	('abandonment', 0.747)
Gradient Desent	divorced	('walkways', 0.799)	('trained', 0.795)	('totaled', 0.785)	('forthcoming', 0.76)	('ralph', 0.755)
Gradient Desent	innocent	('wolcott', 0.807)	('origin', 0.786)	('succeeded', 0.761)	('concluded', 0.749)	('fortin', 0.748)
Gradient Desent	suburban	('cambridge', 0.782)	('knights', 0.775)	('address', 0.773)	('richardson', 0.772)	('dough', 0.772)
Gradient Desent	grand	('screw', 0.814)	('bankruptcy', 0.808)	('smiths', 0.8)	('sirens', 0.794)	('federal', 0.785)
Gradient Desent - Fixed	bath	('bargaining', 0.906)	('ground', 0.892)	('singing', 0.888)	('knecht', 0.888)	('simonelli', 0.887)
Gradient Desent - Fixed	weather	('maids', 0.895)	('denverarea', 0.875)	('receive', 0.866)	('swept', 0.861)	('bailey', 0.858)
Gradient Desent - Fixed	offensive	('internal', 0.872)	('around', 0.861)	('elementary', 0.853)	('dancing', 0.851)	('fur', 0.85)
Gradient Desent - Fixed	ambassador	('ailing', 0.909)	('order', 0.906)	('lyon', 0.894)	('allocated', 0.893)	('lawmakers', 0.893)
Gradient Desent - Fixed	divorced	('struggle', 0.864)	('diem', 0.854)	('household', 0.848)	('decisions', 0.847)	('marcile', 0.846)
Gradient Desent - Fixed	innocent	('coles', 0.885)	('park', 0.884)	('threshold', 0.884)	('broken', 0.883)	('thing', 0.882)
Gradient Desent - Fixed	suburban	('animal', 0.896)	('help', 0.881)	('ditmar', 0.87)	('ideas', 0.868)	('liner', 0.867)
Gradient Desent - Fixed	grand	('caught', 0.91)	('considerably', 0.897)	('maid', 0.897)	('widow', 0.894)	('livelihood', 0.894)
Gradient Desent - Line Search	bath	('burden', 1.0)	('allergic', 1.0)	('husky', 1.0)	('riverside', 1.0)	('guest', 1.0)
Gradient Desent - Line Search	weather	('missionary', 1.0)	('killing', 1.0)	('disappointments', 1.0)	('odds', 1.0)	('costumes', 1.0)
Gradient Desent - Line Search	offensive	('patti', 1.0)	('newest', 1.0)	('payment', 1.0)	('people', 1.0)	('purdues', 1.0)
Gradient Desent - Line Search	ambassador	('commuted', 1.0)	('bright', 1.0)	('recorded', 1.0)	('shreveport', 1.0)	('pole', 1.0)
Gradient Desent - Line Search	divorced	('lynn', 1.0)	('dynamo', 1.0)	('hastened', 1.0)	('quoted', 1.0)	('britains', 1.0)
Gradient Desent - Line Search	innocent	('commuted', 1.0)	('games', 1.0)	('somebody', 1.0)	('supply', 1.0)	('mean', 1.0)
Gradient Desent - Line Search	suburban	('load', 1.0)	('lawmakers', 1.0)	('stolen', 1.0)	('corporations', 1.0)	('debut', 1.0)
Gradient Desent - Line Search	grand	('outstanding', 1.0)	('same', 1.0)	('attempted', 1.0)	('represents', 1.0)	('mickey', 1.0)
Adam	bath	('violations', 0.481)	('kubek', 0.475)	('impinging', 0.462)	('chemistry', 0.442)	('weight', 0.439)
Adam	weather	('jury', 0.61)	('starts', 0.609)	('stimulatory', 0.603)	('halfback', 0.599)	('member', 0.599)
Adam	offensive	('sold', 0.541)	('stayed', 0.53)	('took', 0.517)	('lloyd', 0.499)	('handed', 0.498)
Adam	ambassador	('aplate', 0.537)	('roles', 0.533)	('theyve', 0.492)	('tremendous', 0.486)	('throneberry', 0.48)
Adam	divorced	('just', 0.65)	('about', 0.607)	('station', 0.603)	('district', 0.594)	('law', 0.591)
Adam	innocent	('critical', 0.59)	('soninlaw', 0.585)	('en', 0.564)	('decision', 0.56)	('paid', 0.547)
Adam	suburban	('criticized', 0.549)	('ball', 0.548)	('existence', 0.538)	('controversy', 0.532)	('become', 0.53)
Adam	grand	('last', 0.742)	('before', 0.733)	('first', 0.713)	('so', 0.71)	('in', 0.709)

What we observe above is rather disappointing as the supposed "synonyms" of the 8 test words appear to be nearly non-sensical. Looking further, we observe that the ranges for the cosine similarity of the most similar word varies wildly among the four types of embeddings. We see the cosine similarity of the Adam embeddings are quite low (which one would expect because these words don't appear to be related), while every embedding listed for gradient descent with line search has a cosine similarity of nearly 1 (the results were rounded to fit on the table). This is interesting as it would appear the similarity of the embeddings different greatly depending on which optimizer was used. There are certain combinations of terms that make sense, such as "divorced" and "law" for Adam and "divorce" and "household", "struggle" for fixed learning rate gradient descent. That being said, these combinations are far and few between and could possibly be even random. Therefore, it seems that for this experiment none of the embedding performed very well at all. This is very likely due to the size of our corpus rather than any optimization techniques that were performed. It would seem to me that many words simply did not end up inside of the context window of other related words frequently enough throughout the corpus. This would cause these similarities to be a bit confusing. For example, in the case of Adam, "grand" could be closest associated with "last" because there as a news article in the corpus about a "grand finale" of some kind rather than usage of the term grand to mean large or great. In the future, it seems that it is important that this is trained on a larger, more diverse corpus.

Predictive Power in Simple Classifier

For this problem, the sklearn Python module was used to create a K-Nearest Neighbors classifier ($k = 6$) to classify words as "positive" or "negative". A data set of 60 words was provided in which 31 on the words were positive and 29 were negative. for each set of embeddings the K-Nearest Neighbors model was trained on the same 36 word embeddings and was then tested on the remaining 24 word embeddings to determine whether they were positive or negative. The table below shows the accuracy of each of the embeddings.

Method	Accuracy
Gradient Descent	0.5833333333333334
Gradient Descent - Fixed	0.3333333333333333
Gradient Descent - Line Search	0.5
Adam	0.6666666666666666

We see that again the results are underwhelming. That being said when we look the performance relative to the other embeddings, those from the model that was optimized using Adam performed the best, followed by the gradient descent embeddings, the embeddings optimized by gradient descent with line search, and the embeddings produced by gradient descent with a fixed learning rate. This is interesting as it looks like these results could simply be due to randomness. That said, the fact that there were only 24 test cases does not tell us a lot, but it seems that gradient descent with a fixed learning rate performed a good deal worse than any of the other optimizers. This is surprising because this was likely how these models were trained before Adam became popular. The Adam embeddings performing the best is not surprising due to the fact that our corpus is small, meaning there is more random error and that the models were trained stochastically. If I had better computing resources, I would be interested to see how these embedding perform relative to each other with a larger, more diverse corpus, larger batch sizes, and more iterations through the data.

Conclusions

Overall this was a wonderful and informative exercise relating optimization to machine learning and natural language processing. Unfortunately, the analysis was slightly handicapped by the lack of computational resources available. That said, this gives a glimpse to the real problem that people often have when performing applied optimization problems, and that is not to minimize loss necessarily. It seems like the far more important issue is to comes as closes as possible to minimizing loss given limited computing resources. That being said, Adam's resilience to random error relative to the other three methods make it the best optimization technique for this Word2Vec problem and explains why it performed the best in the classifier. Adam use of low order moments to calculate the proper adaptive leaning rate (somewhat like momentum in physics), is truly a brilliant addition to an algorithm that otherwise is very similar to gradient descent. If one were to use gradient descent for this problem, the best way to implement it would be giving is a learning rate that shrinks as the number of iterations increase, so as to not disturb the algorithms natural convergence. This is exactly how I was always taught gradient descent, but it seems like many natural language processing applications use a fixed learning rate. Using backtracking line search would not be appropriate when training a neural network stochastically as is increases the computational rigor significantly and does not end much value when the gradient of the batch is used as an approximation as the gradient of the entire dataset.

Had I been given unlimited time and computational resources. I would have assembled a massive, diverse corpus, much like the Google team did when creating Word2Vec [Mikolov, Tomas, et al.]. I also would have tried to use a Quasi-Newton method to see how that would perform versus the other methods employed. I

also would have made sure that the classification experiment had plenty of labeled data point to indicated which optimization technique is superior with a high degree of certainty. It's unclear which technique would have been superior in that case, as unlimited computational resources eliminate the need for stochastics gradient descent, but given that this scenario is completely unrealistic, the behavior of the Adam algorithm when used in stochastic gradient descent, makes it the clear choice for this type of problem.

References

[Boyd, Stephen, and Lieven Vandenberghe.] Convex optimization. Cambridge university press, 2004.

[Dennis, Jr, John E., and Jorge J. Moré.] "Quasi-Newton methods, motivation and theory." SIAM review 19.1 (1977): 46-89.

[Kingma, Diederik P., and Jimmy Ba.] "Adam: A method for stochastic optimization." arXiv preprint arXiv:1412.6980 (2014).

[Mikolov, Tomas, et al.] "Distributed representations of words and phrases and their compositionality." Advances in neural information processing systems. 2013.