Gradient Methods for Training Neural Networks on Imbalanced Label Distributions

Felix Wang
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign

Abstract—This project explores some of the different methods for training a neural network with the objective of having the network learn the game Set. In particular, we will be concentrating on gradient methods such as gradient descent, both stochastic and batch, and conjugate gradient. Effectively, the project will be an extension of the labs where we primarily used the backpropagation algorithm along with the standard gradient descent in order to train a neural network. In addition to the trade-offs mentioned in lecture among these gradient methods, such as computational complexity and convergence rates, we will also be exploring the robustness of the methods with respect to imbalanced label distributions. Put another way, we wish to explore how well the different training methods perform on data containing rare events, if it can learn the correct mapping at all.

I. INTRODUCTION

MULTILAYER neural networks, as a generalization of the single layer linear classifier or perceptron model, are a popular tool for learning nonlinear mappings between input data and output classes. As a function approximator, neural networks are incredibly powerful in their representation and are capable of sufficiently approximating any function with bounded support to a chosen level of accuracy. It does this by introducing additional, hidden layers between the input and output of a simple linear classifier enabling subsequent layers to learn linear discriminants on top of outputs of the previous layer. In terms of training, the nonlinear mappings across the hidden layers is learned at the same time as the linear discriminant in the overall [1].

A popular method for doing this is the backpropagation algorithm, or generalized delta rule, where error at the output layer is propagated back through to the input layers, enabling all of the weights in the neural network to be updated simultaneously. The standard method for updating the weights is to modify them in the direction of the negative gradient of the error in an iterative way, hence the name gradient descent. While the standard gradient descent algorithm is fundamental to understanding the nature of the learning mechanism, there are many modifications on top of it that provide better performance with respect to various optimality criterion such as computational complexity per iteration, number of iterations until convergence, overall convergence times, and accuracy of mapping. We will be studying the different gradient training methods under these criterion. Because the particular dataset used in this project, Set, has positive examples as rare events, we will also study how well the methods are able to learn the mapping under this constraint in the first place.

II. DATASET

The dataset we will be using in this project is drawn from the card game Set, developed by Set Enterprises, Inc. [2]. The objective of the game is to identify ‘sets’ of three cards in which each individual feature is either all the same or all different. There are a total of four feature dimensions with three different values per feature: number (one, two, or three), color (red, green, or purple), shape (diamond, oval, or squiggle), and fill (open, striped, or solid). This gives a total of $3^4 = 81$ different possible cards in a Set deck. An example of a set is shown in figure 1.

Fig. 1. A ‘set’ of cards with number, color, shape, and fill all different

Generally, a game of set is played by setting down cards, with a total of twelve in play at a time, and removing sets until the deck has run out. Because the goal of training the neural network is to learn the particular rules governing the classification of a ‘set’ or not, we will only be focusing on examples of three cards at a time. Due to its construction, given any two cards, there is a unique other card that forms a ‘set’ with it. There are 1080 sets out of a total of 85320 possible combinations of three cards. While the rules are fairly straightforward (all same or all different), the classification of a ‘set’ is comparable to the XOR function where the decision boundaries are inherently nonlinear with respect to the feature space. Among other things, this gives a target function that is better suited for neural networks over the various linear classifiers available. One major benefit to knowing the rule beforehand that provides the label classification is the ability to automatically generate the dataset, both for training and for testing. Furthermore, because the features take on discrete values, the preprocessing step of converting a data point to feed into the network is trivialized. In terms of experimental setup, this simplicity allows for better highlighting of the differences in behavior among the training methods.
III. METHODS

As far as methods for training a neural network go, there are many. While many of these are modifications on the standard, first-order gradient descent algorithm, second-order methods are also widely in use due to their effectiveness in evaluating the error surface. With respect to the former, modifications include the batch size (e.g. stochastic gradient descent and mini-batch) and the addition of momentum terms. For the latter, we mostly see approximations of Newton’s method in the form of conjugate gradient and modifications on top of that (e.g. BFGS and L-BFGS for limited memory). The three main gradient methods that were studied in this project were gradient descent (GD), stochastic gradient descent (SGD), and conjugate gradient (CG). An overview of these methods is given below.

A. Backpropagation

Before moving on to the gradient descent algorithm, a brief overview of the backpropagation algorithm, from which we obtain the gradient of the error, is instructive. In a multilayer neural network, the error is computed with respect to a chosen loss function, is the mean square error loss, MSE, \( w \) is a vector of the network weights, \( \text{E} \) is the loss per training example. A common loss function is the mean square error loss, MSE, \( e(i) = \frac{1}{2} (t(i) - y(i))^2 \), where \( y \) is the output of the network and \( t \) is the target label. \( y \) for an \( L \)-layer network is computed via forward propagation starting from the input nodes \( x_0 \):

\[
\begin{align*}
a^{(i)}_{l+1} & = b_{l+1} + w_{l+1} x_l^{(i)} \\
x_{l+1} & = g_{l+1}(a^{(i)}_{l+1}) \\
& \vdots \\
a^{(i)}_L & = b_L + w_L x_{L-1}^{(i)} \\
y^{(i)} & = g_L(a^{(i)}_L)
\end{align*}
\]

where \( w \) are the network weights between layers \( l-1 \) and \( l \), \( b \) is the bias term, and \( g \) is the activation function at layer \( l \). Here, \( w = \{w_1, \ldots, w_L, b_1, \ldots, b_L \} \).

Backpropagation of the error with respect to the weights and biases of any given layer is performed via the chain rule to step back through the layers from the output. This is aided through the introduction of intermediate values given by:

\[
\begin{align*}
\delta_L^{(i)} & = g_L'(a^{(i)}_L) \cdot \nabla_y e(i) \\
& \vdots \\
\delta_l^{(i)} & = g_l'(a^{(i)}_l) \times w_l^T \delta_{l+1}^{(i)}
\end{align*}
\]

where the derivative of the activation function \( g_l' \) is with respect to \( a_l \), and the operation \( \times \) entails piece-wise multiplication. The gradient of the error with respect to the network weights is then found as:

\[
\nabla_w E = \sum_{i=1}^{n} \delta_l^{(i)} x_i^{T} \\
\nabla_b E = \sum_{i=1}^{n} \delta_l^{(i)}
\]

B. Gradient Descent

In gradient descent, or steepest descent, we start with initial guess for the weight vector, \( w \), and iteratively update it along the direction of the greatest decrease of the error [3]. The standard version of GD is batch, where the gradient is computed with respect to the entirety of the training set via the backpropagation equations in the previous section. Here, the choice of learning rate, \( \eta \), is important in the algorithm’s convergence. If the learning rate is too small, the algorithm will take too long, from a practical standpoint, to converge; if the learning rate is too large, the algorithm may never converge due to overshooting oscillations. Although the optimal value of \( \eta \) may change during the course of GD, a constant \( \eta \) is typically used in practice. With respect to computational complexity, GD is on the order \( O(n) \) per iteration, where \( n \) is the size of the weight vector. The total time complexity of GD (until convergence) is on the order of \( O(nk) \) where \( k \) is the conditioning of the error surface, generally \( O(n^{2/3}) \). This gives complexity of \( O(n^2) \) for two-dimensional problems and \( O(n^{5/2}) \) for three-dimensional problems [4]. The GD algorithm is given below.

1. Initialize weight vector \( w_1 = [w_1, \ldots, w_L, b_1, \ldots, b_L] \)
2. Evaluate the gradient vector \( g = \nabla_w E \)
3. At iteration \( j \), evolve weights along negative gradient according to learning rate \( \eta \) (i.e. \( w_{j+1} = w_j - \eta g_j \))
4. Test to see if the stopping criterion is satisfied (e.g. \( ||\nabla_w E|| < \theta \) ), and return \( w \) if the case
5. Else, set \( j = j + 1 \) and go to 2.

C. Stochastic Gradient Descent

Whereas the standard batch version of GD computes the error over the entire training set, stochastic gradient descent evolves the networks weights according to the error gradient computed from a single training example at a time. Mini-batch takes a middle ground, where it evaluates on a substantial fraction of the training examples per iteration, but more than just one. Because of the reduced number of training examples needed, SGD and mini-batch are much faster per iteration. This manifests as a greatly reduced constant term in the computational complexity equations for GD even though the general trend with respect to the number of weights remain the same. One other computational benefit to SGD, and why it is sometimes referred to as an online method, is that there is no need to store the training examples throughout the duration of training as they may be streamed one by one to the learner, computed on, and then discarded. The SGD algorithm is given below.

1. Initialize weight vector \( w_1 = [w_1, \ldots, w_L, b_1, \ldots, b_L] \)
2. Select a data point \( x^{(i)} \), randomly chosen
3. Evaluate the gradient vector \( g = \nabla_w E^{(i)} \)
4. At iteration \( j \), evolve weights along negative gradient according to learning rate \( \eta \) (i.e. \( w_{j+1} = w_j - \eta g_j \))
5. Test to see if the stopping criterion is satisfied (e.g. \( ||\nabla_w E|| < \theta \) ), and return \( w \) if the case
6. Else, set \( j = j + 1 \) and go to 2.
D. Momentum

While oscillations along the error surface may show up in GD for too large of a learning rate, it can be even more pronounced in SGD where the gradient step is performed with respect to a single training example. One simple modification to these algorithms that helps to smooth the minimization is through the addition of a momentum term, \( \mu \) to the weight update (i.e. \( \mathbf{w}_{j+1} = \mathbf{w}_j - (1 - \mu)\eta \mathbf{g}_j + \mu(\mathbf{w}_j - \mathbf{w}_{j-1}) \)). Using momentum, we evolve the weights with respect to the gradient at the current iteration as well as the previous change in weight. This increases the learning rate from \( \eta \) to \( \eta/(1 - \mu) \) in regions where the error surface has low curvature, leading to faster convergence. At the same time, in regions where oscillations would typically occur, or high curvature in the error surface, successive contributions from the momentum term will tend to cancel out, returning the effective learning rate to \( \eta \) [3].

E. Conjugate Gradient

The method of conjugate gradients can be seen as an iterative approximation to Newton’s method for solving quadratic optimization problems without having to compute the inverses of large matrices, which would be on the order of \( \mathcal{O}(n^3) \) with respect to the network weights of size \( n \). The key idea of CG is to successively minimize the error along orthogonal, or conjugate directions, with respect to the error surface [3]. For a perfectly quadratic error surface, this entails the convergence of CG to the minimum error in at most \( n \) iterations. The goal, then, is to ensure that the complexity per iteration of CG is less than \( \mathcal{O}(n^2) \).

The first part of the algorithm, minimizing along a chosen search direction, is relatively straightforward. We may perform a line search along the given search direction, \( \mathbf{d} \), to find \( \alpha_{\text{min}} = \arg\min_{\alpha} \mathcal{E}(\mathbf{w} + \alpha \mathbf{d}) \) that minimizes the error. There are many methods for performing line search, such as golden section search, that are of complexity \( \mathcal{O}(n) \) in the number of weights.

The second part, finding an orthogonal search direction, is a bit trickier. Given an error surface approximated to second order, \( \mathcal{E}(\mathbf{w}) = \mathcal{E}_0 + \mathbf{b}^T \mathbf{w} + \frac{1}{2} \mathbf{w}^T \mathbf{H} \mathbf{w} \), where the hessian, \( \mathbf{H} \), is assumed to be positive definite, we obtain the conjugacy condition for the search directions, \( \mathbf{d}_j^T \mathbf{H} \mathbf{d}_i = 0 \) for \( j \neq i \). Naively, if we choose the search direction iteratively based off of the previous search direction, the formulation appears quadratic, giving at best \( \mathcal{O}(n^2) \) complexity per iteration.

\[
\mathbf{d}_{j+1} = -\mathbf{g}_{j+1} + \beta_j \mathbf{d}_j \\
\beta_j = \frac{\mathbf{g}_{j+1}^T \mathbf{H} \mathbf{d}_j}{\mathbf{d}_j^T \mathbf{H} \mathbf{d}_j}
\]

Fortunately, the conjugacy condition does not depend explicitly on \( \mathbf{H} \), and instead of matrix-vector computations, we may use the substitution \( \mathbf{H} \mathbf{d}_j = \mathbf{g}_{j+1} - \mathbf{g}_j \), giving the Hestenes-Stiefel formulation. By performing vector-vector multiplications instead we achieve the desired \( \mathcal{O}(n) \) complexity.

\[
\beta_j = \frac{\mathbf{g}_{j+1}^T (\mathbf{g}_{j+1} - \mathbf{g}_j)}{\mathbf{d}_j^T (\mathbf{g}_{j+1} - \mathbf{g}_j)}
\]

Another popular form is the Polak-Ribiere formulation, below, which simplifies the Hestenes-Stiefel form while at the same time giving slightly better results in that if successive gradients are similar, the search direction tends to be ‘reset’ to the direction of the error gradient [3].

\[
\beta_j = \frac{\mathbf{g}_{j+1}^T (\mathbf{g}_{j+1} - \mathbf{g}_j)}{\mathbf{g}_j^T \mathbf{g}_j}
\]

Compared to the GD and SGD methods, one major benefit of CG is that we no longer have ad-hoc terms such as learning rate or momentum, which need to be chosen a priori. In contrast, the term \( \alpha_{\text{min}} \) replaces \( \eta \) by choosing precisely how far to move along any given search direction, and \( \beta \) replaces \( \mu \) in that it guides the search direction. In terms of computational complexity, per iteration, CG is on the order of \( \mathcal{O}(n) \), although the constant is greatly increased due to the more elaborate computations. Overall, CG has a time complexity of \( \mathcal{O}(n\sqrt{k}) \), where again, \( k \) is the conditioning of the error surface. This puts CG at \( \mathcal{O}(n^{3/2}) \) for two-dimensional problems and \( \mathcal{O}(n^{1.2}) \) for three-dimensional problems, which is less than GD [4]. The CG algorithm is given below.

1. Initialize weight vector \( \mathbf{w}_1 = [w_1, \ldots, w_L, b_1, \ldots, b_L] \)
2. Evaluate the gradient vector \( \mathbf{g}_1 = \nabla_{\mathbf{w}_1} \mathcal{E} \) and set initial search direction \( \mathbf{d}_1 = -\mathbf{g}_1 \)
3. At iteration \( j \), minimize \( \mathcal{E}(\mathbf{w}_j + \alpha \mathbf{d}_j) \) with respect to \( \alpha \) to give \( \mathbf{w}_{j+1} = \mathbf{w}_j + \alpha_{\text{min}} \mathbf{d}_j \)
4. Test to see if the stopping criterion is satisfied (e.g. \( ||\nabla_{\mathbf{w}_j} \mathcal{E}|| < \theta \), and return \( \mathbf{w} \) if the case
5. Else, evaluate the new gradient vector \( \mathbf{g}_{j+1} \)
6. Evaluate the new search direction \( \mathbf{d}_{j+1} = -\mathbf{g}_{j+1} + \beta_j \mathbf{d}_j \)
7. Set \( j = j + 1 \) and go to 3.

IV. EXPERIMENT

We used a simple, two-layer network with 36 hidden nodes and 1 output node. Given the form of the input feature vector, discussed below, this network setup was chosen because it could theoretically learn the classification rule perfectly. Essentially, the weights could be set such that each of the hidden nodes would be sensitive to exactly one conjunction of the features equivalent to satisfying an all same or all different condition. Activation at the output would subsequently only occur if four of the hidden nodes, each corresponding to a different feature dimension, were active. Activation units were chosen as the standard sigmoidal function, hyperbolic tangent, specifically, and the loss function was chosen as mean squared error. With a single output node, this gives the Bayes posterior probability. A schematic of the network is shown in figure 2.

The feature vector fed into the networks input layer was presented as a concatenation of indicator functions that specified whether a particular value was present on a card or not. This gave a feature vector of size \( 4 \times 3 = 12 \) per card, and an overall length of 36. The order of the features for any given card was number (one, two, or three), color (red, green, or purple), shape (diamond, oval, or squiggle), and fill (open, striped, or solid). For example, the feature vector of a card with two solid green diamonds is 010010100001.

Training examples were generated randomly, either according to the natural probabilities of picking three cards (without
replacement) from a full deck, or such that the ‘set’/‘non-set’ labeling was evenly split. A test set was generated similarly. All three gradient methods were then trained and tested on the same sets for multiple training and test sets to obtain mean performance measures. Training was performed on 5000 examples and testing on 10000 examples. For SGD, the training examples were randomly reordered each pass through the training set.

V. RESULTS

A. Convergence Curves

Sample convergence curves for learning on an evenly split training set are given below for GD, SGD, and CG in figure 3. One particularly striking artifact in the curves for GD and SGD compared with that of CG is the oscillation in the training loss as the network weights settle. Although we may take the oscillations as an indication that the learning rate is too high, in this case, it is likely more accurate that the oscillations are a result of the lack of continuity in the error surface. That is, the rules of Set are highly nonlinear with respect to the feature space, and adjusting the weights across these decision boundaries is highly sensitive. When the learning rate was reduced in these cases, the only major effect was that the convergence rate slowed accordingly. This can be seen in figure 4.

For an unevenly labeled training set, in several cases, all three algorithms failed to converge aside from labeling every data point as a ‘non-set’. With respect to the priors, this gave reasonable training error as well as testing error for the unbalanced test set (error rate of 3%), but on the balanced test set it was essentially no better than chance. When GD converged to accommodate for the rare events, the convergence times were much longer than that for the balanced case. Similarly for CG. SGD never converged to accommodate rare events in any of the training cycles. Of the three gradient methods CG was the only one able to achieve error rate of 0% on the training set. An example convergence curve for CG in this case is shown in figure 5. We observe an initial drop in the training error that accounts for the unbalanced
priors, and then a more gradual decrease as the weights adjust for the rare events.

![Convergence Curve for CG (rare events)](image)

**Fig. 5.** Convergence Curve for CG (rare events)

**B. Convergence Rate**

For the balanced training sets, the rate of convergence fell in line roughly with the theory presented in the methods section. On average, GD took 61s to converge with on the order of 2750 iterations, giving roughly 22ms per iteration. SGD took 40s to converge with on the order of 35000 iterations, giving roughly 114ms per iteration. CG took 36s to converge with on the order of 115 iterations, giving roughly 313ms per iteration. We experimented with momentum in order to improve convergence rates of GD and SGD. In particular, we were able to decrease the convergence times for SGD substantially. Figure 6 shows the progression of some sample convergence curves for SGD with momentum. Although the addition of momentum initially decreased convergence times, we observed that choosing $\mu$ too large became detrimental.

![Convergence Curve for SGD with momentum ($\mu_A = 0.0$, $\mu_B = 0.05$, $\mu_C = 0.1$, and $\mu_D = 0.2$)](image)

**Fig. 6.** Convergence Curve for SGD with momentum ($\mu_A = 0.0$, $\mu_B = 0.05$, $\mu_C = 0.1$, and $\mu_D = 0.2$)

With respect to the unbalanced set, because of the lack of convergence in training, a good measure of the convergence times could not be established for SGD. For GD, in accounting for rare events, convergence times slowed considerably with the method tapering slowly even after 10000 iterations. Of the three gradient methods, CG had the most consistent convergence while accounting for rare events. On average, CG took 70s to converge with on the order of 220 iterations.

**C. Test Accuracy**

Although the network was constructed such that the weights could fully capture the label generating rules, only CG routinely achieved 0% error rate on the training set. SGD, for the balanced training set, was also able to achieve this on occasion, in spite of the method failing to converge for an unbalanced training set. Table 1 provides the confusion matrices of how well each method performed in terms of test accuracy for balanced and unbalanced training and test sets.

<table>
<thead>
<tr>
<th>Method</th>
<th>$B_{train}$</th>
<th>$U_{train}$</th>
<th>$B_{test}$</th>
<th>$U_{test}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>GD</td>
<td>0.925%</td>
<td>38.9%</td>
<td>1.95%</td>
<td>2.20%</td>
</tr>
<tr>
<td>SGD</td>
<td>0.458%</td>
<td>30.4%</td>
<td>1.02%</td>
<td>3.04%</td>
</tr>
<tr>
<td>CG</td>
<td>0.152%</td>
<td>7.84%</td>
<td>0.298%</td>
<td>0.583%</td>
</tr>
</tbody>
</table>

**Tab. 1.** Confusion matrices of test accuracy for balanced (B) and unbalanced (U) training and test sets

**VI. DISCUSSION**

Overall, the conjugate gradient method for training neural networks performed the best with respect to the various optimality criterion. CG had the highest test accuracy across all variations of balanced or unbalanced training and test sets. In particular, CG was the algorithm most sensitive and accommodating to learning rare events. Looking at the convergence curves, we also saw that CG was much smoother, following a monotonically decreasing trend as opposed to exhibiting oscillatory behavior as in the case of GD and SGD. In terms of computational complexity, although the time per iteration was much larger than either GD or SGD, because the method required substantially fewer iterations to converge, the resulting convergence times were faster. Finally, CG is much cleaner algorithmically, if more complex, in the sense that it does not contain any hyperparameters such as learning rate or momentum, instead replacing them with line search minimization and choosing orthogonal search directions, respectively.

**REFERENCES**