HW5: A More Realistic Problem

Collin Conway

Abstract

This paper explores the effect of mini batch size on the conjugate gradient algorithm. We implement one from scratch, explain the implementation, and theory. Then we do a performance comparison with the steepest descent algorithm. We examine it through the lens of training performance, and generalization performance and perform several training runs of varying batch size and present the results. What we find is that our implementation of the conjugate gradient algorithm does not do as well with this architecture as steepest descent. We identify a sources of error, and other avenues of research.

Table of Contents

Introduction

Conjugate gradient is an optimization to backpropagation that is supposed allow backpropagation networks to make better weight updates. In this project we will explore conjugate gradient backpropagation, and the effects, of mini batch size on training and generalization performance. To do so we are going to compare this training algorithm to a plain vanilla steepest descent backpropagation network, by running both of them on the MNIST data set, collecting data, and examining the results. [1]

This paper will discuss the data & network architecture algorithms, investigate sources of error, and perform analysis on what data we could gather. For our performance comparison we will be examining the effects of batch size on the percent error and performance index of the algorithms The purpose of this introduction is to present the theoretical knowledge required to interpret the results and clearly convey the implementation details of the the network, and any design decisions that guided them.

Network Design

Early on in the project I discovered that conjugate gradient was very involved algorithmically, and would be difficult to get working given my time constraints. At that point I made a judgement call, I wanted to keep all other parts of my network, including data, and architecture, as simple as possible to allow me to focus on the algorithm more. This guided my choices in how I designed my network.

Data

We were instructed to use the MNIST data set [1]. It is very well catalogued and tested on many different architectures. Our goal is classify elements in the data set and put them in categories. We will have 10 categories, one for each digit between 0-9. It is already broken into a training set and a test set. The training set contains a relatively uniform distribution of digits, between many different authors. The choices we have when it comes to the data are whether to preprocess the data, and whether a validation set for early stopping will be necessary. As far as preprocessing, I felt that could be a whole project on it's own and decided to leave the images alone. I also decided against early stopping, so the validation set was not necessary.

Architecture

For network architecture, I kept the spirit of keeping simple things simple and reused code from previous assignment. I chose to do a 3 layer back propagation network with log sigmoid transfer functions. I had good results with using one neuron on the output layer per category in the previous assignment. So I chose to have 10 neurons on the output layer. Next I referred, to the MNIST canonical site, and decided to pick the minimal number of hidden neurons that was written about in a paper. Total accuracy was not important for the context of my project, I just wanted something I could iterate over quickly to rapidly develop my new training algorithm. Browsing from the list, they seemed to range from 300 to 1500 in 3 layer networks, so I picked 300. So my final network architecture was 784 inputs -> 300 hidden neurons -> 10 output neurons.

Plumbing

So to load the data set into memory I reused some code provided by Stanford to load the data sets into memory[2]. In addition to this I needed to format the labels to match the pattern I needed to display the different categories, which I show in figure 1.

Figure 1: Label Processing

```
function [ T_n ] = formatOutput( T_o )
T_n = zeros(10,size(T_o,1));
lookup= diag(ones(1,10));
getCode = @(x) lookup(x,:);
for i = 1:size(T_o,1)
   T_n(:,i) = getCode(T_o(i)+1);
end
end
```

Algorithms

There are quite a few algorithms being used in this project, I will use this section to attempt to separate them and talk about them one at a time. Sometimes it will be appropriate to describe things math, and sometimes in code. The code for this project will be attached in full detail at the end of the report.

I use a code formatter to help illustrate my code, but to keep things brief, I will gloss over details by putting an ellipses in a comment describing what I am deleting like so:

```
%...this is some non relevant code
```

I do my best to follow the same notation as Hagan when referring to my network. [3] There are a few pieces of important pieces of notation and language to discuss before we dive in.

I use Hagan's notation where $F(x)$ refers to the error in the output layer of the network with respect to the training set in supervised learning.

Error shall refer to what Hagan calls the performance index, $E^T \bullet E$ where $E = T - A_n$.

All networks described in this paper are 3 layer backpropagation networks. All transfer functions are log sigmoid, the reasons why are described in the data section.

Also whenever I mention "vanilla" backpropagation I am referring to a no-frills no-modifications steepest descent backpropagation network.

Conjugate Gradient

The conjugate gradient algorithm is useful for solving linear systems of equations. [4] It has several basic steps and a few desirable qualities that make it a good candidate for application in neural networks. It shares the property of quadratic convergence with newton's algorithm, but does not require computing the Hessian, which might be intractable for large data sets. [3] So conjugate gradient provides a good level of compromise and is still performant.

The basic steps of the algorithm are very simple, and can be used to train networks with a few small modifications we will detail in subsequent sections. Figure 2 shows the basic steps of the conjugate gradient algorithm.

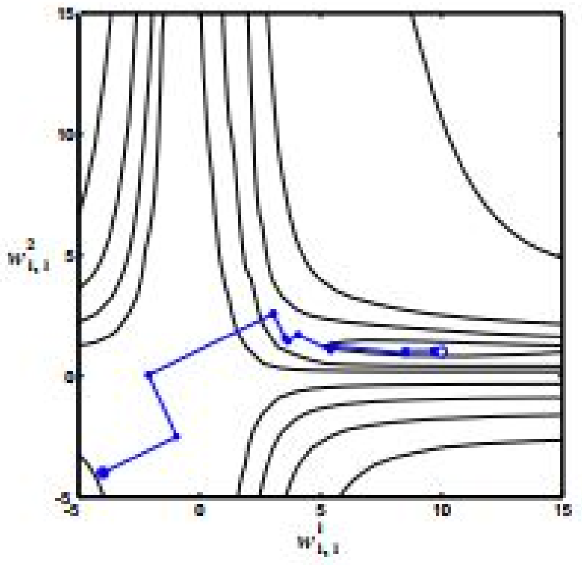
Figure 2: Conjugate Gradient Algorithm

Step	Equation	Description
1	$p_0 = -g_0$	Pick an initial search direction. Most version us the steepest descent direction.
2	where $g_k = \nabla F(x)$	Gradient is with respect to the error function.
3	$x_{k+1} = x_k + \alpha p_k$	Take a step along the search direction by varying the learning rate. Do it until $F(x)$ is minimized.
4	$p_{k+1} =- g_k + \beta p_k$, where $\beta = g_{k+1}{}^T \bullet g_{k+1} / g_k{}^T \bullet g_k$	Pick a new search direction based on the Fletcher-Reeves update.
5	Repeat from step 3	Until the algorithm converges.

Now this algorithm cannot directly be applied to the task of training a neural network, it needs a few modifications as described in the following sections.

Before proceeding let’s discuss a graph of the algorithm in action. Figure 3 shows a graph of successive iterations of the algorithm.

Figure 3: Conjugate gradient



So you will notice when the algorithm is on a flat portion of the graph with little change it follows a path very close to steepest descent. When the graph is steep, the fletcher reeves update "mixes" in more of the new gradient picking a direction that guides convergence closer towards the global minimum. You will also notice because the algorithm always picks a distance to travel that minimizes the error, it has far more predictable and smooth behavior in canyons in the graph.

Mini Batch Training

So it's pretty critical to the algorithm to be able to take the gradient of the error function. Figure 4 shows how to calculate the gradients of the function at a point.

Figure 4: Gradient Calculation

```
function [ gW1, gb1, gW2, gb2] = getGradients( W1, b1, W2, b2, A0, T )
        %get the first layer output
        A1 = logsigmoid(W1*A0 + b1);
        %get the second layer output
        A2 = logsigmoid(W2*A1 + b2);
        %error
        E = T - A2;
        %back propigation
        S2 = -2*diag((1 - A2).*A2)*E;
        S1 = diag((1 - A1).*A1)*(W2'*S2);
        %calculate gradients
        gW1 = S1*(A0');
        gb1 = S1;
        gW2 = S2*(A1');
        gb2 = S2;
end
```

Now the problem is that this only tells us the gradient of the function at a point, and conjugate gradient is a batch algorithm.

This means that to work effectively, and converge conjugate gradient needs an estimate of global error of the function. We need a way to estimate this, and an effective approach is to take an average of the gradient at each point and accumulate them. Figure 5 shows how to take an average of our gradient's based on our previous gradient calculation code.

Figure 5: Accumulating Gradients

```
function [ gW1, gb1, gW2, gb2] = accumulateGradients( W1, b1, W2, b2, P, T )
%... preallocation here
%count of elements in training set
Q = size(P,2);
for j = 1:Q
    [gW1t, gb1t, gW2t, gb2t] = getGradients(    W1, b1, ...
                                                W2, b2, ...
                                                P(:,j),T(:,j));
    %accumulate an average gradient
    gW1 = gW1 + gW1t/Q;     gb1 = gb1 + gb1t/Q;
    gW2 = gW2 + gW2t/Q;     gb2 = gb2 + gb2t/Q;
end
end
```

Now this approach is a little naive and despite being mathematically correct does not give the best results with real world data sets. The problem is that batch algorithms like conjugate gradient, only update their weights and biases, once per iteration, which with a full batch strategy is only once per epoch. That means that while conjugate gradient might take only 20 iterations to converge, an approach like steepest descent will take far fewer calculations to complete.

That being said batch learning is a tradeoff of *convergence accuracy* vs. *convergence speed*. Doing large batches allows for the most accurate gradient accuracy at a given time, while sacrificing frequent updates, and potentially many extra calculations saved. [5] There is another factor that we need to weigh when considering, for batch training we need to load the entire training set into memory, and perform operations on it. For some very large sets this can be unwieldy, or cannot take advantage of specialized hardware.

So we have discussed a lot of problems with batch training, how do we leverage the benefits of more accurate gradients, but converge in a reasonable amount of time. The answer is using mini batches. This means randomizing the order of the data set, breaking it into manageable chunks, then performing the algorithm on each section. Figure 6 shows a pretty abbreviated version of this strategy.

Figure 6: Mini Batch Algorithm

```
for i = 1:epochs
    %get a random ordering
    idx = randperm(size(P_o,2));
    %get initial mini batch
    P1 = P_o(:,idx(1:(1+batchSize)-1));
    T1 = T_o(:,idx(1:(1+batchSize)-1));
    for batch = 2:batchSize:size(P_o,2)   %for each mini batch
        %train on each strategy
        %test our modifications
        %grab next batch
        P1 = P_o(:,idx(batch:(batch+batchSize)-1));
        T1 = T_o(:,idx(batch:(batch+batchSize)-1));
    end
end
```

We can still perform multiple epochs if we wish, we just need to perform the mini batch multiple times. This is a very common strategy that yields pretty good performant results for many algorithms. [6] For our intents and purposes, it will be a sufficient means of estimating gradients of error.

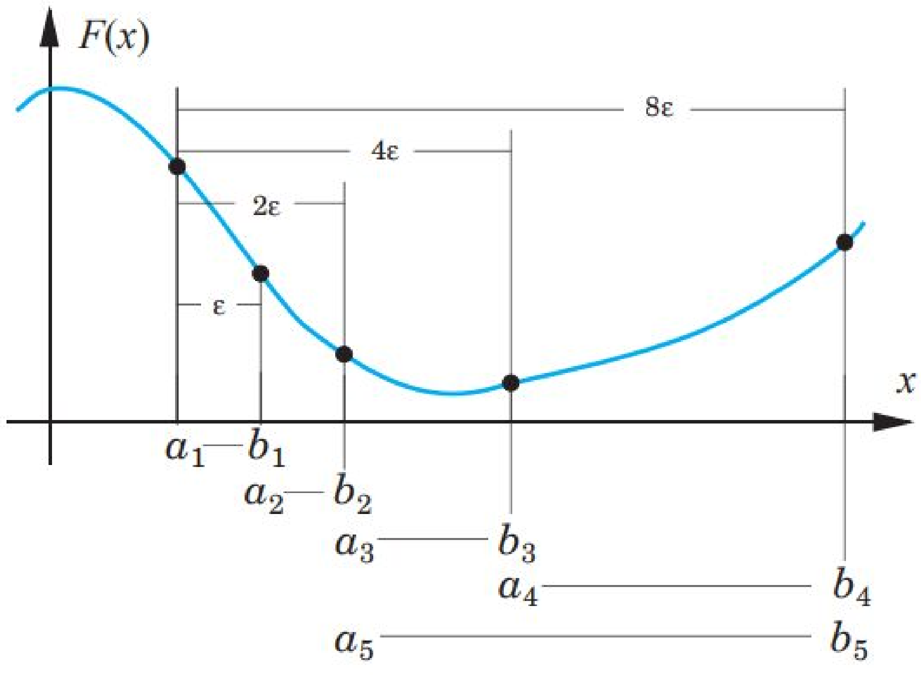
Optimization

So now that we have a way of estimating the gradient of error in terms of learning rate, weights and biases we want a way to minimize the error of the function as a whole in terms of the learning rate in the direction of our gradients. This falls into two sub-problems: interval selection, and interval reduction.

Interval Selection

We are going to use the interval selection algorithm described by Hagan. [3] The basic idea is to select a value ε and a starting point a. We evaluate $F(a)$ & $F(a + \varepsilon)$. In each iteration of the algorithm we successfully double ε and evaluate the function until F increases on two successive iterations, or we reach some cap. Figure 7 shows this approach graphically, while figure 8 shows my implementation.

Figure 7: Interval Selection Algorithm



[1]

Figure 8: Interval Selection Implementation

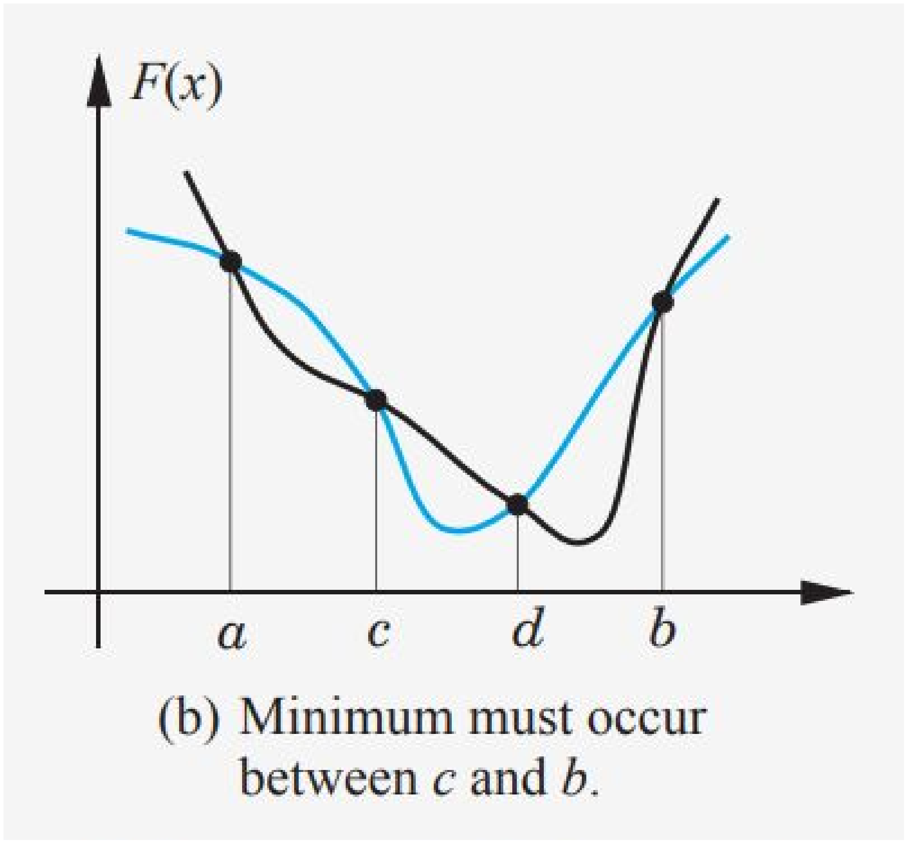
```
rate = ep;
for i = 1:max_itr
    b = a+rate;   %get the end of the interval
    f_b = f(b);
    if f_a < f_b %are we ascending
        if desc   %if we previously were descending we have a minimum
            a = old;
            break
        end
        desc = false;
    else
        if ~desc
            desc = true;
        end
        old = a; %move the interval forward if continuously descending
        a = b;
        f_a = f_b;
    end
    rate = rate * 2; % double the rate
end
```

This algorithm can yield a very large learning rate very quickly, and can cause the algorithm to diverge or reduce on too coarse of an interval.

Interval Reduction

So now that we have an algorithm that can quickly grab us an interval with a minimum in it, we need to find a way to reduce that interval. To do this we will perform a golden section search. This algorithm uses 4 points to figure out which ~2/3rds of the function contains the minimum. It uses the golden ratio to reduce the interval in an efficient manner, and only needs to calculate one point per iteration which makes it pretty efficient. Figure 9 shows the reduction interval pictorially, and Figure 10 shows my implementation. This is a canned algorithm from the Hagan book. [3]

Figure 9: Golden Section Search



[3]

Figure 10: Golden Section Implementation

```
function out = golden(f,a, b)
tau = 0.618; %golden ratio
tol = 0.0005; %tolerance for answer
c = a + (1 - tau)*(b - a);
f_c = f(c);
d = b - (1 - tau)*(b - a);
f_d = f(d);
while b - a > tol
    if (f_c < f_d) %shift the window left
        b = d;
        d = c;
        c = a + (1 - tau)*(b - a);
        f_d = f_c;
        f_c = f(c);
    else %shift the window right
        a = c;
        c = d;
        d = b - (1 - tau)*(b - a);
        f_c = f_d;
        f_d = f(d);
    end
end
out = a;
end
```

Conjugate Gradient Backpropagation

Now that we have assembled all of the pieces: conjugate gradient, gradient estimation, mini batch training, interval selection, and interval reduction we can start assembling the pieces of the whole. We introduced all of these modifications to the conjugate gradient algorithm to make it more suited to training neural networks on real data with hardware. However there are a few odds and ends that need cleaning up.

Search Direction Update

Let's talk about updating the search direction. We use the Fletcher reeves update to determine the β constant:

$$\beta = {g_1}^T \bullet g_1 / {g_0}^T \bullet g_0$$

This compares the magnitudes of the gradients, and "mixes" in more of the old direction based on the ratio of the gradient, the new direction is the steepest descent direction. With the following equation:

$$p_1 = -g_0 + \beta p_0$$

The initial direction is the initial steepest descent. Now the problem is that the linear systems version of the algorithm is guaranteed to converge in n iterations where n is the number of elements in the gradient vector.

In backpropagation training, we have multiple gradient vectors, and sometimes we have gradient matrices, so starts to break down a little. I use the compromise that Hagan suggests [2] where I reset the search direction to the steepest descent direction every m iterations where m is the total number of parameters we are tuning.

My version of the fletcher reeves update takes an accumulated average of all the columns in the gradient matrixes, as described in figures 11, and 12.

Figure 11: Accumulated Average

```
function out = accumulate(x,y)
out = sum(x,2)/y;
end
```

Figure 12: Fletcher Reeves Update

```
function beta = getGains(gn,g)
sgn = accumulate(gn,size(gn,2)); sg = accumulate(g,size(g,2));
beta = dot(sgn,sgn)/dot(sg,sg);
end
```

Line Optimization

So the line that we are optimizing for the conjugate gradient update, is the parameters of the network based on learning rate. To start I define an equation that calculates, the mean squared error of weights and biases, as shown in figure 13. Note it is configured to handle batches of inputs, and single values.

Figure 13: Performance Index

```
function index = perfIndex(W1, b1, W2, b2, P, T)
E = accumulate((T - evaluate(W1,b1,W2,b2,P)),size(P,2));
index = dot(E,E);
end
```

The next step is optimize the performance index of this function base on the learning rate parameter. I do this by taking a leaf out of the functional programming playbook and define a wrapper based on a lexical closure, as shown in figure 14.

Figure 14: Wrapped Index

```
wrapper = @(rate) perfIndex(W1 + rate*pW1, ...
                                            b1 + rate*pb1, ...
                                            W2 + rate*pW2, ...
                                            b2 + rate*pb2, ...
                                            P,T)
```

This function captures the current values of the weights and biases whenever it gets declared and gets passed in a learning rate. I pass this around in as an anonymous function to my interval selection and reduction algorithms.

Training Algorithm

Figure 15 shows my abbreviated implementation for the conjugate gradient backpropagation.

Figure 15: Conjugate Gradient Backpropagation

```
function [ W1, b1,  W2,  b2, ...
           pW1n,pb1n, pW2n, pb2n, ...
           gW1n, gb1n, gW2n, gb2n, ...
           iteration, index] = ...
                        conjugateTrain( P, T, ...
                                        W1, b1, W2, b2, ...
                                        pW1, pb1, pW2, pb2, ...
                                        gW1,gb1, gW2, gb2, ...
                                        iteration, isGraph)
%important constants for training
epsilon = 0.001;         %rate to increase search interval
startingRate =  0.000;     %minimum jump
```

```
%reset search direction every n iterations
R = size(P,1);
s1 = size(W1,1);
s2 = size(W2,1);
resetPeriod = R*s2 + s2 + s2*s1 + s1;
%select an interval to minimize
[a, b] = getInterval( @(rate) perfIndex(    W1 + rate*pW1, ...
                                b1 + rate*pb1, ...
                                W2 + rate*pW2, ...
                                b2 + rate*pb2, ...
                                P,T), ...
                                startingRate, epsilon);
%minimize the interval to tolerance
rate = golden(@(rate) perfIndex(    W1 + rate*pW1, ...
                                b1 + rate*pb1, ...
                                W2 + rate*pW2, ...
                                b2 + rate*pb2, ...
                                P,T),...
                                a, b);
%conjugate gradient weight update
W1 = W1 + rate*pW1;     b1 = b1 + rate*pb1;
W2 = W2 + rate*pW2;     b2 = b2 + rate*pb2;
%get error
index = perfIndex(  W1 + rate*pW1, ...
                    b1 + rate*pb1, ...
                    W2 + rate*pW2, ...
                    b2 + rate*pb2, ...
                    P, T);
[gW1n, gb1n, gW2n, gb2n] = accumulateGradients( W1, b1, W2, b2, P, T );
iteration = iteration +1;
%get directional gain
if mod(iteration, resetPeriod) == 0
    bW1 = 0;    bb1 = 0;
    bW2 = 0;    bb2 = 0;
else
    %decide how much of the new gradient to "mix" based on magnitude
    %uses fletcher reeves update
    bW1 = getGains(gW1n, gW1);    bb1 = getGains(gb1n, gb1);
    bW2 = getGains(gW2n, gW2);    bb2 = getGains(gb2n, gb2);
end
%calculate new directions
pW1n = -gW1n + pW1*bW1;    pb1n = -gb1n + pb1*bb1;
pW2n = -gW2n + pW2*bW2;    pb2n = -gb2n + pb2*bb2;
end
```

So this algorithm performs one iteration of the algorithm with a provided iteration count, initial gradient, initial search direction, and weights and biases. It is set up like this so it can be wrapped by a mini batch algorithm that will take care of the setup, and provide it data. This keeps the algorithm relatively compact, and extensible.

Vanilla Backpropagation Training

So as mentioned previously we need to compare our new fancy conjugate gradient algorithm to some baseline network. I made a plain vanilla steepest descent algorithm that can be trained concurrently with our conjugate gradient network. All of the interesting code for this section has already been written in some shape or form for the conjugate gradient algorithm. My top level script is in figure 16.

Figure 16: Vanilla Backpropagation

```
function [W1,b1,W2,b2, index] = vanillaTrain(P,T,W1,b1,W2,b2)
rate = 0.09;
for j = 1:size(P,2)
    [ gW1, gb1, gW2, gb2] = getGradients(W1, b1, ...
                                                    W2, b2, ...
                                                    P(:,j),T(:,j) );
    W1 = W1 - rate*gW1;    b1 = b1 - rate*gb1;
    W2 = W2 - rate*gW2;    b2 = b2 - rate*gb2;
end
Index =  perfIndex(W1,b1,W2,b2,P,T);
end
```

Methods

So as described previously we want to compare the performance of the two algorithms. This begs the questions: What does performance mean? How do we collect this information? How do they compare?

Metrics

So to compare our networks we are interested in how well they do on real world data. So the best metric for comparison will be how well they do on the test set. We will do this by defining a metric called percent error. Percent error is defined by taking the rounded outputs of the network on some set of data, counting the failures, and dividing by the total number of inputs. Because this is a classification problem this is pretty easy to do, either the answer was correct or incorrect for a given category with no realm of error in between. My code for calculating percent error is shown in figure 17.

Figure 17: Percent Error Calculation

```
function err = getPercError(f,P,T)
A = round(f(P));
errCount = 0;
for i = 1:size(P,2)
```

```
    if sum((T(:,i)-A(:,i)).^2)> 0
        errCount = errCount + 1;
    end
end
err = errCount./size(T,2)*100;
end
```

Another way to track the performance of an algorithm is by examining its behavior during training, we can do that very easily by examining performance index of the function at each mini batch.

So to tie it together percent error is a measure of *generalization performance*, while performance index is a measure of *training performance*, which will be sufficient observable effects to monitor when varying batch size of the network.

Experiment Design

Since we want to compare apples to apples we want to take a few steps to make sure that our conjugate gradient and vanilla algorithm are compared evenly. This section will detail the script that trained these two networks, their starting conditions and what we tuned the various parameters in the network to.

Training Script

In this section we will break down the training script section by section and talk about what we did to set up training

Weights

So to make sure that our networks start out at the same point we want to share some initial random weights. We make a slight optimization here, I want to make sure my networks start out with some good initial weights in a given range. Some need to be positive, and some need to be negative, zero valued parameters are hard to tune, so we want to avoid those. Figure 18 shows a function that generates a matrix with random positive and negative values between a given set of absolute value magnitudes.

Figure 18: Weight Generation

```
function out = rangedRand(magLow, magHigh, R, C)
out = magLow + (magHigh - magLow) * rand(R,C);
for i =1:size(out,1)
    for j = 1:size(out,2)
       if round(rand) == 1
           out(i,j) = out(i,j)*-1;
       end
    end
end
end
```

I use this technique to generate the initial weights and biases that I share between both networks, as shown in figure 19. I input the number of hidden units as a parameter.
Figure 19: getWeights

```
function [W1,b1,W2,b2] = getWeights(P,T, h1)
R = size(P,1);
S1 = h1;
S2 = size(T,1);
%get randomized weights and biases
maxMag = 0.5;
minMag = 0.2;
W1 = rangedRand(minMag, maxMag, S1, R);
b1 = rangedRand(minMag, maxMag, S1, 1);
W2 = rangedRand(minMag, maxMag, S2, S1);
b2 = rangedRand(minMag, maxMag, S2, 1);
end
```

Initial Gradients

Before I can start my mini batch algorithm I need to get an initial direction to search, I do this by grabbing a random mini batch from the training set and getting it's gradient directly, as shown in figure 20. I pass this as an initial parameter to my training functions.
Figure 20: Calculate initial gradients

```
idx = randperm(size(P_o,2));
P1 = P_o(:,idx(1:(1+batchSize)-1));
T1 = T_o(:,idx(1:(1+batchSize)-1));
%get initial gradient
[gW1, gb1, gW2, gb2] = accumulateGradients(W1, b1, W2, b2, P1, T1);
pW1 = -gW1;    pb1 = -gb1;
pW2 = -gW2;    pb2 = -gb2;
```

We previously discussed how the mini batch algorithm worked and gave an implementation of it in Figure 6.

Training Set Evaluation

As a final step, we need to evaluate the performance on the training set described in figure 21.

Figure 21: Training Set Evaluation

```
disp('Load Test Set :')
disp('    loading test images')
P2 = loadMNISTImages('./data/t10k-images-idx3-ubyte');
disp('    loading test labels')
T2 = formatOutput(loadMNISTLabels('./data/t10k-labels-idx1-ubyte'));
evaluate1 = @(x) evaluate(cW1,cb1,cW2,cb2, x);
err1 = getPercError(evaluate1,P1,T1);
evaluate2 = @(x)  evaluate(vW1,vb1,vW2,vb2, x);
err2 = getPercError(evaluate2,P1,T1);
```

Important Parameters

So there are a lot of various dials that can be turned up or down in this algorithms figure 22, shows all of the parameters that I hold constant or set to a discrete value.

Figure 22: Parameters

Location	Parameter	Value	Description
Main Script	*Epochs*	3	The total number of runs through every element in the training set to perform.
Conugate Training	*Epsilon*	0.001	The rate to increase the interval selection width. This doubles every iteration, so needs to start small for good resolution.
	Starting Rate	0	The minimum learning rate for conjugate training. In all of my experiments setting this to a value did not improve performance.
	Reset Period	238510	The number of mini batches perform before resetting the search direction, as suggested by Hagan. [3]
Interval Selection	*Max Iterations*	10	The maximum number of iterations to allow for performance
	Max Rate	3	The maximum learning rate for the function. This needed to be capped at a reasonable value for stability reasons.
Interval Reduction	*Tau*	0.618	The ratio to reduce the interval, set to the default golden ratio value
	Tolerance	0.0005	The minimum acceptable error tolerance for a calculated learning rate.
Weight Selection	*Minimum Magnitude*	0.2	The minimum absolute value magnitude for a weight

	Maximum Magnitudine	0.5	The maximum absolute value magnitude for a weight
Vanilla Training	*Learning Rate*	0.09	The fixed learning rate for vanilla backpropagation.

Results

So the one variable we are modifying is batch size. After much experimenting I found the algorithm was very tempermental when it came to varying batch size. The only acceptable range of sizes were between 100 and 1000. For our comparison today we will be examining 100, 300, 600, and 1000 elements in the mini batch set. I specifically chose values that would cleanly divided into 60,000 which is the size of the training set, for an even coverage of mini batches. We will present the data side by side for each different interval.

Training Size 100

Figure 23: Size 100 Performance Index

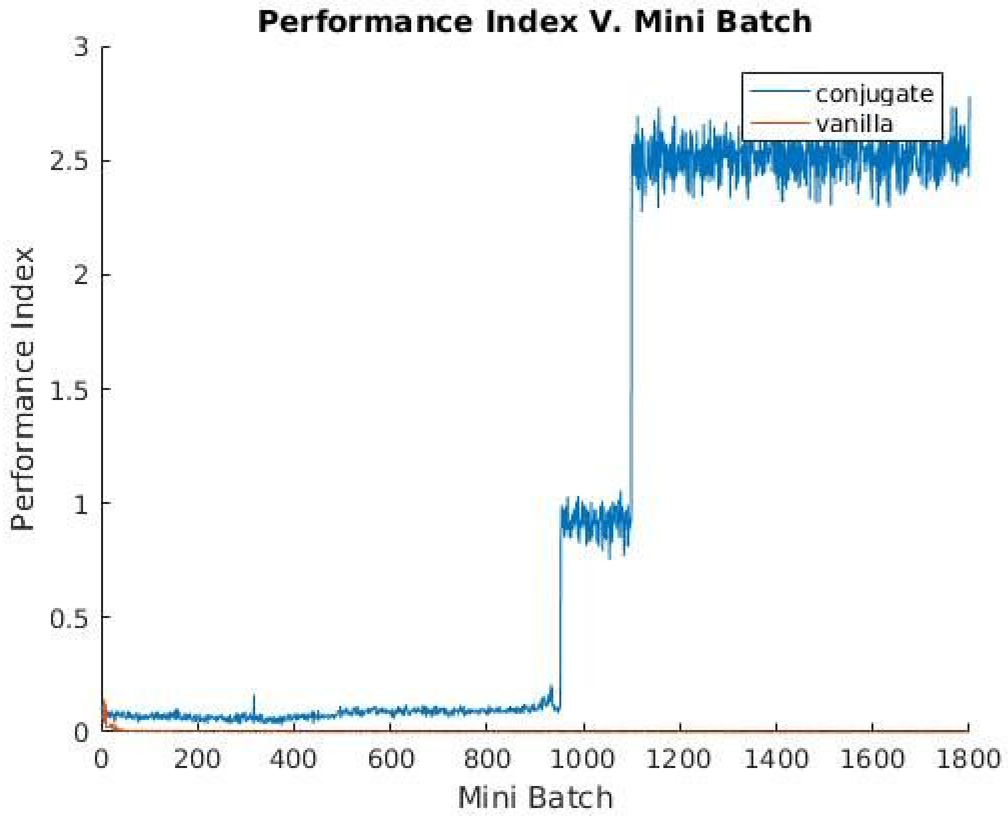
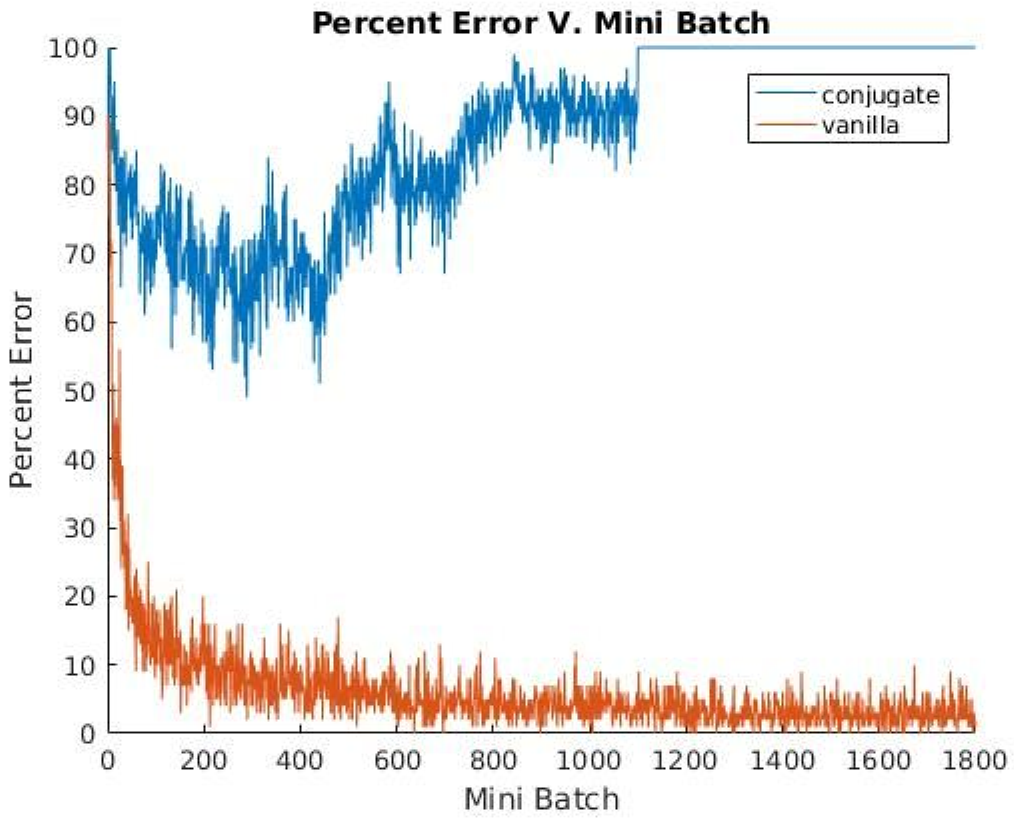


Figure 24: Size 100 Percent Error



Training Size 300

Figure 25: Size 300 Performance Index

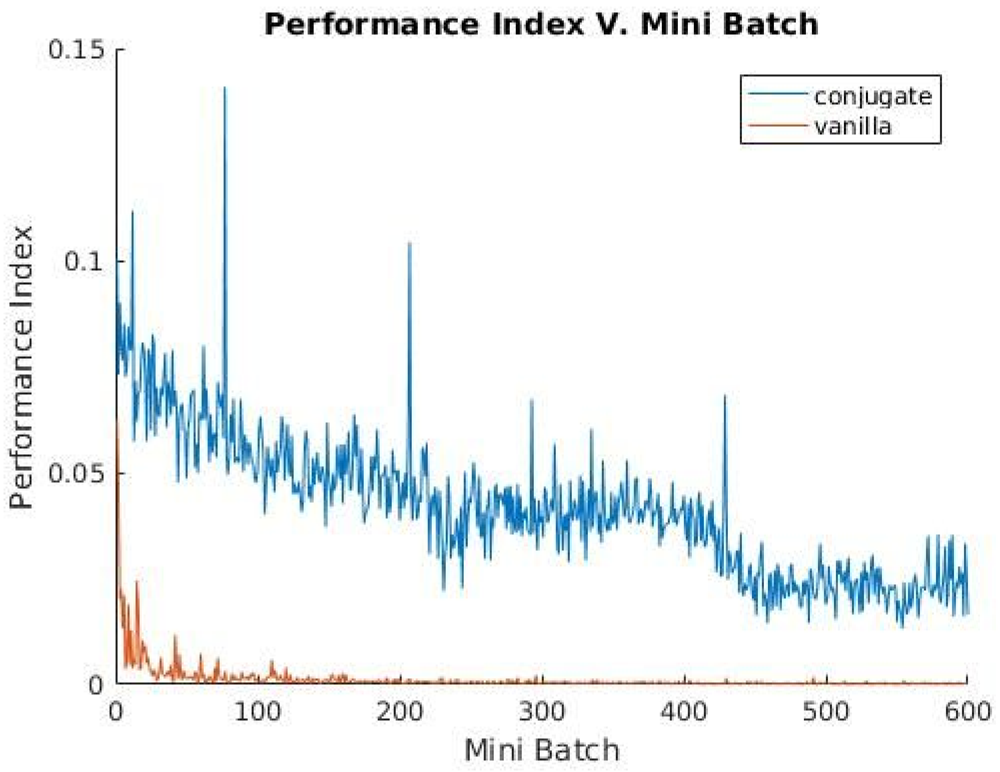
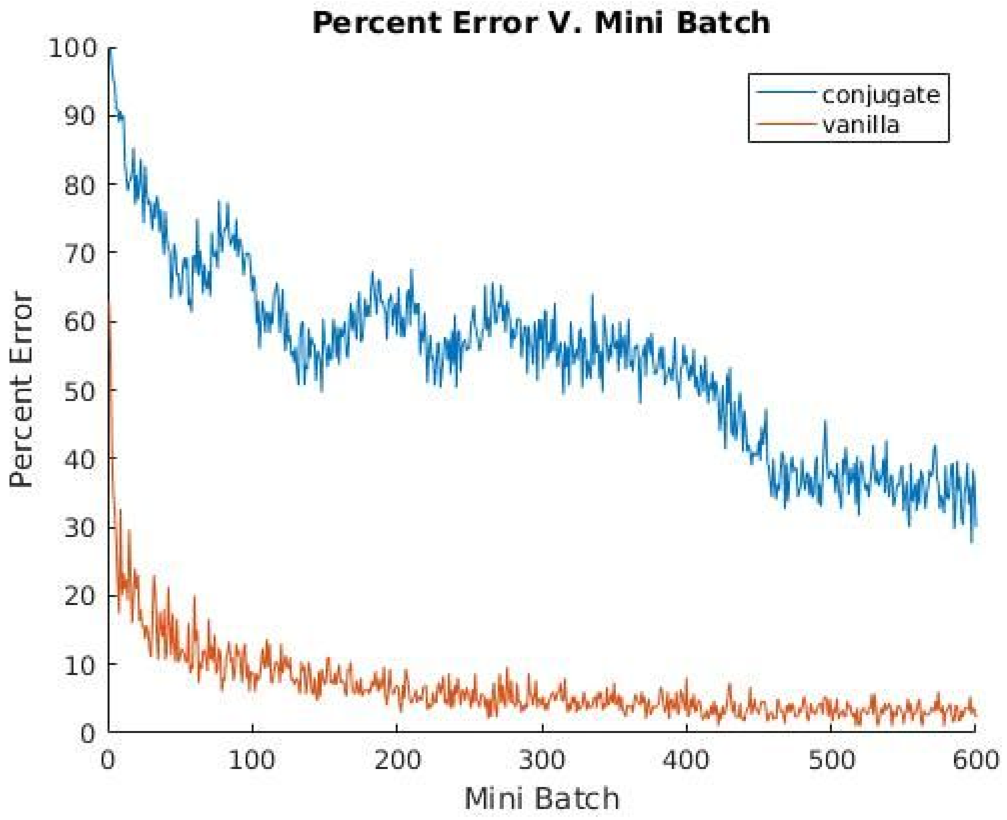


Figure 26: Size 300 Percent Error



Training Size 600

Figure 27: Size 600 Performance Index

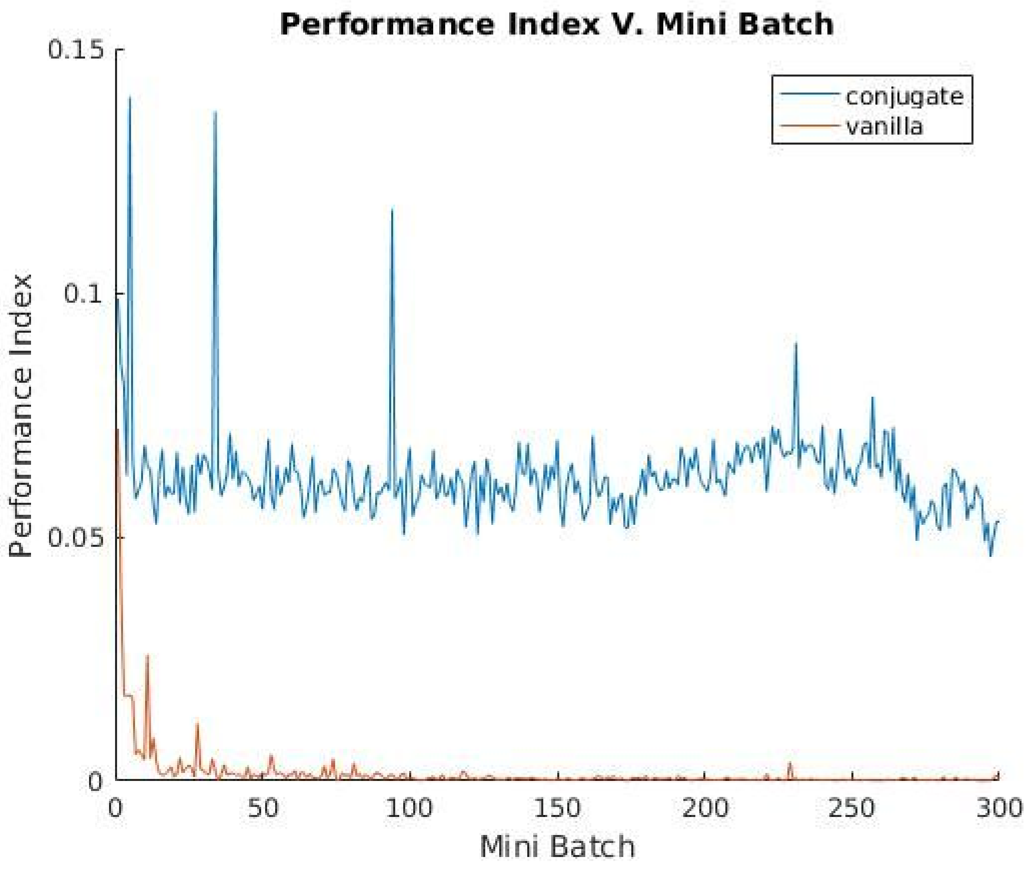
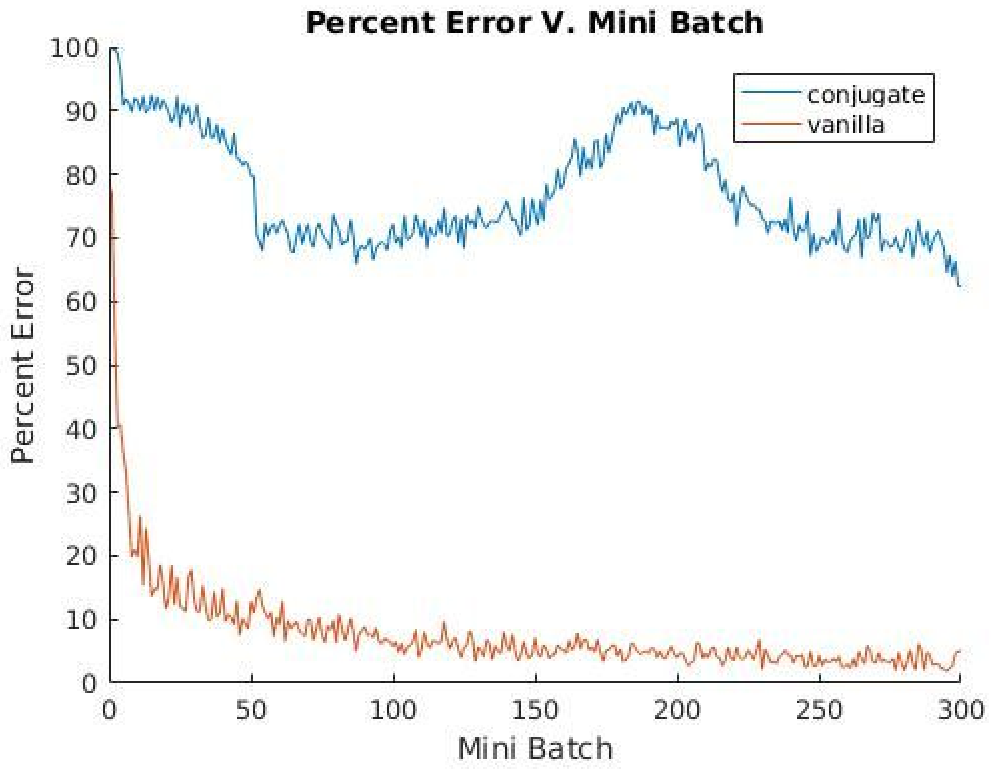


Figure 28: Size 600 Percent Error



Training Size 1000

Figure 29: Size 1000 Performance Index

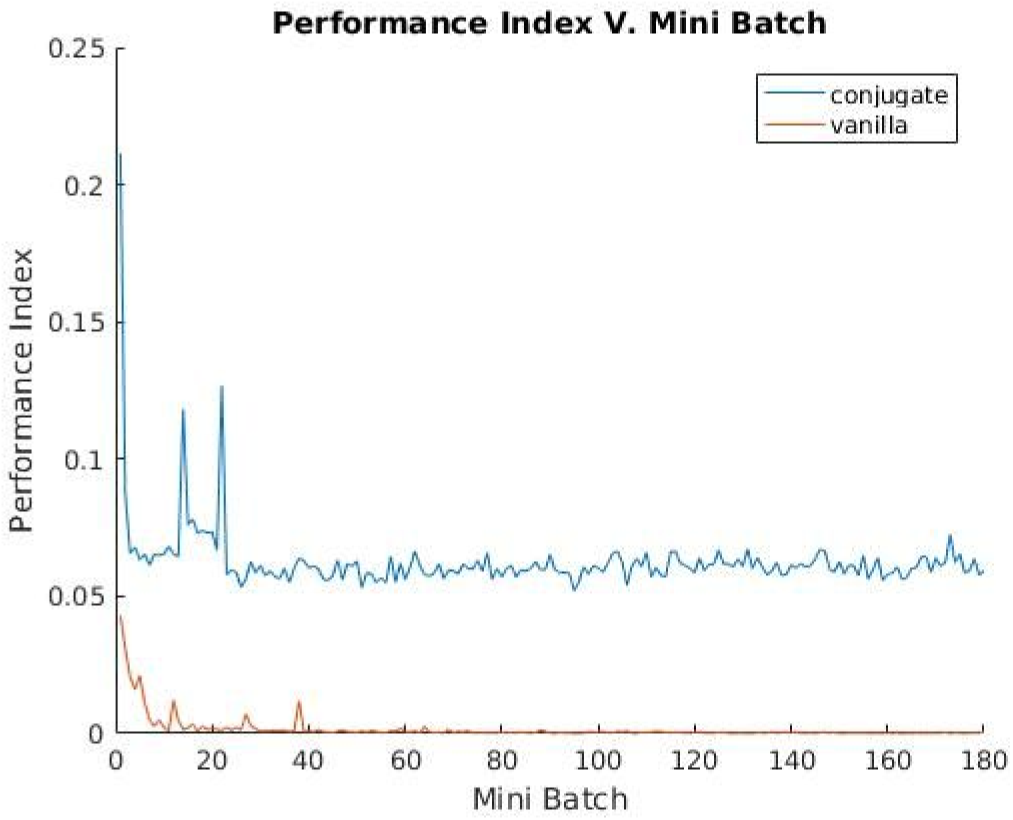
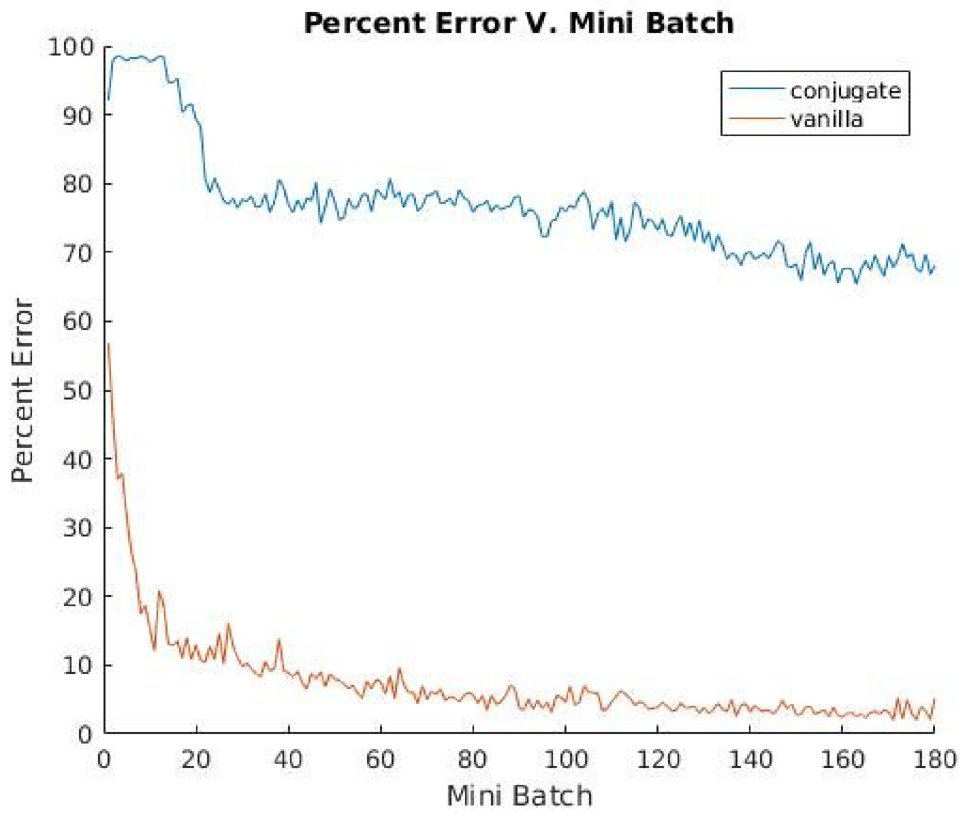


Figure 29: Size 1000 Percent Error



Test Set Performance

Data Summary

Figure 30: Size 100 Test Set Performance

Metric	Conjugate Gradient	Vanilla
Performance Index	2.5	1.809 e-4
Percent Error	100	1

Figure 31: Size 300 Final Results

Metric	Conjugate Gradient	Vanilla
Performance Index	0.0230	1.8249 e-4
Percent Error	30	2.33

Figure 32: Size 600 Final Results

Metric	Conjugate Gradient	Vanilla
Performance Index	0.0499	6.2854 e-4
Percent Error	62.33	5.1667

Figure 33: Size 1000 Final Results

Metric	Conjugate Gradient	Vanilla
Performance Index	0.0616	7.2983 e-4
Percent Error	68.1	5.2

Conjugate Gradient Test Set Performance

Figure 34: Conjugate Gradient Performance Index V. Batch Size

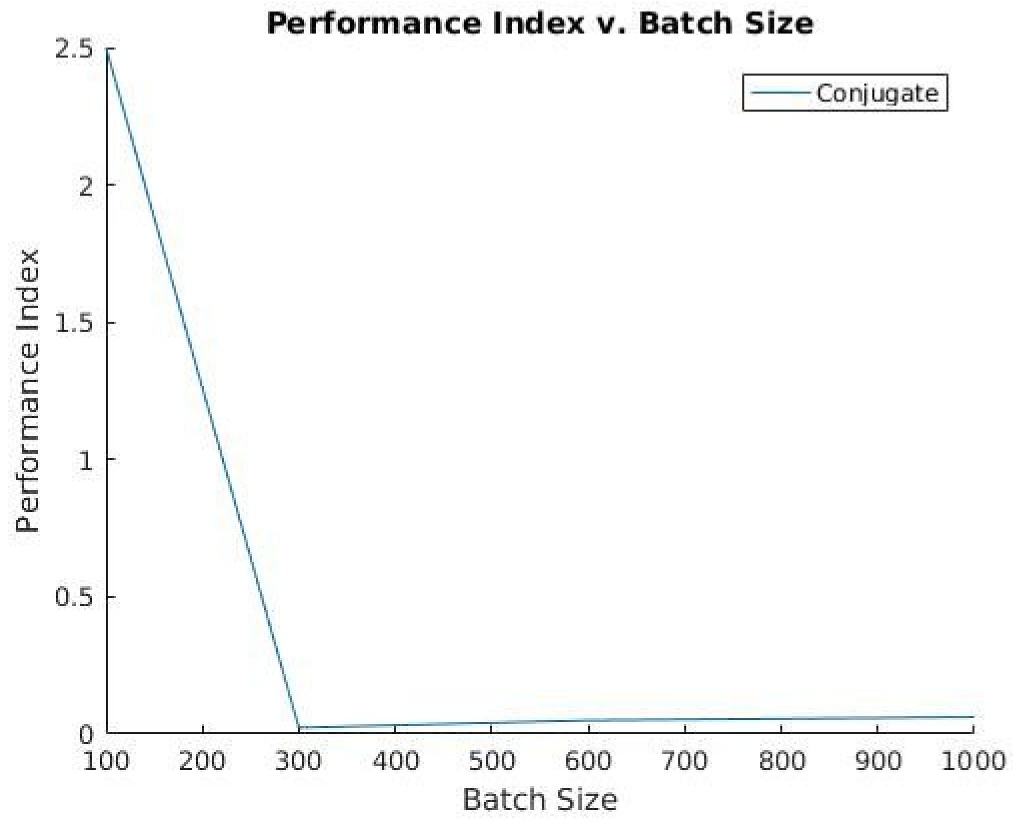
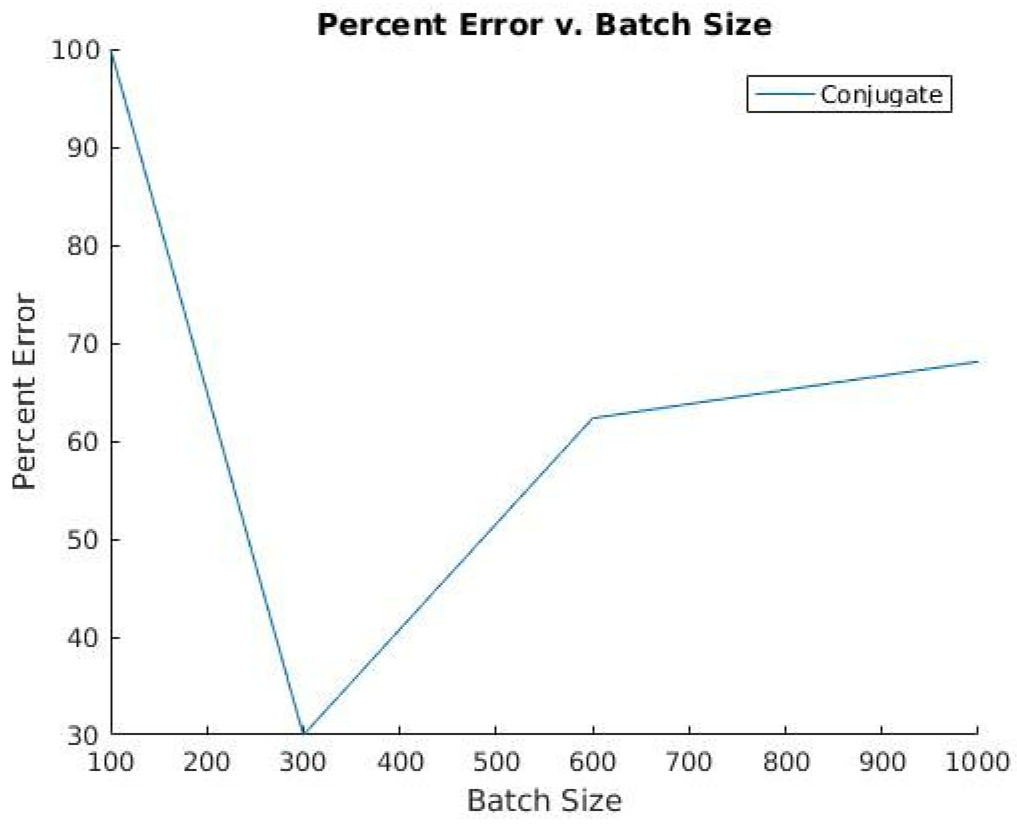


Figure 35: Conjugate Gradient Percent Error V. BatchSize



Vanilla Backpropagation Test Set Performance

Figure 36: Vanilla Backpropagation Performance Index V. Batch Size

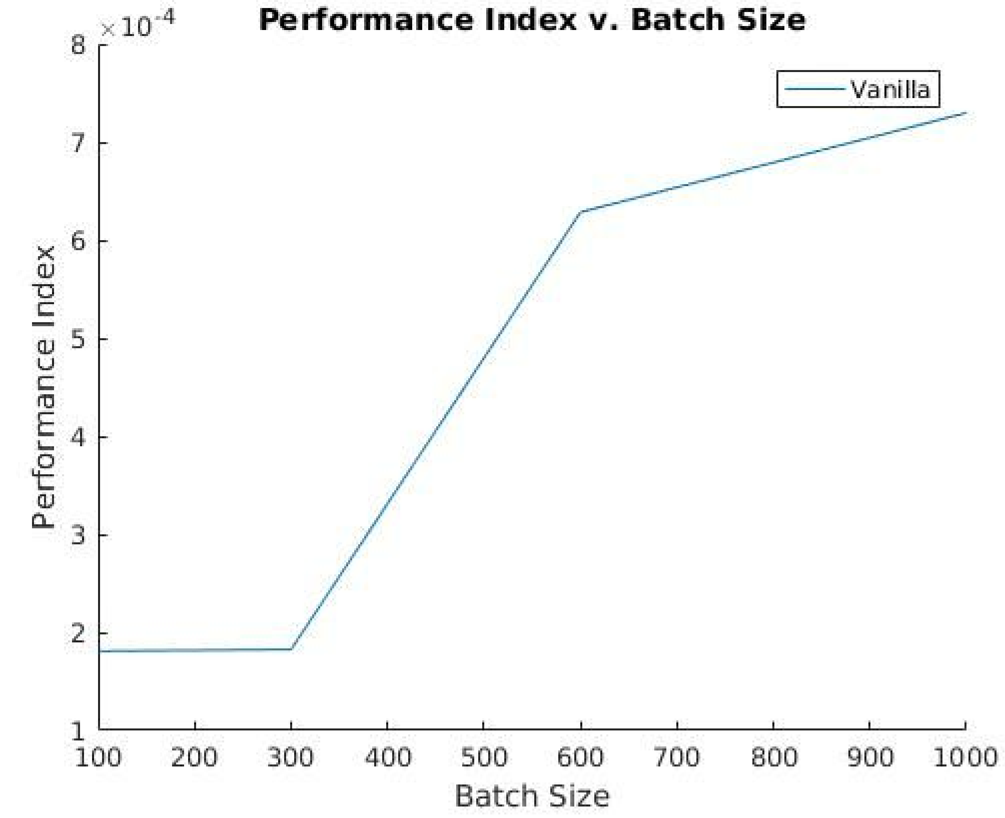
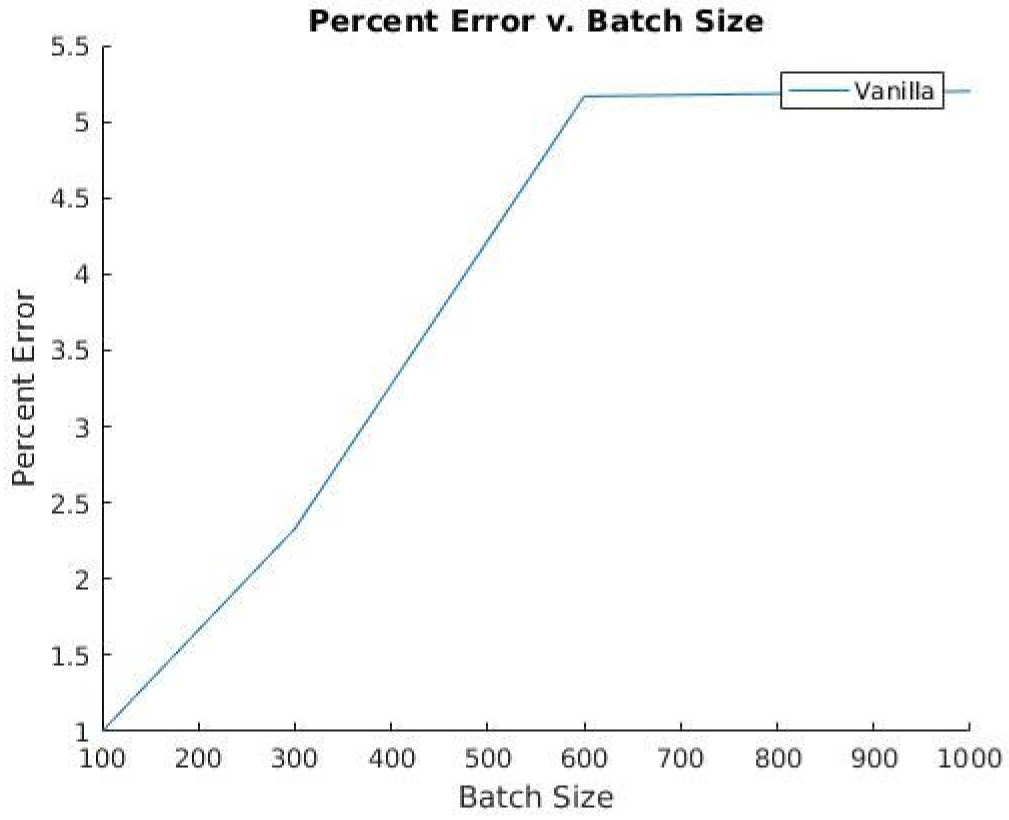


Figure 37: Vanilla Backpropagation Percent Error V. BatchSize



Conclusion

Let’s examine the data from two perspectives, one examining through the lens of training performance, and one through generalization. Then finally talk about what we learned and what we can draw from these data.

Training Performance

So let’s first discuss the vanilla backpropagation performance so we have something to compare to. Vanilla backpropagation fared well with this data set and architecture. It’s best run had a 1% failure rate on the test set. As far as convergence, it seemed to reach some value very close to a final minimum, before the end of the 1st epoch, in each run, then performance index seemed to hover around similar values. Figures 23-29 all show this behaviour This is what we expect compared to previous assignments, and the convergence behaviour we saw there.

Conjugate gradient had some difficulty training with certain batch sizes. On a batch size of 100 as shown in figures 23 -24, this algorithm very clearly diverged in a way that it could not recover from at approximately batch 1000. On batch size of 300, the algorithm had the best training performance. It generalized to a failure percent of 30 on the test set, and figures 25 & 26 show a pattern that looks like convergence. On a batch sizes of 600 and 1000 it seems to exhibit similar behavior. It reaches some performance index very quickly, and tends to hover around it for some time without a clear convergence trend. Figures 27 -29 all exhibit this trend.

Generalization Performance

Vanilla backpropagation generalization seemed to do slightly worse as the size of the mini batch increased. Figures 36 and 37 exhibit this behavior In theory this doesn’t make sense, because batch size does not factor into the algorithm. Either I have a bug, or it’s simply chance. However, it seems to get enough right about the training set to consider it credible enough to discuss.

Conjugate Gradient did not have stellar generalization performance. It only seemed to converge on one of the batch sizes, and when it did it was only to 30% failure rate. This is an order of magnitude larger than vanilla backpropagation, so it is pretty safe to say this particular implementation performed worse.

Final Thoughts

The purpose of this assignment was to tackle a large data set, and apply what we learned in this class to a large data set, and pick a particular topic in neural networks to explore deeply. After 60 hours of tuning and modifying my conjugate gradient algorithm best I have been able to capture is a failure rate of 30% on the test set. Given more time, or a little more exposure to linear algebra I would have liked to train this algorithm to be more accurate.

What we can observe is that conjugate gradient happened to do better when the number of elements in the batch was larger than the number of neurons in the hidden layer. In fact it seemed to do best when the number of inputs matched exactly. One could vary the number of neurons on the hidden layer and reproduce this experiment to verify this, however this goes beyond the scope of this paper.

There are some other observations we can draw from this, is that maybe a more sophisticated network architecture, or more sophisticated modifications might be needed to the conjugate gradient algorithm to make performance more satisfactory. Things like momentum, or an adaptive interval expansion strategy might also improve performance. This is purely qualitative, but from I observed when watching conjugate gradient train was that it would do smalls runs of mini batches where would make a bunch of small performance improvements and there would be a marked decrease in error. Then after this run of improvements it would not update the network for several batches. This means that the search algorithm could only find minima intermittently with the given gradient.

There are problems with the algorithm provided. Sometimes it would overlook minima because the search interval grows very fast. Sometimes if growth was left unchecked with interval selection it would cause the algorithm to diverge.So this tells me I might have been better off using a different interval selection algorithm. However that goes beyond the scope of this paper. What we can draw is that my current implementation needs some improvement or architectural change.

Sources

[1] "MNIST handwritten digit database, Yann LeCun, Corinna Cortes and Chris Burges", *Yann.lecun.com*, 2017. [Online]. Available: http://yann.lecun.com/exdb/mnist/. [Accessed: 05- Dec- 2017].

[2] "Using the MNIST Dataset - Ufldl", *Ufldl.stanford.edu*, 2017. [Online]. Available: http://ufldl.stanford.edu/wiki/index.php/Using_the_MNIST_Dataset. [Accessed: 05- Dec- 2017].

[3] M. Hagan, H. Demuth, M. Beale and O. De Jesús, *Neural network design*. [S. l.: s. n.], 2016, pp. Ch. 12 P. 14-19.

[4] "Conjugate gradient method", *En.wikipedia.org*, 2017. [Online]. Available: https://en.wikipedia.org/wiki/Conjugate_gradient_method. [Accessed: 04- Dec- 2017].

[5] J. Mcaffrey, "Understanding Neural Network Batch Training: A Tutorial -- Visual Studio Magazine", *Visual Studio Magazine*, 2017. [Online]. Available: https://visualstudiomagazine.com/Articles/2014/08/01/Batch-Training.aspx?Page=2. [Accessed: 04- Dec- 2017].

[6] S. Ruder, “An overview of gradient descent optimization algorithms,” Sebastian Ruder, 03-Dec-2017. [Online]. Available: http://ruder.io/optimizing-gradient-descent/index.html#batchgradientdescent. [Accessed: 04-Dec-2017].

Appendix A: Complete Source Code

Here is the complete nitty gritty source code for the project. I ranked it in order of what considered important. I did not include the loadMNIST image functions written by stanford, as they are work I am citing not producing. [2] There are also some mundane plotting scripts that I considered context specific and didn't consider important enough to include in the report. If you want to see or use the code, it's on github for public use. https://github.com/collinsc/NeuralNet

HW5.m

```
%reset system state
clear
close all
disp('Load Training Set :')
disp('    loading images')
P_o = loadMNISTImages('./data/train-images-idx3-ubyte');
disp('    loading labels')
%transform to binary output
T_o = formatOutput(loadMNISTLabels('./data/train-labels-idx1-ubyte'));
disp('    getting sets')

%get an order of samples for the batches
batchSize =100;
epochs = 3;
%stuff for plotting
xrange = 1:((size(P_o,2)*epochs)/batchSize);
cidx = zeros(size(xrange));
cerr = zeros(size(xrange));
verr = zeros(size(xrange));
vidx = zeros(size(xrange));
plotIdx = 1;
iteration = 0;
disp('    generating weights')
h1 = 300;     %neurons in hidden layer 1
[W1,b1,W2,b2] = getWeights(P_o,T_o, h1);
cW1 = W1; cb1 = b1; cW2 = W2; cb2 = b2;
vW1 = W1; vb1 = b1; vW2 = W2; vb2 = b2;
disp('    generating initial gradient')
idx = randperm(size(P_o,2));
P1 = P_o(:,idx(1:(1+batchSize)-1));
T1 = T_o(:,idx(1:(1+batchSize)-1));
%get initial gradient
[gW1, gb1, gW2, gb2] = accumulateGradients(W1, b1, W2, b2, P1, T1);
pW1 = -gW1;    pb1 = -gb1;
pW2 = -gW2;    pb2 = -gb2;
```

```
disp('Training:')
count = 0;
for i = 1:epochs
    idx = randperm(size(P_o,2));

    for batch = 1:batchSize:size(P_o,2)   %for each mini batch
        P1 = P_o(:,idx(batch:(batch+batchSize)-1));
        T1 = T_o(:,idx(batch:(batch+batchSize)-1));
        %train on each strategy
        [ cW1, cb1,  cW2,  cb2, ...
           pW1,pb1, pW2, pb2, ...
           gW1, gb1, gW2, gb2, ...
           iteration,cid] = ...
                        conjugateTrain( P1, T1, ...
                                        cW1, cb1, cW2, cb2, ...
                                        pW1, pb1, pW2, pb2, ...
                                        gW1,gb1, gW2, gb2, ...
                                        iteration, true);
        cidx(plotIdx) = cid;
        [vW1,vb1,vW2,vb2, vid] = vanillaTrain(   P1, T1, ...
                                                    vW1, vb1, ...
                                                    vW2, vb2, ...
                                                    true);
        vidx(plotIdx) = vid;
        %test our modifications
        evaluate1 = @(x) evaluate(cW1,cb1,cW2,cb2, x);
        cerr(plotIdx) = getPercError(evaluate1,P1,T1);
        evaluate2 = @(x)  evaluate(vW1,vb1,vW2,vb2, x);
        verr(plotIdx) = getPercError(evaluate2,P1,T1);
        count = count +1;
        fprintf('    batch %i\n\tconjugate error percent: %f\n\tvanilla error percent: 
%f\n', count,cerr(plotIdx),verr(plotIdx));
        plotIdx = plotIdx + 1;
        %let's save our data
    end
end
%evaluate network performance
disp('Load Test Set :')
disp('    loading test images')
P2 = loadMNISTImages('./data/t10k-images-idx3-ubyte');
disp('    loading test labels')
T2 = formatOutput(loadMNISTLabels('./data/t10k-labels-idx1-ubyte'));
evaluate1 = @(x) evaluate(cW1,cb1,cW2,cb2, x);
err1 = getPercError(evaluate1,P1,T1);
evaluate2 = @(x)  evaluate(vW1,vb1,vW2,vb2, x);
err2 = getPercError(evaluate2,P1,T1);
disp('Run Test Set Conjugate Gradient:')
```

```
fprintf('    percent error: %f\n',err1);
disp('Run Test Set Steepest Descent:')
fprintf('    percent error: %f\n',err2);
disp('run complete')
plot(xrange, cerr, xrange, cidx, xrange, verr, xrange, vidx)
```

conjugateTrain.m

```
function [ W1, b1,  W2,  b2, ...
           pW1n,pb1n, pW2n, pb2n, ...
           gW1n, gb1n, gW2n, gb2n, ...
           iteration, index] = ...
                        conjugateTrain( P, T, ...
                                        W1, b1, W2, b2, ...
                                        pW1, pb1, pW2, pb2, ...
                                        gW1,gb1, gW2, gb2, ...
                                        iteration, isGraph)
%important constants for training

epsilon = 0.001;         %rate to increase search interval
startingRate =  0.000;     %minimum jump
%reset search direction every n iterations
R = size(P,1);
s1 = size(W1,1);
s2 = size(W2,1);
resetPeriod = R*s2 + s2 + s2*s1 + s1;
%select an interval to minimize
[a, b] = getInterval( @(rate) perfIndex(    W1 + rate*pW1, ...
                                b1 + rate*pb1, ...
                                W2 + rate*pW2, ...
                                b2 + rate*pb2, ...
                                P,T), ...
                                startingRate, epsilon);
%minizize the interval to tolerance
rate = golden(@(rate) perfIndex(    W1 + rate*pW1, ...
                                b1 + rate*pb1, ...
                                W2 + rate*pW2, ...
                                b2 + rate*pb2, ...
                                P,T),...
                                a, b);
%conjugate gradient weight update
W1 = W1 + rate*pW1;     b1 = b1 + rate*pb1;
W2 = W2 + rate*pW2;     b2 = b2 + rate*pb2;
%get error
index = perfIndex(  W1 + rate*pW1, ...
                    b1 + rate*pb1, ...
                    W2 + rate*pW2, ...
                    b2 + rate*pb2, ...
                    P, T);
if isGraph
    fprintf('\tconjugate performance index: %f,    learning rate: %f\n',index, rate)
end
[gW1n, gb1n, gW2n, gb2n] = accumulateGradients( W1, b1, W2, b2, P, T );
iteration = iteration +1;
%get directional gain
if mod(iteration, resetPeriod) == 0
    bW1 = 0;    bb1 = 0;
    bW2 = 0;    bb2 = 0;
else
    %decide how much of the new gradient to "mix" based on magnitude
    %uses fletcher reeves update
    bW1 = getGains(gW1n, gW1);    bb1 = getGains(gb1n, gb1);
    bW2 = getGains(gW2n, gW2);    bb2 = getGains(gb2n, gb2);
end
%calculate new directions
pW1n = -gW1n + pW1*bW1;    pb1n = -gb1n + pb1*bb1;
pW2n = -gW2n + pW2*bW2;    pb2n = -gb2n + pb2*bb2;
end
```

vanillaTrain.m

```
function [W1,b1,W2,b2, index] = vanillaTrain(P,T,W1,b1,W2,b2,isPlot)
Q = size(P,2);
%important constants for training
rate = 0.09;
for j = 1:Q
    [ gW1, gb1, gW2, gb2] = getGradients(W1, b1, ...
                                                    W2, b2, ...
                                                    P(:,j),T(:,j) );
    W1 = W1 - rate*gW1;    b1 = b1 - rate*gb1;
    W2 = W2 - rate*gW2;    b2 = b2 - rate*gb2;
end
index=  perfIndex(W1,b1,W2,b2,P,T);
if isPlot
    fprintf('\tvanilla performance index: %f\n', index)
end
end
```

getInterval.m

```
function [a,b]  = getInterval(f,a,ep)
```

```
max_itr = 10;
max_learn = 3;
f_a = f(a);
int = ep;
dec = false;
old = a;
orig = f_a;
for i = 1:max_itr
    b = a + int;
    f_b = f(b);
    if b >= max_learn
      b = a;
      a = old;
      break
    end
    if f_a <= f_b
        if dec
            a = old;
            break
        end
    else
        dec = true;
        old = a;
        a = b;
        f_a = f_b;
    end
    int = int *2;
end
end
```

golden.m

```
function out = golden(f,a, b)
tau = 0.618;
tol = 0.0005;
c = a + (1 - tau)*(b - a);
f_c = f(c);
d = b - (1 - tau)*(b - a);
f_d = f(d);
while abs(b - a) > tol
    if (f_c < f_d)
        b = d;
        d = c;
        c = a + (1 - tau)*(b - a);
        f_d = f_c;
        f_c = f(c);
    else
        a = c;
        c = d;
        d = b - (1 - tau)*(b - a);
        f_c = f_d;
        f_d = f(d);
    end
end
out = a;
end
```

getGradients.m

```
function [ gW1, gb1, gW2, gb2] = getGradients( W1, b1, W2, b2, A0, T )
        %get the first layer output
        A1 = logsigmoid(W1*A0 + b1);
        %get the second layer output
        A2 = logsigmoid(W2*A1 + b2);
        %error
        E = T - A2;
        %back propigation
        S2 = -2*diag((1 - A2).*A2)*E;
        S1 = diag((1 - A1).*A1)*(W2'*S2);
        %accumulate normalized gradients
        gW1 = S1*(A0');
        gb1 = S1;
        gW2 = S2*(A1');
        gb2 = S2;
end
```

accumulateGradients.m

```
function [ gW1, gb1, gW2, gb2] = accumulateGradients( W1, b1, W2, b2, P, T )
gW1 = zeros(size(W1,1),size(P,1));    gb1 = zeros(size(b1,1),1);
gW2 = zeros(size(W2,1),size(W1,1));    gb2 = zeros(size(b2,1),1);
%count of elements in training set
Q = size(P,2);
%initialize gradients for first iteration
for j = 1:Q
    [gW1t, gb1t, gW2t, gb2t] = getGradients(    W1, b1, ...
                                                W2, b2, ...
                                                P(:,j),T(:,j));
    %accumulate an average gradient
    gW1 = gW1 + gW1t/Q;     gb1 = gb1 + gb1t/Q;
    gW2 = gW2 + gW2t/Q;     gb2 = gb2 + gb2t/Q;
end
```

```
end
```

getWeights.m

```
function [W1,b1,W2,b2] = getWeights(P,T, h1)
R = size(P,1);
S1 = h1;
S2 = size(T,1);
%get randomized weights and biases
maxMag = 0.5;
minMag = 0.2;

W1 = rangedRand(minMag, maxMag, S1, R);
b1 = rangedRand(minMag, maxMag, S1, 1);
W2 = rangedRand(minMag, maxMag, S2, S1);
b2 = rangedRand(minMag, maxMag, S2, 1);
end
```

getGains.m

```
function beta = getGains(gn,g)
sgn = accumulate(gn,size(gn,2)); sg = accumulate(g,size(g,2));
beta = dot(sgn,sgn)/dot(sg,sg);
end
```

getPercError.m

```
 function err = getPercError(f,P,T)
A = round(f(P));
errCount = 0;
for i = 1:size(P,2)
    if sum((T(:,i)-A(:,i)).^2)> 0
        errCount = errCount + 1;
    end
end
err = errCount./size(T,2)*100;
end
```

perfIndex.m

```
function index = perfIndex(W1, b1, W2, b2, P, T)
E = accumulate((T - evaluate(W1,b1,W2,b2,P)),size(P,2));
index = dot(E,E);
end
```

rangedRand.m

```
function out = rangedRand(magLow, magHigh, R, C)
out = magLow + (magHigh - magLow) * rand(R,C);
for i =1:size(out,1)
    for j = 1:size(out,2)
       if round(rand) == 1
           out(i,j) = out(i,j)*-1;
       end
    end
end
end
```

Logsigmoid.m

```
function out = logsigmoid(x)
out = 1./(1+exp(-x));
end
```

formatOutput.m

```
function [ T_n ] = formatOutput( T_o )
T_n = zeros(10,size(T_o,1));
lookup= diag(ones(1,10));
getCode = @(x) lookup(x,:);
for i = 1:size(T_o,1)
   T_n(:,i) = getCode(T_o(i)+1);
end
end
```

evaluate.m

```
%evaluate the 2 layer logsig neural network
function [ out ] = evaluate( W1, b1, W2, b2, x)
out = zeros(size(b2,1),size(x,2));
for i = 1:size(x,2)
    out(:,i) = logsigmoid(W2*logsigmoid(W1*x(:,i)+b1) + b2);
end
```

accumulate.m

```
function out = accumulate(x,y)
out = sum(x,2)/y;
end
```