Training Deep Learning Algorithms with PMBSolve

Ang Li	Biying Wu	Jeroen Vlak	Patrick van der Reijden	Peer van Paasen
431373	430135	432045	430530	433940

Bachelor Seminar in Business Analytics and Quantitative Marketing
Erasmus School of Economics, Erasmus University Rotterdam

Abstract

At the moment, Stochastic Gradient Descent (SGD) is mostly used as optimization method for neural networks due to its ease of implementation. In this paper we discuss the implementation and performances of a new optimization method, the Preconditioned Model Building Solver (PMBSolve). We also introduce Minibatch PMBSolve, where we use PMBSolve on small batches of the training set. Moreover, we combine PMBSolve and Minibatch PMBSolve as a candidate method and call this combination PMBSolve with warm-up. To compare these different methods, we run different experiments in a feed-forward two-layered neural network using the MNIST dataset and an artificial credit rating dataset. We find that PMBSolve performs worse than SGD for the MNIST dataset, but better for a smaller dataset. PMBSolve with warm-up is the best optimization algorithm in terms of accuracy rate and reliability among all proposed optimization algorithms. This paper shows the advantages and downsides of PMBSolve in deep learning and proposes a good alternative optimization method PMBSolve with warm-up for deep learning.

Keywords: deep learning; two-layer neural network; feed-forward neural network; preconditioned model building solver (PMBSolve); stochastic gradient descent; backpropagation

1 Introduction

Some famous applications of deep learning, which are used to train computer programs to recognize different patterns, have been widely applied to various tasks in different industries. A crucial step in training such a deep learning network is to find the optimal weights for the connections between individual units. Hence, different optimization methods have been advocated for the training process. At the moment, stochastic gradient descent (SGD) method with the incorporation of error backpropagation has been widely used due to its ease of implementation. However, SGD converges very slowly in terms of iterations and requires prior knowledge or a large amount of manual tunning to get the appropriate optimization parameters. Thus, other optimization methods need to be considered to support the training process. In our research, we aim to employ two alternative optimization methods, the Preconditioned Model Building Solver (PMBSolve) and Minibatch PMBSolve for deep learning and compare their performances with the minibatch implementation of the SGD method.

A common way to optimize the weight parameters in different neural network layers is minimizing a predefined error function with respect to the weights. This optimization step is one of the most crucial elements in network training for two reasons: First, this optimization step directly influences the prediction accuracy of the neural network as the outputs are determined by the inputs and the weight parameters. Second, this optimization step determines the efficiency of the algorithm because the rate of convergence in this step influences the total training time dramatically. Therefore, investigating the performances of different optimization methods in training deep learning is of high relevance to increase the prediction power of the algorithms and speed up the training process.

In this research, we discuss the implementation of PMBSolve proposed by Öztoprak and Ş. İlker Birbil [2018]. The fundamental idea for this strategy is to use extra information gathered by multiple trial points to come up with an approximation of the objective function as a better model in each iteration. The ultimate goal of this research is to compare performances of PMBSolve and Minibatch PMBSolve against SGD in deep learning training. Hence, we focus on solving two research questions to achieve our goal. The first one is how to implement the PMBSolve in training the standard neural network. The other one is that, regarding the convergence rate and overall efficiency, how PMBSolve performs compared with the commonly used SGD optimization method, and what the advantages and disadvantages of PMBSolve are in such training process.

To answer our research questions, we consider a standard neural network for solving classification tasks. The standard neural network is a feed-forward network with an input layer, one hidden layer and an output layer. We incorporate the logistic sigmoid function as the activation function to match the actual input to its corresponding output for every unit in the hidden and output layer. With this setup, we conduct a supervised training, which in this context, refers to finding the optimal weights in connecting every unit to the units in the succeeding layer with specific input and target values. Finally, we use the Bayes' decision rule for pattern classification. Our standard neural network is an ideal case for us to experiment the performance of PMBSolve versus SGD because it is a basic model for solving classification tasks with deep learning. The SGD method with backpropagation rule and the PMBSolve method are implemented to minimize the error function in the training. In addition, as the computational costs for PMBSolve can be high for large datasets, we also propose the PMBSolve with minibatch and study its performance on a large dataset. We use the same number of hidden units and batch size for each method and compare the performances of the three methods based on the prediction accuracy and the running time.

To obtain a solid conclusion on the performance of PMBSolve specifically, we use two different datasets. Particularly, we use the MNIST dataset that trains the algorithm to recognize handwritten images and an artificial credit rating (CREDIT) dataset to train the algorithm to

assign individuals to different credit classes. The two datasets vary hugely in terms of the size of attributes and the number of instances, which allows us to investigate the performance of PMBSolve versus SGD for deep learning in different conditions.

We refer to the definition to neural networks and other related knowledge in Stutz [2014] where it implements SGD as the optimization algorithm for the neural network for recognizing handwritten images in the MNIST dataset. The optimization method we focus on, PMBSolve, is a new optimization algorithm that makes its debut in the literature by Öztoprak and Ş. İlker Birbil [2018]. Le et al. [2011] do similar comparisons for SGD, Limited memory Broyden-FletcherGoldfarbShanno (L-BFGS) and Conjugate Gradient (CG) based on the same dataset as that of Stutz [2014].

Our main finding in this research is that, unless the dataset and neural network are sufficiently small, PMBSolve is not a good alternative optimization method for deep learning applications in terms of efficiency. This becomes increasingly apparent as either the neural network or dataset increases in size, which is normal for deep learning applications. Although for large scale deep learning problems, PMBSolve can give higher accuracy than SGD with a good starting point but this cost much longer running time and when the starting point is bad, PMBSolve fails. In addition to this result, we find that Minibatch PMBSolve rarely uses the inner iteration and second order information of the error function. Moreover, to reduce the chances for PMBSolve to start with a bad point and fail, we find the method PMBSolve with warm-up. This method first uses Minibatch PMBSolve to do a preparatory training then continue with PMBSolve, which lets PMBSolve get a fairly good starting point and give higher accuracy rate than SGD. Our results prove that this method is the best optimization method in terms of accuracy rate and reliability among all proposed optimization algorithms. Thus, PMBSolve with warm-up is a good alternative optimization algorithm for deep learning. Therefore, our research contributes the deep learning study through the proposal of the good alternative optimization method PMBSolve with warm-up.

In section 2, we give a brief introduction to neural networks. Then we introduce the previous study about different optimization algorithms and the relevance with our research. In section 3, we give a detailed introduction to the structure of our neural network. Then we determine the activation function, training method and pattern classification respectively. We focus on methodology in section 4. In its first subsection, the objective function of all optimization algorithms, the cost function, is formulated, which is followed by initialization of weights in next subsection. In the other subsections, we provide explanations to SGD, Backpropagation, PMBSolve and Minibatch PMBSolve in more detail. Section 5 concerns the procedure of implementing simulations including preparation of data, setup for simulations and numerical results of the simulations. Our conclusion and further discussion are found in section 6.

2 Related Literature

The first neural network training dates back to the work of Rosenblatt [1958] that presents a single-layer network training. After that, Minsky and Papert [1969] point out the limitation of a single-layer network and then the multi-layer network is proposed. Today, a large amount of research is done by using a two-layer network. As an example, Stutz [2014] mentions that theoretically, two-layer networks can model any continuous function using an appropriate activation function, and it gives a thorough explanation of a standard two-layer network.

In neural network training, the optimization step is one crucial step. For that reason, many related works have been done to study the performances of different optimization methods. When it comes to large deep learning problems, SGD is the most extensively considered method. With multiple training experiments using MNIST dataset, Le et al. [2011] conclude the prominent advantages of SGD are the simple implementation and fast convergence for problems with many training examples. However, a significant drawback of SGD is that it requires

manually choosing optimization parameters such as learning rate, momentum coefficient and convergence criteria. A standard framework to tackle such problem is to run the training algorithm with all possible parameters and pick the model that gives the best performance. But Le et al. [2011] also point out this has poor efficiency.

Another popular family of optimization methods for training neural networks is the Quasi-Newton family. Le et al. [2011] propose the use L-BFGS with line search in parameter optimization. The study confirms the superior ability of L-BFGS to give a fast convergence for low dimensional problems with large datasets. It also concludes L-BFGS converges even faster with the incorporation of line search. However, L-BFGS performs badly for low-rank estimates when the Hessian cannot be well approximated. Therefore, Le et al. [2011] also introduce CG with line search, which is an efficient algorithm for training high dimensional problems. In general, the study concludes that L-BFGS and CG are much more efficient compared with SGD in many cases, especially when the number of parameters are relatively small.

For our research we consider a newly proposed optimization algorithm, PMBSolve. This globalization strategy is introduced in the literature by Öztoprak and Ş. İlker Birbil [2018] to solve any unconstrained optimization problem. The main idea of this new strategy is to use multiple trial points to collect additional information and build a proper local model of the objective function in each iteration. Before making a step, PMBSolve first collects the information of different trial points to improve both the length and the direction of the step. The promising performance of PMBSolve has been confirmed when solving a set of unstrained optimization problems of CUTEst collection. In Öztoprak and Ş. İlker Birbil [2018], the PMBSolve method is tested and compared with two line search routines, namely Backtracking and More-Thuente line search. From the numerical test results, PMBSolve outperforms Backtracking and More-Thuente line search in the sense that it attains the lowest objective function value after three inner iterations. Furthermore, the numerical test shows that direction updates of PMBSolve indeed increases the efficiency as it requires fewer number of function evaluations.

In our research, we aim to investigate whether the conclusions of SGD and PMBSolve methods from the previous literatures also apply to our problem that is training a supervised feed-forward two-layered network. In addition, our ultimate goal is to apply PMBSolve and Minibatch PMBSolve to the network training algorithm, study the performance of both PMBSolve and Minibatch PMBSolve on supervised deep learning and investigate whether these new globalization strategies outperform the SGD method.

3 Preliminaries

In this section, we present the preliminary knowledge. We first provide a comprehensive introduction to the standard neural network that is used to investigate the performances of different optimization methods. Next, we present the activation function that is incorporated. Finally, we give training details and the pattern classification method.

3.1 A Standard Neural Network

An artificial neural network, hereinafter referred to as neural network, has a high similarity to the biological neural network. A neural network consists of a set of interconnected processing units that are similar to neurons interconnected by synapses in the human brain. Specifically, there are two important sets of units, the input units set and the output units set. For the rest of this research, we call these two sets the input layer and output layer respectively.

The input layer receives inputs from external sources, computes the outputs of this layer and propagates the outputs to other layers. Eventually, it reaches the output layer and gives the final output values. This process is similar as a function in which the corresponding output value can be computed given a certain input value. Therefore, the neural network models an

unknown function $y = f(x)$ where its input dimension equals the size of the input layer and its output dimension equals the size of the output layer.

We visualize the neural network using network graphs and introduce two basic network topologies: feed-forward and recurrent networks. Feed-forward means that a unit can only propagate its output to a unit in the layer after the current one it is in. Conversely, recurrent networks allow the existence of a closed cycle within the network graph.

In this research, we only consider a feed-forward neural network. We use the network to model a classification problem. That is, given some input values, the neural network can map the inputs to a certain class. To illustrate how our neural network works in more detail, we first consider a simple perceptron introduced by Rosenblatt [1958]. Rosenblatt's perceptron can be viewed as a binary classifier that decides whether an input vector belongs to a specific class or not. Such a perceptron is a single-layer network which contains an input layer and an output layer only. As an example, Figure 1a shows a single-layer perceptron with D input units and C output units. Each input unit is connected to every output unit.

The process of the propagation is as the follows: Every input unit first receives an input value x from the outside. Then a propagation rule generates the actual input to be used in next layer by distributing different weights to every input unit. In this research, we use the weighted sum propagation rule for all units except the input units. This propagation rule gives the c^{th} actual input unit z_c as

$$z_c = \sum_{d=1}^{D} w_{cd} x_d \quad for\ 1 \leq c \leq C, \tag{1}$$

where D denotes the number of input units, C denotes the number of output units, x_d is the d^{th} input unit and w_{cd} represents the weight of x_d while being mapped to z_c. After that, an activation function f uses the actual input z_c to calculate the c^{th} output $y_c = f(z_c)$ in the output layer.

The single-layer network is the simplest network to apply. However, in the literature by Minsky and Papert [1969], they point out the limitation of single-layer network to model Exclusive-OR(XOR) problems. To overcome such a limitation, Stutz [2014] argues that a two-layer perceptron, with an additional hidden layer in between the input and output layers, can model any problems with continuous functions. Therefore, we also use a two-layer perceptron in this research. Figure 1b illustrates our two-layer network with D input units in the input layer, M hidden units in the hidden layer and C output units in the output layer. Let $y_m^{(1)}$ denotes the output of the m^{th} hidden unit in the hidden layer and $y_c^{(2)}$ the output of the c^{th} output unit in the output layer. Then, the output value $y_m^{(1)}$ of the m^{th} hidden unit is given by

$$y_m^{(1)} = f\left(\sum_{d=1}^{D} w_{md}^{(1)} x_d\right) \quad for\ 1 \leq m \leq M \tag{2}$$

and the output value $y_c^{(2)}$ of the c^{th} output unit is given by

$$y_c^{(2)} = f\left(\sum_{m=1}^{M} w_{cm}^{(2)} y_m^{(1)}\right) \quad for\ 1 \leq c \leq C, \tag{3}$$

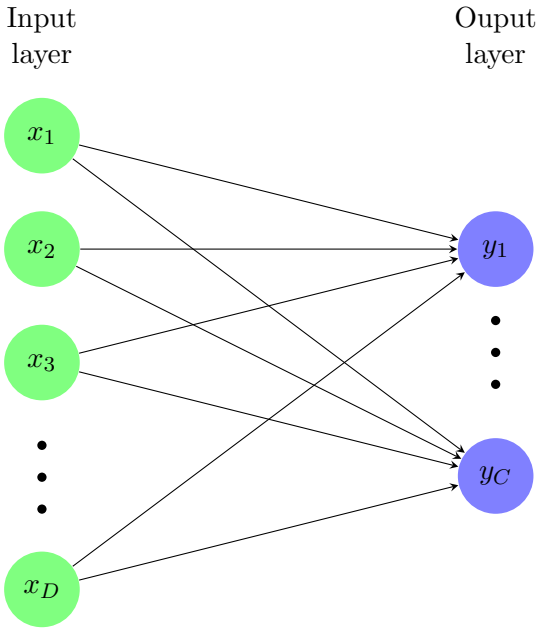
where $w_{md}^{(1)}$ represents the weight of x_d while being mapped to $y_m^{(1)}$ and $w_{cm}^{(2)}$ represents the weight of $y_m^{(1)}$ while being mapped to $y_c^{(2)}$.

To simplify subsequent discussions, we introduce the following notation for the modeled

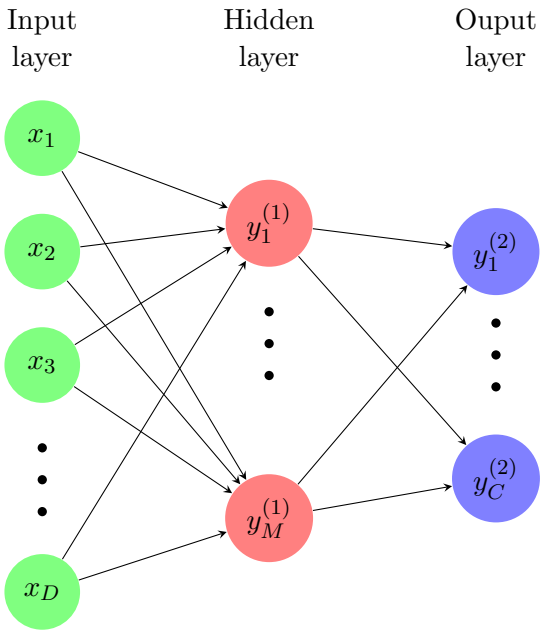
function, which is given by

$$y(\cdot, w) : \mathbb{R}^D \to \mathbb{R}^C, x \mapsto y(x, w) = \begin{pmatrix} y_1(x, w) \\ \vdots \\ y_C(x, w) \end{pmatrix}, \tag{4}$$

where x is a D-dimensional input vector, w is a weight vector comprising of all weights and $y_c(x, w) := y_c^{(2)}(x, w)$ is calculated following equation (2) and (3).



(a) A single-layer neural network with D input units and C output units, each unit in the input layer is connected to all the units in the output layer, but units within the same layer are not connected. Here, x_d, $d = 1, ..., D$ denote the input values and y_c, $c = 1, ..., C$ denote the output values of the output layer.



(b) A two-layer neural network with D input units, M hidden units and C output units. Here, x_d, $d = 1, ..., D$ denote the input values, $y_m^{(1)}$, $m = 1, ..., M$ denote the output values in the hidden layer, and $y_c^{(2)}$, $c = 1, ..., C$ denote the output values of the output layer.

Figure 1: Standard Neural Network Graphs

3.2 Activation Function

At the core of a neural network lies an activation function which converts actual input units into output units that are input units for the next layer. One type of commonly used activation functions is a threshold function Haykin [1999], for example, the heaviside function

$$h(z) = \begin{cases} 1, & \text{if} \quad z \geq 0; \\ 0, & \text{if} \quad z < 0. \end{cases} \tag{5}$$

In this research, we use a smooth version of the heaviside function, the logistic sigmoid function given by

$$\sigma(z) = \frac{1}{1 + exp(-z)}, \tag{6}$$

where z denotes the actual input vector of a given layer. Figure 2 shows the graph and characters of a sigmoid function. From the graph, we can conclude several properties of this function that are tailored for neural networks: Firstly, this function is smooth and differentiable, which is essential for implementing various optimization methods discussed in section 4. Secondly, according to Duda et al. [2001] the nonlinearity of this function can enhance the computational power of the network. Lastly, the function value ranges from 0 to 1 which allows a probabilistic interpretation. Using this sigmoid function, for a layer $l \geq 1$, the i^{th} output unit of this layer is given as

$$y_i^{(l)} = \sigma\left(\sum_{m=1}^{M} w_{im} y_m^{l-1}\right), \tag{7}$$

where $y_m^{(l-1)}$ is the m^{th} output unit of the previous layer. The derivative of this logistic sigmoid function takes the following form:

$$\frac{\partial \sigma(z)}{\partial z} = \sigma(z)(1 - \sigma(z)), \tag{8}$$

where z is an input vector.

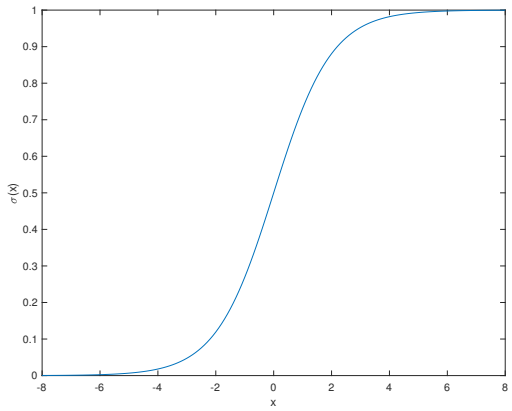


Figure 2: The commonly used logistic sigmoid function in neural network training is a s-shaped function that is smooth and nonlinear. It maps the independent variable to an output value ranges from 0 to 1, which allows a probabilistic interpretation.

3.3 Training

In this section, we introduce the method to train the neural network described in the previous sections. The iterative process of adjusting the weights in the neural network to get a better mapping from input to output is referred to as training or teaching. We use a supervised training to train the network with a labeled training set. In supervised training, the training set includes both the input values and the intended target values. Suppose we have N instances, let x_n with $n = 1, ..., N$ denotes a specific input vector with D dimensions and t_{nc} denotes the c^{th} entry of the n^{th} target output value t_n. Thus, we have a training set $T := (x_n, t_n) : 1 \leq n \leq N$.

To train the network, the objective is to find the optimal parameters in the mapping process such that the distance between the target values and the estimated output values is minimized. In our case, it is equivalent to find the optimal propagation weights in each layer. For that purpose, we use some error functions expressed by all the weights. The optimal weights can then be found by minimizing the error function with respect to the weights. The detailed training approaches are presented in section 4.

3.4 Pattern Classification

After training, we can obtain the optimal weights $\hat{w}$. We then need to make a decision about which class the given input vector belongs to. In this section, we present the method for this pattern classification problem.

Specifically, we need to assign the given input vector x_n with $1 \leq n \leq N$ to one of the discrete classes, c, with $1 \leq c \leq C$. In general, we want to assign each input vector x_n to the correct class with as few decision errors as possible. We follow the work of Ney [1995] and use the Bayes' decision rule, which is proven to give the minimum number of decision errors. According to the Bayes' decision rule, we assign x_n to the class which has the highest probability. Using the logistic sigmoid activation function, the estimated probability of x_n belonging to class c is given by the calculated output given by equation 3 in which the estimates weights are used.

4 Methodology

In this section, we discuss the methods we use to train our neural network by finding the optimal weights. We first propose the cost function to be used for the error measurement, which will be the objective function of the optimization problems. Next, we discuss the implementation details of the weight initialization. Finally, we look at three optimization methods for finding the optimal weights and talk about their implementations in neural networks one by one. For each method, we quickly run through the optimization algorithm and how the method is used in training.

4.1 Error Measurement

To train the network, the objective is to find the optimal weights for the hidden and output layers such that the distance between the target values and the estimated output values is minimized. For that purpose, in our neural network with weights w, input values x_n propagated through the network resulting in output values $y_c(x_n, w)$ and target output values t_{nc}, we define our sum-of-squared error function

$$E(w) = \sum_{n=1}^{N} E_n(w) = \frac{1}{2} \sum_{n=1}^{N} \sum_{c=1}^{C} (y_c(x_n, w) - t_{nc})^2, \tag{9}$$

where $y_c(x_n, w)$ is the function representing how the neural network maps the input x_n to the output in node. To minimize this error function, calculating the exact gradient is necessary in the implementation of the optimization methods. To simplify the first-order derivative of this equation, the error function is defined as the sum-of-squared errors times a half.

For the exact evaluation of the gradient of this error function, we rewrite it in matrix form using output and hidden weight matrices W_o and W_h, target value matrix T and input value matrix X:

$$E(w) = \frac{1}{2} \|\sigma(W_o \sigma(W_h X)) - T\|_2^2. \tag{10}$$

Using this notation, we get the gradient

$$\nabla E(w) = \begin{bmatrix} \delta_o \sigma(W_h X)^T \\ [(W_o^T \delta_o) \odot \sigma'(W_h X)] X^T \end{bmatrix}, \tag{11}$$

where

$$\delta_o = (\sigma(W_o \sigma(W_h X)) - T) \odot \sigma'(W_o \sigma(W_h X)). \tag{12}$$

We will use equation 11 for the implementation of the optimization methods in following sections.

4.2 Implementation Details

Before implementing different methods for training the network, we should choose the initial weights for every layer randomly. LeCun et al. [1998] suggests that one should draw every weight from a random distribution with mean 0 and standard deviation $\frac{1}{\sqrt{m}}$, where m is the number of units in the previous layer in the network. This form of weight initialization helps alleviate the problem of slow learning and, in our case, optimization methods failing to properly train the network because of large gradients. We initialize our weights drawing from an uniform distribution using the suggested mean and standard deviation. In each trial, we make sure the different optimization methods get the same set of initial weights to get a better idea of relative performance.

4.3 Stochastic Gradient Descent

In this section we explain the SGD algorithm. SGD is a frequently used optimization method in neural networks. The idea behind SGD is to use a single randomly picked training example for estimating the gradient of the error function instead of using all the training examples.

Gradient Descent(GD) is a first-order optimization algorithm. In each iteration we update the weights w using information about the gradient of the error function at the current point, $\nabla E(w)$. The error function is estimated using all the training examples $E(w) = \sum_{n=1}^{N} E_n(w)$. Thus, after taking a step in the direction of the gradient at w_t, the new weights w_{t+1} are

$$w_{t+1} = w_t - \gamma \frac{\partial E}{\partial w_t}, \tag{13}$$

where γ is the learning rate. The error rate decreases in every iteration of training procedure. In this method, the entire training dataset is used in each optimization step. We will refer to training that uses the whole dataset at once as batch training. Because we are using all the training examples in every iteration, the batch training with GD here takes long and converges slowly.

Therefore a modification of GD is used, namely SGD Becker and Le Cun [1988]. This algorithm estimates the gradient of the error function based on a single training example in each iteration and determines the new weights w_{t+1} by moving in the direction of the estimated gradient based on this single observation:

$$w_{t+1} = w_t - \gamma \frac{\partial E_n}{\partial w_t}. \tag{14}$$

SGD does not converge to the minimum and will stay around the minimum. In minibatch stochastic gradient descent, the number of training examples to estimate the gradient is between one and all the training examples. A certain number of training examples is propagated through the neural network and the weights are updated based on these training examples. This is the version of SGD we commonly use and refer to throughout this research. The error function may not decrease in every single iteration but there is a downward trend in the error function.

Large datasets will generally not be passed through a neural network in one iteration. Frequently datasets are divided into batches. The batch size is the number of training examples in a single batch. When an entire dataset is passed forward and backward through the neural network once, it is called an epoch. The number of iterations needed for one epoch depends on the batch size.

4.4 Backpropagation

In this part we explain the backpropagation algorithm, which is an efficient algorithm to evaluate the gradient of the error function, ∇E_n, what we use in equation 14.

To explain error backpropagation, we follow the work of Stutz [2014] and Christopher [2016]. We first look at the i^{th} output unit. The derivative for the error function E_n for the weight $w_{ij}^{(L+1)}$ is given by

$$\frac{\partial E_n}{\partial w_{ij}^{(L+1)}} = \frac{\partial E_n}{\partial z_i^{(L+1)}} \frac{\partial z_i^{(L+1)}}{\partial w_{ij}^{(L+1)}}, \tag{15}$$

where $z_i^{(L+1)}$ equals the total input of the i^{th} unit of the output layer as given in equation 1. We denote $\frac{\partial E_n}{\partial z_i^{(L+1)}}$ as $\delta_i^{(L+1)}$ and call it error. We rewrite it with the usage of chain rule as:

$$\delta_i^{(L+1)} := \frac{\partial E_n}{\partial z_i^{(L+1)}} = \frac{\partial E_n}{\partial y_i^{(L+1)}} \frac{\partial y_i^{(L+1)}}{\partial z_i^{(L+1)}} = \frac{\partial E_n}{\partial y_i^{(L+1)}} f'(z_i^{(L+1)}), \tag{16}$$

where $\delta_i^{(L+1)}$ can be interpreted as the influence of the i^{th} output unit on the total error rate and these are usually simply computable. Moreover, we rewrite $\frac{\partial z_i^{(L+1)}}{\partial w_{ij}^{(L+1)}}$ in equation 15 as:

$$\frac{\partial z_i^{L+1}}{\partial w_{ij}^{(L+1)}} = \frac{\partial}{\partial w_{ij}^{(L+1)}} \left[\sum_{k=0}^{m^{(L)}} w_{ik}^{(L+1)} y_k^L \right] = y_j^L. \tag{17}$$

Substituting with equation 16 and 17 in equation 15, we can estimate the gradient for the i^{th} unit of the output layer using

$$\frac{\partial E_n}{\partial w_{ij}^{(L+1)}} = \delta_i^{(L+1)} y_j^L. \tag{18}$$

For the hidden layers, we need a different evaluation of the gradient. Considering an arbitrary hidden layer l, we use the following formula to compute the derivative of the error measure in hidden unit i:

$$\frac{\partial E_n}{\partial w_{ij}^{(l)}} = \frac{\partial E_n}{\partial z_i^l} \frac{\partial z_i^l}{\partial w_{ij}^{(l)}}. \tag{19}$$

The second part stays the same as that in the computation of the derivative for the output units. However, the first part, the error, is different and it is given by

$$\delta_i^{(l)} := \frac{\partial E_n}{\partial z_i^{(l)}} = \sum_{k=0}^{m^{(l+1)}} \frac{\partial E_n}{\partial z_k^{(l+1)}} \frac{\partial z_k^{(l+1)}}{\partial z_i^{(l)}} = f'(z_i^{(l)}) \sum_{k=0}^{m^{(l+1)}} w_{ik}^{(l+1)} \delta_i^{(l+1)}. \tag{20}$$

So, we find a recursive algorithm to evaluate $\delta_i^{(l)}$ for every hidden layer. We can use the backpropagation in SGD to evaluate the gradient in our research since we use the logistic sigmoid as activation function and we use the sum-of-squared error function.

4.5 PMBSolve

In this section, we present the PMBSolve algorithm to solve any unconstrained optimization problems and give the details of its implementation to deep learning. We follow the research done by Öztoprak and Ş. İlker Birbil [2018], which presents two PMBSolve algorithms, one for solving convex problems and one for all the unconstrained problems in general. As the error function of our research is not convex, we only present and implement the algorithm for the general case to see how PMBSolve performs generally. The idea of PMBsolve is to use multiple trial points to collect additional information and build a proper local model in each iteration to be optimized. As the deep learning algorithm contains an optimization step for the unconstrained error function, we use the PMBSolve method for this purpose.

We first give the algorithmic details of PMBSolve. The algorithm contains an outer iteration to update each iterate and an inner iteration in each outer iteration to find the optimal step length. Suppose the objective function to be optimized is given by f. Let x_k and s_k denote the current iterate and the k^{th} step length of the algorithm, respectively. Then, the next outer iterate is given by $x_{k+1} = x_k + s_k$. To simplify further discussion, we define the following notation:

$$f_k := f(x_k),\ g_k := \nabla f(x_k),\ x_k^t := x_k + s_k^t,$$
$$f_k^t := f(x_k^t),\ g_k^t := \nabla f(x_k^t),\ y_k^t := g_k^t - g_k.$$

In each iteration, the step length s_k is the optimal one among a series of trial steps s_k^t for $t = 0, 1, \cdots$ that provides sufficient decrease satisfying the Armijo condition:

$$f_k^t - f_k \leq \rho g_k^{\mathrm{T}} s_k^t \text{ for some } \rho \in (0, 1). \tag{21}$$

This process is called an inner iteration. In each inner iteration, we compute every trial step s_k^t by solving a subproblem. The subproblem includes an extended model function, which is a linear combination of f at x_k and x_k^t. That is

$$m_k^t(s) = \alpha_k^0(s) l_k^0(s) + \alpha_k^t(s) l_k^t(s - s_k^t), \tag{22}$$

where

$$l_k^0(s) = f_k + g_k^\intercal s \quad \text{and} \quad l_k^t(s - s_k^t) = f_k^t + (g_k^t)^\intercal (s - s_k^t).$$

To ensure the weight $\alpha_k^0(s)$ increases when s is closer to s_k^t and similarly for $\alpha_k^t(0)$ when $(s - s_k^t)$ gets closer to $(-s_k^t)$, we set

$$\alpha_k^0(s) = \frac{(s - s_k^t)^{\mathrm{T}}(-s_k^t)}{(-s_k^t)^{\mathrm{T}}(-s_k^t)} \quad \text{and} \quad \alpha_k^t(s) = \frac{s^{\mathrm{T}} s_k^t}{(s_k^t)^{\mathrm{T}} s_k^t}.$$

Further, to restrict the length of the step s and its deviation from the previous trial point s_k^t, we impose the constraint:

$$\|s\|^2 + \|s - s_k\|^2 \leq \left\|s_k^t\right\|^2. \tag{23}$$

The pseudo code of this algorithm is given in section 2 of the literature by Öztoprak and Ş. İlker Birbil [2018].

We can conclude several observations based on the above information. Firstly, $\alpha_k^0(s) + \alpha^t(s) = 1$. Secondly, the region of constraint 23 is never empty that implies that the backtracking operation on the previous trial step s_k^t can always give a new feasible trial step s_k^{t+1}. Lastly, if a condition $g_k^\intercal s_k^t \leq 0$ holds, there are always feasible steps in the steepest descent direction. In other words, as long as the initial trial step s_k^0 is gradient related, there is always a feasible point that is also gradient related in the feasible region constructed around the subsequent trial step. This last observation gives the requirement for the PMBSolve algorithm to converge.

After some simplification, the subproblem is solvable using the Karush-Kuhn-Tucker optimality condition and we obtain:

$$s_k^{t+1} = c_g(\sigma) g_k + c_y(\sigma) y_k^t + c_s(\sigma) s_k^t \tag{24}$$

where

$$c_g(\sigma) = -\frac{\left\|s_k^t\right\|^2}{2\sigma}, \quad c_y(\sigma) = -\frac{\left\|s_k^t\right\|^2}{2\sigma\theta}[-((y_k^t)^\intercal s_k^t + 2\sigma)((s_k^t)^\intercal g_k) + \left\|s_k^t\right\|^2 ((y_k^t)^\intercal g_k)],$$
$$c_s(\sigma) = -\frac{\left\|s_k^t\right\|^2}{2\sigma\theta}[-((y_k^t)^\intercal s_k^t + 2\sigma)((y_k^t)^\intercal g_k)],$$

with

$$\theta = ((y_k^t)^\intercal s_k^t + 2\sigma)^2 - \left\|s_k^t\right\|^2 \left\|y_k^t\right\|^2$$

and

$$\sigma = \frac{1}{2}\left(\left\|s_k^t\right\|\left(\left\|y_k^t\right\| + \frac{1}{\eta}\|g_k\|\right) - (y_k^t)^\intercal s_k^t\right).$$

Finally, the preconditioning of this optimization method means that the subproblem needs a predetermined initial trial step s_k^0. In this research, we follow the details of the implementation done by Öztoprak and Ş. İlker Birbil [2018] and use the L-BFGS method as a preconditioner to find the initial trial step s_k^0 in each outer iteration.

In this research, we implement the PMBSolve method in our deep learning algorithm to minimize the error function with respect to the weights in every layer. Öztoprak and Ş. İlker Birbil [2018] proves this algorithm guarantees convergence if the objective function f is bounded below and its gradient ∇f is Lipschit continuous. Based on that, we expect PMBSolve converge in minimizing our error function.

4.6 Minibatch PMBSolve

Theoretically, PMBSolve uses a series of inner iterations where calculating second order information is necessary. It is therefore a very costly way of training our model for a large training set. Thus, instead of implementing PMBSolve to the whole training set, we can use smaller subsamples of the training and implement PMBSolve on every such small training set one after another. This algorithm is similar to the minibatch implementation of SGD. We call this algorithm Minibatch PMBSolve. We also run this algorithm and compare it with the other two algorithms.

The idea of this algorithm is that we continuously take a random sample set that is the minibatch from the whole training set and let PMBSolve optimize the weights for this taken set use the optimal weights found so far. The specific process is as the follows: first we randomly draw pairs of input and output vectors from the training set to create our minibatch. Then we redefine the error function based on this minibatch. The final step is letting PMBSolve minimize this error function over the initialized weights, only allowing a limited amount of optimization iterations. The resulting weights are used as the starting point for optimization for the next minibatch. We repeat this process until the stopping conditions for training are met.

This Minibatch PMBSolve requires two new parameters relative to batch PMBSolve: minibatch size and number of iterations per minibatch. Minibatch size is the number of samples a minibatch contains and the number of iterations per minibatch determines how many iterations a minibatch can be used before being replaced by another minibatch. Thus, these two parameters need to be predetermined given a specific dataset and we talk about this in the computational study.

5 Computational Study

In this section, we present the computational study. Our experiments are based on the two-layer perceptron described in section 3. We train this neural network for two datasets with the three optimization algorithms proposed in 4 and compare their performances against each other. In this section, we first give a detailed setup, which includes a description of our two datasets, the accessibility of our programming code, the machine we use and main statistics we report. Next, we give the detail of our parameter tests to get good parameter settings before conducting the experiments. Finally, we present our results.

5.1 Setup

Since our goal is to compare PMBSolve and Minibatch PMBSolve with SGD, to get a solid and convincing result, we look at how these methods perform for different types of dataset. In this research, we consider two datasets. One is the standard MNIST dataset that is widely used in neural network studies. Another one is an simulated dataset for Credit Rating. The two dataset varies a lot in terms of the input dimension and the total number of examples. We implement the three optimization methods and compare the experimental results. In this section, we discuss the characteristics of the two datasets.

The MNIST dataset is a combination of Special Database 1 (SD-1) and Special Database 3 (SD-3) of the original NISTs database, a special designed database for training image processing systems. Because of different collection sources, SD-3 contains less noise and is much easier to be recognized than SD-1. Thus, the MNIST combining SD-1 and SD-3 allows a more general training experiment over the complete set of samples.

MNIST provides 60,000 images of handwritten digits as the training set and 10,000 images as the validation set. Each image contains a digit ranging from 0 to 9 and is provided in the IDX file format with 28×28 pixels. An image can be regarded as a matrix with 28 columns and rows. However, it is too complicated to input a matrix to neural network. Here, every pixel in an image is used as a predictor. Therefore, to simplify the training procedure, we rewrite each sample matrix as a single vector by stacking all columns to get 70,000 vectors with a length of $28 \times 28 = 784$. We follow the work of Stutz [2014] using his **loadMNISTImage** and **loadMNISTLables** MATLAB functions (available online at `http://yann.lecun.com/exdb/mnist/`).

The alternative dataset, the CREDIT dataset, is an artificial dataset providing financial ratios, industry sector and credit ratings of 3,932 corporate consumers in total. Financial ratios consists of WC_TA, RE_TA, $EBIT\_TA$, MVE_BVTD and S_TA. Industry sector uses an unordered integer varying from 1 to 12 to represent the industry this consumer is in. Credit rating is labeled with letters AAA, AA, A, BBB, BB, B and C, where risk level is ascending from AAA to C. The most reliable consumer gets a rating of AAA and the least reliable gets C.

The dataset is a matrix with 3932 rows and 8 columns. Thus, it is necessary to split and extract predictors and responses for our neural network training. The first column is the ID of all consumers, which is useless for our case. Columns from the second to the sixth are financial ratios and the seventh contains an integer signifying industry sector of every consumer. We merge these six columns into one predictor matrix X with 6 rows and 3,293 columns. We also extract the last column, the ratings, as a 3,293-dimensional vector and name it Y. However, there is an issue that ratings are letters that cannot be applied to our existing neural network. In which case, transforming ratings to numerical values is necessary in our implementation. Considering that rating is an ordered multinomial dependent variable, we replace letters by integers from 0 to 6 to represent the magnitude of risk and store these corresponding integers into vector Y. Out of the 3,932 samples, we use 2,932 samples for training and the remaining 1,000 as validation set. (The dataset is available in MATLAB following the instructions at `https://mathworks.com/help/stats/select-data-and-validation-for-classification-problem.html`)

For the design of our neural network in MATLAB, we borrow the code by Stutz [2014] for a two-layer network for digit recognition (available online at `https://github.com/davidstutz/matlab-mnist-two-layer-perceptron`). For our research, we add a training algorithm based on his code, which implements the PMBSolve optimization algorithm by Öztoprak and Ş. İlker Birbil [2018] (available online at `https://github.com/sibirbil/PMBSolve`). Our code is available upon request.

For our experiments, we use a single machine with an Intel Core i5-2320 Processor (@ 3.00GHz), 7.98GB RAM and a GeForce GTX 1060 3GB GPU. We run our experiments using MATLAB without using any plug-ins.

In the subsequent sections, we mainly report the following statistics from our experiments:

The validation error, which is the error function evaluation for the validation set. The training error, which is the error function evaluation for the training set. The success rate, which is the prediction success rate of the network after training for the validation set.

5.2 Parameter Settings

Before comparing performances among the three optimization methods, we need to decide specific parameters beforehand to ensure a fair comparison. In addition, we need to decide the number of hidden units of the neural network. To get a convincing result, we need to make sure all three algorithms optimize the same cost function. This means the neural networks using these optimization algorithms have the same structure in terms of the number of hidden layers and the number of hidden units. We find our optimal parameter settings for the two datasets as follow.

For the MNIST dataset with SGD, we fix the learning rate at 0.1, the batch size at 100 and pass each batch only once. According to Stutz [2014], for the number of hidden units that are smaller than 700, this combination of parameter settings gives the best performance. With this setting, we then investigate how the size of hidden layer effects the relative performance of PMBSolve versus SGD to decide the optimal hidden layer size. For that purpose, we perform simulations with the usage of different numbers of hidden units. Since MNIST has a large input dimension, increasing the size of the hidden units drastically increases the number of weights to be optimized over. This increases the size of the gradient and thus causes longer running time for each iteration. Additionally, we have to use fewer hidden units so that PMBSolve will not reach stopping conditions in the first iteration of training. If the gradient is too large, the PMBSolve algorithm cannot make a good first step and will reach the maximum number of inner iterations. Removing the limit on inner iterations causes the training process to get stuck in the first iteration. Through the experiments with different numbers of hidden units, we find the performance of SGD always improves as the number of hidden units increases up to 700, whereas this is not the case for PMBSolve. Figure 3 shows the result of average success rate of 50 runs using PMBSolve versus the number of hidden units. We obtain the results after PMBSolve has conducted 300 iterations of training. From figure 3, the performance of PMBSolve only improves up to 20 hidden units for MNIST dataset. Thus, we set the number of hidden units to 20 to compare the best performance of PMBSolve with the performance of SGD.

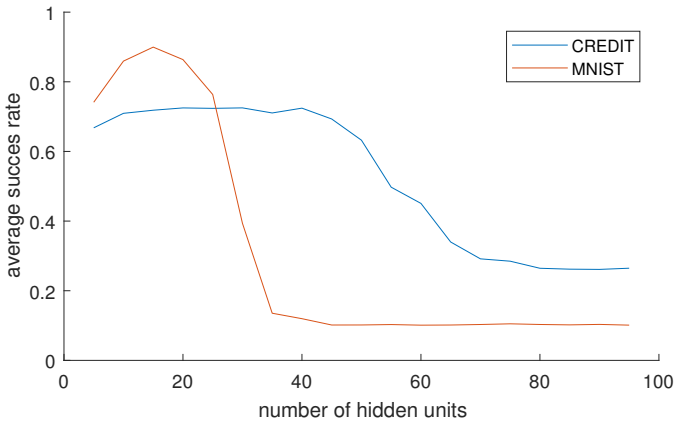


Figure 3: Average success rate versus the number of hidden units for network training with PMBSolve algorithm for MNIST and CREDIT datasets after 300 iterations of training. For each dataset, we obtain the result after running the entire process with PMBSolve 50 times and record the average success rate of the 50 trials for each number of hidden units.

Afterwards, using our baseline network with 20 hidden units, we need to find a proper minibatch size and number of iterations per minibatch for the Minibatch PMBSolve algorithm. We select candidate values {500,1000,2000,5000,10000} for minibatch size and {1,5,20,50} for number of iterations per minibatch. Next, we implement all combinations and compare the validation results. From our result, with 20 hidden units, {5000,10000} and {5,20} are the parameter values which yield the best validation results relative to other combinations. With this preliminary result, we perform larger simulation tests with different combinations of the four candidate parameters. Figure 4 shows the result of average validation error of 50 runs versus the time elapsed for the four different combinations. With the result given by figure 4, we pin down the parameters at batch size 5000 and 20 iterations per minibatch.

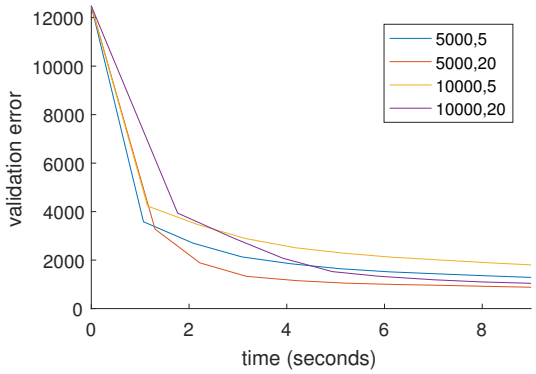


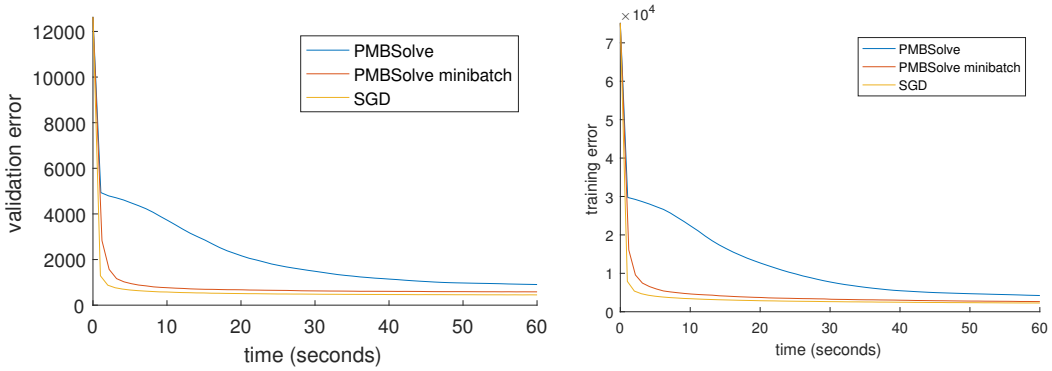
Figure 4: Average validation error versus the time elapsed for the four different combinations of minibatch size {5000,10000} and number of iterations per minibatch {5,20} using Minibatch PMBSolve algorithm. For each of these combination, we obtain the result by running the entire program of Minibatch PMBSolve 50 times and record the average success rate of the 50 trials at specific time points from 0 seconds to more than 8 seconds. We use MNIST dataset and 20 hidden units.

Following similar steps, we do parameter tests for the CREDIT dataset. After trying different numbers of hidden units range from 5 to 500 for both PMBSolve and SGD, We find in general, PMBSolve outperforms SGD for all different hidden sizes. Therefore, we set the number of hidden units at 30, which is the best for SGD, to compare the best performance of SGD with PMBSolve. We then determine the optimal learning rate and batch size for SGD. The candidate learning rates are {0.1, 0.3, 0.5, 0.7, 0.9} and candidate batch sizes are {50, 100, 150, 200, 250, 300, 350, 400, 450, 450, 500}. We find among different combinations, SGD performs best with a learning rate at 0.1 and a batch size at 300 for 30 hidden units. We also try different combinations of the batch size and number of iterations per batch for Minibatch PMBSolve method. We find the best combination among all the candidate values for this method is batch size equals to 1000 and number of iterations per batch equals to 50. We use these parameter settings for the CREDIT dataset for the subsequent study.

5.3 Results

In this section, we present the results with the optimal parameter settings discussed in section 5.2. To help avoid conclusions based on accidental results, we run 100 different trials to train the network for all the experiments in this section and report the average result. Among these 100 trials, the only difference is the randomly initialized weights. We use the initialized weights as the same starting point for each optimization method.

We first look at the result for MNIST dataset. Figure 5a shows the average validation error of 100 trials against time for the three methods using MNIST dataset. We record the average



(a) Average validation error versus time elapsed for the three methods using the optimal parameter settings in section 5.2. The vertical axis represents the average validation error of 100 trials. The validation errors after 60 seconds using SGD, PMBSolve and Minibatch PMBSolve are 450.46, 904.20 and 583.58.

(b) Average training error versus time elapsed for the three methods using the optimal parameter settings in section 5.2. The vertical axis represents the average training error of 100 trials. The training errors after 60 seconds using SGD, PMBSolve and Minibatch PMBSolve are 2279.10, 4216.45 and 2644.91.

Figure 5: Validation error and training error versus time elapsed with twenty hidden units for MNIST dataset.

validation errors for 60 seconds. With the size of the validation set equals to $10,000$, the average validation error rates for SGD, PMBSolve and Minibatch PMBSolve are approximately 4.50%, 9.04% and 5.84%. From these statistics and figure 5a, SGD outperforms PMBSolve according to the accuracy. Additionally, Minibatch PMBSolve has a strikingly similar trend as SGD during the training process but still cannot outperform SGD in terms of accuracy.

To investigate if there is an overfitting issue when use the three methods, we look at the training errors. Figure 5b shows the average training error of 100 trials against time. With the size of the training set equals to $60,000$, the average training error rates after 60 seconds for SGD, PMBSolve and Minibatch PMBSolve are approximately 3.80%, 7.03% and 4.41%. From figure 5b, the average training errors versus time have very similar trends as 5a. Although all the methods give lower error rates in the training set, the difference between the training errors and validation errors are not significantly large. This means, only from this result, we cannot conclude that there are overfitting issues use the three methods.

With the above elementary results, we dive into the performances of PMBSolve and Minibatch PMBSolve compared to SGD and discuss what possibly cause these performances. To make a more convincing comparison for the accuracy rates and running time, we set a limit of 500 iterations for each method and record the average success rates and running times of 100 trials. Table 1 shows the corresponding results.

Method	Average Running time (seconds)	Average Success Rate
PMBSolve	91.6653	0.8937
Minibatch PMBSolve	8.4047	0.8970
SGD	3.9107	0.9233

Table 1: Average results of 100 trials using SGD, PMBSolve and Minibatch PMBSolve with MNIST dataset. We use the optimal parameter settings in 5.2 with 20 hidden units. The results are obtained after 500 iterations for each method. All the methods use the same initial weights in each trial.

First of all, we compare the performance of PMBSolve with SGD. According to table 1,

PMBSolve needs to run around 23 times as long as SGD to complete 500 iterations. In addition, the average success rate for SGD is approximately 3.3% higher than that for PMBSolve. We infer the big input dimension of MNIST dataset causes the low speed of PMBSolve. As in section 4.5, in each outer iteration, PMBSolve includes inner iterations to find the optimal step length by minimizing an extended model. This step uses the gradient and the second order information and thus can take long when the input dimension is big and the size of the gradient is large. To gain an idea why PMBSolve gives lower accuracy rate, we first check the success rates of all the 100 trials. We find that in three out of these 100 trials, PMBSolve fails to properly train the network, with accuracy rates less than 50%. These three trials weigh down the average accuracy rates of PMBSolve. Because the only difference between these 100 trials is the randomly set initial weights, we attribute these exceptional failures to bad starting points. Although Öztoprak and Ş. İlker Birbil [2018] verifies the direction updates of the PMBSolve algorithm can reduce the total number of function evaluations required, from our experiment with a fairly large dataset, this does not necessarily reduce the total running time using PMBSolve.

Secondly, it is nature to wonder why the performance of Minibatch PMBSolve is similar to that of SGD as shown in figure 5. For that purpose, we investigate the number of inner iterations used in the procedure of implementing Minibatch PMBSolve. The inner iteration step of PMBSolve is a crucial step that makes Minibatch PMBSolve distinct from SGD since it uses second order information rather than only first order information as in GD. We run our deep learning algorithm five times using Minibatch PMBSolve and record the percentage of times Minibatch PMBSolve uses inner iterations over the total number of outer iterations. According to the results, on average only in 7.8% of the outer iterations, the algorithm uses inner iterations. This implies that in the minibatch implementation, PMBSolve effectively uses first order information in the outer iterations and hardly uses second order information. A possible reason for second order information being left unused is the shape of our error function. As we use a logistic sigmoid function as the activation function with sum-of-squared errors, we cannot ensure that the curvature of our error function demands the use of second order information for better optimization steps. From table 1, Minibatch PMBSolve performs better than batch PMBSolve in terms of speed but is still around twice as slow as SGD and it is slightly less accurate than SGD.

Method	Average Running time (seconds)	Average Success Rate
PMBSolve	88.6837	0.8951
Minibatch PMBSolve	8.3923	0.8787
SGD	3.8210	0.9236
PMBSolve with warm-up	79.8306	0.9250

Table 2: Average results of 100 trials using SGD, PMBSolve, Minibatch PMBSolve and PMBSolve with warm-up for MNIST dataset. We use the optimal parameter settings in 5.2 and 5.3 with 20 hidden units. We obtain the results after 500 iterations for each method. All the methods use the same initial weights in each trial.

After that, in order to reduce the number of times PMBSolve performs poorly, we try a combination of Minibatch and batch PMBSolve. The idea is to use one or several minibatches to do preparatory training before using PMBSolve. By doing this, PMBSolve is less likely to start with a bad point and result in a bad accuracy. We call this method PMBSolve with warm-up. Additionally, we call it a round when one minibatch is passed. In this method, we use a smaller minibatch size because we want to keep the preparatory training short. We try several combinations of different number of rounds, minibatch sizes and iterations per batch. Among all the combinations, we get the best result using 2 rounds of Minibatch PMBSolve with batch size 2000 and 20 iterations per batch. After these 40 iterations, we use PMBSolve

for the entire training set for the rest of training. Table 2 shows the result of PMBSolve with warm-up using the optimal settings together with the other three methods. To give a cogent conclusion of whether this warm-up increases the performance of PMBSolve, we ensure all the methods presented in table 2 start with the same initial weights in each run. Again, the result is based on 20 hidden units, 500 iterations and the average of 100 trials. From table 2, only with 2 rounds of the warm-ups, the starting point is already good enough for PMBSolve to give slightly higher success rate than SGD. With our current settings, SGD is still competitive in terms of the running time. However, depends on the dataset, it is possible that with more rounds of warm-ups using Minibatch PMBSolve, the running time of this method significantly decreases. Overall, this suggests PMBSolve with warm-up can be a good alternative for neural network training.

After the MNIST dataset results, we present the results for the CREDIT dataset. We conduct the same experiments for the CREDIT dataset using the optimal parameters for this dataset described in section 5.2. We also perform an overfitting test by comparing the validation errors and training errors for the three methods. Similar as the MNIST dataset, there is no significant indication of overfitting issue for CREDIT dataset.

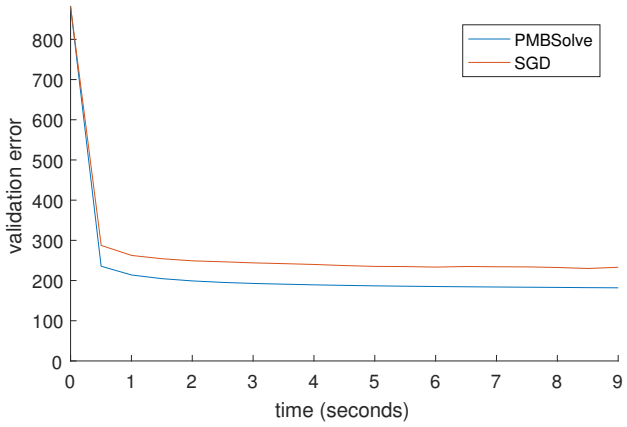


Figure 6: Validation error versus time elapsed for PMBSolve and SGD with CREDIT dataset using the optimal parameter settings in section 5.2. The vertical axis represents the average validation error of 100 trials. The validation error is the value of the error function for the validation set. The validation errors after 9 seconds using SGD and PMBSolve are 233.44 and 185.92 respectively.

Figure 6 shows the average validation error of 100 trials against time for PMBSolve and SGD methods using CREDIT dataset. From our experiments, for CREDIT dataset, the Minibatch PMBSolve method is highly similar to PBMBSolve in both accuracy and running time. Thus, we omit the result and discussion of Minibatch PMBSolve. According to Figure 6, the validation error decreases remarkably within the first second for both SGD and PMBSolve. With the size of the validation set equal to $1,000$, the average validation error rates after 9 seconds for SGD and PMBSolve are 23.34% and 18.59% respectively. Unlike the result of MINIST, PMBSolve outperforms SGD for this dataset. As is introduced in section 5.1, the CREDIT dataset has a much smaller input dimension, training set and validation set. We see these as the reasons that lead to the different performances of PMBSolve relative to SGD for MNIST and CREDIT. Furthermore, from Figure 3, PMBSolve is sensitive to the number of hidden units with both datasets. In general, PMBSolve performs worse as the size of hidden layer increases, which is not the case for SGD shown in Stutz [2014]. We attribute this to the large number of parameters that need to be optimized over and large computational costs for PMBSolve when the number of hidden units increases. Overall, we expect PMBSolve to perform worse than SGD when using large datasets and expect it to perform better than SGD when using smaller datasets with a

small input dimension size.

6 Conclusion and Future Research

In our experiments we find that, given the right conditions, PMBSolve can get better results than SGD in a simple deep learning setting. The ability of PMBSolve to use the entire dataset with second order information of the error function leads to a smaller training error than that of SGD. However, the usage of two datasets with different sizes displays the dependence of PMBSolve on dataset size. With a relatively small dataset, as is the case for CREDIT in our research, PMBSolve gives higher success rates than SGD with a higher speed. However, we observe that when we use a larger dataset, specifically the MNIST dataset, PMBSolve has several downsides that make it an undesirable optimization algorithm for a deep learning setting. Firstly, PMBSolve takes long time to achieve a satisfactory prediction result. Secondly, PMBSolve gives very low accuracy if it is not provided with a good starting point. Lastly, PMBSolve is more sensitive to the input dimension size and, similarly, the number of hidden units than SGD. In general, PMBSolve performs worse when the number of hidden units increases. Therefore, we expect that when a problem becomes more complicated, for example using a deeper network with more hidden layers or a larger dataset, PMBSolve may perform increasingly worse compared with SGD. The only advantage of using PMBSolve for large datasets is that it can get better results than SGD if it has a good starting point.

To alleviate the problem of large computational costs when using PMBSolve with large dataset, we introduce Minibatch PMBSolve algorithm. We find with our MNIST dataset, this method performs very similar to SGD and on average cannot outperform SGD. This similarity is caused by the fact that Minibatch PMBSolve does not use inner iterations, and thus, second order information for better optimization steps very often. A possible reason for this is the relative simplicity of the error function. Therefore, in our case, Minibatch PMBSolve does not provide a useful alternative to batch PMBSolve even though it is much faster.

To reduce the chances that PMBSolve starts with a bad point, we also introduce PMBSolve with warm-up, which uses Minibatch PMBSolve for a limited amount of iterations to do a preparatory training before using PMBSolve for the entire dataset. In our deep learning setup with MNIST dataset, this method gives the highest average success rate. The preparatory training makes significant steps, providing the batch PMBSolve method with a good starting point to succeed in the training. In terms of the accuracy rate and reliability, PMBSolve with warm-up is a good alternative optimization algorithm.

In our research, we point out several drawbacks of using PMBSolve to train neural networks with large datasets. However, we do not go into detailed investigation of the causes for these drawbacks. This is a potential direction for further research. From our result, we attribute the random failures of PMBSolve to the bad starting point. However, more studies are needed to find out why the starting point has so much influence on the performance of the PMBSolve algorithm for deep learning. A preliminary conjecture is that if the starting point is very far from the global minimum, PMBSolve might need many iterations to finally reach the minimum. In this case, PMBSolve algorithm may reach the limit of the running time and stop at a point still far away from the global minimum. It is also possible that a bad starting point results in the algorithm lingering at a saddle point or local minimum. In any case, it is yet to be concluded what the exact reason for these failures is. It is also interesting to study the exact causes for PMBSolve to have a very long computation time when the dataset is large and check if there is room for modification. In addition, though we cannot see any indication of an overfitting issue, it could use a more sophisticated testing approach, for example, bootstrapping.

Moreover, we only partially test the performance of PMBSolve in deep learning training. Future research may use different activation function, error function or more datasets to see if the performances of PMBSolve are different. Finally, we do not have a parallel implementation

for PMBSolve in our research. Future research may conduct a parallel implementation for PMBSolve to train deep learning algorithms and see if there is any improvement.

References

Figen Öztoprak and Ş. İlker Birbil. An alternative globalization strategy for unconstrained optimization. *Optimization*, 67(3):377–392, 2018. doi: 10.1080/02331934.2017.1401070.

David Stutz. Introduction to neural networks. *Selected topics in human language technology and pattern recognition 13/14*, 2014.

Quoc V. Le, Jiquan Ngiam, Adam Coates, Abhik Lahiri, Bobby Prochnow, and Andrew Y. Ng. On optimization methods for deep learning. In *Proceedings of the 28th International Conference on International Conference on Machine Learning*, ICML'11, pages 265–272, USA, 2011. Omnipress. ISBN 978-1-4503-0619-5.

F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, pages 65–386, 1958.

Marvin Minsky and Seymour Papert. *Perceptrons: An Introduction to Computational Geometry.* MIT Press, Cambridge, MA, USA, 1969.

Simon Haykin. *Neural Networks: A Comprehensive Foundation.* NJ, Englewood Cliffs: Prentice Hall, 1999.

Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification.* Wiley, New York, 2 edition, 2001. ISBN 978-0-471-05669-0.

Hermann Ney. On the probabilistic interpretation of neural network classifiers and discriminative training criteria. *IEEE Trans. Pattern Anal. Mach. Intell.*, 17(2):107–119, 1995. URL `http://dblp.uni-trier.de/db/journals/pami/pami17.html#Ney95`.

Yann LeCun, Leon Bottou, Genevieve B. Orr, and Klaus-Robert Mller. Efficient backprop. *Neural Networks: Tricks of the Trade*, pages 9–50, 1998.

Sue Becker and Yann Le Cun. Improving the convergence of back-propagation learning with second order methods. In *Proceedings of the 1988 connectionist models summer school*, pages 29–37. San Matteo, CA: Morgan Kaufmann, 1988.

M Bishop Christopher. *Pattern Recognition and Machine Learning.* Springer-Verlag New York, 2016.

Simon S. Haykin. *Neural networks and learning machines.* Pearson Education, Upper Saddle River, NJ, third edition, 2009.

Franz Josef Och. Minimum error rate training in statistical machine translation. In *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics - Volume 1*, ACL '03, pages 160–167, Stroudsburg, PA, USA, 2003. Association for Computational Linguistics. doi: 10.3115/1075096.1075117. URL `https://doi.org/10.3115/1075096.1075117`.