Trust Region Methods for Training Neural Networks

by

Colleen Kinross

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Science
in
Computer Science

Waterloo, Ontario, Canada, 2017

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Artificial feed-forward neural networks (ff-ANNs) serve as powerful machine learning models for supervised classification problems. They have been used to solve problems stretching from natural language processing to computer vision. ff-ANNs are typically trained using gradient based approaches, which only require the computation of first order derivatives. In this thesis we explore the benefits and drawbacks of training an ff-ANN with a method which requires the computation of second order derivatives of the objective function. We also explore whether stochastic approximations can be used to decrease the computation time of such a method. A numerical investigation was performed into the behaviour of trust region methods, a type of second order numerical optimization method, when used to train ff-ANNs on several datasets. Our study compares a classical trust region approach and evaluates the effect of adapting this method using stochastic variations. The exploration includes three approaches to reducing the computations required to perform the classical method: stochastic subsampling of training examples, stochastic subsampling of parameters and using a gradient based approach in combination with the classical trust region method. We found that stochastic subsampling methods can, in some cases, reduce the CPU time required to reach a reasonable solution when compared to the classical trust region method but this was not consistent across all datasets. We also found that using the classical trust region method in combination with mini-batch gradient descent either successfully matched (within 0.1s) or decreased the CPU time required to reach a reasonable solution for all datasets. This was achieved by only computing the trust region step when training progress using the gradient approach had stalled.

Acknowledgements

I would like to thank my supervisors, Professor Yuying Li and Professor Justin Wan, for all of their help and guidance through this process. I learned a lot from both of you and really appreciate all of the hard work you each put into helping me succeed. I would also like to thank all of my labmates for their encouragement and support.

Dedication

This is dedicated to my family.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Machine learning has become an increasingly popular subject in industry and academia in recent years. Machine learning is an approach in which computers learn solutions from data directly. In this thesis we will be considering artificial neural networks, often referred to more simply as neural networks. Neural networks, particularly deep neural networks, are of great value for use across many industries, such as self-driving cars [4], computer vision [20], natural language processing [23] and even for mastering the game of Go [33]. In this thesis we study multi-layer feedforward neural networks which are neural networks of more than one layer which only contain connections that are in the direction from input to output. The power of a multi-layer feedforward neural network lies in its ability to learn complex functions. In fact, the multi-layer feedforward neural network is referred to as a universal approximator [18].

In this thesis, we are interested in supervised training methods for feedforward neural networks. Supervised training refers to training a model based on data that has an ‘answer key’, meaning that there are dependent variables provided for their respective independent variables in the data [17]. In other words, the network is learning how to make predictions based on provided data, where the correct prediction is provided as well as the features used to make that prediction. The typical training approach used for feedforward neural networks is what is sometimes called the ‘Backpropagation Training Procedure’ [28]. Training involves the formulation of an objective function based on the discrepancy between expected prediction from the ‘answer key’, and the actual prediction made by the neural network based on its current parameters. Once the objective function has been formulated it is minimized using a numerical optimization method. Backpropagation is a method used to compute partial derivatives of the objective function in terms of each parameter value, also known as a weight value. These partial derivatives are then used to construct the gradient, the vector of partial derivatives of the objective function in terms of weights of the network. Using the gradient or stochastic approximation to the gradient, a numerical method can now be applied to reduce the value of the objective function. Typically the objective function is the mean squared prediction error of all of the training samples.

The standard method for minimizing the objective function of a neural network is gradient descent [1][13]. More recently, there has been interest in deeper networks which require solving very large optimization problems for which online learning methods such as stochastic gradient descent or mini-batch gradient descent are typically used [14][34]. These methods are useful because each step calculation only requires computation of the first derivatives for a subset of the training set and is therefore typically fast, which can be a benefit to some problems. However, the gradient direction is only the direction of fastest decrease up to the first order. These methods have difficulty in valley type structures where the direction of negative curvature is perpendicular to the gradient direction. In order to improve reduction in the objective value at each step, some methods use first derivatives to approximate second derivatives. These are known as quasi-Newton methods. Examples of successful methods that approximate second order information include the Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm which solves the secant equation for the second derivative [30]. This increases the potential for reduction of the objective function at each step compared to a step of the same magnitude in the gradient direction. However, we can improve upon the accuracy of the step direction by computing second derivatives of the objective function.

Second order numerical optimization methods, those which use second derivatives of the objective function in terms of its parameters, take more time to compute but provide more information and therefore can compute step directions that are closer to the optimal reduction in objective function value over a step. The curvature information, given by second derivatives, is helpful for descending valley type structures effectively and can be more effective at passing through nearly flat areas in the objective function surface [30]. This information can be significantly slower to compute at each iteration, therefore the benefit of the method must be very large for the speed to be comparable to a first order method. Their relative behaviours depend on the characteristics of the optimization problem being considered.

Newton's method is a common second order method which solves for the first order critical point of the second order approximation to the objective function at each iteration. One issue with this algorithm is that the Hessian, the matrix containing the second order derivatives of the objective function, can be indefinite if it is not close to a minimum. When the Hessian is not positive semi-definite, Newton's method would point towards a critical point which is not a local minimizer. Because of this, Newton's method is only locally convergent. There are many methods, such as BFGS mentioned earlier, that approximate the second order information while keeping the approximate Hessian positive definite. Other approaches include the Hessian free method which approximates the Hessian by adding an extra value to each eigenvalue of the finite difference approximation of the Hessian matrix [25]. There are also trust region type methods during which the minimizer of a bounded second order trust region subproblem is solved at each step where the bound is adjusted based on the model's accuracy [40].

In this thesis we study the performance, in terms of CPU time and training error, of the trust region method and variations of the method to train neural networks on several datasets. The trust region method is chosen as the topic of study because it is globally convergent under mild assumptions [32], compared to Newton's method which is only locally convergent. The basic idea of the algorithm is that at each step we consider a region around the current point where the model, constructed based on the first and second order information, is trusted. At this point we solve a constrained minimization problem for this model within the trusted bounds called the trust region subproblem. In order to study whether the basic second order trust region method can be sped up in terms of CPU time required for a reasonable solution, we study stochastic subsampling effects on the method. This is a different approach to second order approximation than that taken by quasi-Newton methods. Rather than approximating second derivatives from first derivatives we are approximating second derivatives of the objective function from the true second order derivatives of randomly subsampled training examples. This is the method used in stochastic gradient descent in order to speed up gradient descent. We wish to determine whether the same approach benefits trust region methods. We also study the effect of subsampling parameters or 'weights', which is a method inspired by co-ordinate descent type approaches, where we only consider a subset of the parameters at each step. This reduces the dimensionality of the trust region subproblem.

Through this study we aim to shed light on the behaviour of trust region methods and stochastic variants of trust region methods when used to train feedforward neural networks. We are looking for promising directions of speed up to the trust region method while maintaining the benefit of using second order information, which is improved step direction at each iteration. As feedforward neural networks are used to solve more diverse problems we expect that a greater understanding of the behaviour of a wider range of numerical optimization methods in the specific context of training neural networks will become increasingly valuable.

1.1 Summary of Contributions

We compute second order information from feedforward neural networks using a backpropagation technique proposed by Pearlmutter in 1993 [27]. We also implement the trust region method using a recent approach that involves solving a single generalized eigenvalue problem to determine the trust region subproblem solution at each iteration, rather than an iterative procedure [29]. With a complete trust region algorithm we then perform experiments that compare the trust region algorithm to those based on variations of the complete trust region algorithm to collect empirical information of their behaviour as well as seeking quantitative information to indicate which variations, if any, decrease the computational time of the trust region method. More specifically, our contributions are as follows:

1. A study of the benefits and drawbacks of using stochastic subsampling of training examples upon which to solve the trust region subproblem at each iteration compared to using the full training set which is the case for the full trust region algorithm. For gradient methods, subsampling is a very effective speed up approach. This motivated us to study this appraoch in the context of a second order method.

2. We compare the effect of the full trust region algorithm as well as its stochastic variations with and without parameter subsampling. This reduction in dimensionality of the trust region subproblem can reduce the time complexity to compute the solution of the trust region subproblem considerably. We run numerical experiments to see if this change benefits any of the methods (trust region method and it's stochastic subsampling variations) overall.

3. We study the effects of combining mini-batch gradient descent, a stochastic gradient method, with the full trust region algorithm. We determine whether this is an effective way to improve the final objective function value of a stochastic gradient method without the burden of CPU time required to compute second order information at each iteration.

1.2 Outline

We begin by providing background in Chapter 2 where we define the feedforward artificial neural network (ff-ANN) as well as the trust region method and the specific approach we will be taking to compute the solution to the trust region subproblem from [29]. In

Chapter 3 we cover the full details of the trust region approach and implementation used in our study. In Chapter 4 we define the various algorithms that will be studied and which variations are used to develop them based on the trust region method from Chapter 3. Chapter 5 contains the main contributions of this thesis. Here we present the results and analysis of the numerical exploration into trust region methods and its variations when used to train neural networks on data from five different datasets. Finally, in Chapter 6 we outline the conclusions from the thesis and highlight directions for future work.

Chapter 2

Training Feed Forward Artificial Neural Networks

In this chapter we provide the necessary background information for our investigation. This includes a definition and description of feedforward artificial neural networks (ff-ANNs), some numerical optimization methods used for training ff-ANNs, as well as trust region methods. In addition, we provide the details of the specific trust region method we will be using in this thesis to train our networks.

2.1 Feed Forward Artificial Neural Networks

An artificial neural network is a set of artificial neurons attached to each other in a network using weighted connections. This type of network can be used as a machine learning model and trained to approximate vectors of dependent variables from their provided vectors of independent variables. An ff-ANN has only single direction connections going in the direction from independent variables to prediction of dependent variables. Training is performed on a set S that contains m instances of independent and dependent variable vector pairs, represented in this thesis as:

$$(\mathbf{x}^k, \mathbf{y}^k) \quad k = 1, 2, ..., m \ , \tag{2.1}$$

where m is the size of the training set S, $\mathbf{x}^k \in S$ is the k^{th} vector of independent variables and $\mathbf{y}^k$ is the associated vector of dependent variables. In this section we briefly define the perceptron then generalize and build on this concept to construct an ff-ANN .

2.1.1 Perceptron

A perceptron is a specific ff-ANN with only a single node and binary output. As an ff-ANN, a perceptron can be trained using supervised learning when given a training set S, as in (2.1), where y^k is a scalar value rather than a vector. Formally, the operation between independent and dependent variables, $(\mathbf{x}^k, y^k)$, using a perceptron, is described as:

$$z^k = \sigma_H(\mathbf{w}^T \mathbf{x}^k + b), \tag{2.2}$$

where z^k is the network's prediction of y^k based on $\mathbf{x}^k$, the vector, $\mathbf{w}$, and scalar, b, are parameters which are trained on the training set S, and σ_H is the Heaviside step function defined as:

$$\sigma_H(x) = \begin{cases} 1, & \text{if } x > 0, \\ 0, & \text{otherwise.} \end{cases} \tag{2.3}$$

Perceptrons are often presented visually as a directed graph. Figure 2.1 is an visual representation of a single perceptron where:

$$\mathbf{x}^k = \begin{pmatrix} x_1^k \\ x_2^k \\ x_3^k \end{pmatrix}, \quad \mathbf{w} = \begin{pmatrix} w_1 \\ w_2 \\ w_3 \end{pmatrix}, \tag{2.4}$$

and h^k is used as an intermediate value defined as:

$$h^k = \mathbf{w}^T \mathbf{x}^k + b. \tag{2.5}$$

Perceptrons are linear classifiers, as can be seen from (2.2). The perceptron's k^{th} output, z^k, indicates whether $\mathbf{x}^k$ is above or below the hyperplane defined by $\mathbf{w}$ and b. Therefore, perceptrons cannot accurately classify data which cannot be separated by a hyperplane, meaning such problems require a more complex model.

2.1.2 Multiple Layers for Complex Models

An ff-ANN is a network of neurons. A perceptron is specific case of a single artificial neuron, one which contains the Heaviside step function as it's activation function. In general, neurons in ff-ANNs can contain any monotonically increasing function as the activation function. In our thesis all neurons use activation functions, $\sigma(.)$, which approximate the Heaviside step function by meeting the criteria:

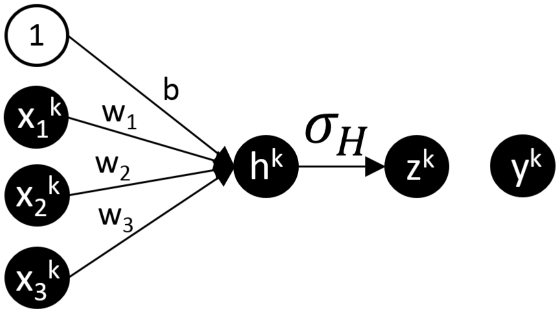

Figure 2.1: Graphical representation of an example perceptron which predicts the output for the k^{th} sample, z^k with $\mathbf{x}^k \in R^3$. The "1" is a constant multiplier which differentiates regular weights values from the bias value b.

$$\sigma(x) \in \begin{cases}]0.5, 1], & \text{if } x > 0, \\ [0, 0.5], & \text{otherwise.} \end{cases} \tag{2.6}$$

Meeting this criteria for all activation functions, we can still use the concept of neurons as hyperplanes as described in the previous section for perceptrons. The hyperplane defined by a neuron with an activation function satisfying (2.6) separates the two value ranges in (2.6) rather than determining a simple binary output.

An ff-ANN can contain several layers of neurons stacked on top of each other as well as several neurons per layer. By adding multiple layers, the network can learn more complex functions. The output of the neurons at each layer, which is the more general version of (2.2), becomes:

$$\mathbf{z}_l^k = \sigma_l(\mathbf{W}_l \mathbf{z}_{l-1}^k + \mathbf{b}_l), \quad \text{for } l = 1, ..., L, \tag{2.7}$$

where l represents the layer number, $\sigma_l()$ is the activation function at layer l, L is the total number of layers, $\mathbf{W}_l$ and $\mathbf{b}_l$ contain the weights and biases for layer l where each row contains the parameters for a single neuron, and:

$$\mathbf{z}_0^k = \mathbf{x}^k, \quad \text{for } k = 1, 2, ..., m. \tag{2.8}$$

The final classification prediction is the output of the final layer, $\mathbf{z}_L^k$, for the k^{th} training example. The purpose of having multiple layers is to model more complex non-linear functions. Considering the neurons as hyperplanes, by assuming the neurons' activation functions meet (2.6), a neuron in the second layer then separates the results from the first

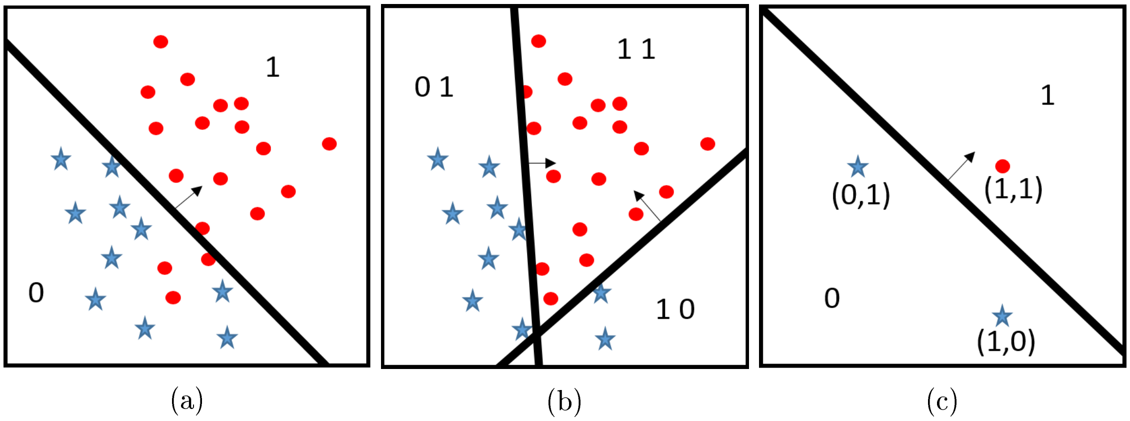

Figure 2.2: Visualization of the capacity increase from having multiple layers in a network. Subplot a) shows the classification potential for a single perceptron. Subplots b) and c) show the potential classification performed by 3 perceptrons, 2 in the first layer and 1 in the second.

layer. See Figure 2.2 for an example of a dataset that can be separated using a two layer network but not a single neuron, illustrated using hyperplane representations of neurons. In this example we show that a single hyperplane cannot separate the given example data, however two hyperplanes whose outputs are combined using a single hyperplane can be used to perfectly separate the data.

2.1.3 Training

Now that we have provided intuition for why ff-ANNs have the potential to model complex functions, we focus this section on how the parameters of a network are tuned or "trained" to meet potential. The goal of training is to fit the network to the training data by solving an optimization problem. Typically the objective function, $f(\mathbf{W}_1, ..., \mathbf{W}_L, \mathbf{b}_1, ..., \mathbf{b}_L)$, used for this optimization problem is the mean squared error of the set of all m training examples, which can be written as:

$$f(\mathbf{W}_1, ..., \mathbf{W}_L, \mathbf{b}_1, ..., \mathbf{b}_L) = \frac{1}{m}\sum_{k=1}^{m}(\mathbf{y}^k - \mathbf{z}_L^k)^T(\mathbf{y}^k - \mathbf{z}_L^k), \tag{2.9}$$

where $\mathbf{z}_L^k$ is the network's prediction for $\mathbf{y}^k$ which depends on weights $\mathbf{W}_l$ and biases $\mathbf{b}_l$ for all layers l, $l = 1, ..., L$. The optimization problem for training the network is therefore:

$$\min_{\mathbf{W}_1,...,\mathbf{W}_L,\mathbf{b}_1,...,\mathbf{b}_L} \quad \frac{1}{m}\sum_{k=1}^{m}(\mathbf{y}^k - \mathbf{z}_L^k)^T(\mathbf{y}^k - \mathbf{z}_L^k). \tag{2.10}$$

The ultimate objective of training a ff-ANN, and machine learning models in general, is to maximize the accuracy of predicted dependent variable $\mathbf{z}$ for independent variable $\mathbf{x}$ that is outside of the set S, in addition to maintaining model accuracy for the training data in S. Accuracy of the model on the training set is attained through the use of a numerical optimization method. Various methods are used, each with different benefits and drawbacks. This will be discussed in §2.2.

2.1.4 Overfitting

Overfitting describes fitting a model to a training set such that it performs poorly on unseen data compared to on the training data. In other words, the trained model does not generalize well. As previously discussed, ff-ANNs can model very complex functions. When there is a small training set, it may learn a function influenced by the noise of the training set rather than purely the signal shared by unseen and training data alike, leading to this lack of generalizability [35]. Since many problems are tackled using machine learning where the underlying model dimensionality is not known prior to learning, we do not know how large our ff-ANN should be to prevent overfitting without over-simplifying the model. There are many techniques to avoid overfitting without shrinking the network. In our experiments we use a regularization term which penalizes large values of the parameters. It is a way to reduce the capacity of the network and therefore helps prevent overfitting. Specifically the final objective function with our added regularization term has the following form:

$$f(\mathbf{x}) = g(\mathbf{x}) + \lambda_r \|\mathbf{x}\|_2^2, \tag{2.11}$$

where $g(\mathbf{x})$ is the initial objective function, $\mathbf{x}$ is the parameters for the objective function and $f(\mathbf{x})$ is the resulting objective function when using regularization for the problem, λ_r is what is known as the 'regularization' parameter which tunes the level of impact of the regularization term on the original objective function and $\|\mathbf{x}\|_2^2$ is the 2-norm of the parameters, $\mathbf{x}$.

2.2 Optimization Methods for Training ff-ANNs

For simplicity of discussion, we consider a general unconstrained optimization problem:

$$\min_{\mathbf{w} \in \mathbb{R}^n} \quad f(\mathbf{w}). \tag{2.12}$$

Note that (2.10) can be transformed into the form of (2.12) by adding the regularization component as in (2.11) and vectorizing parameters $\mathbf{W}_l$, and $\mathbf{b}_l$ for $l = 1, ..., L$ into a single vector $\mathbf{w}$, which is discussed in Appendix A.0.1.

As described in §2.1.3, a numerical optimization method is required to solve (2.10). A lot of work has been done in numerical optimization for ff-ANNs that can be sorted into two categories, those that use only first-order information and those that also use second-order information of an ff-ANN objective function, (2.9).

The particular challenges for solving large ff-ANN training problems are:

1. Large number of training examples, m,
2. Large number of dimensions (weights, including biases), n.

For instance, a network with 7 layers and 100 hidden nodes per layer would have $n = 70700$ (100^2 weights and 100 biases for each layer) and the value of m can be in the millions. The following characteristics of a ff-ANN objective function surface pose additional challenges to solving (2.12) are [30]:

1. Large plateaus (flat or nearly flat areas),
2. Narrow valleys,
3. Many saddle points (exponentially increasing with number of parameters, n).

This means that numerical optimization methods need to perform **well** on a problem with these characteristics to be considered suited for ff-ANNs. Typically the definition of **well** is based on a mixture of speed (training time), training error, prediction accuracy of test set and the final objective function value. In this thesis we will mainly focus on training time and training error.

2.2.1 First Order Methods

Here we will describe some common optimization methods used for training ff-ANNs using only first order information.

1. **Gradient Descent** is typically only used for small datasets which has the update defined by
$$\mathbf{w}_k = \mathbf{w}_{k-1} - \eta \nabla f(\mathbf{w}_{k-1}), \tag{2.13}$$
where η is the learning rate. When η is small enough and kept constant, with random initialization, gradient descent does not converge to a saddle point almost surely [19].

2. **Stochastic Gradient Descent** is a commonly known and used method which approximates gradient descent using only a single example at each step. Each iteration i has the update:
$$\mathbf{w}_k = \mathbf{w}_{k-1} - \eta \widetilde{\nabla} f(\mathbf{w}_{k-1}), \tag{2.14}$$
where η is the learning rate chosen by the user which is decreased in magnitude throughout training using a predefined schedule, and $\widetilde{\nabla} f(\mathbf{w}_{k-1})$ refers to the approximation of the gradient of the objective function at point $\mathbf{w}_{k-1}$ based on a single, uniformly randomly selected, training example.

3. **Mini-batch Gradient Descent** is a method that uses the same update function as shown in (2.14) but with $\widetilde{\nabla} f(\mathbf{w}_{k-1})$ computed as an approximation to the gradient of the objective function at point $\mathbf{w}_{k-1}$ based on a **subset** of the set of training examples S.

First order methods compute steps very quickly. However, they can experience difficulties in some regions, such as in valleys. A valley is an example of a characteristic often seen in ff-ANN problems [30] that slows down first order methods. First order methods experience difficulties in this case because the gradient direction is almost perpendicular to the direction of the local minimum. Figure 2.3, which is from [25], depicts this concept. These gradient methods also have difficulties in small gradient areas such as large pseudo-plateaus, since the length of the step taken at each iteration is proportionate to the size of the gradient for a constant learning rate.

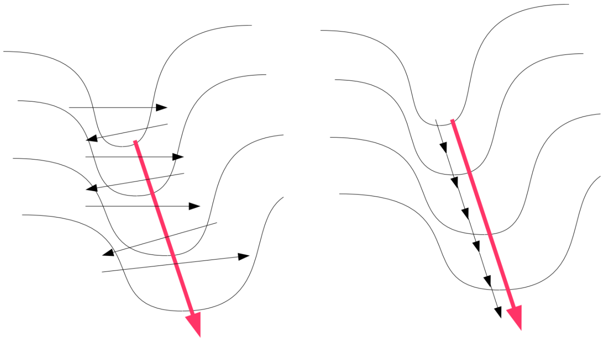

Figure 2.3: This is Figure 1 from [25] which shows an example of a valley characteristic. On the left the arrows point towards the gradient direction and the red arrow points towards the shortest path to the smallest value visible. On the right are arrows pointing in the direction of a method considering curvature and therefore avoiding the gradient direction which is, in this case, in a direction of high positive curvature.

2.2.2 Second Order Methods

The first group of methods presented below are approximate second order methods which are faster to compute and do not contain exact second order information. Instead, they estimate Hessian information using various approaches. Some examples are:

1. **Quasi-Newton methods** estimate the Hessian matrix (second order information) using only first order derivatives. The full Newton's method updates the iterate as follows:
$$\mathbf{w}_k = \mathbf{w}_{k-1} - (\nabla^2 f(\mathbf{w}_{k-1}))^{-1} \nabla f(\mathbf{w}_{k-1}), \tag{2.15}$$
where $\nabla^2 f(\mathbf{w}_{k-1})$ is the Hessian matrix of $f(\mathbf{w}_{k-1})$. This step is the solution to the first order critical point of the second order approximation of $f(\mathbf{w}_{k-1})$ using the Taylor series expansion, (2.17). Quasi-Newton methods replace $(\nabla^2 f(\mathbf{w}_{k-1}))$ with an approximation to the Hessian. A common quasi-Newton method is the **BFGS (Broyden-Fletcher-Goldfarb-Shanno) algorithm** which approximates the Hessian using the secant equation [30].

2. **Finite Difference Newton Methods** are methods where $(\nabla^2 f(\mathbf{w}_{k-1}))$ in (2.15) is replaced by a finite difference approximation to the Hessian which uses only first order information [30].

3. **Levenberg-Marquardt** is an algorithm for non-linear least squares problems, which has an update of the following form [30]:

$$\mathbf{w}_k = \mathbf{w}_{k-1} + [\mathbf{J}_{k-1}^T \mathbf{J}_{k-1} + \lambda \mathbf{I}]^{-1} \mathbf{J}_{k-1}^T \mathbf{r}_{k-1}. \tag{2.16}$$

   Here $\mathbf{r}_{k-1}$ is the residual vector which contains values $(\mathbf{y}^k - \mathbf{z}_L^k)_i$ for all indices $i = 1, ..., n_L$ where $\mathbf{y}^k \in \mathbb{R}^{n_L}$, and all training examples $k = 1, ..., m$, identified by the superscript. The iteration index is $k-1$, identified by the subscript. The matrix $\mathbf{J}_{k-1}$ is the Jacobian of $\mathbf{r}_{k-1}$ in terms of each weight, and λ is a damping factor which is adapted based on descent speed [30].

These methods are quite effective at approximating curvature in many instances without the computational time cost of actually computing it directly. BFGS in particular is a very well known approach for incorporating second order information based on recent updates. These, however, do not have the full second order information, e.g., the negative curvature information may be absent in the Hessian update. There are also methods that use exact second order information which we are interested in for this thesis.

1. **Newton's Method** is a classical second order optimization method which is only locally convergent. The Newton's method update is defined by (2.15). This method is the inspiration for some second order methods for neural network learning such as the "Approximate saddle-free Newton" method proposed by [6].

2. **Trust Region Methods** are globally convergent methods. At each iteration a solution to a subproblem which approximates the original problem, in a trusted region, based on the first and second order information of the objective function, is used to determine a step towards the solution for the original problem. This is the approach we focus on for this thesis.

We specifically investigate a trust region method with exact Hessian information in this thesis as well as modified versions based on subsampling of parameters and training examples. Part of our motivation in looking at trust region methods is that they are well suited to non-convex problems since a global minimizer to the subproblem is computed at each iteration. This does not mean a global minimizer will be computed for the problem but does lead to convergence to a local minimizer.

2.3 Trust Region Methods

The idea of a trust region method is to solve a subproblem at each iteration based on a model that is trusted within a certain radius around the current point. Trust region methods consist of the following steps [3]:

1. Subproblem definition.
2. Step calculation: solving the "Trust Region Subproblem".
3. Acceptance of the trial point.
4. Trust-region radius update.

In this section we will go through each step and specify the approach that will be used in our numerical exploration. Initialization specifies a guess for solution $\mathbf{w}$. In our investigation, the initial guess is uniformly pseudo-random values in the range of [-0.5,0.5]. The weight matrices, $\mathbf{W}_l$ at each layer l, are then normalized by dividing each matrix by the sum of the absolute value of all their matrix elements.

2.3.1 Subproblem Definition

Trust region methods typically use a quadratic approximation as the model for the objective function, $f(\mathbf{w})$. At the current iterate, the second order approximation to the objective function, $f(\mathbf{w}+\mathbf{p})$ is:

$$\begin{aligned}\tilde{f}(\mathbf{w}+\mathbf{p}) =& f(\mathbf{w}) + \nabla f(\mathbf{w})^T\mathbf{p} + \frac{1}{2}\mathbf{p}^T\nabla^2 f(\mathbf{w})\mathbf{p} \\ =& f(\mathbf{w}) + \mathbf{g}^T\mathbf{p} + \frac{1}{2}\mathbf{p}^T\mathbf{H}\mathbf{p},\end{aligned} \tag{2.17}$$

where $\mathbf{p}$ is the step (change in parameter values) we wish to find, $\mathbf{w}$ is the current parameter vector of the function f, $\mathbf{g} = \nabla f(\mathbf{w})$ is the gradient and $\mathbf{H} = \nabla^2 f(\mathbf{w})$ is the Hessian matrix at $\mathbf{w}$.

In this thesis $\mathbf{H}$ denotes the exact Hessian rather than an approximation, which is common among numerical methods for ff-ANNs as discussed in §2.2.2.

2.3.2 Step Calculation: Solving the Trust Region Subproblem

The step calculation of a trust region method computes the minimizer of the quadratic approximation within a trust region [3] formally written as:

$$\begin{aligned} \min_{\mathbf{p}} \quad & \mathbf{g}^T\mathbf{p} + \frac{1}{2}\mathbf{p}^T\mathbf{H}\mathbf{p} \\ \text{s.t.} \quad & \mathbf{p}^T\mathbf{p} \leq \gamma^2, \end{aligned} \tag{2.18}$$

where γ is the radius of the trusted region. Throughout this thesis, we will refer to the objective function of the trust region subproblem as $\delta(\mathbf{p})$, therefore:

$$\delta(\mathbf{p}) = \mathbf{g}^T\mathbf{p} + \frac{1}{2}\mathbf{p}^T\mathbf{H}\mathbf{p}. \tag{2.19}$$

Traditionally, the trust region subproblem is solved using an iterative method involving repeatedly computing solutions to linear equations or eigenvalue problems. It is demonstrated in a recent paper, [29], that the trust region subproblem can be solved by a single generalized eigenvalue problem and linear equation. This makes the trust region method more appealing since it is simple to implement with the possibility of improved computational efficiency. We provide the details of this computation.

Optimality Conditions

In this section we present and explain the optimality conditions for the solution to the trust region subproblem (TRS), (2.18), for the simple case when the solution is unique. The TRS is a special example of a non-convex problem with strong duality [5]. A solution to the dual problem is the Lagrangian multiplier which can be used to compute the solution to the primal problem. In the following section we will derive the optimality conditions for the TRS using the strong duality property of the problem. Firstly, in order to satisfy primal feasibility (2.18), we have:

$$\mathbf{p}^T\mathbf{p} \leq \gamma^2. \tag{2.20}$$

The Lagrangian function of (2.18) is:

$$\begin{aligned} \mathcal{L}(\mathbf{p}, \lambda) =& \mathbf{g}^T\mathbf{p} + \frac{1}{2}\mathbf{p}^T\mathbf{H}\mathbf{p} + \frac{1}{2}\lambda(\mathbf{p}^T\mathbf{p} - \gamma^2), \\ =& \mathbf{g}^T\mathbf{p} + \frac{1}{2}\mathbf{p}^T(\mathbf{H} + \lambda\mathbf{I})\mathbf{p} - \frac{1}{2}\lambda\gamma^2, \end{aligned} \tag{2.21}$$

where λ is the Lagrange multiplier. To satisfy complimentary slackness, λ and $\mathbf{p}$ must satisfy:

$$\lambda(\mathbf{p}^T\mathbf{p} - \gamma^2) = 0. \tag{2.22}$$

The dual function becomes:

$$\begin{aligned} d(\lambda) &= \inf_{\mathbf{p}} \mathcal{L}(\mathbf{p}, \lambda) \\ &= -\frac{1}{2}\lambda\gamma^2 + \inf_{\mathbf{p}} (\mathbf{g}^T\mathbf{p} + \frac{1}{2}\mathbf{p}^T(\mathbf{H} + \lambda\mathbf{I})\mathbf{p}). \end{aligned} \tag{2.23}$$

The infimum of the quadratic function in (2.23) is $-\infty$ when $(\mathbf{H} + \lambda\mathbf{I})$ is indefinite. For a value of λ such that $(\mathbf{H} + \lambda\mathbf{I})$ is singular we refer the reader to the "hard case" described in [29]. When $(\mathbf{H} + \lambda\mathbf{I}) \succ \mathbf{0}$ we can compute $d(\lambda)$ by computing the first order stationary point of $\mathcal{L}(\mathbf{p}, \lambda)$ in terms of $\mathbf{p}$, which, given positive definiteness of $(\mathbf{H} + \lambda\mathbf{I})$, means that this is a global minima of (2.21) in terms of $\mathbf{p}$. The first order condition gives us:

$$\begin{aligned} \nabla_{\mathbf{p}}\mathcal{L}(\mathbf{p}, \lambda) = \mathbf{g} + (\mathbf{H} + \lambda\mathbf{I})\mathbf{p} = \mathbf{0} \\ \to (\mathbf{H} + \lambda\mathbf{I})\mathbf{p} = -\mathbf{g}. \end{aligned} \tag{2.24}$$

We can now simplify the infimum from (2.23) by using (2.24) and assuming $(\mathbf{H} + \lambda\mathbf{I}) \succ \mathbf{0}$:

$$\begin{aligned} &\inf_{\mathbf{p}}(\mathbf{g}^T\mathbf{p} + \frac{1}{2}\mathbf{p}^T(\mathbf{H} + \lambda\mathbf{I})\mathbf{p}) \\ &\quad = -\mathbf{g}^T(\mathbf{H} + \lambda\mathbf{I})^{-1}\mathbf{g} + \frac{1}{2}\mathbf{g}^T(\mathbf{H} + \lambda\mathbf{I})^{-1}(\mathbf{H} + \lambda\mathbf{I})(\mathbf{H} + \lambda\mathbf{I})^{-1}\mathbf{g} \\ &\quad = -\mathbf{g}^T(\mathbf{H} + \lambda\mathbf{I})^{-1}\mathbf{g} + \frac{1}{2}\mathbf{g}^T(\mathbf{H} + \lambda\mathbf{I})^{-1}\mathbf{g} \\ &\quad = -\frac{1}{2}\mathbf{g}^T(\mathbf{H} + \lambda\mathbf{I})^{-1}\mathbf{g}. \end{aligned} \tag{2.25}$$

Using (2.25) to simplify (2.23), the dual function, we get:

$$d(\lambda) = -\frac{1}{2}\lambda\gamma^2 - \frac{1}{2}\mathbf{g}^T(\mathbf{H} + \lambda\mathbf{I})^{-1}\mathbf{g}, \tag{2.26}$$

for values of λ such that:

$$(\mathbf{H} + \lambda\mathbf{I}) \succ \mathbf{0}. \tag{2.27}$$

The remaining discussion is on a technique by Adachi et al [29] to determine the value of λ which maximizes (2.26). A stationary point must satisfy:

$$\begin{aligned} d'(\lambda)) &= \frac{1}{2}(-\gamma^2 + \mathbf{g}^T(\mathbf{H} + \lambda\mathbf{I})^{-2}\mathbf{g}) \\ &= 0. \end{aligned} \tag{2.28}$$

Using the eigenvalue decomposition for $\mathbf{H}$,

$$\mathbf{H} = \mathbf{V}^T\mathbf{D}\mathbf{V}, \tag{2.29}$$

where $\mathbf{D}$ is the diagonal matrix such that $\mu_i = \mathbf{D}_{ii}$ is the i^{th} eigenvalue of $\mathbf{H}$ by decreasing algebraic size (ie μ_1 is the largest eigenvalue of $\mathbf{H}$), and $\mathbf{V}$ is the matrix of eigenvectors where $\mathbf{V}_i$, the i^{th} column vector of $\mathbf{V}$, is the eigenvector corresponding to μ_i. We can now write (2.28) as:

$$\begin{aligned} 2d'(\lambda) &= -\gamma^2 + \mathbf{g}^T(\mathbf{V}^T(\mathbf{D} + \lambda\mathbf{I})\mathbf{V})^{-2}\mathbf{g} \\ &= -\gamma^2 + \mathbf{g}^T(\mathbf{V}^T(\mathbf{D} + \lambda\mathbf{I})\mathbf{V}\mathbf{V}^T(\mathbf{D} + \lambda\mathbf{I})\mathbf{V})^{-1}\mathbf{g} \\ &= -\gamma^2 + \mathbf{g}^T(\mathbf{V}^T(\mathbf{D} + \lambda\mathbf{I})(\mathbf{D} + \lambda\mathbf{I})\mathbf{V})^{-1}\mathbf{g} \\ &= -\gamma^2 + \mathbf{g}^T(\mathbf{V}^T(\mathbf{D} + \lambda\mathbf{I})^2\mathbf{V})^{-1}\mathbf{g} \\ &= -\gamma^2 + \mathbf{g}^T\mathbf{V}(\mathbf{D} + \lambda\mathbf{I})^{-2}\mathbf{V}^T\mathbf{g} \\ &= -\gamma^2 + \sum_{i=1}^{n} \frac{(\mathbf{g}^T\mathbf{V}_i)^2}{(\mu_i + \lambda)^2} = 0 \\ &\rightarrow \sum_{i=1}^{n} \frac{(\mathbf{g}^T\mathbf{V}_i)^2}{(\mu_i + \lambda)^2} = \gamma^2. \end{aligned} \tag{2.30}$$

We define $\phi(\lambda)$ as:

$$\phi(\lambda) = \sum_{i=1}^{n} \frac{(\mathbf{g}^T\mathbf{V}_i)^2}{(\mu_i + \lambda)^2}. \tag{2.31}$$

Now, note that $(\mu_i+\lambda)$ is the i^{th} eigenvalue of $(\mathbf{H}+\lambda\mathbf{I})$ so in order to satisfy the requirement in (2.27), $\mu_n + \lambda > 0$ where μ_n is the smallest eigenvalue of the Hessian matrix, $\mathbf{H}$. The λ which fulfills this constraint as well as (2.30), exists and is unique when $\mathbf{g}^T\mathbf{V}_n \neq 0$ since the value of $\phi(\lambda)$ is ∞ when $-\lambda = \mu_n$ and then always decreasing for $\lambda > |\mu_n|$ and approaches zero. A visualization of $\phi(\lambda)$ can be seen in [3], Figure 7.3.2. There is a case where the value of λ which maximizes (2.26) makes $(\mathbf{H}+\lambda\mathbf{I})$ semi-positive definite and singular which is discussed in [29].

To summarize, in order for a solution $\mathbf{p}$ to be optimal in the case that $(\mathbf{H} + \lambda\mathbf{I}) \succ \mathbf{0}$, it must satisfy primal feasibility, (2.20), (2.27), and complementary slackness, (2.22). Now that we have established optimality conditions for the TRS, we need to determine how to compute a solution that satisfies these conditions. The dual problem (2.26) can be solved by an iterative method to determine a solution to $\phi(\lambda) = \gamma^2$ where $\mathbf{H} + \lambda\mathbf{I} \succ 0$. For example, [3], finds a model minimizer using a numerical approach of adjusting λ and solving for (2.24) and re-adjusting λ until a suitable $\mathbf{p}$ is achieved. As previously mentioned, in this thesis we follow a more recent approach by [29] which, for the boundary case, developed a generalized eigenvalue problem with a solution from which a solution to the TRS can be directly computed. It also takes advantage of complementary slackness to devise a two step approach described in the following section.

Solving TRS via a Generalized Eigenvalue Problem

The method proposed by [29] uses a two pronged approach. They compute two steps, $\mathbf{p}_1$ and $\mathbf{p}_0$ where $\mathbf{p}_1$ is the optimal step for the trust region subproblem when (2.20) is an active constraint, and $\mathbf{p}_0$ is a point such that $\nabla\delta(\mathbf{p}_0) = 0$ where $\delta(.)$ is defined by (2.19). If $\mathbf{p}_0$ is a feasible solution, meaning (2.20) is satisfied, then $\mathbf{p}$ is the step, out of $\mathbf{p}_1$ and $\mathbf{p}_0$, which produces the lower objective function value of the TRS, $\delta(\mathbf{w} + \mathbf{p})$. This decision is presented as Algorithm 1.

Here we describe the details of the two parts of the method proposed in [29] as well as some explanation to show that this solution does, in fact, correspond to a solution to the TRS, (2.18). For further details on the derivation of this approach we refer a reader to [29].

Interior Case

If a solution to (2.18) is in the interior then the following must be satisfied:

$$\mathbf{H}\mathbf{p} = -\,\mathbf{g}, \tag{2.32a}$$

$$\mathbf{H} \succeq \mathbf{0}, \tag{2.32b}$$

$$||\mathbf{p}|| < \gamma, \tag{2.32c}$$

adapted from (2.24), (2.27) and (2.20) respectively for this case, since $\lambda = 0$ in order to satisfy complementary slackness, (2.22).

Define $\mathbf{p}_0$ to be a solution to (2.32a), which is unique when $\mathbf{H}$ is non-singular. If $\mathbf{H}$ is singular any of the feasible solutions can be submitted as the $\mathbf{p}_0$ solution. In the case where the hyperplane of optimal solutions intersects the trust region sphere, there will be at least one boundary case solution. Otherwise none of the possible solutions to $\mathbf{p}_0$ would satisfy (2.32c) and would therefore not be valid steps. Because of this, we can ensure that there is always a boundary solution which is a global minimizer in the case that $\mathbf{H}$ is singular. In our approach we do not check for singularity. Rather, we set a maximum number of iterations on our linear equation solver, which is the conjugate gradient method as will be discussed in the following chapter, and only consider solutions that satisfy the all of the above equations.

Boundary Case via a Generalized Eigenvalue Problem

If a solution is on the boundary then it satisfies (2.27), (2.24), and:

$$\mathbf{p}^T\mathbf{p} = \gamma^2. \tag{2.33}$$

We define $\mathbf{p}_1$ to be the vector that satisfies these conditions. In order to compute $\mathbf{p}_1$, [29] formulated a generalized eigenvalue problem such that $\mathbf{p}_1$ can be computed based on the eigenvalue, λ_*, and associated eigenvector:

$$\mathbf{v} = \begin{bmatrix} \mathbf{v}_1 \\ \mathbf{v}_2 \end{bmatrix}. \tag{2.34}$$

where $\mathbf{v}_1 \in \mathbb{R}^n$, $\mathbf{v}_2 \in \mathbb{R}^n$, together with λ_*, satisfy:

$$\begin{bmatrix} -\mathbf{I} & \mathbf{H} \\ \mathbf{H} & -\mathbf{g}\mathbf{g}^T/\gamma^2 \end{bmatrix} \begin{bmatrix} \mathbf{v}_1 \\ \mathbf{v}_2 \end{bmatrix} = -\lambda_* \begin{bmatrix} \mathbf{0} & \mathbf{I} \\ \mathbf{I} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{v}_1 \\ \mathbf{v}_2 \end{bmatrix}, \tag{2.35}$$

and λ_* is the largest eigenvalue satisfying this equation. We define the lefthand side of (2.35) to be matrix $\mathbf{M}$ for simplicity. That is:

$$\mathbf{M} = \begin{bmatrix} -\mathbf{I} & \mathbf{H} \\ \mathbf{H} & -\mathbf{g}\mathbf{g}^T/\gamma^2. \end{bmatrix} \tag{2.36}$$

The paper by Adachi et al [29] states that the optimal boundary solution is:

$$\mathbf{p}_1 = -\frac{\gamma^2}{\mathbf{g}^T\mathbf{v}_2}\mathbf{v}_1. \tag{2.37}$$

We can verify this by checking that the optimality conditions are satisfied. Firstly, we re-write (2.35) as two equations:

$$-\mathbf{v}_1 + \mathbf{H}\mathbf{v}_2 = -\lambda_* \mathbf{v}_2, \tag{2.38}$$

and

$$\mathbf{H}\mathbf{v}_1 - \mathbf{g}\mathbf{g}^T\mathbf{v}_2/\gamma^2 = -\lambda_* \mathbf{v}_1. \tag{2.39}$$

Rearranging (2.39) we get:

$$(\mathbf{H} + \lambda_* \mathbf{I})(-\frac{\gamma^2}{\mathbf{g}^T\mathbf{v}_2}\mathbf{v}_1) = -\mathbf{g}, \tag{2.40}$$

which is equivalent to (2.24) with the RHS of (2.37) replaced by $\mathbf{p}_1$. Therefore the dual feasibility, (2.24), is satisfied. Rearranging (2.38) we get:

$$\mathbf{v}_1 = (\mathbf{H} + \lambda_* \mathbf{I})\mathbf{v}_2. \tag{2.41}$$

Now we are able to show that $\mathbf{p}_1$ satisfies (2.33):

$$\begin{aligned}
\mathbf{p}_1^T\mathbf{p}_1 =&(-\frac{\gamma^2}{\mathbf{g}^T\mathbf{v}_2}\mathbf{v}_1)^T(-\frac{\gamma^2}{\mathbf{g}^T\mathbf{v}_2}\mathbf{v}_1) && \text{using (2.37)},\\
=&\frac{\gamma^4}{(\mathbf{g}^T\mathbf{v}_2)^2}\mathbf{v}_1^T\mathbf{v}_1, &&\\
=&\frac{\gamma^4}{(\mathbf{g}^T\mathbf{v}_2)^2}\mathbf{v}_2^T(\mathbf{H} + \lambda_* \mathbf{I})^T\mathbf{v}_1 && \text{using (2.41)},\\
=&\frac{\gamma^4}{(\mathbf{g}^T\mathbf{v}_2)^2}\mathbf{v}_2^T(\mathbf{g}\frac{\mathbf{g}^T\mathbf{v}_2}{\gamma^2}) && \text{using (2.40)},\\
=&\frac{\gamma^4}{(\mathbf{g}^T\mathbf{v}_2)^2}\frac{(\mathbf{g}^T\mathbf{v}_2)^2}{\gamma^2}, &&\\
=&\gamma^2. &&
\end{aligned}$$

The condition (2.27) is satisfied by taking the eigenvalue/eigenvector pair of (2.35) with the eigenvalue of the largest algebraic value which is explained in [29]. In the case where $(\mathbf{H} + I\lambda_*)$ is singular, we have what is considered the “hard case” which rarely occurs. See [29] for an approach to dealing with this case.

Choosing the trial point

The trial point $\mathbf{p}$ is chosen based on the decision as described in Algorithm 1 which considers both $\mathbf{p}_0$ and $\mathbf{p}_1$ to determine the global minimizer, $\mathbf{p}$, to (2.18).

Algorithm 1 Selecting Global Minimizer, $\mathbf{p}$, to (2.18)

```
$\delta(.)$ from (2.19), $\mathbf{w}$ and $\gamma$ are known
if $||\mathbf{p}_0|| \leq \gamma$ and $\delta(\mathbf{p}_0) < \delta(\mathbf{p}_1)$ then
    $\mathbf{p} \leftarrow \mathbf{p}_0$
else
    $\mathbf{p} \leftarrow \mathbf{p}_1$
```

2.3.3 Acceptance of the trial point

In order to determine the acceptability of the step, as well as adjust the trust region size adaptively, we use a ratio ρ defined as:

$$\rho = \frac{f(\mathbf{w} + \mathbf{p}) - f(\mathbf{w})}{\delta(\mathbf{p})}, \tag{2.42}$$

which is the ratio of true change in objective function to the estimated objective function value change based on our model, $\delta(\mathbf{p})$, defined by (2.19).

We follow the method presented in Fletcher (1987) [10] where a parameter, we refer to as lb, that satisfies $0 < lb < 1$, is used as the threshold for whether or not the step leads to a sufficient decrease in the objective function. This decision is presented as Algorithm 2.

Algorithm 2 Acceptance of the trial point [10]

```
$lb \in (0, 1)$
compute $\rho$ using (2.21)
if $\rho \geq lb$ then
    $\mathbf{w} \leftarrow \mathbf{w} + \mathbf{p}$
else
    $\mathbf{w}$ is unchanged
```

2.3.4 Trust region radius update

In this section we describe the method in [10] for adaptive adjustment of the trust region radius, γ. This update method is presented in Algorithm 3. In this method we use two parameters ub and lb where lb is the same value as in Algorithm 2 and ub is used to determine when the trusted region can be expanded. These parameters must satisfy:

$$0 < lb < ub < 1. \tag{2.43}$$

Conceptually this algorithm increases the radius of the trusted area when the model at the previous step is deemed very accurate, $\rho > ub$, shrink it if it was deemed insufficiently accurate, $\rho < lb$, and keep it constant otherwise. The shrinking is controlled by a parameter, α_s where $0 < \alpha_s < 1$, and the growth is controlled by a parameter $\alpha_g > 1$. When $\mathbf{w}$ is kept constant, $\mathbf{g}$ and $\mathbf{H}$ remain unchanged and can be re-used for the next iteration, reducing computation time.

Algorithm 3 Adaptive adjustment of trust region size

```
$lb$, $ub$, $\alpha_g$ and $\alpha_s$ are given where $lb$ and $ub$ satisfy (2.43), $\alpha_g > 1$ and $0 < \alpha_s < 1$
compute $\rho$ using (2.21)
if $\rho < lb$ then
    $\gamma \leftarrow \gamma\alpha_s$
else if $\rho > ub$ then
    $\gamma = \max(\gamma, \alpha_g||\mathbf{p}||)$
```

2.3.5 Method Summary

For clarity and the reader's reference, we present the full trust region algorithm used in this thesis in Algorithm 4 which combines all steps described in this section and includes the stopping criteria, e.g. stopping when $||\mathbf{g}|| > 10^{-8}$.

Algorithm 4 Full Trust Region Algorithm to Solve (2.12)

```
$lb$, $ub$, $\alpha_g$ and $\alpha_s$ are given where $lb$ and $ub$ satisfy (2.43), $\alpha_g > 1$ and $0 < \alpha_s < 1$
$\mathbf{w} \leftarrow$ initialized randomly
$\gamma \leftarrow 1$
while $||\mathbf{g}|| > 10^{-8}$ do
    if $\mathbf{w}$ was updated on the previous iteration then
        $\mathbf{g}, \mathbf{H} \leftarrow$ computed based on $\mathbf{w}$
        compute $\mathbf{p}_0$ s.t. $\mathbf{Hp}_0 = -\mathbf{g}$
    compute $\mathbf{p}_1$ from (2.37)
    if $||\mathbf{p}_0|| \leq \gamma$ and $\delta(\mathbf{p}_0) < \delta(\mathbf{p}_1)$ then
        $\mathbf{p} \leftarrow \mathbf{p}_0$
    else
        $\mathbf{p} \leftarrow \mathbf{p}_1$
    compute $f(\mathbf{w} + \mathbf{p})$ and $\delta(\mathbf{p})$
    compute $\rho$ from (2.42)
    if $\rho < lb$ then
        $\gamma \leftarrow \gamma\alpha_s$
    else
        $\mathbf{w} \leftarrow \mathbf{w} + \mathbf{p}$
        if $\rho > ub$ then
            $\gamma = \max(\gamma, \alpha_g||\mathbf{p}||)$
return $\mathbf{w}$
```

Chapter 3

Solving the Trust Region Subproblem for Feedforward Neural Networks

In the previous chapter we discussed trust region methods and an approach to solving the trust region subproblem by using a generalized eigenvalue problem proposed by Adachi et al. [29]. In this chapter we will propose a complete algorithm for using the trust region algorithm to train an ff-ANN. The algorithm follows the generalized eigenvalue framework using methods that are suited to training feedforward neural networks (ff-ANNs) to build upon Algorithm 4. The following problems need to be addressed in order to solve the TRS:

1. Computing second order derivatives for the objective function of an ff-ANN.
2. Computing the eigenvector associated with the largest eigenvalue for the generalized eigenvalue problem (2.35).
3. Solve the system of linear equations defined by (2.32a) where **H** is symmetric, and possibly indefinite.

This chapter discusses known algorithms that will be used to address the three problems in order to use the generalized eigenvalue framework.

3.1 Computing the Solution to the TRS

The trust region subproblem, (2.18), is solved using the approach described in Chapter 2. The approach is left where two solutions need to be computed: one to solve the linear system of equations, (2.32a), and the other requires computing the largest eigenvalue and it's associated eigenvector for the generalized eigenvalue problem (2.35). In this section we fill in these gaps to show how our solution was implemented.

3.1.1 Computing Second Order Derivatives

We compute second order information of the objective function, (2.9) with regularization, by using the "Pearlmutter Trick". This method uses a second round of forward and back-propagation after the first, which computes the gradient. Using this method, the product $\mathbf{H}_k\mathbf{v}$ can be computed for any vector $\mathbf{v}$ where $\mathbf{H}_k$ is the Hessian matrix of the objective function for the ff-ANN being trained, at the k^{th} iteration. The details of the method as it applies to ff-ANNs, are described in Appendix A.

3.1.2 Conjugate Gradient

As mentioned, the solution to the system of linear equations (2.32a) must be computed in order to compute the solution to the TRS. The conjugate gradient method is used as a fast descent method for solving this system of linear equations. The particular implementation used is that in the MATLAB *pcg* function. This method can be used either with a function call to an implementation of the Pearlmutter Trick, defined in Appendix A, or by being provided the computed Hessian.

The Hessian of the objective function at iteration k, $\mathbf{H}_k$, is sometimes indefinite. In context of the TRS algorithm however, the $\mathbf{p}_0$ step is only considered when $\mathbf{H}_k$ happens to be positive definite. If conjugate gradient does not converge to a minimizer because $\mathbf{H}_k$ is indefinite, we know that $\mathbf{p}_1$, the boundary solution, is the solution to the TRS.

3.1.3 Implicitly Restarted Arnoldi Method

The Implicitly Restarted Arnoldi Method (IRAM) is used to compute the eigenvalue/vector pair corresponding to the smallest eigenvalue of (2.35). A definition of this approach is presented in [41]. The particular implementation of this algorithm used is the MATLAB *eigs* function.

Algorithm 5 Conjugate Gradient, Algorithm 6.11 from [7]

```
procedure CG($\mathbf{A}$,$\mathbf{p}$)
    Given $\delta$
    $\mathbf{x}_0 \leftarrow \mathbf{0}$
    $\mathbf{r}_0 \leftarrow \mathbf{b}$
    $\mathbf{p}_0 \leftarrow \mathbf{b}$
    $k \leftarrow 0$
    while $\|\mathbf{r}_k\| > \delta$ do
        $k = k + 1$
        $\alpha_k = \mathbf{r}_{k-1}^T \mathbf{r}_{k-1} / (\mathbf{p}_{k-1}^T \mathbf{A} \mathbf{p}_{k-1})$
        $\mathbf{x}_k = \mathbf{x}_k + \alpha_k \mathbf{p}_{k-1}$
        $\mathbf{r}_k = \mathbf{r}_k + \alpha_k \mathbf{A} \mathbf{p}_{k-1}$
        $\mu_k = (\mathbf{r}_k^T \mathbf{r}_k) / (\mathbf{r}_{k-1}^T \mathbf{r}_{k-1})$
        $\mathbf{p}_k = \mathbf{r}_k + \mu_k \mathbf{p}_{k-1}$
    return $\mathbf{x}_k$
```

3.1.4 Computing $\mathbf{H}_k$

Using the "Pearlmutter Trick" proposed by [27] and it's specific application to ff-ANNs, we can now write a simple function to compute the Hessian matrix of the ff-ANN at each iteration k, $\mathbf{H}_k$, as we show in Algorithm 6. We do not require computation of the full $\mathbf{H}_k$ since the product $\mathbf{H}_k \mathbf{v}$, for any vector $\mathbf{v}$, is sufficient for the algorithm. However, computing $\mathbf{H}_k$ is found to be beneficial in some instances, which will be discussed in Chapter 4.

Algorithm 6 Compute $\mathbf{H}_k$, using the Pearlmutter Trick [27]

```
procedure COMPUTEHK:
    $\mathbf{H} \leftarrow$ initialize $n \times n$ matrix
    $n \leftarrow$ number of parameters
    $\mathbf{I} \leftarrow$ identity matrix in $\mathbb{R}^{n \times n}$
    for $i = 1$ to $n$ do
        $\mathbf{H}_i = Hv(\mathbf{I}_i)$
    return $\mathbf{H}$
procedure $Hv(\mathbf{v})$:
    [Four Pass Procedure in §3.1.1]
    return $\mathbf{H}_k \mathbf{v}$
```

3.1.5 Computing The Boundary Solution, $\mathbf{p}_1$

As described in Chapter 2, we can compute the boundary case solution, $\mathbf{p}_1$, from the solution to the generalized eigenvalue problem, (2.35), which is computed using the MATLAB *eigs* function. In Algorithm 7 we show our implementation of the $getp1$ method which returns the boundary case solution. In this algorithm we call a method $Mv()$ defined below $getp1$ which, given a vector $\mathbf{v} \in \mathbb{R}^{2n}$, returns the product, $\mathbf{Mv}$, where $\mathbf{M}$ is defined in (2.36). This method calls $Hv()$ which is presented as a subroutine of Algorithm 6.

Algorithm 7 Compute $\mathbf{p}_1$

```
procedure $getp1$
    given $\gamma$, $\mathbf{g}$ =vectorized($\nabla f(\mathbf{w})$)
    eigvec = eigs($Mv()$)
    $\mathbf{y}_1 \leftarrow eigvec[0 : n - 1]$
    $\mathbf{y}_2 \leftarrow eigvec[n : 2n - 1]$
    $\mathbf{p}_1 \leftarrow -sign(\mathbf{g}^T\mathbf{y}_2)\gamma\frac{\mathbf{y}_1}{|\mathbf{y}_1|}$
    return $\mathbf{p}_1$
procedure $Mv(v)$
    $\mathbf{v}_1 \leftarrow \mathbf{v}[1 : n]$
    $\mathbf{v}_2 \leftarrow \mathbf{v}[n + 1 : 2n]$
    $Mv[1 : n] \leftarrow -\mathbf{v}_1 + Hv(\mathbf{v}_1)$
    $Mv[n + 1 : 2n] \leftarrow Hv(\mathbf{v}_2) - \mathbf{g}\mathbf{g}^T\mathbf{v}_2/\gamma^2$
    return $Mv$
```

3.1.6 Summary of TRS Solution Method

We now have all the pieces to put together a full algorithm. This algorithm is presented as Algorithm 8 and can be fully implemented to solve the trust region subproblem [29] which is a *step* or iteration of the trust region method. This is the implementation used for experimentation in our numerical investigation, Chapter 5. Note that *pcg* refers to the MATLAB function, $getp1$ is the procedure defined in Algorithm 7 and $Hv()$ is from Algorithm 6. The $Mv()$ function is called from $getp1$ and is therefore still present in the complete algorithm.

Algorithm 8 Solving the TRS

```
procedure Step k:
    $\alpha_s$, $\alpha_g$ as defined in Algorithm 4
    $\mathbf{g}_k$ is computed using the backpropagation, see Appendix A.0.1
    $\mathbf{p}_1 \leftarrow getp1$
    $\mathbf{p}_0 \leftarrow pcg$
    $\delta(\mathbf{p}_1) \leftarrow (\mathbf{p}_1)^T g + \frac{1}{2}(\mathbf{p}_1)^T Hv(\mathbf{p}_1)$
    $\delta(\mathbf{p}_0) \leftarrow (\mathbf{p}_0)^T g + \frac{1}{2}(\mathbf{p}_0)^T Hv(\mathbf{p}_0)$
    if $||\mathbf{p}_0|| \leq \gamma$ and $\delta(\mathbf{p}_1) > \delta(\mathbf{p}_0)$ then
        $\mathbf{p}^* \leftarrow \mathbf{p}_0$
        $\delta(\mathbf{p}^*) \leftarrow \delta(\mathbf{p}_0)$
    else
        $\mathbf{p}^* \leftarrow \mathbf{p}_1$
        $\delta(\mathbf{p}^*) \leftarrow \delta(\mathbf{p}_1)$
    $\Delta f \leftarrow f(\mathbf{w} + \mathbf{p}^*) - f(\mathbf{w})$
    $\rho \leftarrow \frac{\Delta f}{\delta(\mathbf{p}^*)}$
    if $\rho < lb$ then
        $\gamma \leftarrow \gamma\alpha_s$
    else
        $\mathbf{w} = \mathbf{w} + \mathbf{p}^*$
        if $\rho > ub$ then
            $\gamma \leftarrow \gamma\alpha_g$
```

Chapter 4

Modified Trust Region Methods

This thesis seeks to study the behaviour of the trust region method, **TRM**, and variations of this method for training ff-ANNs. We take several different approaches to variations of **TRM**. Most of our modifications are meant to reduce the runtime of the method for each step computation. This means that we can expect each iteration will contribute less to the solving the overall problem, but it will be computed faster. Numerical investigation into the speed of the objective function reduction in terms of CPU time for each method will provide us with information as to which approaches are effective at speeding up the **TRM**.

4.0.1 Challenge: Stochastic Variations on the TRM

The trust region radius update is one of the main components of the trust region method. It is the mechanism that ensures the objective of the trust region subproblem is an accurate approximation to the change of the true objective while keeping the trust region size as large as possible to ensure fast progress. The idea being that when the quadratic model is not accurate for the current point, we increase its accuracy by shrinking the trust region size at that step. This relies on continuity between iterations, where we can predict how accurate the approximation at a particular iteration will be, based on the previous iteration. Unfortunately, this is not necessarily possible when we are subsampling training examples to estimate the gradient and Hessian information. This leads to an interesting question that motivates most of the investigation in this thesis: Is it possible to harness the benefits of TRM while using a stochastic subsampling method at each iteration? This is a concern for the **TRM** and is not for Newton's method which does not rely on information from a previous iteration to determine step size, which means this problem is interesting but challenging. We kept this in mind when developing the following approaches to stochastic subsampling of training samples.

4.1 Stochastic Subsampling of Training Samples

In the context of this thesis, stochastic subsampling of training samples (SSTS) refers to approximating the objective function, gradient, and Hessian using only a subset of training samples at each iteration. We define $\mathbf{c}_k \in \mathbb{R}^{b_m}$ to be an array that contains the parameter indices randomly selected for the stochastic approximation at iteration k, where $b_m < m$ is the batch size of training samples used each iteration. The approximated objective function becomes:

$$f^m(\mathbf{w}_k) = \frac{1}{2b_m} \sum_{r=1}^{b_m} (\mathbf{z}_L^{(c_k)_r} - \mathbf{y}^{(c_k)_r})^T (\mathbf{z}_L^{(c_k)_r} - \mathbf{y}^{(c_k)_r}) + \lambda^{reg} \mathbf{w}_k^T \mathbf{w}_k, \tag{4.1}$$

where f^m refers to the approximation of the objective function based on training example subsampling and λ^{reg} is the regularization constant. The approximate objective function for the TRS is then:

$$\delta^m(\mathbf{p}_k) = (\mathbf{g}_k^m)^T \mathbf{p}_k + \frac{1}{2} (\mathbf{p}_k)^T \mathbf{H}_k^m \mathbf{p}_k, \tag{4.2}$$

where $\mathbf{p}_k \in \mathbb{R}^{b_m}$,

$$\mathbf{g}_k^m = \nabla f^m(\mathbf{w}_k), \tag{4.3}$$

and

$$\mathbf{H}_k^m = \nabla^2 f^m(\mathbf{w}_k). \tag{4.4}$$

We want to determine whether SSTS can be used with a standard trust region size, γ, update approach, Algorithm 3. In order to determine the effectiveness of SSTS in the context of the otherwise unchanged **TRM** we implement the Batch Trust Region Method (**BTRM**). The value of b_m for this method is given by a hyperparamter b_m^{large} which has a value specifically chosen for the dataset. The approach for choosing the value is presented in Appendix B.

For our second group of stochastic variations we note that this SSTS method for approximating the gradient, $\mathbf{g}_k$, by $\mathbf{g}_k^m$ is the same method used for stochastic gradient methods such as stochastic gradient descent (**SGD**) and mini-batch gradient descent (**MBGD**), where the update is:

$$\mathbf{w}_{k+1} = \mathbf{w}_k + \eta \mathbf{g}_k^m, \tag{4.5}$$

where η is the learning rate and **SGD** refers to the special case of **MBGD** where $b_m = 1$. We want to study whether this approach, as used for stochastic gradient methods, is also effective for **TRM**. Therefore, we use subsampling with a single training sample and with a batch size to mirror **SGD** and **MBGD** respectively. We develop two methods for numerical experimentation: stochastic trust region method (**STRM**) and mini-batch trust region method (**MBTRM**). For **STRM** we have $b_m = 1$ and for **MBTRM** we use $b_m = b_m^{small}$ where b_m^{small} is a hyperparameter chosen based on the dataset. The hypothesis is that we can gain some of the benefit that is seen in **SGD** and **MBGD** compared to plain gradient descent (**GD**). Both of these methods, in general, greatly decrease the CPU time it takes to compute an approximate solution, an example of which that was performed on the MNIST dataset [21] is shown in Figure 4.1. This example illustrates the improvements stochastic subsampling methods make to the speed of gradient descent. Note that **GD** does eventually reduce the objective function value in this example, Figure 4.1 is meant to demonstrate timing differences only.

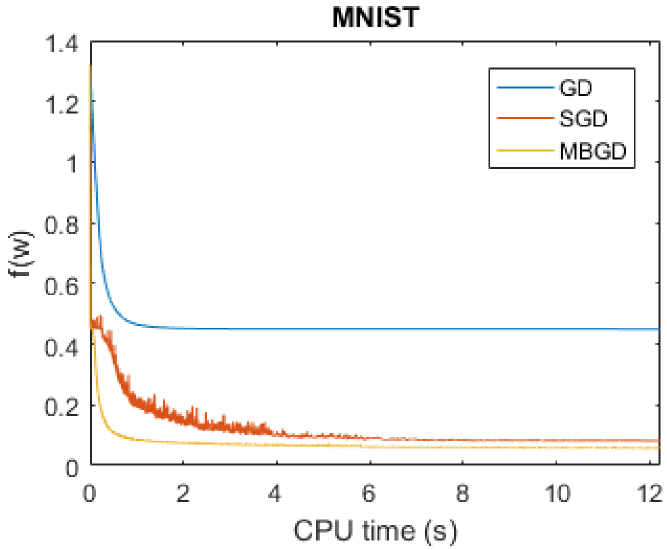

Figure 4.1: Training an ff-ANN to predict MNIST using **SGD**, **MBGD** and **GD** for comparison.

A learning rate schedule that is often used in practice for stochastic gradient methods is to half the learning rate when progress slows averaging over an epoch [2]. We use a trust region update schedule for our stochastic trust region methods inspired by this learning rate schedule. This schedule is shown in Algorithm 9, where a is a block size, k is the

Algorithm 9 Adaptive Trust Region Reduction Schedule (ATRRS)

```
function $\gamma =$ UPDATEGAMMA($a$, $k$, $e_c$, $e_{c-1}$ AND $\gamma$)
    if $k$ mod $a == 0$ then
        if $e_c > e_{c-1} + tol$ then
            $\gamma \leftarrow \gamma/2$
        else
            do nothing
    else
        do nothing
```

number of steps taken so far and the value computed as:

$$e_c = \frac{1}{a} \sum_{k=a(c-1)+1}^{ac} f^m(\mathbf{w}_k), \tag{4.6}$$

where e_c is the average gradient during c^{th} epoch and

$$c = \lfloor k/a \rfloor . \tag{4.7}$$

Recall that γ is the trust region size, or "step size" in the context of adjusted trust region methods. We refer to this schedule we use as the adaptive trust region reduction schedule (ATRRS) and compute only the $\mathbf{p}_1$ step from (2.37). This schedule takes into account the average approximated objective function over a set number of iterations, which we refer to as a **block**, that has number of iterations a. For our experiments we define a to be:

$$a = \min(\frac{m}{b_m^{small}}, b_m^{large}), \tag{4.8}$$

where b_m^{small} is the value of b_m for **MBTRM** and b_m^{large} is approximately the number of training samples that can adequately represent the dataset in order to run **TRM** which is b_m for **BTRM**. The parameters b_m^{small} and b_m^{large}, known as hyperparameters, are chosen for our experiments using an hyperparameter protocol defined in Appendix B, which is run previous to training. This choice of iteration span is to either wait until m training samples have been considered (with possible overlap), or consider a **block** of b_m^{large} iterations which is not dependent on the dataset size itself but rather the characteristics of the data. This means that in cases where b_m is small, b_m^{large} is used as an upper bound for how often to check progress of the stochastic method. It can also be used for stochastic trust region implementations in online learning contexts where the total training set size is not set.

Since these methods use a small value of b_m the computation of $\mathbf{p}_1$ is faster using the “Pearlmutter Trick” each time the product $\mathbf{H}_k\mathbf{v}$ must be computed for some vector $\mathbf{v}$ it will take $O(b_m n)$ computations rather than $O(mn)$.

4.2 Weight Subsampling

The other dimension which we can reduce, in the hopes of reducing CPU time for training, is the number of parameters n, which is the size of $\mathbf{w}_k$ for any iteration k. In this thesis we refer to weight subsampling as the random selection of indices of $\mathbf{w}_k$, which are the variables of the TRS, at iteration k, over which to minimize. We define $\mathbf{a}_k \in \mathbb{R}^{b_w}$, where $b_w < n$ is the number of weights in the subset, to be an array that contains the random indices selected for the weight subset at iteration k. Considering only the parameters which have indices contained in $\mathbf{a}_k$, the corresponding objective function for the approximate TRS (trust region subproblem) is:

$$\delta^w(\mathbf{p}_k^w) = (\mathbf{g}_k^w)^T\mathbf{p}_k^w + \frac{1}{2}(\mathbf{p}_k^w)^T\mathbf{H}_k^w\mathbf{p}_k^w \tag{4.9}$$

where $\mathbf{p}^w \in \mathbb{R}^{b_w}$,

$$(\mathbf{g}_k^w)_i = (\mathbf{g}_k)_{(\mathbf{a}_k)_i}, \tag{4.10}$$

and

$$(\mathbf{H}_k^w)_{ij} = (\mathbf{H}_k)_{(\mathbf{a}_k)_i(\mathbf{a}_k)_j}. \tag{4.11}$$

In our investigation we want to see how weight subsampling affects the behaviour of the **TRM**, therefore we implement an algorithm called the trust region method with weight subsampling, **TRMWS**, which follows Algorithm 8 with the following changes. At the beginning of each iteration a new set of weight indices, $\mathbf{a}_k$, are randomly chosen and throughout the iteration, the weight subset version of the gradient, $\mathbf{g}_k^w$ and that of the Hessian $\mathbf{H}_k^w$ is used in the place of the full gradient and Hessian at that iteration, $\mathbf{g}_k$ and $\mathbf{H}_k$ respectively. This means that the variables whose indices are not present in $\mathbf{a}_k$ are not updated in iteration k.

We also wish to study the effect of weight subsampling on stochastic variations of **TRM** that already incorporate SSTS, therefore, we implemented three methods: stochastic trust region method with weight subsampling, **STRMWS**, mini-batch trust region method with weight subsampling, **MBTRMWS**, and batch trust region method with weight subsampling, **BTRMWS**, which are all extensions of methods from the previous section with the addition of weight subsampling.

4.3 Hybrid Approach

The final approach we consider is a hybrid method combining mini-batch gradient descent, **MBGD**, and **TRM** in order to retain the speed benefit generally seen with **MBGD**, a well known approach to training ff-ANNs, while decreasing the objective function value using a **TRM** step occasionally. Note that for the experiments in this thesis the implementation of **MBGD** and stochasti gradient descent, **SGD**, updates the learning rate using Algorithm 9 where trust region size, γ, is replaced by learning rate, η, for the stochastic-type gradient method 2.14.

Algorithm 10 MBGD η Update and **TRM** Step Incorporation

```
function $\eta$ = CHECKPROGRESS($a$, $k$, $e_c$, $e_{c-1}$ AND $\eta$)
    if $r \bmod a == 0$ then
        if $e_c > e_{c-1} + tol$ then
            $\eta \leftarrow \eta/2$
            [Take one TRM step with $\gamma = \eta$]
        else
            do nothing
    else
        do nothing
```

Our hybrid approach, which we refer to as **TRMMBGD**, is to follow **MBGD** but each time the learning rate is halved due to a lack of progress between iteration blocks, the learning algorithm solves the TRS for the next step rather than using the approximate gradient step and then returns to **MBGD** on the following iteration. This is shown in Algorithm 10. By using a TRS step only when we are not seeing progress in reducing the objective function with **MBGD**, we hope to benefit heavily from the fast first order decent by **MBGD** and reserve solving the TRS for when it would be most beneficial.

4.4 Full Hessian vs Hessian Free

Our methods can all be performed using either a function which computes the product $\mathbf{H}_k\mathbf{v}$ for any given vector $\mathbf{v}$, or by using the full Hessian $\mathbf{H}_k$ which must be computed first. When using a Hessian free approach, the “Pearlmutter Trick” (see Appendix A) must be performed each time $\mathbf{H}_k\mathbf{v}$ is computed for some vector $\mathbf{v}$. This takes $O(mn)$ computations where m is the number of training samples and n is the number of parameters. Computing

$\mathbf{H}_k\mathbf{v}$ when we compute $\mathbf{H}_k$ only takes $O(n^2)$ and often $n < m$. This, however, requires the added cost of computing $\mathbf{H}_k$. We simplified this discussion by using symbols n, m, and $\mathbf{H}_k$ which are the symbols relevant to the **TRM**. This discussion is also true for b_w, any value of b_m and the approximate or lower dimension Hessian matrix being used. It is therefore relevant to all method variations presented in this chapter.

In order to decide for which methods we compute the full Hessian matrix and for which we did not, we timed the first 1000 iterations (or fewer for those that converged prior to 1000 iterations) of all methods on all datasets using both approaches. These results are presented in Appendix C. We chose to keep the Hessian computation approach for each method consistent across datasets for our experimental results. If a method computed each step taking on average less CPU time using the Hessian free approach for the majority of datasets, the Hessian free approach was used for all datasets in experimentation and otherwise the full Hessian approach was used. We show the resulting choice of Hessian computation in Table 4.1.

Method	Hessian Computation Approach
TRM	FH
TRMWS	FH
BTRM	**HF**
BTRMWS	FH
MBTRM	**HF**
MBTRMWS	FH
STRM	**HF**
STRMWS	FH
TRMMBGD	**HF**

Table 4.1: The approach used for computing Hessian information in each method. HF refers to “Hessian Free” and FH refers to “Full Hessian”. HF is in bold to more easily distinguish between the two values.

4.5 Summary

Descriptions of labels for the methods used in this investigation are presented in Table 4.2, which use the definitions of weight subsampling and stochastic subsampling of training samples (SSTS) from this chapter. These methods cover a broad range of approaches to reduce the computations required for a step of the base method, **TRM**. The method of choosing hyperparameters (b_w, b_m^{large}, b_m^{small} etc) is defined in Appendix B.

Label	Method definition / description
TRM	standard trust region method, Algorithm 4.
TRMWS	**TRM** with weight subsampling.
BTRM	**TRM** with SSTS.
BTRMWS	**TRM** with SSTS and weight subsampling.
MBTRM	**TRM** with SSTS, using only the $\mathbf{p}_1$ step and the adaptive trust region reduction schedule (ATRRS), Algorithm 9.
MBTRMWS	**MBTRM** with weight subsampling as well.
STRM	special case of **MBTRM** where $b_m = 1$.
STRMWS	special case of **MBTRMWS** where $b_m = 1$.
MBGD	Mini-batch gradient descent with learning rate updated using the Algorithm 9 after replacing γ by the learning rate.
SGD	**MBGD** with $b_m = 1$.
TRMMBGD	**MBGD** where **TRM** step is used when **MBGD**'s progression slows.

Table 4.2: Training algorithm definitions used for numerical exploration, and their labels, which will be used to reference them.

Chapter 5

Numerical Results

In this chapter we present the results from a large array of experiments. We compare methods, defined in Table 4.2, performed on various datasets in order to gain insight into their behaviour when used for training ff-ANNs.

5.1 Experimental Set-up

The algorithms presented in Chapter 4 are implemented in MATLAB and available at https://github.com/cwkinros/Stochastic_TRM_Exploration. Experiments were run on individual nodes of the University of Waterloo Chardonnay cluster consisting of 8 nodes. Each of these nodes have 2x Intel E5-2671 (8C) CPUs, 128 GB of memory and 15T LSI SAS2308 of disk space. The nodes run Linux and the specific experiments are run in the 64-bit R2013b version of MATLAB on these machines.

5.1.1 Datasets

Sizes of datasets are displayed in Table 5.1. The datasets were chosen to have a variety of sizes in terms of the number of independent variables, n_0, and the number of classes, both which affect the number of parameters, n, in the resulting ff-ANN. They also have differing numbers of training examples, m. They cover a few application domains as shown in Table 5.2, and are expected to have different properties. For instance, most datasets have separable classes, except Habe where the class label appears virtually unrelated to the independent variables as can be seen in Figure 5.1.

label	m	n_0	**# classes**	n
MNIST	60000	100	10	1120
Derm	366	33	6	406
IRIS	150	4	3	83
Nurs	12960	8	5	145
Habe	306	3	2*	51

Table 5.1: Datasets used for experimentation. *Habe uses a single dependent variable with two states, each to represent one of the classes.

label	**Description**	**Source**
MNIST	handwritten digits	[21]
Derm	differential diagnosis of erythemato-squamous diseases	[15]
IRIS	predict class of iris plant	[9]
Nurs	rank applications to nursery schools	[8]
Habe	predict survival status for patients with breast cancer	[16]

Table 5.2: Datasets used for experimentation.

5.1.2 Network Structure

The networks used for experimentation are two-layer, fully connected ff-ANNs. The activation function used is the sigmoid function, both at the hidden layer and at the final layer:

$$\sigma_1(x) = \sigma_2(x) = \frac{1}{1 - e^{-x}}. \tag{5.1}$$

An example of the network structure, applied to the Derm dataset, is shown in Figure 5.2. The value for n_0 depends on the dataset, and can be seen in Table 5.1, and n_2 is the number of classes in the networks shown in Table 5.1 for our experiments, with the exception of the Habe dataset where we use $n_2 = 1$ for a two class problem. All networks for the five datasets in Table 5.1 contain 10 hidden units.

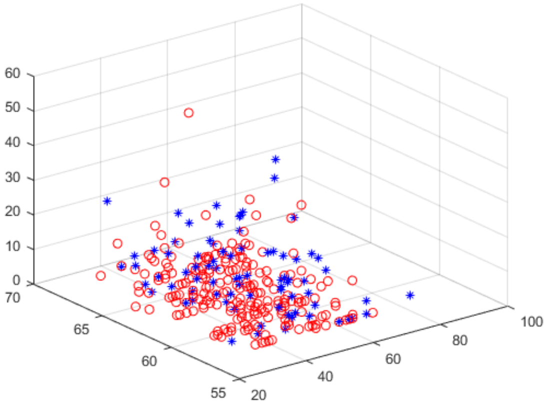

Figure 5.1: Visualization of Habe dataset where classes are represented by shape and colour. This shows the difficulty in learning this dataset, there is no clear separation between classes.

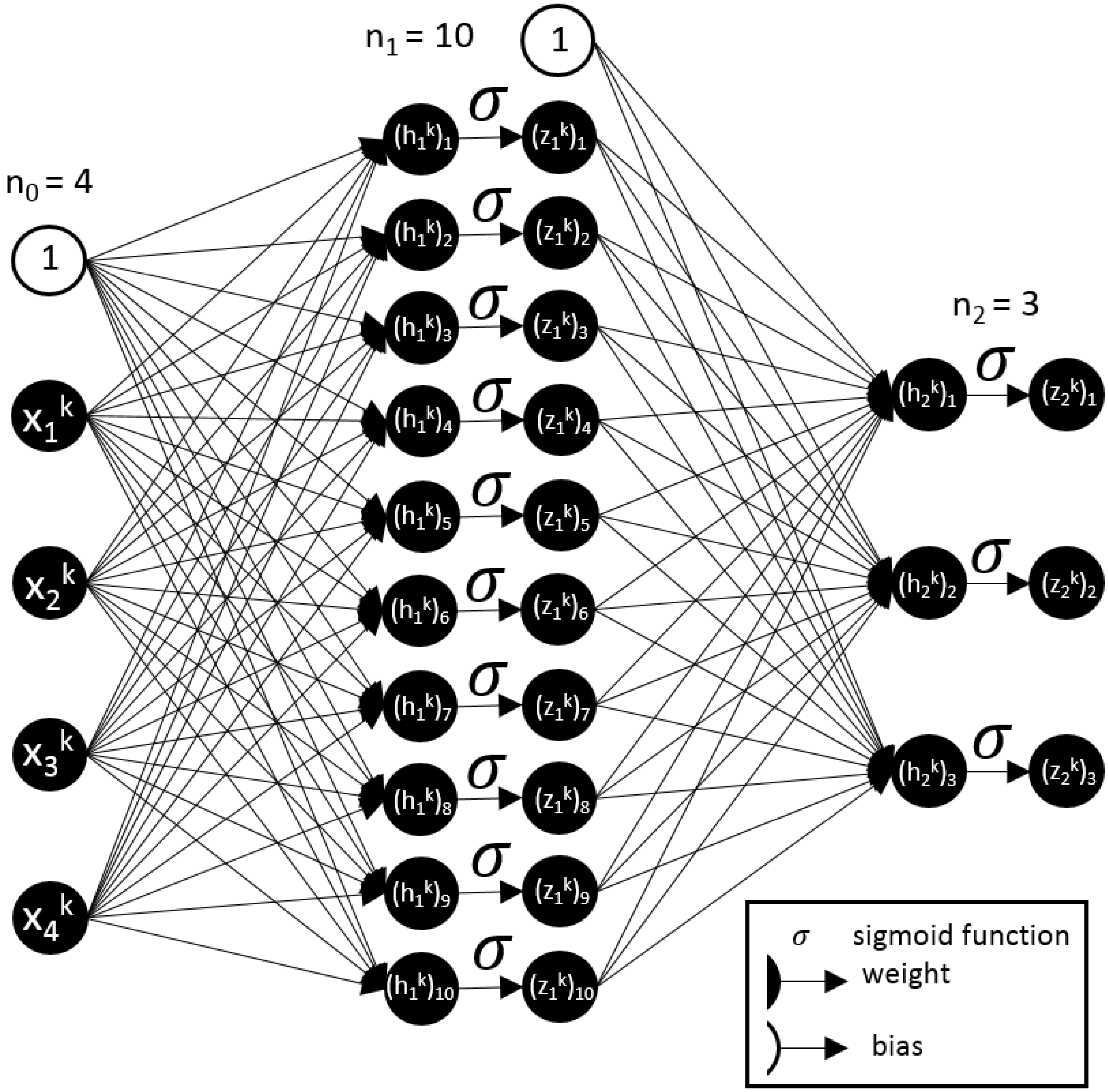

Figure 5.2: Structure of ff-ANN for learning the Derm dataset. Used as visual example to show resulting structure for a given n_0 and n_2 which are based on the dataset.

5.2 Methods for Comparison

In this section we define criteria we use for ranking optimization curves, the curve representing the objective function value over time, while undergoing an iterative optimization method. It can be difficult to make comparisons between two different optimization curves that display different behaviours. In order to be specific in our analysis of optimization algorithms we define a ranking method. To begin, we first present the two possible comparison approaches include:

1. CPU time taken to reach a specific objective function value.
2. Objective function value achieved within a certain time.

Both of these metric comparisons involve choosing either a specific objective function value or time to set as a fixed value and then compare using the other metric. The final measurement can depend on this choice, making it critical that there is an objective procedure used to make the choice. We will analyze the majority of the experimental results using the metric defined in the following section, which uses the data to define an objective function value for CPU time measurements. Simpler comparison methods will be explained when they are used.

5.2.1 Time Required for a Reasonable Solution

The main comparison method we use in our numerical investigation is what we refer to as time required for a reasonable solution (TRRS). This follows the first metric concept where we compare the CPU time taken to reach a specific objective function value. The specific objective function value used is dependent on the methods being compared which means that the TRRS can only be used to compare two methods, otherwise this objective function may differ and change the result.

Before we define TRRS, we will first introduce a few notations. Suppose K_a and K_b are the final iteration numbers for method a and b, respectively. Let $cpu(K_a)$ ($cpu(K_b)$) be the CPU time taken for performing K_a (K_b) iterations by method a (method b). For the purpose of comparison, we will fix the time for both methods and the fixed time is chosen to be the minimum of the two:

$$T = min(cpu(K_a), cpu(K_b)). \tag{5.2}$$

Accordingly, let K_T^a (K_T^b) be the smallest iteration number for method a (method b) such that $cpu(K_T^a) >= T$ ($cpu(K_T^b) >= T$).

We then consider the values of the objective function for all iterates generated by the two methods and find the minimum one. More precisely, we define:

$$\kappa = min(f(\mathbf{w}_{k_a}^a), f(\mathbf{w}_{k_b}^b)) \quad k_a = 0, 1, ..., K_T^a, \; k_b = 0, 1, ..., K_T^b \tag{5.3}$$

where $\mathbf{w}_{k_a}^a$ and $\mathbf{w}_{k_b}^b$ are the approximate solutions generated by method a and method b, respectively.

Now we call a solution, $\mathbf{w}_k$, a reasonable solution if $f(\mathbf{w}_k)$ is close to k. Specifically,

$$f(\mathbf{w}_k) <= \psi, \tag{5.4}$$

where

$$\psi = k + \Delta|f(\mathbf{w}_0) - k|. \tag{5.5}$$

Here $\mathbf{w}_0$ is the vector of initial parameters and Δ is given by the user.

Finally, TRRS for method a is defined to be $cpu(K_a^\psi)$ where K_a^ψ is the first iteration such that $\mathbf{w}_{K_a^\psi}^a$ is considered a reasonable solution. TRRS for method b is defined similarly.

5.3 CPU time of TRM vs SGD

The first question we wanted to answer in this investigation is simply: How much slower is the **TRM** compared to **SGD**. We run **SGD** to train each dataset in Table 5.2. At this point we search for the smallest objective function value achieved during **SGD** training and define our CPU time metric to be when the CPU time at the first iteration that a method reaches within 1% (based on the initial objective function) of the smallest objective function value achieved by **SGD**. The results are presented in Table 5.3.

These results show that it takes **TRM** much longer, in terms of CPU time, to reach the goal objective function value computed using the **SGD** results, than **SGD**. The proportional time discrepancy is particularly large for MNIST and Derm datasets which both have a larger number of parameters. MNIST has by far the largest magnitude increase in time, which makes sense since it has both the largest number of parameter and the largest number of training samples.

Dataset	CPU time (s) **TRM**	CPU time (s) **SGD**
Derm	5.0535	**0.23578**
Habe	0.14527	**0.092241**
IRIS	0.46323	**0.18152**
MNIST	2607.145	**4.2025**
Nurs	39.7811	**5.3361**

Table 5.3: Time taken to reach within 1% of minimum $f(\mathbf{w})$ achieved by **SGD** for both **SGD** and **TRM**.

5.4 Using TRM and MBGD in Hybrid: TRMMBGD

In this section we explore whether we can make an improvement to traditional stochastic gradient methods, specifically **MBGD**, using the **TRM** step when necessary. Recall the mechanism for switching between the two described in §4.3, which combines them by prioritizing **MBGD** and only using **TRM** when **MBGD** is not effectively reducing the objective function.

The results of this experiment are found in Figure 5.3. The two methods, **MBGD** and **TRMMBGD** perform very similarly at low CPU time with **MBGD** slightly outperforming **TRMMBGD** in terms of values of TRRS for $\Delta = 5\%$ and 10%, shown in Tables 5.5 and 5.6 respectively. In all TRRS tables, the faster method based on the Δ used in each set of results, is in bold. The dash represents when there is no TRRS for a method, which means that the method did not achieve a reasonable solution based on the definition of reasonable in §5.2.1. Looking at values of TRRS for $\Delta = 1\%$, shown in Table 5.4, we see that either **TRMMBGD** performs as well as **MBGD**, where there is less than a 0.1 second discrepency for the TRRS, or in the case of Habe, IRIS and Nurs, **TRMMBGD** achieves a better solution overall. The most extreme improvement in objective function value is seen by IRIS in Figure 5.4, where **TRMMBGD** continues to make significant improvements to the objective function while the objective function being minimized by **MBGD** has stopped decreasing in magnitude.

In conclusion, we find that **TRMMBGD** is an effective improvement over **MBGD**. In some cases the two methods perform very similarly, whereas for some datasets the optimization curve of **TRMMBGD** performs similarly early on but achieves a lower final objective function eventually.

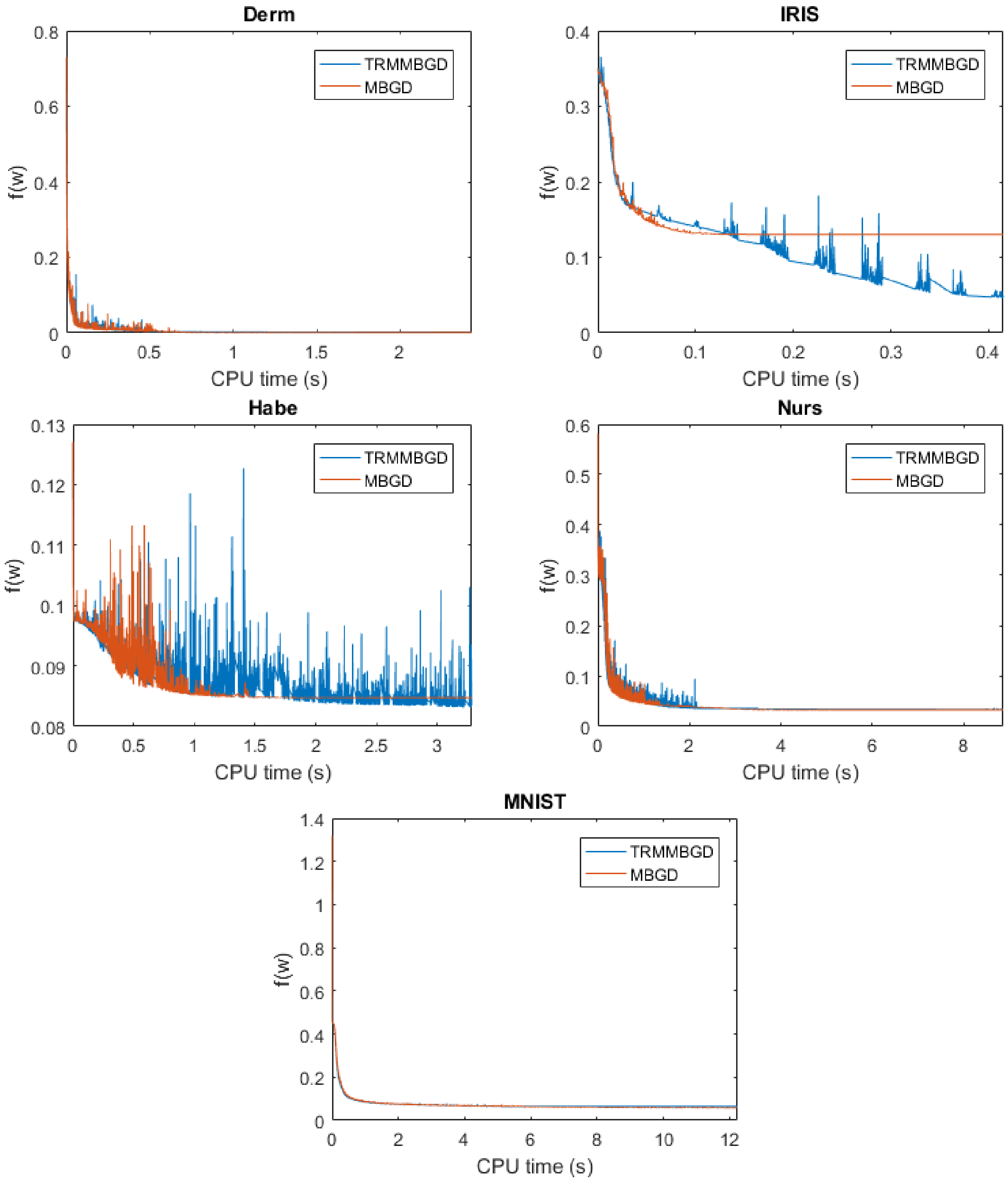

Figure 5.3: Running TRMMBGD vs MBGD for training ff-ANNs on five datasets.

Dataset	CPU time (s) **MBGD**	CPU time (s) **TRMMBGD**
Derm	**0.18436**	0.23661
Habe	-	**1.8206**
IRIS	-	**0.37217**
MNIST	2.096	**1.925**
Nurs	1.9246	**1.4415**

Table 5.4: TRRS results for $\Delta = 1\%$

Dataset	CPU time (s) **MBGD**	CPU time (s) **TRMMBGD**
Derm	**0.040571**	0.044899
Habe	**0.35471**	0.45047
IRIS	-	**0.28553**
MNIST	0.38535	**0.35748**
Nurs	**0.40615**	0.51935

Table 5.5: TRRS results for $\Delta = 5\%$

Dataset	CPU time (s) **MBGD**	CPU time (s) **TRMMBGD**
Derm	0.026375	**0.025359**
Habe	0.16741	**0.15084**
IRIS	-	**0.23544**
MNIST	0.22061	**0.19866**
Nurs	0.23353	**0.21426**

Table 5.6: TRRS results for $\Delta = 10\%$

5.5 Stochastic TRMs

We devote this section to evaluating whether or not stochastic subsampling improves the performance of **TRM** in terms of the TRRS measure, from §5.2.1, for either of the trust region size update schemes discussed in Chapter 4. We also present figures of the methods' progress over time, as we have done in the previous section, to provide insight into their behaviour.

5.5.1 Adaptive Trust Region Reduction Scheme

In this section we study the effectiveness of applying stochastic sampling methods, used with the adaptive trust region reduction scheme (ATRRS) in Algorithm 9, to **TRM**. We want to measure TRRS for **TRM** and its variatons based on stochastic sampling, **MBTRM** and **STRM**, in order to determine if stochastic sampling variations are an effective way to speed up the **TRM** method on ff-ANN when paired with the adaptive trust region reduction scheme.

We set up an experiment to determine whether or not stochastic subsampling of training examples is an effective method for speeding up **TRM**. The experiment involves running **TRM**, **STRM** and **MBTRM** on the five datasets listed in Table 5.1 and recording CPU time and the true objective function, $f(\mathbf{w}_k)$, at each iteration k. The results of this experiment are displayed in Figure 5.4. Results in Figure 5.4 are cutoff on the x-axis at the minimum total CPU time between all three methods, based on the protocol for measuring TRRS. These graphs show significant differences in the objective function values at any time between **MBTRM** and **STRM** that are not seen between **SGD** and **MBGD**. Specifically, **MBTRM** seems to reduce the objective function in all cases, despite sometimes not doing as well as **TRM**. On the other hand, in the case of the Derm dataset, **STRM** does not even decrease the objective function from the initial value.

We use TRRS to quantitatively evaluate **MBTRM** and **STRM** as potential faster variations of **TRM**. We measure the time required to obtain a reasonable solution (TRRS) where $\Delta = 1\%$, 5%, 10%. We compare both **MBTRM** and **STRM** against **TRM**, and the results are displayed in Tables 5.7, 5.8, and 5.9 respectively. Recall that TRRS is a measure between two methods only, therefore we compare **MBTRM** to **TRM** and **STRM** to **TRM** separately. In each table (5.7, 5.8, 5.9) columns 2 and 3 contain the results from measuring TRRS for **TRM** and **MBTRM** and columns 4 and 5 show the results from measuring TRRS for **TRM** and **STRM**.

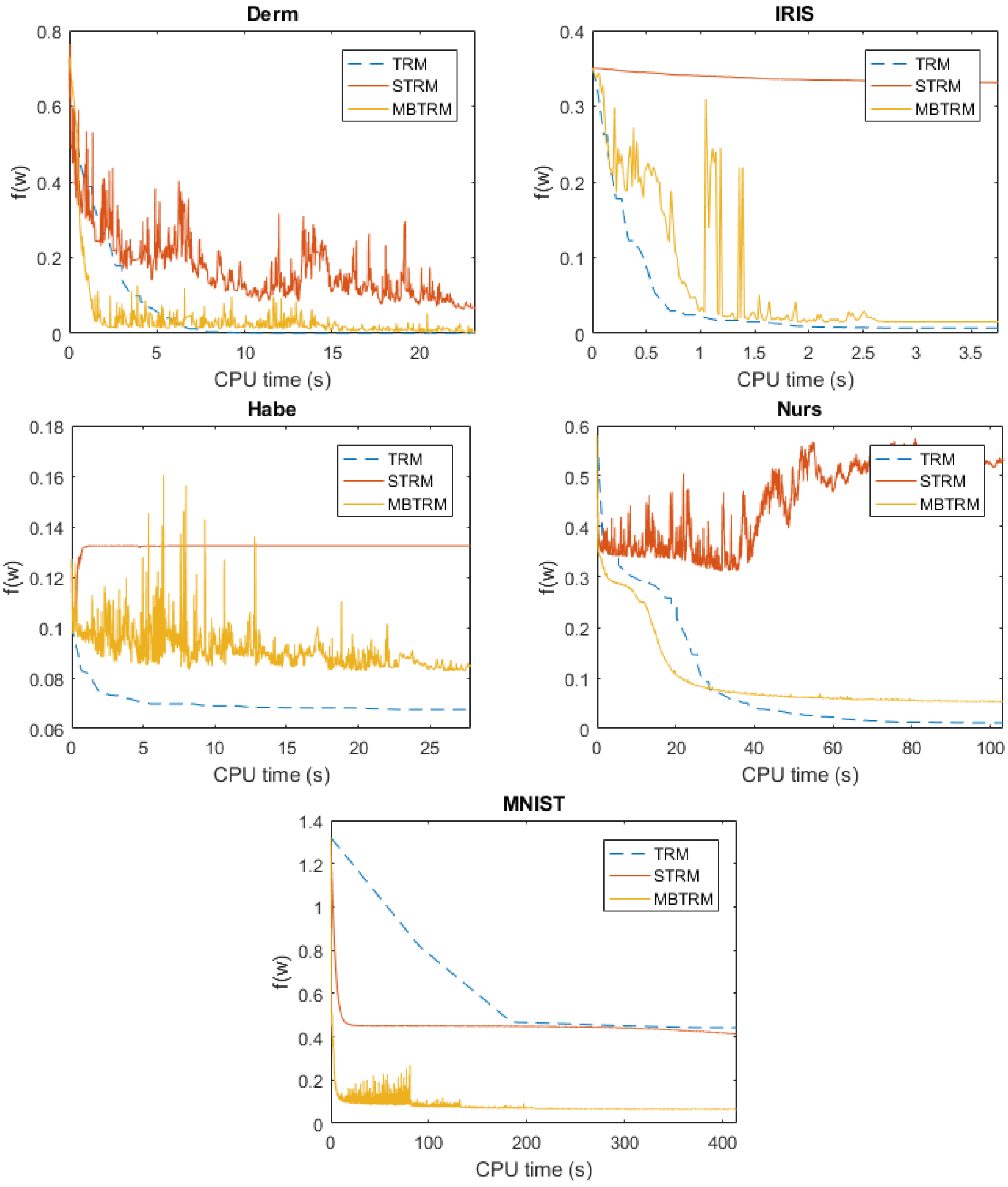

Figure 5.4: Comparing **TRM** with stochastic training example subsampling **TRM** methods (**STRM** and **MBTRM**), based on objective function, $f(\mathbf{w})$, vs CPU time.

We find that **TRM** has a lower TRRS than **STRM** for all training runs, except for in the case of MNIST where **STRM** gets to $\Delta = 10\%$ faster than **TRM**. Once we decrease Δ to 5% and 1%, **TRM** outperforms **STRM**. This indicates that generally, **STRM** as a variation of **TRM**, is not an effective speed up approach compared to the performance of the original **TRM** method.

Results were found to be more mixed when we compare TRRS between **MBTRM** and **TRM**. We find that for $\Delta = 5\%$ and 10% **MBTRM** actually achieves a faster TRRS than **TRM** on MNIST and Derm datasets. This is somewhat unexpected since Nurs has a larger number of training samples than Derm and therefore would appear to be a better candidate for a mini-batch method.

Recall from §4.4 that the algorithms **MBTRM** and **STRM** both use the Hessian Free approach where the 'Pearlmutter Trick' is called with $O(nb_m)$ time complexity each time we compute the product $\mathbf{H}_k\mathbf{v}$ for any $\mathbf{v} \in \mathbb{R}^n$ where $\mathbf{H}_k$ is the Hessian of the objective function at iteration k. This is in contrast to the method in **TRM** where we compute the Hessian matrix fully before solving for $\mathbf{p}_1$ and $\mathbf{p}_0$.

Finally, we note that **MBTRM** has a lower TRRS when used to train using the MNIST dataset, even for the case when $\Delta = 1\%$. In conclusion, for some datasets we find that **MBTRM** can improve upon **TRM** however it is not a consistent trend across a diverse range of datasets.

Dataset	**TRM**	**MBTRM**	**TRM**	**STRM**	**TRM**	**BTRM**
Derm	**7.3726**	14.8783	**7.3726**	-	7.3726	**2.827**
Habe	**15.8215**	-	**11.8777**	-	**21.0525**	-
IRIS	**1.8912**	-	**1.8912**	-	**1.8912**	-
MNIST	-	**83.7168**	**1574.484**	-	-	**1799.908**
Nurs	**67.7825**	-	**67.7825**	-	**67.7825**	-

Table 5.7: TRRS results for $\Delta = 1\%$

5.5.2 Traditional TRM γ Update Scheme

We train the same networks using **BTRM** and **TRM** in order to test whether using stochastic sampling of training examples on the full **TRM** method, including the traditional **TRM** update scheme, is an effective method for improving **TRM** performance in terms of TRRS. The batch size is chosen using the hyperparameter value search scheme

Dataset	**TRM**	**MBTRM**	**TRM**	**STRM**	**TRM**	**BTRM**
Derm	5.8078	**1.455**	**5.8078**	-	5.8078	**1.383**
Habe	**2.652**	-	**2.3852**	-	**3.477**	-
IRIS	**0.84902**	1.1691	**1.0384**	-	**1.0384**	1.8596
MNIST	-	**6.4677**	**1389.245**	-	-	**146.8234**
Nurs	**41.1667**	-	**41.1667**	-	41.1667	**32.9154**

Table 5.8: TRRS results for $\Delta = 5\%$

Dataset	**TRM**	**MBTRM**	**TRM**	**STRM**	**TRM**	**BTRM**
Derm	4.4035	**1.2892**	**4.4035**	20.9612	4.4035	**1.024**
Habe	**1.3666**	-	**1.3274**	-	**1.4662**	-
IRIS	**0.71805**	0.97309	**0.71805**	-	**0.71805**	1.3308
MNIST	-	**3.932**	1111.175	**605.5431**	-	**9.3117**
Nurs	**32.7128**	37.7004	**32.7128**	-	32.7128	**11.8563**

Table 5.9: TRRS results for $\Delta = 10\%$

(see Appendix B). The hyperparameter which defined the batch size is b_m^{large}. The value of b_m^{large} is greater than that of b_m^{small} for all datasets, which is why we use the label "large". We expect that a greater batch size will mean that the approximation of the trust region subproblem more closely resembles the actual subproblem. Therefore, we expect that the use of the traditional trust region size update in Algorithm 3 will be effective at predicting what size of region can be accurately modeled using the previous step accuracy.

The optimization curves produced by running this experiment on each dataset, are presented in Figure 5.5. Visually, we can see that the **BTRM** curves start out faster than **TRM** for training most datasets besides Iris, where the speed is similar, and Habe, where **BTRM** is found to be essentially ineffective at minimizing the objective function. This result for Habe is expected for any stochastic method since subsets are not sufficiently representative of the full set as we discovered in Figure 5.1. The low signal to noise ratio means that it is possible that the noise of the stochastic approximation has more effect on the step choice than the underlying signal we wish to learn.

We can see the quantitative TRRS measurements in Tables 5.7, 5.8 and 5.9 in columns 6 and 7. In the results for the MNIST and Derm datasets we see that for all values of Δ, **BTRM** outperforms **TRM** in terms of TRRS. On the Nurs dataset we see that for $\Delta = 5\%$ and 10% **BTRM** reaches a reasonable solution faster than **TRM** but does not reach within 1% of the final solution for the **TRM**. This means that **BTRM** successfully speeds up **TRM** for training Nurs, but it is at the expense of the quality of final solution. For IRIS the results are slightly worse when trained with **BTRM** compared to **TRM**, but reasonably similar. Training using the Habe dataset resulted in no cases where **BTRM** outperforms **TRM** which we have discussed during the discussion of Figure 5.5.

We conclude that **BTRM** can decrease the TRRS compared to **TRM** for training ff-ANNs on datasets where the signal to noise ratio is high. We hypothesize that stochastic methods perform poorly since the poor signal to noise ratio means that the direction of each stochastic step may be more influenced by the noise of the small subset of samples rather than the underlying trend. To understand this more clearly, imagine performing a linear regression problem on a dataset with a very small signal to noise ratio. With 1000 samples, the signal may be picked up but with only 2 the predicted linear relationship will be heavily influenced by noise effecting the two points.

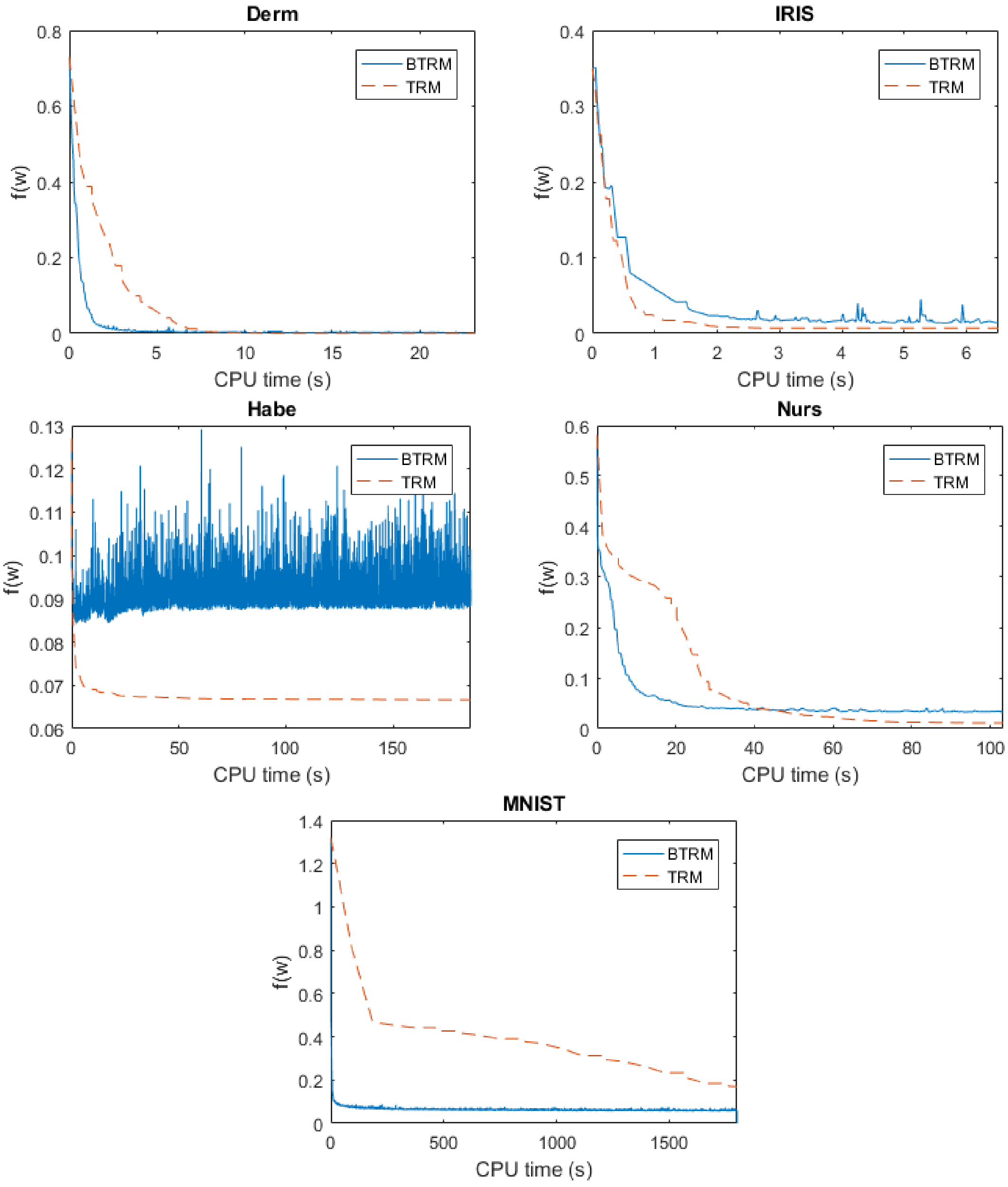

Figure 5.5: Comparing **TRM** with stochastic training example subsampling **TRM** method, **BTRM**, based on objective function over CPU time.

5.6 Reducing the Dimentionality of TRS

We experiment with weight subsampling added to the **TRM** algorithm to train ff-ANNs using the five datasets with both **TRMWS** and **TRM** and comparing the results. The results of these experiments are shown in Figure 5.6. Specifically, we plot the objective function vs the CPU time as we have done for stochastic methods in the previous sections. In terms of the qualitative behaviour we can see that for datasets Derm, IRIS and MNIST, this method is beneficial. For Habe, weight subsampling actually slows down the overall progress initially but it eventually surpases **TRM** in objective function over CPU time.

In order to compare these methods quantitatively, we display the TRRS in Tables 5.10, 5.11 and 5.12 for the TRRS parameter $\Delta = 1\%$, 5% and 10% respectively. We observe improvement in terms of TRRS for training ff-ANNs on Derm, IRIS and MNIST for all values of Δ when adding weight subsampling to **TRM**. For Nurs we see that **TRM** has a shorter TRRS for all values of Δ displayed, however we note in Figure 5.6 that Nurs is faster at getting a very approximate solution and is passed before the $\Delta = 10\%$ mark. Finally, we see that **TRMWS** outperforms **TRM** on the Habe dataset for only the lowest value of Δ. These are mixed results, but we observe that **TRMWS** is faster at the beginning of training for 80% of our datasets, and has a better TRRS for $\Delta = 1\%$ on 80% datasets as well. Therefore we can conclude that the variation, **TRMWS**, improves **TRM** in many cases.

Dataset	**TRM**	**TRMWS**	**BTRM**	**BTRMWS**
Derm	7.3726	**4.2109**	**2.827**	-
Habe	-	**152.2109**	-	**19.1976**
IRIS	-	**2.9561**	**32.9446**	-
MNIST	-	**1218.143**	**1799.908**	-
Nurs	**67.7825**	-	251.4025	**226.7159**

Table 5.10: TRRS results for $\Delta = 1\%$

We have evaluated the effect of adding weight subsampling to the basic **TRM** method, now we wish to see if weight subsampling improves other methods such as methods that already include stochastic subsampling of training samples, SSTS. Firstly, we test the effect of weight subsampling on **BTRM** by comparing the behaviour and TRRS of the method with it's weight subset variation, **BTRMWS**. The results of these experiments are shown in Figure 5.7 and the TRRS values are shown in columns 4 and 5 of Tables 5.10, 5.11 and 5.12. Qualitatively and quantitatively we observe a much smaller effect of weight

Dataset	TRM	TRMWS	BTRM	BTRMWS
Derm	5.8078	**1.7998**	**1.383**	2.1279
Habe	**4.4892**	53.3771	-	**0.64488**
IRIS	1.1462	**0.77798**	2.4622	**1.6066**
MNIST	-	**564.9604**	**146.8234**	-
Nurs	**41.1667**	-	23.1261	**20.7021**

Table 5.11: TRRS results for $\Delta = 5\%$

Dataset	TRM	TRMWS	BTRM	BTRMWS
Derm	4.4035	**1.1238**	3.6755	**1.1415**
Habe	**1.6481**	5.6479	1.1263	**0.41935**
IRIS	0.71805	**0.60017**	**0.70417**	0.85703
MNIST	-	**176.2933**	57.9639	**13.7871**
Nurs	**32.7128**	42.1636	10.9396	**4.7036**

Table 5.12: TRRS results for $\Delta = 10\%$

subsampling when it is applied to **BTRM** vs **TRM**, though in all cases except for IRIS, the TRRS for $\Delta = 10\%$ is lower for **BTRMWS** than for **BTRM**. These findings suggest that weight subsampling has a smaller and more consistent effect, across different datasets, when applied to **BTRM** than when applied to **TRM**.

Next, we test weight subsampling applied to stochastic trust region methods, **MBTRM** and **STRM**, to see if we get the same results as for **BTRM**. These results can be seen in Figure 5.8. We find that applying weight subsampling to these methods does not follow the consistent and slight improvement seen when applying weight subsampling to **BTRM** to get **BTRMWS**. Rather, we find that for $\Delta = 1\%$ and 5%, both **STRM** and **MBTRM** do better than their weight subset equivalents, **STRMWS** and **MBTRMWS**. For training on Derm, Habe and IRIS we do see an improvement in TRRS for $\Delta = 10\%$ for **MBTRMWS** compared to **MBTRM**.

By looking at all of our results from applying weight subsampling together, the trend appears to be that the smaller the stochastic subset of training samples chosen at each iteration, the less effective adding weight subsampling will be to reduce TRRS. This result, though not entirely expected, is not surprising as every time a variation is made using a stochastic approach, the steps become less likely to lead to the solution of the overall problem. Once the sum of all approximated steps is no longer leading towards a minima for the full objective function, $f(\mathbf{w})$, they are no longer useful.

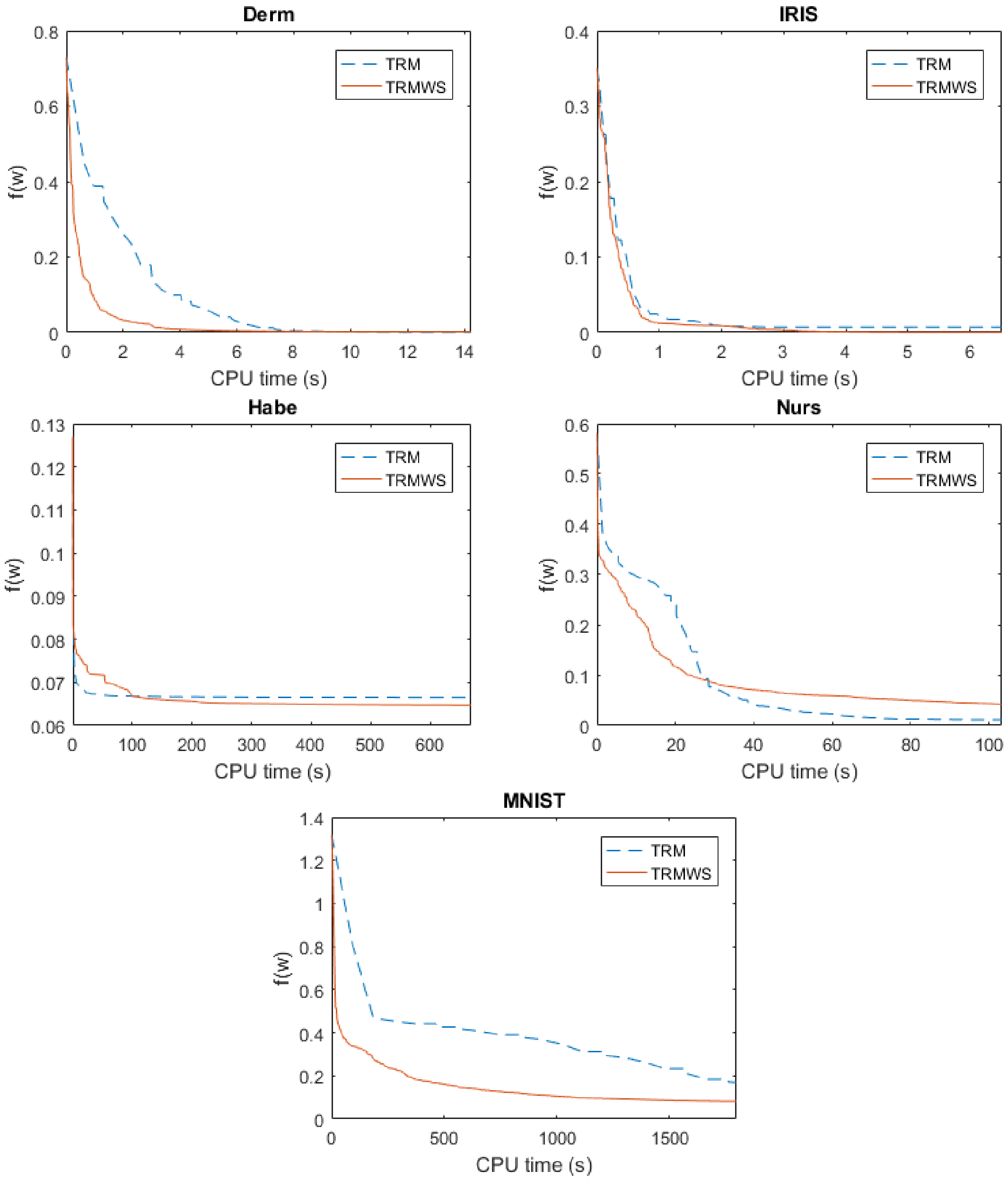

Figure 5.6: Running TRM vs TRMWS for training ff-ANNs on five datasets.

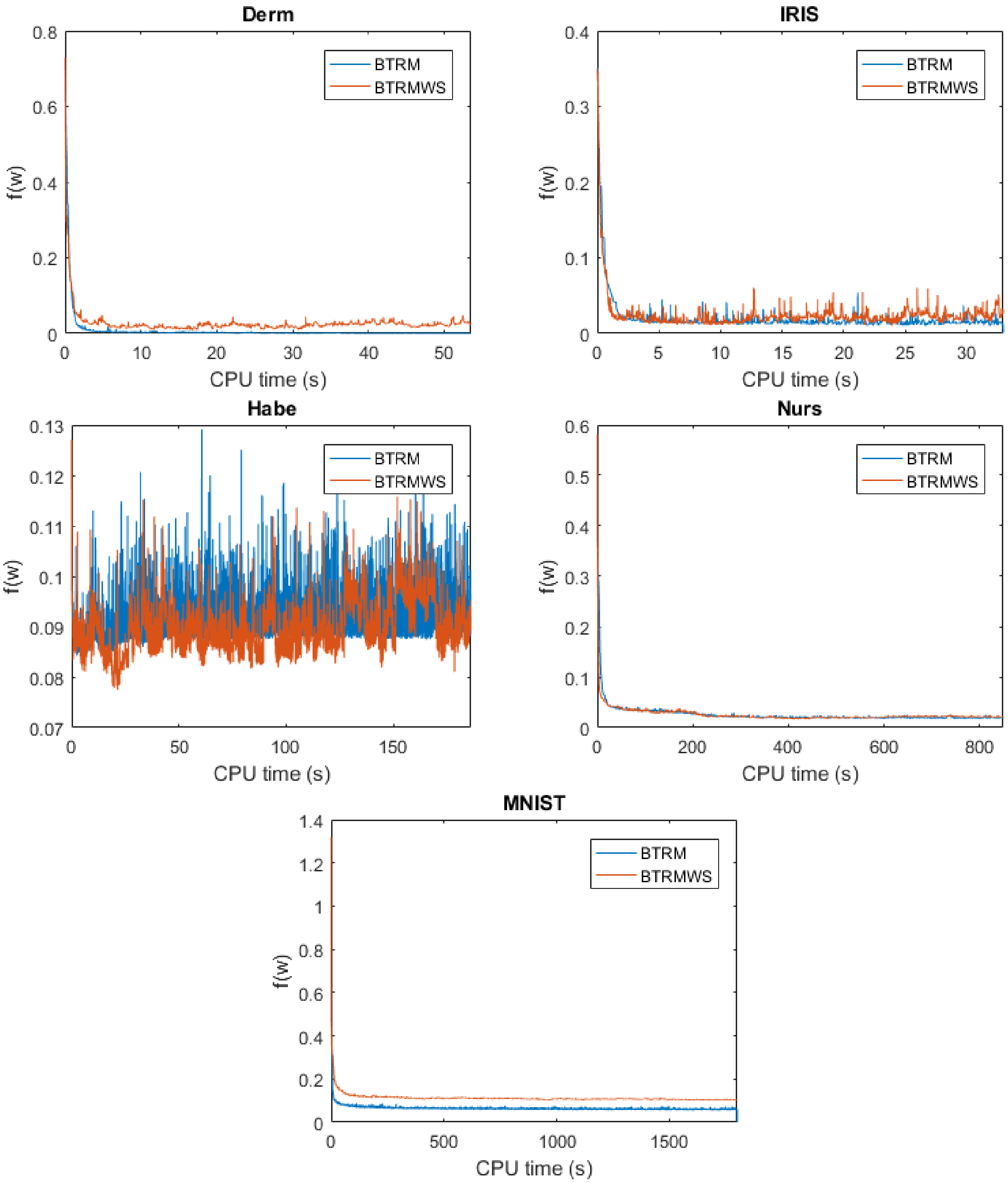

Figure 5.7: Running BTRM vs BTRMWS for training ff-ANNs on five datasets.

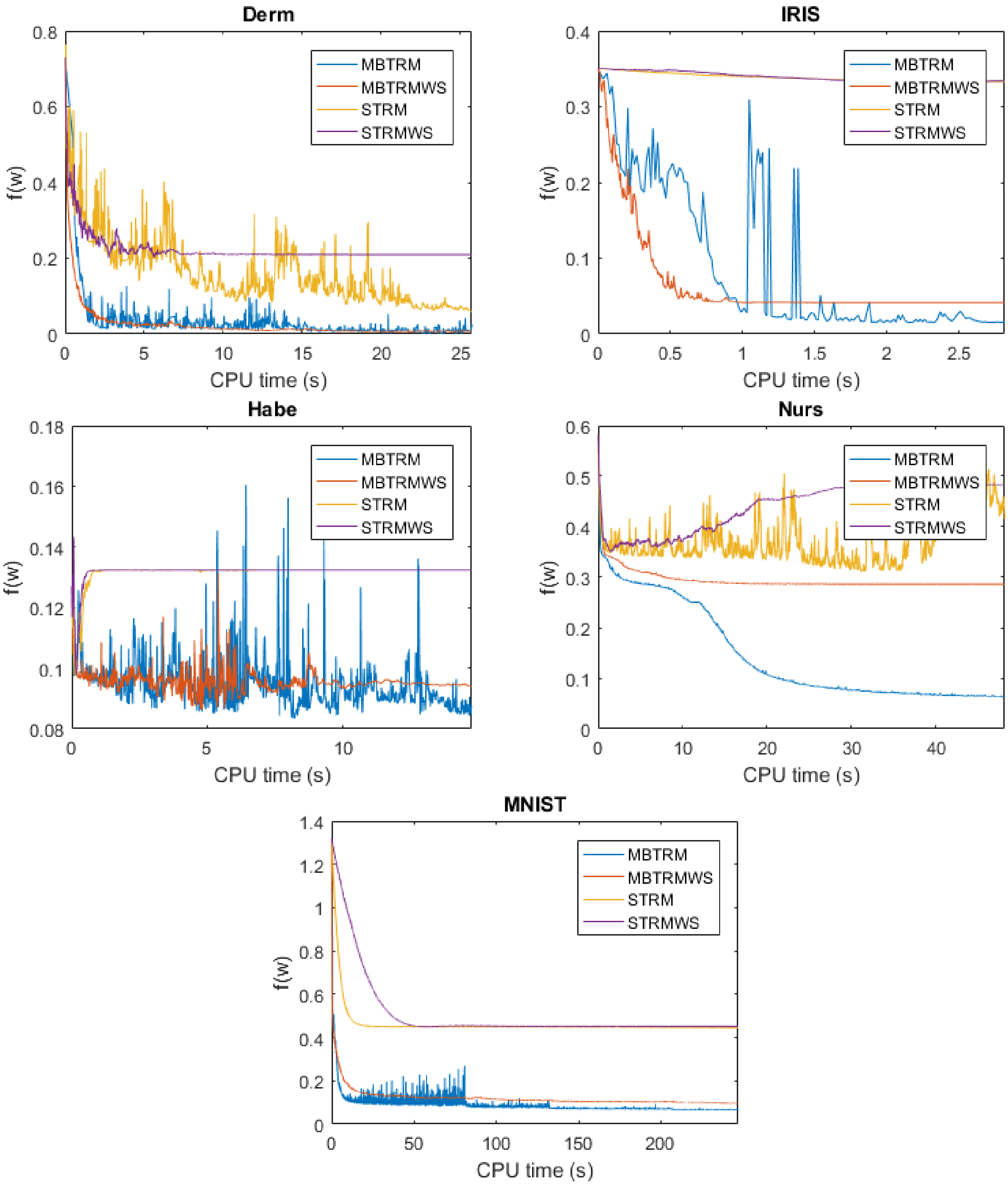

Figure 5.8: Test results for using stochastic TRM methods.

Dataset	**MBTRM**	**MBTRMWS**	**STRM**	**STRMWS**
Derm	**14.8783**	17.1257	**22.0584**	-
Habe	**8.1992**	-	**0.097114**	0.15717
IRIS	**1.4322**	-	**14.1681**	-
MNIST	**83.2774**	-	**17.2798**	46.9834
Nurs	**55.4531**	-	**27.9826**	-

Table 5.13: TRRS results for $\Delta = 1\%$

Dataset	**MBTRM**	**MBTRMWS**	**STRM**	**STRMWS**
Derm	**1.455**	2.0583	**10.3782**	-
Habe	**2.0487**	4.4119	**0.079468**	0.13661
IRIS	**0.98844**	-	**2.4714**	3.2203
MNIST	**6.4677**	31.6931	**9.7466**	35.0008
Nurs	**26.2484**	-	**9.0291**	-

Table 5.14: TRRS results for $\Delta = 5\%$

Dataset	**MBTRM**	**MBTRMWS**	**STRM**	**STRMWS**
Derm	1.2892	**1.168**	**7.9418**	-
Habe	0.96801	**0.65215**	**0.061815**	0.11599
IRIS	0.90709	**0.54737**	0	0
MNIST	**3.932**	7.9736	**7.3748**	28.1962
Nurs	**19.4037**	-	**0.50192**	0.9181

Table 5.15: TRRS results for $\Delta = 10\%$

5.7 CPU Time Analysis

We study the profile of the CPU time expenditure for each Trust region derived method applied to each dataset. We exclude **TRMMBGD** which computes mainly stochastic mini-batch gradient steps and therefore cannot be compared directly. In order to make the comparison between remaining methods, we timed each component of an algorithm to determine if there are any outliers. It turns out that computing $\mathbf{H}_k$ for each iteration k, and $\mathbf{p}_1$ take the most time for all methods. This can be seen in Figures 5.9 and 5.10. For methods that require the computation of both $\mathbf{p}_1$ and $\mathbf{p}_0$, $\mathbf{p}_0$ always takes less than 12% of the CPU time for an average step computation. If we sum together the time taken to compute $\mathbf{H}_k$, $\mathbf{p}_1$ and $\mathbf{p}_0$ (if $\mathbf{p}_0$ is applicable for that method) we find that these components contribute over 98% of the CPU time. This gives us an explanation for why the second order methods, despite progressing more at each step, still produce longer TRRS values when compared to the gradient methods. It does show how Newton's method, equivalent to the instances where $\mathbf{p} = \mathbf{p}_0$ for **TRM**, is effective at computing a solution efficiently compared to computing the boundary step since the $\mathbf{p}_0$ computation takes much less time than the $\mathbf{p}_1$ computation as we can gather by comparing Figures 5.9 and 5.11.

The $\mathbf{p}_1$ and $\mathbf{p}_0$ steps both involve iterative methods to determine their solution, namely "eigs" in MATLAB and conjugate gradient, respectively. The algorithm is stopped when the relative residual is less than 10^{-3} for both "eigs" and the conjugate gradient computation. Since convergence times depend on properties of the optimization problem that are out of our control, we decided to reduce these times by setting a very small limit on the number of iterations for each of the methods and accepting steps where the methods did not converge. We are changing the stopping criteria reaching a residual of 10^{-3} to reaching a max iteration number we refer to as **submaxiter**. In other words, the tests so far have been performed where convergence is achieved in order to solve the TRS at each step of the original problem. For the following methods, rather than waiting for convergence, we use the result achieved after **submaxiter** iterations regardless of it's accuracy to see how this performs. We compare these results to those where the method must converge to see if it speeds up the method. For most datasets the resulting behaviour is not notably different, this is likely because the datasets have a low number of parameters, n, and therefore may not need as many iterations to converge. On the other hand, we found a big difference in performance on the MNIST dataset, which is trained on the ff-ANN with the most parameters (1120), more than double the number of the next largest network. These results are displayed in Figure 5.12. When TRM with $\mathbf{p}_1$ and $\mathbf{p}_0$ convergence is compared with TRM run with a very low max iteration number for these two computations, and no convergence requirement, we see that TRM with convergence is more productive per epoch but much

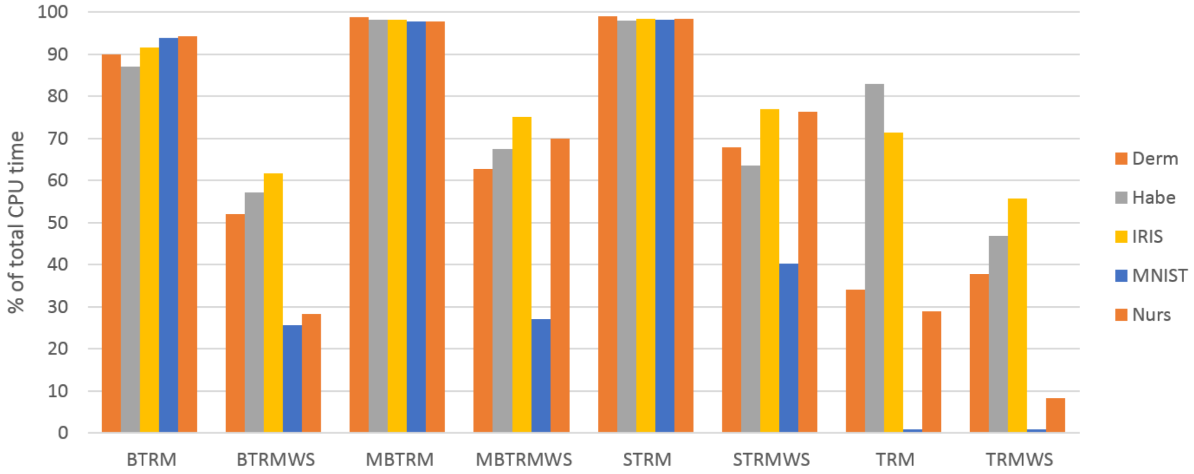

Figure 5.9: Percentage of time taken up by solving the Generalized Eigenvalue Problem (2.35) and computing $\mathbf{p}_1$ from the result at each step.

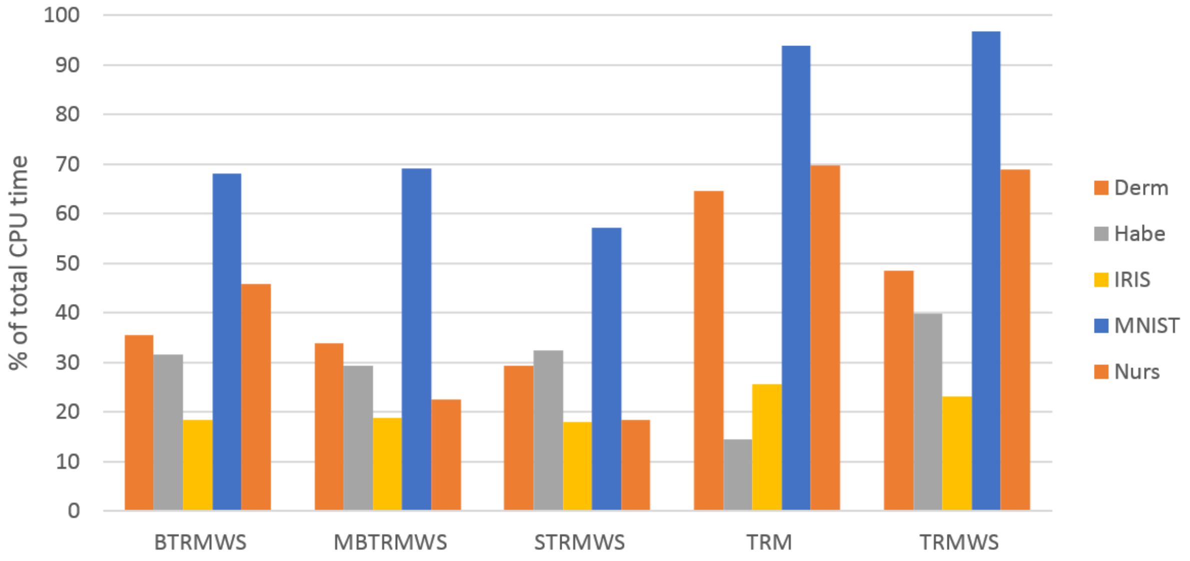

Figure 5.10: Percentage of time taken for computing the Hessian matrix at each change in weight values $\mathbf{w}$. Recall that only five of our methods include the computation of the Hessian matrix.

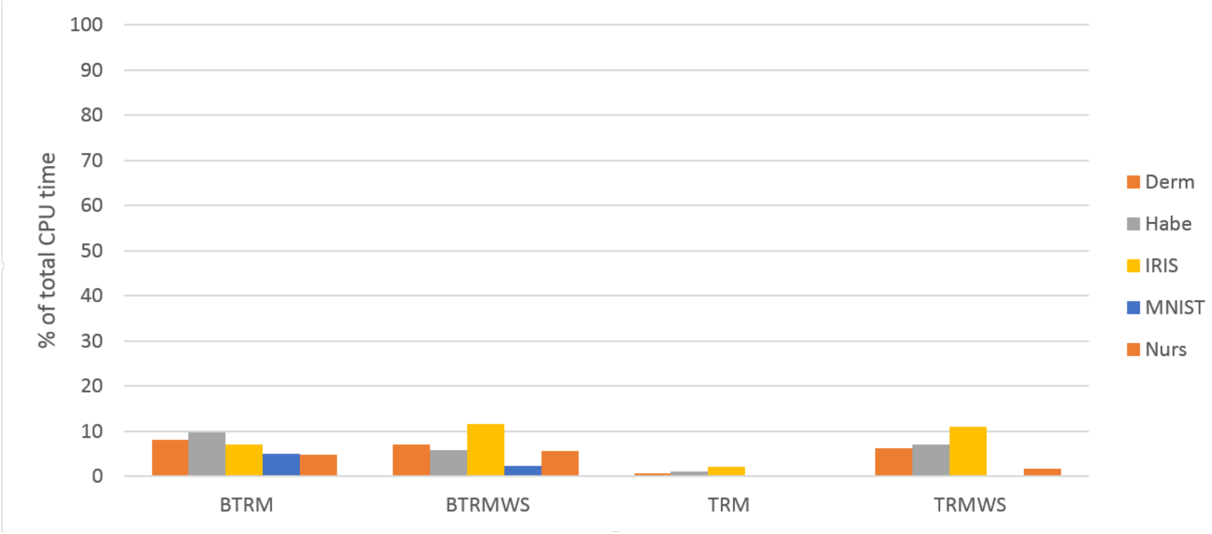

Figure 5.11: Percentage of time taken to solve for the $\mathbf{p}_0$, which is solving the linear equation (2.32a). Recall that those methods which use the ATRRS do not compute $\mathbf{p}_0$ (see Algorithm 9).

slower in terms of CPU time. Early stopping of $\mathbf{p}_0$ and $\mathbf{p}_1$ computations is shown here to produce less precise results in terms of step direction, but is significantly faster on problems with a large (in our case > 1000) number of parameters. We compare with **SGD** to put the speed of the **submaxiter TRM** method in perspective. This comparison is shown in the bottom subfigure of Figure 5.12. We observe that despite seeing a speed up with **TRM**, the speed of objective function reduction still is much slower than that of **SGD**.

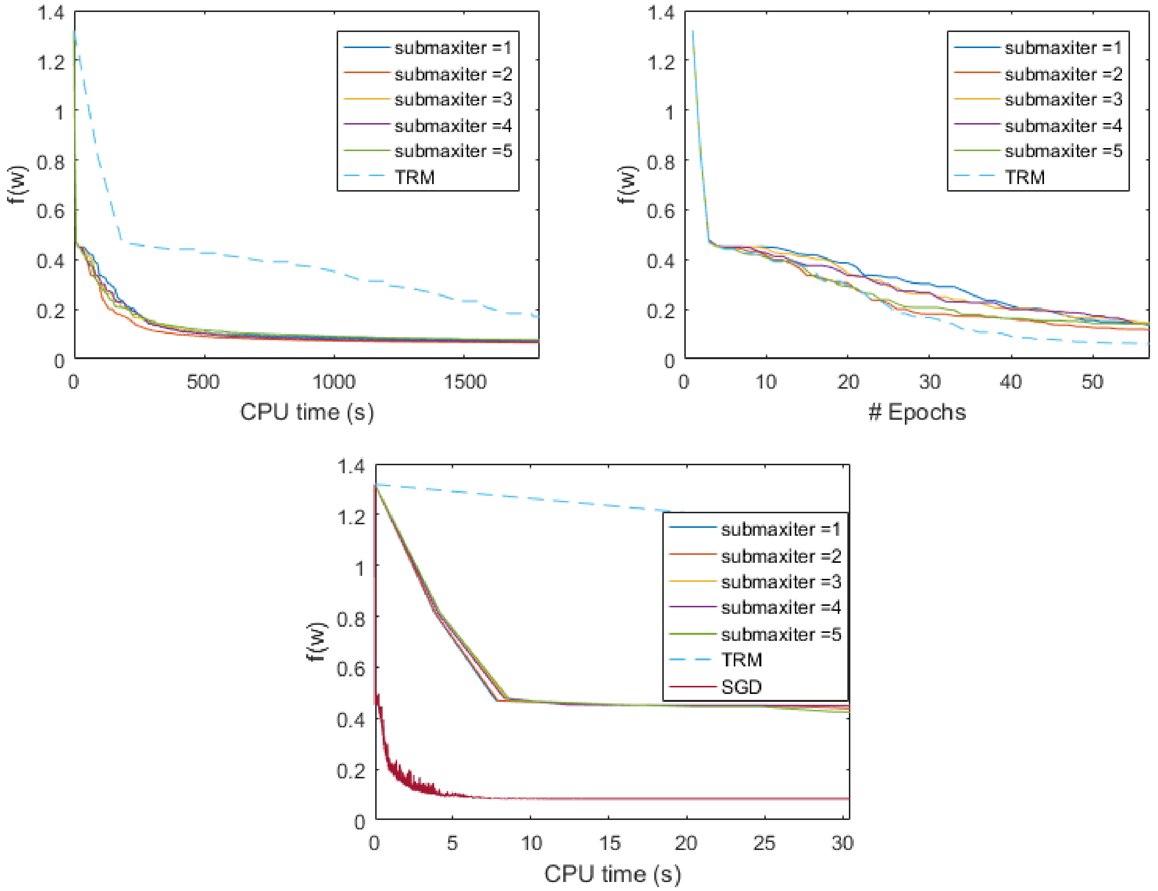

Figure 5.12: Convergence of **TRM** with $\mathbf{p}_0$ and $\mathbf{p}_1$ using stopping criterion of residual magnitude less than 10^{-3} (dashed line) and stopping criterion of reaching the max iteration (**submaxiter**=1, 2, 3, 4, 5). (Upper Left) Convergence in terms of CPU time. (Upper Right) Convergence in terms of Epoch. (Bottom) Convergence in terms of CPU time with **SGD** for reference.

5.8 Robustness Test

It is important to have some measure of robustness for an algorithm which is non-deterministic since outcomes may differ between experiment runs. We performed a robustness test on our algorithms by running them many times on the same small network and checking the accuracy of the solution. In order to perform this experiment we used a small network to compute the XOR output for two inputs on a two layer. This network can be trained quickly and also has a known saddle point, making it well suited to measure robustness.

5.8.1 The XOR Problem

The XOR problem that we use is specifically the two feature case of the XOR function. The total number of distinct values that a training sample can take is four, all of which are displayed in Table 5.16. In this test problem we use four training samples, one of each distinct value.

$\mathbf{x}_1$	$\mathbf{x}_2$	y
0	0	0
0	1	1
1	0	1
1	1	0

Table 5.16: The 2-feature XOR problem.

This is a very simple problem that contains classes which are not linearly separable. When trained on a two-layer network with two hidden nodes, the problem has a known saddle point, two local minima and a global mininum which are displayed in the Table 5.17.

$f(\mathbf{w})$	**Point Type**
0.125	saddle point
0.0625	local minimum
0.0833	local minimum
0	global minimum

Table 5.17: First order critical point classification for the XOR problem when trained on a 2-layer ff-ANN with two hidden nodes.

Because of these known values, small size for fast training, and the known results of training a network on the XOR problem using various methods this is a well suited problem to test our methods for robustness by running each method many times and grouping the results.

5.8.2 XOR Training Results

In order to see quantitatively how well all of these methods perform, we train a two layer ff-ANN to learn the XOR solution, and run it 1000 times on 100 initial weight settings in order to test robustness. We analyze our results by categorizing the final value of the objective function for each trial for each of our methods. We say a method has reached one of the four known points, $\mathbf{x}$, if the final objective function value achieved by the method is within 10^{-3} of $f(\mathbf{x})$. The value of these known points are displayed in Table 5.17. The results of this experiment are displayed in Figure 5.13.

The hyperparameter procedure resulted in values for training example sampling: $b_m^{small} = b_m^{large} = m$, therefore **TRM** is equivalent to **BTRM** for this experiment so we do not show **BTRM** results. The mini-batch trust region methods, **MBTRM** and **MBTRMWS**, are still distinct from **TRM** and **TRMWS** respectively because of difference in γ update method.

The results of the experiment are shown in Figure 5.13. The first, most striking result, is the stochastic gradient descent, **SGD**, converges to the saddle point 100% of the time. Its important to point out that this result could be dependent on the initial learning rate and learning rate schedule. It is still an interesting result and shows that with certain parameter choices, **SGD** may have difficulty converging to a minimizer. The promising takeaway for our methods is that **TRM**, **MBTRM**, **MBTRMWS**, and **TRMWS** all converge to a local minima for all 1000 trials. This shows that in this context, weight subsampling does not seem to affect the global convergence property of **TRM**, that is the method still converges to minima in all cases.

Another interesting result from this experiment is the improvement in performance between **MBGD** and **TRMMBGD**. We see that **TRMMBGD** increases the chances of at least resulting in a saddle point solution, compared to reaching no critical point for **MBGD**, and in some cases reaching the global minimum, compared to **MBGD** which most of the time does not reach a saddle point and never reaches a local or global minimum. When we look at all the results together that the percentage of times that a local minimum is reached can be predicted by the number of steps calculated using all 4 training examples of the XOR problem. That is, all methods that used $b_m = m$ at each step reached a local minimum 100% of the time. **TRMMBGD**, which uses $b_m = 2$ most of the time and $b_m = m$ occasionally, reaches a local minimum occasionally. Finally those methods that never consider all training samples at the same step, $b_m < m$, never reach a local minimum. This is to be expected that stochastic subsampling of training samples is ineffective for the XOR problem since the classes for any set of three or fewer samples is linearly separable, therefore all four samples are required to represent the more complex XOR problem.

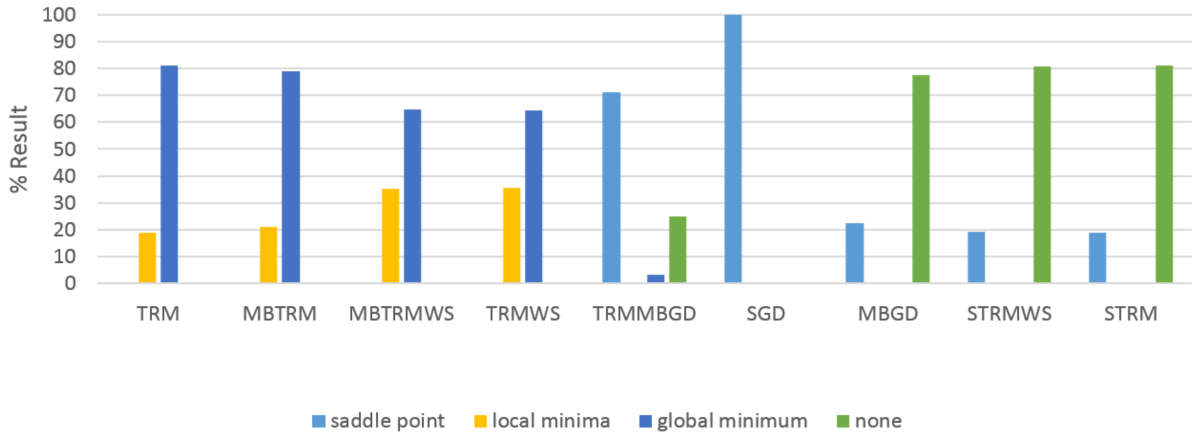

Figure 5.13: Final points are recorded when the final objective function value is within a tolerance of 10^{-3} from one of the known points.

Finally, we compare these results with those from other methods used to train the same two layer ff-ANN from [30]. This comparison is shown in Figure 5.14. Methods with the suffix $-tr$ are methods that use the trust region model approach, those with $-B$ uses line minimization and those ending in $-qn$ are quasi-Newton methods. The full list of methods and their definitions are available on p. 134 of [30].

In Figure 5.14 we see that **TRM** manages to outperform all methods studied in [30], with the exception of **BD-tr** which is the trust region model method with the bold driver method for adjusting the step size. The bold driver method simply increases the trust region radius when the previous step reduces the objective function, and decreases it otherwise. This is different from **TRM** which updates the trust region size based on the reliability of the second order model used within the bounds. This method therefore does not determine "trust" of the second order model based on model accuracy but rather on reduction of the objective function, $f(\mathbf{w})$, a measure which does not take the model into consideration. In this thesis we decided to focus on methods derived from the more classical measure ρ, from (2.42), which is based on model accuracy.

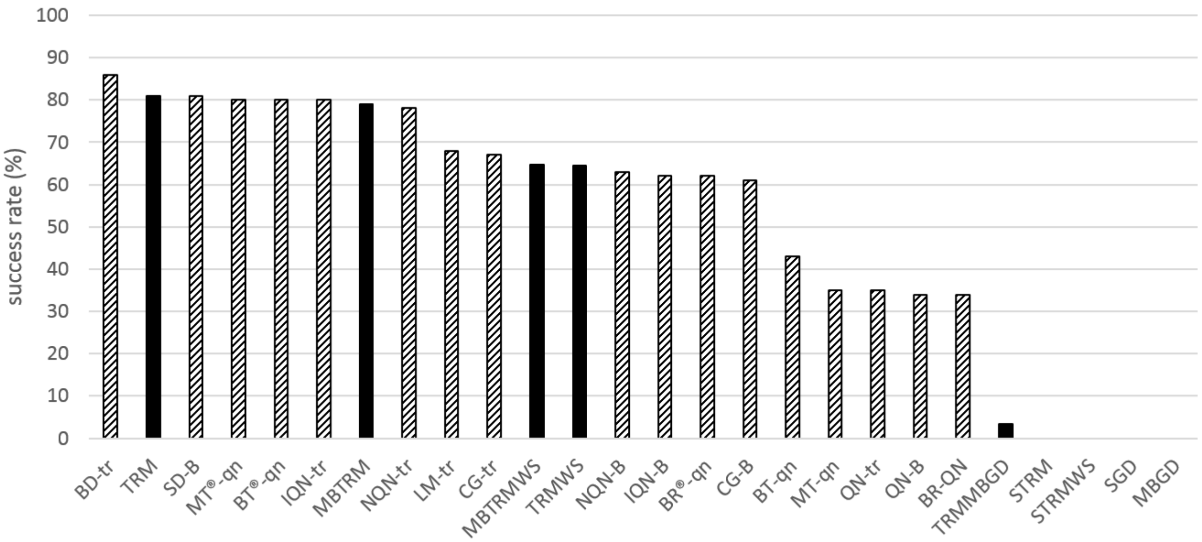

Figure 5.14: Percentage of runs which successfully converge to the global minimum of the XOR problem. The methods from our exploration are displayed as solid black and methods from [30] are coloured with a diagonal pattern.

All of the methods from this thesis that use $b_m = m$ perform similarly to the trust region approaches tested in [30], in terms of percentage of trials that reach the global minimum on the XOR problem. This simple problem does not show a very detailed comparison for training ff-ANNs in general but provides some context to show how these methods compare to a wide array of methods that have been studied in the context of training ff-ANNs.

Chapter 6

Conclusion

In this thesis, we studied a trust region approach, that uses the recent generalized eigenvalue method to solve the trust region subproblem (TRS), on training feedforward neural networks (ff-ANNs) by using the 'Pearlmutter Trick' to compute the second order information. From this full method, we developed several variations of this learning algorithm that either estimate the TRS at an iteration, reduce the dimension of the TRS, or combine the trust region method with a stochastic gradient method. We used each of these methods to train networks on five datasets of different sizes and application domains. We analyzed these results to provide some insight as to the behaviour of these methods as well as a measure of CPU time taken to reach a reasonable solution, which provides an indicator of speed.

Our results suggest that combining the trust region method presented in this thesis, along with mini-batch gradient descent, we can match or improve upon the speed of mini-batch gradient descent for all datasets tested. In terms of stochastic approximation approaches, we find that second order approximations using stochastic sampling were not beneficial for all datasets. However, results for using a large batch trust region method with the trust region size update showed that for most datasets a speed up in training results from large batch size sampling. We also find that with the largest dataset, MNIST, which requires the most parameters in a two-layer network, smaller batch stochastic sampling can be effective at speeding up the method. These results show promise for use on larger datasets.

We find that weight subsampling at each iteration, which reduces the dimension of the TRS, can speed up the trust region method in some cases. It was particularly effective for the largest dataset tested, MNIST, on the largest network. This benefit however, was not seen, in a significant way, when used in combination with a stochastic sampling methods. Therefore, our results suggest that weight sampling can benefit trust region methods for problems with a larger number of test samples, m, and a larger number of parameters n. It should be considered, however, as an alternative to stochastic subsampling rather than a complementary method since we do not see the same benefits when used in tandem with a stochastic sampling method.

The most useful result found in this investigation was the improvement or matching of solution and the time required for a reasonable solution (TRRS) for trust region used in tandem with mini-batch gradient descent as compared to mini-batch gradient descent on its own. This is a very promising result and indicates that other tandem methods may also be successful.

Despite some reduction of time required for a reasonable solution using stochastic subsampling and weight subsampling, these time reductions are still not large enough to improve upon the TRRS of a stochastic gradient descent method. Therefore, the benefit of a trust region method, without used in combination with a first order method, still lies purely in its ability to produce precise results and not in its speed.

Overall, we have shed some light into the behaviour and performance changes of a specific approach to approximation of the trust region method used to train feedforward neural networks. We have also identified one method, the hybrid of the trust region method with mini-batch gradient descent, which does provide some improvements in speed and precision to mini-batch gradient descent.

6.1 Future Work

The most promising method that was tested in our numerical investigation was a hybrid method combining the complete trust region method with mini-batch gradient descent as was mentioned previously. We leave it as future work to determine what other methods can be improved for training neural networks when used in combination with the trust region method, or even a locally convergent method such as Newton's method.

Another avenue of future work is to expand this survey to deeper networks and larger scale datasets. Since the trust region method has scaling difficulties, it is likely that the benefits of using approximate information become more obvious with larger datasets and a higher number of parameters in the network. This is something we see already with training on the MNIST dataset compared to the smaller datasets tested.

The final interesting area that is left open for future exploration, is how to choose hyperparameters more effectively. This has been considered for training neural networks using more common approaches, but could be investigated further specifically for variations on trust region methods.

APPENDICES

Appendix A

Pearlmutter Trick for Computing Second Order Information

In this section we discuss the use of the “Pearlmutter Trick” which can be used to exactly compute the matrix vector product of the Hessian, $\mathbf{H} \in \mathbb{R}^{n\times n}$, of an ff-ANN and any vector $\mathbf{v} \in \mathbb{R}^n$ [27]. In order to quickly compute the product $\mathbf{Hv}$, Pearlmutter developed a method based upon the “R” operator which is a differential operator [27] defined as:

$$R_{\mathbf{v}}\{f(\mathbf{w})\} = \frac{\partial}{\partial r} f(\mathbf{w} + r\mathbf{v}) \mid_{r=0}, \tag{A.1}$$

where $f(\mathbf{w})$ is the objective function we are looking to minimize, $\mathbf{w}$ is a vector of the parameters of the objective function (in the case of an ff-ANN these are the weights), and $\mathbf{v}$ can be interpreted as a direction vector to indicate in which direction to measure the derivative. For simplicity we will refer to $R_{\mathbf{v}}\{.\}$ simply as $R\{.\}$ Using this operator we are able to compute the $\mathbf{Hv}$ more easily since:

$$\mathbf{Hv} = \frac{\partial}{\partial r} \nabla f(\mathbf{w} + r\mathbf{v}) \mid_{r=0} = R\{\nabla f(\mathbf{w})\}. \tag{A.2}$$

Our goal is therefore to compute the result of using the R operator on the gradient of our objective function. Reaching this result requires a four pass method. The first two are the well known forward and back propagation passes for computing the gradient where we compute $f(\mathbf{w})$ using forward propagation, and $\nabla f(\mathbf{w})$ using back propagation. At this point we do a second forward/back propagation method at the end of which we are able to compute elements of $\mathbf{Hv}$. The specific computations that make up these passes are described in §3.1.1 and are referred to collectively as the “Pearlmutter Trick”.

A.0.1 Forward and Back Propagations

In this section we describe the forward and back propagation passes required to compute $R\{\nabla f(\mathbf{w})\}$ for an ff-ANN. Forward and back propagation methods are a way to reformat approaches to problems so that we can modularize layers in the ff-ANN setting and therefore implement general solutions that apply to all ff-ANNs rather than only a specific network. This is the concept for the use of forward and backpropagation to compute $\nabla f(\mathbf{w})$ and will be used as well to compute $R\{\nabla f(\mathbf{w})\}$.

In this section there are some variables such as our weight variable $\mathbf{w}$ that are represented differently in the optimization setting than they are in the ff-ANN setting for ease of computations. Here is a list of equivalent values where on the left we have the representation used in optimization settings and on the right we have the representation most useful for training ff-ANNs:

$$\begin{aligned} n &= \sum_{l=1}^{L} n_l(n_{l-1}+1), \\ \mathbf{w} &\leftrightarrow \{(\mathbf{W}_1, \mathbf{b}_1)\,(\mathbf{W}_2, \mathbf{b}_2), ..., (\mathbf{W}_L, \mathbf{b}_L)\}, \\ \mathbf{v} &\leftrightarrow \{(\mathbf{V}_1, \mathbf{b}_1^v)\,(\mathbf{V}_2, \mathbf{b}_2^v), ..., (\mathbf{V}_L, \mathbf{b}_L^v)\}, \end{aligned} \tag{A.3}$$

where n_l represents the number of nodes computed at layer l, meaning that each layer computes a vector of size n_l from a vector of size n_{l-1}. In the implementation of the TRM, $\mathbf{w}$ and $\mathbf{v}$ are concatenated vectorized matrices of their set on the right. The set on the right is the form of these weights for computations involved in the four passes of the "Pearlmutter trick". A reverse mapping is also implemented. Let index q be defined as:

$$q = \sum_{c=1}^{l-1} n_c(n_{c-1}+1) + (i-1)(n_{l-1}+1) + j, \tag{A.4}$$

where i and j are the row and column indices respectively, for the matrix at layer l. The mapping by index is defined as:

$$\mathbf{v}_q = (\mathbf{V}_l, \mathbf{b}_l^v)_{ij} = \begin{cases} (\mathbf{b}_l^v)_i & \text{if } j = (n_{l-1}+1), \qquad \text{(A.5a)} \\ (\mathbf{V}_l)_{ij} & \text{otherwise.} \qquad \text{(A.5b)} \end{cases}$$

We note that we set $j = n_{l-1} + 1$ for bias values. Since we are computing $\mathbf{Hv}$ using backpropagation, the same mapping will apply here. That is to say, we want to compute

	forward	back
start	0	L
end	L	0
incr	+1	-1

Table A.1: Values of parameters for Algorithm 11 based on propagation direction.

all elements:

$$(\mathbf{H}\mathbf{v})_q = R\{(\nabla f(\mathbf{w}))_q\} = R\Big\{\frac{\partial f(\mathbf{w})}{\partial \mathbf{w}_q}\Big\} = R\Big\{\frac{\partial f(\mathbf{w})}{\partial (\mathbf{W}_l)_{ij}}\Big\}, \tag{A.6}$$

where q corresponds to a weight value, otherwise $(\mathbf{H}\mathbf{v})_q = R\Big\{\frac{\partial f(\mathbf{w})}{\partial (\mathbf{b}^v_l)_i}\Big\}$. In this section we show that in four passes, $R\{\frac{\partial f(\mathbf{w})}{\partial (\mathbf{W}_l)_{ij}}\}$ and $R\Big\{\frac{\partial f(\mathbf{w})}{\partial (\mathbf{b}^v_l)_i}\Big\}$ can be computed exactly for all l and all possible associated values of i and j.

The four propagation steps, or passes, that make up the "Pearlmutter Trick" will be presented by what will be referred to as a respective **propagation equation** for each pass. How the **propagation equation** of an arbitrary pass is used to propagate throughout the layers of a network is shown in Algorithm 11. The values of some of the parameters used in Algorithm 11 are dependent on whether the pass involves forward or backward propagation. Their values based on direction of propagation are presented in Table A.1.

Algorithm 11 Pseudocode of a Single Pass

```
procedure PROPAGATE EXAMPLE $k$
    initialize $\mathbf{a}^k_{start}$
    $l \leftarrow start + incr$
    while $l \neq end$ do
        $\mathbf{a}^k_l \leftarrow$ [propagation equation: unique for each pass]$(\mathbf{a}^k_{l-incr})$
        $l \leftarrow l + incr$
```

In this Algorithm 11, $\mathbf{a}_l$ is the **propagating vector** for the pass at layer l and the variables are based on the direction of the pass, defined in Table A.1. **Note**: This section assumes that all non-linear functions, $\sigma_l(.)$ map a single dependent variable from one independent variable for all layers l. This is not the case for all activation functions including the popular Softmax function for multi-class classification problems in machine learning and the difference in the form that the affected partial derivatives take is outlined at the end of §3.6.1.

Forward Propagation I

In the first forward pass $\mathbf{a}_l^k = \mathbf{z}_l^k$, which is the propagating vector for step 5 of Algorithm 11. $\mathbf{z}_0^k$ is initialized as the vector input for the training example k, $\mathbf{x}^k$, which is set prior to training. The propagation equation for this pass is given by:

$$(\mathbf{z}_l^k)_i = \sigma_l((\mathbf{W}_l)_i \mathbf{z}_{l-1}^k + (\mathbf{b}_l)_i), \tag{A.7}$$

where $\sigma_l(.)$ is the non-linear function at layer l, and $(\mathbf{W}_l)_i$ is the i^{th} row vector of the weight matrix at layer l. This allows us to compute the output of our final layer, $\mathbf{z}_L^k$, where L is the number of layers. This forward propagation step is performed for each training example k in our set of m training examples. With this set of m vectors $\mathbf{z}_l^k$, which depend on $\mathbf{w}$, the objective function is now computed as:

$$f(\mathbf{w}) = \frac{1}{2m}\sum_{k=1}^{m}(\mathbf{z}_L^k - \mathbf{y}^k)^T(\mathbf{z}_L^k - \mathbf{y}^k), \tag{A.8}$$

where $\mathbf{y}^k$ represents the vector of dependent variables of the k^{th} training example, where training examples are presented as in (2.1), which is set at the time of training along with it's paired independent variable vector $\mathbf{x}^k = \mathbf{z}_0^k$.

Back Propagation I

In this pass the propagation vector, $\mathbf{a}_l^k$ from Algorithm 11, for each training example is the vector where the element at index i is defined as $\frac{\partial f(\mathbf{w})}{\partial(\mathbf{z}_l^k)_i}$ and the vector is of length n_l. Initializing in this case involves defining this propagation vector at layer L following our propagation procedure. Based on the objective function, the elements of this initial vector are:

$$\frac{\partial f(\mathbf{w})}{\partial(\mathbf{z}_L^k)_i} = \frac{1}{m}((\mathbf{z}_L^k)_i - \mathbf{y}_i^k). \tag{A.9}$$

This pass begins at the final layer, therefore $l = L$ for which $\frac{\partial f(\mathbf{w})}{\partial(\mathbf{z}_L^k)_i}$ has already been computed using equation (A.9) for $i = 1, ..., n_L$. The value of $\frac{\partial f(\mathbf{w})}{\partial(\mathbf{z}_L^k)_i}$ are then each computed based on the previous layer. The propagation begins with the propagation equation used in Algorithm 11 defined as:

$$\frac{\partial f(\mathbf{w})}{\partial(\mathbf{z}^k_{l-1})_i} = \sum_{j=0}^{n_l} \frac{\partial f(\mathbf{w})}{\partial(\mathbf{z}^k_l)_j}\sigma'_l((\mathbf{W}_l)_j\mathbf{z}^k_{l-1} + (\mathbf{b}_l)_i)(\mathbf{W}_l)_{ij}. \tag{A.10}$$

From this propagation procedure there is now sufficient information to compute the gradient defined as $\nabla f(\mathbf{w})$ where each element indexed by q is defined as $\frac{\partial f(\mathbf{w})}{\partial(\mathbf{W}_l)_{ij}}$ using the index mapping from (A.4). Each of these components is computed using:

$$\frac{\partial f(\mathbf{w})}{\partial(\mathbf{W}_l)_{ij}} = \sum_{k=1}^{m} \frac{\partial f(\mathbf{w})}{\partial(\mathbf{z}^k_l)_i}\sigma'_l((\mathbf{W}_l)_i\mathbf{z}^k_{l-1} + (\mathbf{b}_l)_i)(\mathbf{z}^k_{l-1})_j, \tag{A.11}$$

and

$$\frac{\partial f(\mathbf{w})}{\partial(\mathbf{b}_l)_i} = \sum_{k=1}^{m} \frac{\partial f(\mathbf{w})}{\partial(\mathbf{z}^k_l)_i}\sigma'_l((\mathbf{W}_l)_i\mathbf{z}^k_{l-1} + (\mathbf{b}_l)_i), \tag{A.12}$$

where

$$\sigma'_l(x) = \frac{d\sigma_l(x)}{dx} \quad \forall x \in \mathbb{R}. \tag{A.13}$$

For gradient methods this would be the propagation stopping point. The next two passes are required to compute Hessian related information and use values computed during these first two passes.

Forward Propagation II

In the second forward pass $\mathbf{a}^k_l = R\{\mathbf{z}^k_l\}$ for Algorithm 11. Since this is a forward pass, the vector that needs to be initialized is $R\{\mathbf{z_0}^k\}$. Because the R operator is a differential operator and $\mathbf{z}^k_0$ is the k^{th} provided independent variable, $\mathbf{x}^k$, which is set prior to training and therefore treated as a constant:

$$R\{\mathbf{z_0}^k\} = \mathbf{0}. \tag{A.14}$$

The propagation equation for Algorithm 11 is determined by applying the "R" operator to (A.7). The "R" operator follows all the rules of a differential operator so this is results in the following propagation equation:

$$\begin{aligned} R\{(\mathbf{z}^k_l)_i\} &= R\{\sigma_l((\mathbf{W}_l)_i\mathbf{z}^k_{l-1} + (\mathbf{b}_l)_i)\} \\ &= \sigma'_l((\mathbf{W}_l)_i\mathbf{z}^k_{l-1} + (\mathbf{b}_l)_i)R\{(\mathbf{W}_l)_i\mathbf{z}^k_{l-1} + (\mathbf{b}_l)_i\} \\ &= \sigma'_l((\mathbf{W}_l)_i\mathbf{z}^k_{l-1} + (\mathbf{b}_l)_i)((\mathbf{W}_l)_iR\{\mathbf{z^k_{l-1}}\} + (\mathbf{V}_l)_i\mathbf{z}^k_{l-1} + (\mathbf{b}^v_l)_i), \end{aligned} \tag{A.15}$$

since

$$R\{\mathbf{w}\} = \frac{\partial}{\partial r}(\mathbf{w} + r\mathbf{v}) \mid_{r=0} = \mathbf{v}, \tag{A.16}$$

and

$$R\{\mathbf{w}\}_q = \mathbf{v}_q, \tag{A.17}$$

which, when converted to matrix form using the mapping in (A.4), is equivalent to:

$$R\{(\mathbf{W}_l)_{ij}\} = (\mathbf{V}_l)_{ij}, \tag{A.18}$$

and

$$R\{(\mathbf{b}_l)_i\} = (\mathbf{b}_l^v)_i. \tag{A.19}$$

Back Propagation II

In this final pass the propagation vector, $\mathbf{a}_l^k$, is set as the vector of size n_l such that each element i is defined by $R\{\frac{\partial f(\mathbf{w})}{\partial(\mathbf{z}_l^k)_i}\}$. The "R" operator is applied to (A.9) for computing initial values $R\{\frac{\partial f(\mathbf{w})}{\partial(\mathbf{z}_L^k)_i}\}$ which results in:

$$R\{\frac{\partial f(\mathbf{w})}{\partial(\mathbf{z}_L^k)_i}\} = \frac{1}{m}R\{(\mathbf{z}_L^k)_i - \mathbf{y}_i^k\} = \frac{1}{m}R\{(\mathbf{z}_L^k)_i\}, \tag{A.20}$$

where $\mathbf{y}^k$ is the dependent variable vector for the k^{th} training example which is set before optimization begins and therefore $R\{\mathbf{y}^k\} = \mathbf{0}$.

For the propagation equation we can apply the "R" operator on (A.10). To simplify our final equation we can use this definition of an intermediate vector $\mathbf{h}_l^k$ where:

$$\mathbf{h}_l^k = \mathbf{W}_l\mathbf{z}_{l-1}^k + \mathbf{b}_l, \tag{A.21}$$

$$\mathbf{z}_l^k = \sigma_l(\mathbf{h}_l^k). \tag{A.22}$$

Using (A.18) and (A.19) we can compute the result of using the "R" operator on $\mathbf{h}_l^k$ as:

$$\begin{aligned} R\{\mathbf{h}_l^k\} &= R\{\mathbf{W}_l\}\mathbf{z}_{l-1}^k + \mathbf{W}_l R\{\mathbf{z}_{l-1}^k\} + R\{\mathbf{b}_l\} \\ &= \mathbf{V}_l\mathbf{z}_{l-1}^k + \mathbf{W}_l R\{\mathbf{z}_{l-1}^k\} + \mathbf{b}_l^v. \end{aligned} \tag{A.23}$$

To further simplify our equation we define a second intermediate vector made up of the partial derivatives of $\mathbf{h}_l^k$ which are computed as:

$$\frac{\partial f(\mathbf{w})}{\partial(\mathbf{h}_l^k)_i} = \frac{\partial f(\mathbf{w})}{\partial(\mathbf{z}_l^k)_i}\frac{\partial(\mathbf{z}_l^k)_i}{\partial(\mathbf{h}_l^k)_i} = \frac{\partial f(\mathbf{w})}{\partial(\mathbf{z}_l^k)_i}\sigma_l'((\mathbf{h}_l^k)_i), \tag{A.24}$$

and applying the "R" operator we get:

$$\begin{aligned} R\{\frac{\partial f(\mathbf{w})}{\partial(\mathbf{h}_l^k)_i}\} &= R\{\frac{\partial f(\mathbf{w})}{\partial(\mathbf{z}_l^k)_i}\}\sigma_l'((\mathbf{h}_l^k)_i) + \frac{\partial f(\mathbf{w})}{\partial(\mathbf{z}_l^k)_i}R\{\sigma_l'((\mathbf{h}_l^k)_i)\} \\ &= R\{\frac{\partial f(\mathbf{w})}{\partial(\mathbf{z}_l^k)_i}\}\sigma_l'((\mathbf{h}_l^k)_i) + \frac{\partial f(\mathbf{w})}{\partial(\mathbf{z}_l^k)_i}\sigma_l''((\mathbf{h}_l^k)_i)R\{(\mathbf{h}_l^k)_i\}. \end{aligned} \tag{A.25}$$

With the above definitions to simplify the result, we apply "R" to (A.10) which results in the propagation equation:

$$\begin{aligned} R\{\frac{\partial f(\mathbf{w})}{\partial(\mathbf{z}_{l-1}^k)_i}\} =& R\{\sum_{j=1}^{\mathbf{n_l}}\frac{\partial f(\mathbf{w})}{\partial(\mathbf{h}_l^k)_j}(\mathbf{W}_l)_{ij}\} \\ =& \sum_{j=1}^{\mathbf{n_l}}(R\{\frac{\partial f(\mathbf{w})}{\partial(\mathbf{h}_l^k)_j}\}(\mathbf{W}_l)_{ij} + \frac{\partial f(\mathbf{w})}{\partial(\mathbf{h}_l^k)_j}R\{(\mathbf{W}_l)_{ij}\}) \\ =& \sum_{j=1}^{\mathbf{n_l}}(R\{\frac{\partial f(\mathbf{w})}{\partial(\mathbf{h}_l^k)_j}\}(\mathbf{W}_l)_{ij} + \frac{\partial f(\mathbf{w})}{\partial(\mathbf{h}_l^k)_j}(\mathbf{V}_l)_{ij}). \end{aligned} \tag{A.26}$$

This is all the information necessary to compute all values of $R\{\frac{\partial f(\mathbf{w})}{\partial(\mathbf{W}_l)_{ij}}\}$ defined as:

$$\begin{aligned} R\{\frac{\partial f(\mathbf{w})}{\partial(\mathbf{W}_l)_{ij}}\} =& \sum_{k=1}^{m} R\{\frac{\partial f(\mathbf{w})}{\partial(\mathbf{h}_l^k)_i}(\mathbf{z}_{l-1}^k)_j\} \\ =& \sum_{k=1}^{m} R\{\frac{\partial f(\mathbf{w})}{\partial(\mathbf{h}_l^k)_i}\}(\mathbf{z}_{l-1}^k)_j + \frac{\partial f(\mathbf{w})}{\partial(\mathbf{h}_l^k)_i}R\{(\mathbf{z}_{l-1}^k)_j\}, \end{aligned} \tag{A.27}$$

and the equivalent for the bias values $(\mathbf{b}_l)_i$:

$$R\{\frac{\partial f(\mathbf{w})}{\partial(\mathbf{b}_l)_i}\} = \sum_{k=1}^{m} R\{\frac{\partial f(\mathbf{w})}{\partial(\mathbf{h}_l^k)_i}\}. \tag{A.28}$$

Since $\mathbf{Hv} = R\{\nabla f(\mathbf{w})\}$ and we have computed every element of $R\{\nabla f(\mathbf{w})\}$ where

$$\mathbf{Hv}_q = R\{\nabla f(\mathbf{w}_q)\} = \begin{cases} R\{\frac{\partial f(\mathbf{w})}{\partial(\mathbf{b}_l)_i}\} \quad \text{if } j = (n_{l-1}+1), & \text{(A.29a)} \\ R\{\frac{\partial f(\mathbf{w})}{\partial(\mathbf{W}_l)_{ij}}\} \quad \text{otherwise,} & \text{(A.29b)} \end{cases}$$

where the index mapping between $\mathbf{w}_q$ and $(\mathbf{W}_l, \mathbf{b}_l)_{ij}$ is defined by (A.4).

A.1 Complexity Analysis

In order to study speed up methods and to compare the TRM with gradient based methods we take this section to analyze the complexity of a single step of the TRM.

A.1.1 Complexity Analysis of The Pearlmutter Trick for ff-ANNs

This section presents the complexity analysis of this method in order to quantify the scalability of the method and to take a step towards computing the complexity of the full TRS algorithm. Through the analysis of each of the four passes this method is shown to be bound by $O(mn)$ computations where m is the number of training examples and n is the number of parameters (weights).

Forward Propagation I

During this first pass, the algorithm computes $\mathbf{z}_l^k$ using equation (A.7) for every layer, $l = 1, ..., L$, and training example $k = 1, ..., m$. This requires computing $\mathbf{W}_l\mathbf{z}_{l-1}^k + \mathbf{b}_l$ which takes $n_l(n_{l-1} + 1)$ computations and then computing the non-linear function $\sigma_l(.)$ for all n_l values of $\mathbf{W}_l\mathbf{z}_{l-1}^k + \mathbf{b}_l$, each of which takes constant time. Therefore the number of computations for the forward pass is:

$$\begin{aligned} T(n,m) =& \sum_{k=1}^{m}\sum_{l=1}^{L} n_l(n_{l-1}+1) + cn_l \\ =& m\sum_{l=1}^{L} n_l(n_{l-1}+1) + cn_l, \end{aligned} \tag{A.30}$$

using (A.3) we can simplify this to:

$$\begin{aligned} T(n,m) &= m(n + c\sum_{l=1}^{L} n_l) \\ &\leq mn(c+1) \\ &= O(mn) \end{aligned} \tag{A.31}$$

Back Propagation I

During the second pass, the propagation equation (A.10) computes $\sigma_l'((\mathbf{W}_l)_j \mathbf{z}_{l-1}^k + (\mathbf{b}_l)_i)$ which takes constant time determined by the non-linear function since the product $(\mathbf{W}_l)_j \mathbf{z}_{l-1}^k + (\mathbf{b}_l)_i$ has already been computed in the first forward pass. Finally we need then two constant time multiplications. This must happen for all $j \in n_l$ for a single derivative computation of which there are n_{l-1} per layer. By this logic, we have:

$$\begin{aligned} T(n,m) &= \sum_{k=1}^{m} \sum_{l=1}^{L} \sum_{i=1}^{n_{l-1}} \sum_{j=1}^{n_l} (c+2) \\ &= m \sum_{l=1}^{L} (c+2) n_l n_{l-1} \\ &= (c+2)mn \\ &= O(mn). \end{aligned} \tag{A.32}$$

The second set of values that need to be computed are the values of the gradient, $\frac{\partial f(\mathbf{w})}{\partial (\mathbf{W}_l)_{ij}}$ and $\frac{\partial f(\mathbf{w})}{\partial (\mathbf{b}_l)_i}$, which is done using (A.11) and (A.12) respectively. In this case $\sigma_l'((\mathbf{W}_l)_j \mathbf{z}_{l-1}^k + (\mathbf{b}_l)_i)$ has already been computed for (A.10) so we have m products of three scalars computed for each element value of $\mathbf{W}_l \in \mathbb{R}^{n_l n_{l-1}}$ at each layer and m products of two scalars computed for each element value of $\mathbf{b}_l \in \mathbb{R}^{n_l}$ for each layer $l = 1, ..., n_L$. Therefore we count computations contributed from (3.11) to be:

$$\begin{aligned} T(n,m) =& \sum_{l=1}^{L} \sum_{i=1}^{n_l} \sum_{k=1}^{m} (\sum_{j=1}^{n_{l-1}} 2 + 1) \\ <& 2m \sum_{l=1}^{L} n_l (n_{l-1} + 1) \\ =& 2mn \\ =& O(mn), \end{aligned} \tag{A.33}$$

therefore, the number of operations required for this pass is bounded by $O(mn)$.

Forward Propagation II

During the third pass, which uses the propagation equation (A.15), we note that the vector $\sigma_l'((\mathbf{W}_l)_j \mathbf{z}_{l-1}^k + (\mathbf{b}_l)_i)$ has already been computed during the previous pass for (A.10). The two products $(\mathbf{W}_l)_i R\{\mathbf{z^k_{l-1}}\}$ and $(\mathbf{V}_l)_i \mathbf{z}^k_{l-1}$ require n_{l-1} computations each and then are added together so we can say that there are cn_{l-1} computations total for a constant value c. The propagation equation, (A.15), is computed for $i = 1, ..., n_l$, for $l = 1, ..., L$ for all m training examples k. Therefore the runtime is:

$$\begin{aligned} T(m,n) =& \sum_{k=1}^{m} \sum_{l=1}^{L} \sum_{i=1}^{n_l} c(n_{l-1}) \\ =& cm \sum_{l=1}^{L} n_l n_{l-1} \\ =& cmn \\ =& O(mn) \end{aligned} \tag{A.34}$$

Back Propagation II

In (A.23), there are $cn_l(n_{l-1}+1)$ computations to compute $R\{\mathbf{h}^k_l\}$. Looking at (A.25), there are constant time operations for computing each value $(\mathbf{h}^k_l)_i$, call this c_1. Therefore the total computations is:

$$\begin{aligned} T(m,n) =& \sum_{k=1}^{m} \sum_{l=1}^{L} (\sum_{i=1}^{n_{l-1}} \sum_{j=1}^{n_l} c_2 + cn_l(n_{l-1}+1) + \sum_{i=1}^{n_l} c_1) \\ =& \sum_{k=1}^{m} \sum_{l=1}^{L} ((c+c_2)n_l n_{l-1} + c_1 n_l) \\ \leq& m \sum_{l=1}^{L} ((c+c_1+c_2)n_l n_{l-1}) \\ =& (c+c_1+c_2)mn \\ =& O(mn), \end{aligned} \tag{A.35}$$

where c_2 is the amount of time to compute the right side of (3.21) within the sum.

Softmax vs Sigmoid

All analysis was done for non-linear functions, $\sigma_l(.)$, that produce a single dependent variable for a single independent varaible. That is, $\sigma_l(x) = y$ where $x,\ y\ \in \mathbb{R}$. Using a function such as softmax $\sigma_l(\mathbf{x}) = y$ where $\mathbf{x} \in \mathbb{R}^w$ and $y \in \mathbb{R}$ requires an extra bit of analysis. Recall that the number of output values in a layer computation is n_l when we are iterating through multiple layers l. Here we look at what that difference is for completion of the algorithm.

Sigmoid (or any other function of one-to-one):
The sigmoid function refers to:

$$\mathbf{z}_i = \sigma(\mathbf{h}_i) = \frac{1}{(1 + e^{-\mathbf{h}_i})}, \tag{A.36}$$

where $\mathbf{z}_i, \mathbf{h}_i \in \mathbb{R}$ and the operation takes constant time. The first derivative of this function comes out to be:

$$\sigma'(\mathbf{h}_i) = \sigma(\mathbf{h}_i)(1 - \sigma(\mathbf{h}_i)), \tag{A.37}$$

which also takes constant time and maps a single input to a single output. Finally, the second derivative is defined as:

$$\sigma''(\mathbf{h}_i) = \sigma'(\mathbf{h}_i)(1 - 2\sigma(\mathbf{h}_i)), \tag{A.38}$$

which can be computed in constant time as well. Therefore derivatives in terms of the objective function for our problem in terms of the inputs to the non-linear function are:

$$\frac{\partial f(\mathbf{w})}{\partial \mathbf{h}_i} = \frac{\partial f(\mathbf{w})}{\partial \mathbf{z}_i}\frac{\partial \mathbf{z}_i}{\partial \mathbf{h}_i} = \frac{\partial f(\mathbf{w})}{\partial \mathbf{z}_i}\sigma'(\mathbf{h}_i), \tag{A.39}$$

and:

$$\frac{\partial^2 f(\mathbf{w})}{\partial \mathbf{h}_i^2} = \frac{\partial^2 f(\mathbf{w})}{\partial \mathbf{z}_i^2}(\frac{\partial \mathbf{z}_i}{\partial \mathbf{h}_i})^2 + \frac{\partial f(\mathbf{w})}{\partial \mathbf{z}_i}\frac{\partial^2 \mathbf{z}_i}{\partial \mathbf{h}_i^2} = \frac{\partial^2 f(\mathbf{w})}{\partial \mathbf{z}_i^2}\sigma'(\mathbf{h}_i) + \frac{\partial f(\mathbf{w})}{\partial \mathbf{z}_i}\sigma''(\mathbf{h}_i). \tag{A.40}$$

This confirms that applying the sigmoid function element-wise to the output vector of a layer takes $O(n_l)$ complexity which is what we assumed in our analysis.

Softmax: The softmax function is defined as:

$$\sigma(\mathbf{h})_i = \frac{e^{\mathbf{h}_i}}{\sum_{j=1}^{n_L} e^{\mathbf{h}_j}}, \tag{A.41}$$

where the output, $\sigma(\mathbf{h})_i$ is a scalar and it requires all elements of the vector $\mathbf{h}$ to be computed. The runtime for computing a single output value appears to be $O(n_L)$ based on the sum in the denominator, but can be made faster by computing the denominator once and then used for all values of $i = 1, ..., n_L$, therefore the runtime complexity bound for a single training example remains unchanged from sigmoid for forward propagation at $O(n_l)$ where $l = L$ in the case of the final layer.

The partial derivatives $\frac{\partial \mathbf{z}_j}{\partial \mathbf{h}_i}$ for all $j = 1, ..., n_l$ are defined as follows:

$$\frac{\partial \mathbf{z}_j}{\partial \mathbf{h}_i} = -\mathbf{z}_i \mathbf{z}_j \qquad \forall i \neq j, \tag{A.42}$$

and

$$\frac{\partial \mathbf{z}_i}{\partial \mathbf{h}_i} = \mathbf{z}_i(1 - \mathbf{z}_i). \tag{A.43}$$

Finally, $\frac{\partial f(\mathbf{w})}{\partial \mathbf{h}_i}$ can be computed using the sum:

$$\frac{\partial f(\mathbf{w})}{\partial \mathbf{h}_i} = \sum_{j=0}^{n-1} \frac{\partial f(\mathbf{w})}{\partial \mathbf{z}_j} \frac{\partial \mathbf{z}_j}{\partial \mathbf{h}_i} = \frac{\partial f(\mathbf{w})}{\partial \mathbf{z}_i} \mathbf{z}_i - (\mathbf{z}^T \frac{\partial f(\mathbf{w})}{\partial \mathbf{z}}) \mathbf{z}_i, \tag{A.44}$$

which we computed using (A.42) as well as (A.43), and $\frac{\partial f(\mathbf{w})}{\partial \mathbf{z}}$, is the vector such that:

$$(\frac{\partial f(\mathbf{w})}{\partial \mathbf{z}})_i = \frac{\partial f(\mathbf{w})}{\partial \mathbf{z}_i} \tag{A.45}$$

which contains some n_l computations where $\mathbf{h} \in \mathbb{R}^{n_l}$. However, the inner product $(\mathbf{z}^T \frac{\partial f(\mathbf{w})}{\partial \mathbf{z}})$ can be computed once in $O(n_l)$ time and used to compute the output values for all $i \in n_l$. Therefore, the runtime complexity remains unchanged once again for back propagation I.

Finally we look at the second order derivative of the objective function in terms of the input to the non-linear function, $\mathbf{h}$. Firstly, the second derivatives of the softmax function are:

$$\frac{\partial^2 \mathbf{z}_i}{\partial \mathbf{h}_i{}^2} = \mathbf{z}_i(1 - 2\mathbf{z}_i) + \mathbf{z}_i^2(2\mathbf{z}_i - 1), \tag{A.46}$$

$$\frac{\partial^2 \mathbf{z}_j}{\partial \mathbf{h}_i{}^2} = \mathbf{z}_j \mathbf{z}_i(2\mathbf{z}_i - 1) \quad \forall j \neq i. \tag{A.47}$$

Using these results the second derivative, $\frac{\partial^2 f(\mathbf{w})}{\partial \mathbf{h}_i^2}$, is defined as:

$$
\begin{aligned}
\frac{\partial^2 f(\mathbf{w})}{\partial \mathbf{h}_i^2} =& \sum_{j=1}^{n_l} (\frac{\partial^2 f(\mathbf{w})}{\partial {\mathbf{z}_j}^2} (\frac{\partial \mathbf{z}_j}{\partial \mathbf{h}_i})^2 + \frac{\partial f(\mathbf{w})}{\partial \mathbf{z}_j} \frac{\partial^2 \mathbf{z}_j}{\partial {\mathbf{h}_i}^2}) \\
=& \frac{\partial f(\mathbf{w})}{\partial \mathbf{z}_i} \mathbf{z}_i (1 - 2\mathbf{z}_i) + \frac{\partial^2 f(\mathbf{w})}{\partial {\mathbf{z}_i}^2} \mathbf{z}_i^2 (1 - 2\mathbf{z}_i) \\
& + \sum_{j=1}^{n_l} \frac{\partial^2 f(\mathbf{w})}{\partial {\mathbf{z}_j}^2} (\mathbf{z}_i \mathbf{z}_j)^2 + \frac{\partial f(\mathbf{w})}{\partial \mathbf{z}_j} \mathbf{z}_j \mathbf{z}_i (2\mathbf{z}_i - 1).
\end{aligned}
\tag{A.48}
$$

The summation in this equation takes $O(n_l)$ to compute, but is the same for every value of $\frac{\partial^2 f(\mathbf{w})}{\partial \mathbf{h}_i^2}$. Therefore, as before, the same runtime complexity can be achieved for both softmax and sigmoid non-linear functions. This shows that despite assuming there is a 1-to-1 mapping for the non-linear function used, the complexity analysis still applies to Softmax as the activation function used in the network.

Appendix B

Hyperparameters

As mentioned in Chapter 4, there are parameters that need to be chosen prior to training, we refer to these as hyperparameters to differentiate them from parameters or weights, which are tuned during training. The hyperparameters we consider before training are the weight subset size, b_w, the mini-batch size for trust region steps, b_m^{small}, the mini-batch size for gradient steps, b_m^{GD}, the large batch size b_m^{large}, the initial learning rate for stochastic gradient descent, γ_0^{SGD}, the initial learning rate for mini-batch gradient descent, γ_0^{MBGD}, the initial learning rate, or step size, for **STRM**, γ_0^{STRM} and finally the initial learning rate, or step size, for **MBTRM**, γ_0^{MBTRM}. These hyperparameters are chosen using a line search type test protocol shown in Algorithm 12, where $f(\mathbf{w}_{i_t})$ is the objective function achieved at iteration i_t where iteration i_t is the smallest iteration count where the CPU time taken thus far is larger than t. Values of $incr$ and p_0 for each hyperparameter and dataset can be found in Appendix D, and the **Method** is chosen based on the parameter being tested using the mapping in Table B.1.

Algorithm 12 Hyperparameter Protocol

```
 1: procedure Find Suitable Hyperparameter
 2:     Given: Dataset, $p_0$, $incr$, Hyperparameter
 3:     Method $\leftarrow$ from Hyperparameter, see Table B.1
 4:     $t \leftarrow$ CPU time for 10 iterations of TRM
 5:     $p \leftarrow p_0$
 6:     $error_0 \leftarrow f(\mathbf{w}_{i_t})$ running Method on Dataset given $p$
 7:     $i \leftarrow 0$
 8:     $error_{-1} = \infty$
 9:     while $error_i < error_{i-1}$ do
10:         $p \leftarrow p + incr$
11:         $i = i + 1$
12:         $error_i \leftarrow f(\mathbf{w}_{i_t})$ running Method on Dataset given $p$
        return $p - incr$
```

Hyperparameter	**Method**
b_w	**TRMWS**
b_m^{GD}	**MBGD**
b_m^{small}	**MBTRM**
b_m^{large}	**BTRM**
γ_0^{SGD}	**SGD**
γ_0^{MBGD}	**MBGD**
γ_0^{STRM}	**STRM**
γ_0^{MBTRM}	**MBTRM**

Table B.1: The **Method** is the algorithm used to test the **hyperparameter** in the same row. This mapping is used in Algorithm 12, row 3.

Appendix C

Full Hessian vs Hessian Free

Dataset	**IRIS** FH	**IRIS** HF	**Nurs** FH	**Nurs** HF	**Habe** FH	**Habe** HF
TRM	**6.5095**	15.526	**103.2252**	1393.098	**562.1648**	1799.944
BTRM	25.7475	**22.2905**	**254.3873**	331.7311	21.0777	**20.9922**
MBTRM	1.6608	**1.2765**	47.2323	**24.9619**	**17.3623**	17.4178
STRM	5.1491	**3.7489**	35.5067	**17.8883**	18.5672	**17.6631**
TRMMBGD	0.62793	**0.50674**	0.58953	**0.51024**	0.31192	**0.2718**
TRMWS	**6.539**	14.1964	**67.2935**	306.8893	**18.3271**	269.4662
BTRMWS	**5.701**	11.1422	**17.5241**	51.3723	**11.4665**	21.4033
MBTRMWS	**1.9442**	3.4353	**7.2354**	13.1268	**9.8883**	16.7996
STRMWS	**4.6339**	8.0689	**6.4347**	10.7207	**10.3431**	18.0074

Table C.1: Time taken to compute min(1000,max iterations) using the full H approach and the HF approach.

Dataset	**MNIST** FH	**MNIST** HF	**Derm** FH	**Derm** HF
TRM	3569.335	**1731.513**	21.7445	**14.5365**
BTRM	1798.383	**235.8341**	151.3577	**45.5098**
MBTRM	645.3847	**46.7114**	109.1076	**22.4816**
STRM	398.102	**27.7807**	36.0154	**8.4558**
TRMMBGD	1.224	**1.1368**	**0.31795**	0.34063
TRMWS	**203.0851**	1114.619	**9.7145**	28.1894
BTRMWS	**84.9487**	168.2657	**13.1542**	25.5094
MBTRMWS	**31.547**	32.5771	**10.7201**	18.7424
STRMWS	**20.5796**	21.3626	**9.7782**	16.4827

Table C.2: Time taken to compute min(1000,max iterations) iterations using the full H approach and the HF approach.

Appendix D

Hyperparameter Test Results

Dataset	γ_0 **SGD**	γ_0 **MBGD**	γ_0 **STRM**	γ_0 **MBTRM**	$\mathbf{b_w}$	$\mathbf{b}_m^{small}$	$\mathbf{b}_m^{large}$	$\mathbf{b}_m^{MBGD}$
MNIST	10	10	1.000000e-02	1	40	100	1000	100
Derm	1	10	1	1	15	20	75	10
IRIS	1	1	0.01	1	5	15	50	5
Nurs	1	10	0.1	0.1	5	60	1500	40
Habe	0.01	0.1	0.1	0.1	15	10	75	5
XOR	0.1	0.1	0.001	1	4	4	4	2

Table D.1: Final objective function achieved for training an ff-ANN on the set of datasets using SGD and using TRM.

References

[1] Herve Abdi, Dominique Valentin, and Betty Edelman. Neural networks (quantitative applications in the social sciences). *Sage University paper, Series*, (5):10, 1999.

[2] Chuan Yu Foo Yifan Mai Caroline Suen Adam Coates Andrew Maas Awni Hannun Brody Huval Tao Wang Sameep Tandon Andrew Ng, Jiquan Ngiam. Ufldl tutorial. http://ufldl.stanford.edu/tutorial/.

[3] Nicholas I. M. Gould Andrew R. Conn and Philippe L.Toint. *Trust-Region Methods*. MPS and SIAM, Philadelphia, PA, 2000. Series on Optimization.

[4] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016.

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

[6] Yann N Dauphin, Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, Surya Ganguli, and Yoshua Bengio. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In *Advances in neural information processing systems*, pages 2933–2941, 2014.

[7] James W. Demmel. *Applied Numerical Linear Algebra*. SIAM, Philadelphia, PA, 1997.

[8] Vladislav Rajkovic et al. Nursery database, 1989.

[9] R.A. Fisher. Iris plants database, 1936.

[10] R. Fletcher. *Practical Methods of Optimization*. John Wiley Sons, New York, 2nd edition, 1987.

[11] Gene H Golub and Charles F Van Loan. *Matrix computations*, volume 2. JHU Press, 1989.

[12] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[13] Daniel Graupe. *Principles of artificial neural networks*, volume 7. World Scientific, 2013.

[14] Caglar Gulcehre, Jose Sotelo, Marcin Moczulski, and Yoshua Bengio. A robust adaptive stochastic gradient method for deep learning. *arXiv preprint arXiv:1703.00788*, 2017.

[15] Nilsel Ilter H. Altay Gubenir, Gulsen Demiroz. Dermatology database, 1998.

[16] S.J. Haberman. Haberman's survival data, 1976.

[17] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. Overview of supervised learning. In *The elements of statistical learning*, pages 9–41. Springer, 2009.

[18] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[19] Michael I. Jordan Benjamin Recht Jason D. Lee, Max Simchowitz. Gradient descent only converges to minimizers. *JMLR: Workshop and Conference Proceedings vol 49*, pages 1–12, 2016.

[20] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[21] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010.

[22] M. Lichman. UCI machine learning repository, 2013.

[23] Qing Ma. Natural language processing with neural networks. In *Language Engineering Conference, 2002. Proceedings*, pages 45–56. IEEE, 2002.

[24] Maren Mahsereci, Lukas Balles, Christoph Lassner, and Philipp Hennig. Early stopping without a validation set. *arXiv preprint arXiv:1703.09580*, 2017.

[25] James Martens. Deep learning via hessian-free optimization. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 735–742, 2010.

[26] Marvin Minsky and Seymour Papert. *Perceptrons.* M.I.T Press, Oxford, England, 1969.

[27] B. A. Pearlmutter. Fast exact multiplication by the hessian. *Neural Computation 6.1*, pages 147–160, 1994.

[28] Kevin L Priddy and Paul E Keller. *Artificial neural networks: an introduction*, volume 68. SPIE press, 2005.

[29] Yuji Nakatsukasa Satoru Adachi, Satoru Iwata and Akiko Takeda. Solving the trust region subproblem by a generalized eigenvalue problem. Technical report, The University of Tokyo, Tokyo, Japan, 2015.

[30] Adrian J. Shepherd. *Second-Order Methods for Neural Networks: Fast and Reliable Training Methods for Multi-Layer Perceptrons.* Springer, 1997.

[31] Jonathan Richard Shewchuk et al. An introduction to the conjugate gradient method without the agonizing pain, 1994.

[32] Gerald A Shultz, Robert B Schnabel, and Richard H Byrd. A family of trust-region-based algorithms for unconstrained minimization with strong global convergence properties. *SIAM Journal on Numerical Analysis*, 22(1):47–67, 1985.

[33] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[34] Andrew JR Simpson. " oddball sgd": Novelty driven stochastic gradient descent for training deep neural networks. *arXiv preprint arXiv:1509.05765*, 2015.

[35] Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of machine learning research*, 15(1):1929–1958, 2014.

[36] Z Strakoš. On the real convergence rate of the conjugate gradient method. *Linear algebra and its applications*, 154:535–549, 1991.

[37] Igor V Tetko, David J Livingstone, and Alexander I Luik. Neural network studies. 1. comparison of overfitting and overtraining. *Journal of chemical information and computer sciences*, 35(5):826–833, 1995.

[38] Vladimir N Vapnik. An overview of statistical learning theory. *IEEE transactions on neural networks*, 10(5):988–999, 1999.

[39] Vladimir N Vapnik and A Ya Chervonenkis. On the uniform convergence of relative frequencies of events to their probabilities. In *Measures of Complexity*, pages 11–30. Springer, 2015.

[40] Stephen J Wright and Jorge Nocedal. Numerical optimization. *Springer Science*, 35(67-68):7, 1999.

[41] Chao Yang. The implicitly restarted arnoldi method. http://www.caam.rice.edu/software/ARPACK/UG/node45.html, 1997.