EPSRC
Engineering and Physical Sciences Research Council

InFoMM
Industrially Focused Mathematical Modelling

EPSRC Centre for Doctoral Training in Industrially Focused Mathematical Modelling

Optimisation methods for machine learning applications

Ioan Alexandru Puiu

UNIVERSITY OF OXFORD

nag®

Contents

1 Introduction

Machine Learning algorithms perform inference tasks based on an input.

Have you ever wondered how facial recognition works or how *Siri* or *Google Assistant* are able to support you in daily tasks through voice communication? At the core of these artificial intelligence lie Machine Learning (ML) algorithms. These smart algorithms learn to perform inference tasks such as classification, prediction and many others (e.g. face recognition, automatic text correction). But how do these algorithms learn and what makes them "*smart*"? Every Machine Learning algorithm is characterised by a set of parameters, that define its ability to perform a task. These parameters are learned by exposing the algorithm to data, improving its performance. The process of learning translates to a mathematical optimisation problem.

The Numerical Algorithms Group (NAG) is an international technology company which focuses on mathematical software, high performance computing, and associated services. Mathematical optimization is one of the key focus areas of NAG. Machine learning is an ubiquitous source of optimization problems, where a good understanding of the most appropriate optimisation methods can make huge a difference in terms of algorithms' performance.

Glossary of terms

- **Train:** the optimisation process of finding the paramters of ML algorithms.
- **Objective function:** the cost function we are trying to minimise.
- **Step:** the result of one iteration in the optimisation algorithm (the change made).
- **Step size:** the length of the distance taken in the chosen direction.
- **Descent direction:** direction along which the objective function decreases
- **Representation capacity:** the ability of a function to reproduce some behaviour
- **Classification task:** the problem of assigning an input to a class.
- **Autoencoding task:** the problem of reducing the dimensionality of the data.
- **MSE:** Mean Squared Error, a common type of objective function in ML.
- **Hyperparameters:** chosen, fixed parameters of the optimisation algorithm.
- **Epoch:** one pass of the optimisation algorithm through the whole data.

Optimisation is the problem of finding the set of values that minimise a chosen cost function.

An optimisation problem involves finding the best set of the problem's variables according to some performance measure. Without loss of generality, this can always be reduced to finding the set of values that minimise some *cost*. In the machine learning context, the values are the parameters, and the cost is the sum of all *wrong inferences*. By naively looking just in the neighbourhood of a set of values, we can find the best values within, but this does not guarantee they are the absolute best. Thus, an optimisation problem can have many *local* solutions, and sometimes finding the *global* (absolute best one) can be very challenging or even impossible. This is ubiquitous in Machine Learning and usually, at best, *local* solutions are found.

Standard first order methods are slow.

Consider a landscape and assume we wish to find a peak. If we couldn't see very far we would walk a certain distance in the direction which gives largest height increase (largest slope), re-evaluate the largest slope at the new location, walk again, and so on. Mathematically, this is the approach taken by *first order (gradient) methods*. However, although these methods involve using the largest gradient at a particular point, the methods do not guarantee fast convergence to the solution. Thus, more advanced techniques also include the change in the slope (curvature) in order to decide the best walking direction. These are known as second order methods and use slope and curvature information.

Traditionally, second order methods are preferred. They decrease the computational time per step compared to first order methods, because they use more information. However, there is an extra cost associated with computing the curvature information, since we need at least two slopes in one direction to get an estimate. For traditional optimisation problems this extra cost is not an issue. However, in Machine Learning, the number of

In standard form, second order methods are infeasible for ML.

First order stochastic algorithms are the standard optimisation algorithm choice in ML.

parameters (d), and thus possible search directions, can be huge (in the order of millions). Moreover, the data size (n) can also huge, and thus computing the value of the objective function is expensive as well. In this setting, calculating the slope (gradient) becomes very computationally expensive while obtaining the curvature becomes infeasible. This makes the optimisation problems very challenging, with many days of computations potentially being required even on largest super-computers.

Stochastic algorithms have been developed to reduce the computational costs. They use a much smaller subset of data picked at random at each iteration. In this context, first order methods perform well due to their simplicity, and are the standard choice. However, they can sometimes stall or find bad solutions. Unfortunately, second order methods are still infeasible in general, due to quadratic increase of cost with the number of parameters. However, the hope is that by using approximations or problem specific properties, second order methods could become feasible whilst also retaining their core strengths.

Our aim is to develop an overview of the latest algorithmic development of optimisation methods for deep neural networks as a representation of hard optimisation problems within machine learning, in order to determine whether second order methods can efficiently leverage the extra information to perform better in stochastic settings than first order methods.

2 Optimisation Methods for Deep Learning

A deep neural network is composed of layers of interconnected neurons that process the data sequentially.

We restrict our attention to Deep Learning, which is one of the most popular Machine Learning branches. Deep learning is the state-of-the-art method for image or voice recognition problems, and also performs well in other tasks. Deep Learning results in very challenging optimisation problems due to the large number of parameters and data points involved in most applications. Deep Learning algorithms use neural networks, which are inspired in the brain's ability to process data. A network is composed of layers of neurons that take an input and return an output. In the simplest case, the neurons between consecutive layers are interconnected. These connections must be trained and they correspond to the parameters that need to be optimised and determine the network's performance on the given task.

In figure 1, we show an example of a neural network. Information is propagated left to right, where each layer of neurons takes an input and returns an output. In the simplest case, neurons store a single value. Operations are performed based on connections, at neuron level, and the output is then used by the neurons connected to the right.

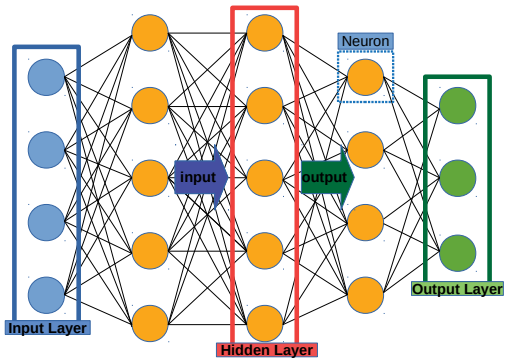

Figure 1 – Example of a deep neural network. The network contains an input layer (blue), hidden layers (orange), where the data is processed, and the output layer (green), that returns the inferred result.

Clearly, the more neurons and connections (parameters), the higher the capacity to make inferences. However, it also becomes much harder to find the optimal parameters as the size of the network increases. Thus, very large neural networks have good representation capacity, but they are very hard to train, since the number of connections is very large. In addition, to avoid wrong inferences, a very large number of training examples are needed, which further complicates the problem.

First and second order order optimisation methods

To begin with, all optimisation methods start with an initial guess, which is then improved iteratively. We refer to the estimated solution at each step as an iterate. The hope is that, by performing good steps or iterations, the algorithm will converge to a local minimum. Once a local minimum is reached, no further improvement can be achieved.

First order methods only consider the slope information for deciding the movement direction. Assuming we know a direction along which the cost decreases, we need to determine the appropriate step size. Due to the stochastic nature of the problem and computational costs, computation of the best step size is infeasible and thus the size is usually just chosen in advance. This makes convergence to the exact local minimum impossible for gradient methods. Moreover, the estimate of the descent direction might be rather poor, potentially meaning small (or sometimes negative) progress per iteration. Nevertheless, the time taken per iteration is small, and so these methods usually perform very well. Adaptive first order methods, that account for the local landscape, were introduced to overcome the limitations on step size choice. The most popular first order methods for deep learning are:

- **Stochastic Gradient Descent** (SGD), which is the simplest algorithm . It uses fixed step size or a simple pre-defined step size decay schedule, and thus is not adaptive.
- **Adaptive momentum** (ADAM), which is the most popular and successful first order method. It includes information about gradient mean and variance to scale the step. This is usually the standard optimisation algorithm in Deep Learning.
- **AMSGrad**, which is variant of ADAM, introduced because ADAM does not always converge in practice. This algorithm performs more cautious steps, and thus is slower in general, but very effective in situations where ADAM fails.

Second order methods re-scale the step size using *curvature* information and thus they naturally adapt to the local landscape. A larger curvature means a greater change in the slope and thus the slope value at the current location is a good estimate in a smaller neighbourhood of the current location. In this situation, the step should be small to avoid arriving at an unpredictable, and possibly bad, location. For this reason, in the simplest, one dimensional, case the effective step is *inversely proportional* to the curvature.

In figure 2, we see that the first order method performs well in the left hand case, but when we double the curvature, as in the right hand case, we find that the solution does not converge when using the same initial guess and step size.

First order methods are generally very sensitive to their step size.

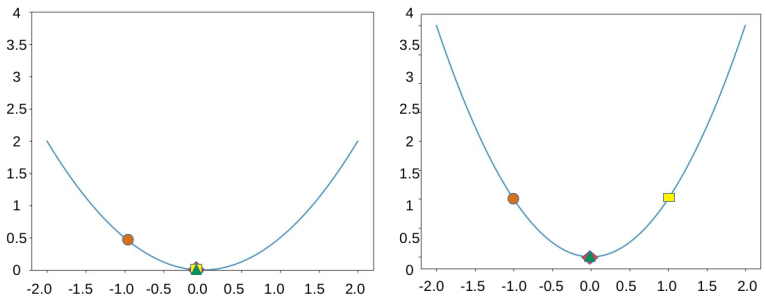

Figure 2 – Graphs illustrating the benefit of second order methods (scale invariance). Starting from the same initial guess (orange disk), the second order method (green triangle) finds the solution (red diamond) in one iteration for both cases.

Although superior, in very high dimensional cases, second order methods become very expensive. This is because for a d dimensional space, we need to evaluate d^2 curvatures. Since in deep learning d is in the order of millions or larger, storing or computing d^2 curvatures is infeasible. Moreover, traditionally, these curvatures need to be computed for every data point (there are n of them), which further complicates the problem. A subset of $m \ll n$ points can be chosen, but this can be detrimental since the curvature is more susceptible to noise.

The computationally infeasible dependence on n is usually addressed by sub-sampling m points at random for each iteration. However, further cost reduction is required. Thus far, there are two types of ideas that result in successful methods:

- **Block diagonal approximations** try to retain and use most of the curvature information by using the network's layer structure. Curvatures are computed either by considering the parameters in isolated layers or in groups of three consecutive layers. The intuition is that parameters in a layer depend mainly on the other parameters in the same layer. Two recent methods use this approximation for the *Fisher Information Matrix* and are known as KFAC and EKFAC, whilst KFRA uses the same ideas for the *Gauss-Newton Matrix*.
- **Using computational tricks for the Gauss Newton method** can linearise the d^2 cost and thus make the computations feasible. To perform this computational trick, a further requirement that the step should be small, is introduced. This algorithm is known as *Gram Gauss-Newton (GGN)* and its practical use is regression problems with a small number of outputs.

3 Comparison of Methods

In the previous section we introduced first- and second-order optimisation methods for deep learning. We restrict our attention to the ones we found to be most successful, which are:

- **First order methods**: SGD, ADAM (denoted in Tables as A) and AMSGrad,
- **Second order methods:** KFAC (denoted in Tables as K), EKFAC (denoted in Tables as E) and GGN,
- **Second order method used as preconditioner for first order**: KFAC + ADAM (denoted in Tables as K+A), EKFAC + ADAM (denoted in Tables as E+A).

Testing Framework

To evaluate these methods, we used 7 test problems. The first six are formed by using the *MNIST*, *CIFAR10* and *FACES* datasets on classification and autoencoding problems. This gave us a range of parameters from $d = 4,712$ to $d = 10,222,326$. Unfortunately, running the current form of the GGN on these problems is infeasible. Thus, the last test problem is designed for the regime where the GGN algorithm is computationally feasible. We describe the test problems below:

The testing data sets and networks were chosen to represent typical small and medium scale deep learning applications.

- **Test Problem I - Classification task** on the MNIST dataset. The neural network contains four layers and has a total of $d_I = 4,712$ parameters to be optimised.
- **Test Problems II and III - Classification task** on CIFAR10 and FACES data sets. The network comprises of 7 layers which gives a total of $d_{II} = 686,538$ and $d_{III} = 703,001$ parameters for the two data sets, respectively;
- **Test Problems IV, V and VI - Autoencoding task** on the MNIST, CIFAR10 and FACES data sets. The network consists of 10 fully connected layers, which gives $d_{IV} = 8,173,814$, $d_V = 9,134,054$ and $d_{VI} = 10,222,326$ parameters for the three data sets, respectively.
- **Test Problem VII - 1D Regression task** on the MNIST data set. The network has 8 layers and the total number of parameters to be optimised is $d_{VII} = 713,251$.

Further, for each of the problems I to VI, we test the methods for a small batch size ($m = 64$) and a large one ($m = 1000$). We introduce four metrics in order to provide a framework for drawing conclusions about the performance of the algorithms. These are:

Four performance metrics are chosen for systematic method comparison.

- convergence speed with respect to time (ST),
- convergence speed with respect to number of iterations (SI),
- generalisation ability, which is given by the best value obtained on the test set (G),
- best (minimum) objective function value achieved (MV).

For each of these test problems, we rank each method for each performance metric, with 1 being the best. Methods that perform virtually the same are given the same rank. Due to the stochastic nature of the algorithms, we performed a small number of runs of the algorithms for each case. However, we only present results for one instance. Nevertheless, our results seemed to be robust with respect to changing the *random computational seed*.

Performance of the methods

We illustrate our results in Table 1 by presenting the average rankings over the four performance metrics for each of the methods. We observe that $KFAC + ADAM$ seems to perform best, whilst KFAC and EKFAC could outperform $ADAM$, for large enough batch size. We summarise the most important results in Section 4. As an example, we show the convergence plot for two problems in figure 3.

In general, $KFAC + ADAM$ outperforms all the other methods.

	A	K	E	K+A	E+A
ST	2.66	3.4	3	**1.33**	2.5
SI	2.9	3.66	2.4	**1.33**	2.5
G	2	3.6	2.33	**1**	2
MV	2.5	3.13	2.4	**1.16**	2

	A	K	E	K+A	E+A
ST	2.4	2.1	3.2	**1.6**	-
SI	2.7	2.1	3.2	**1.6**	-
G	2.6	1.9	3.2	**1.4**	-
MV	2.7	1.9	3	**1.6**	-

Table 1 – Averaged rankings over the first six problems for batch size of 64 **(left) and** 1000 **(right)**

GGN generally achieves best progress per iteration and objective function value.

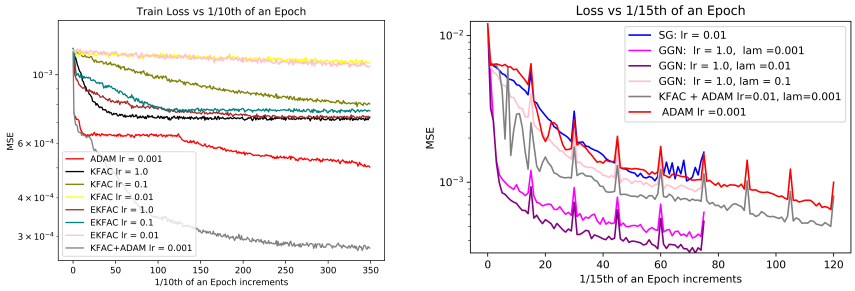

Figure 3 – Problem VI with batch size 1000 **(left) and Problem VII with batch size** 4096 **(right). Different hyper-parameter values are shown to illustrate algorithms' dependence on these.**

We see a strong dependence of KFAC and EKFAC on their hyper-parameters. Moreover, for Problem VI, we could not find hyper-parameter values that would result in these methods outperforming $ADAM$. In contrast $KFAC + ADAM$ and $ADAM$ perform well with standard hyper-parameter values. We observe that GGN is somewhat sensitive to its hyper-parameters, although finding good values seems much easier, and the method outperforms all the others.

4 Discussion, conclusions, & recommendations

We have considered various optimisation routines for problems arising in Deep Learning. In particular we assessed the performance of these routines on seven test problems. We summarise the key points for each algorithm:

KFAC+ADAM is generally the best algorithm choice if good performance is desired with little parameter tuning and relatively low computational cost. The method seems very reliable for standard parameters and the most robust to data sets or network architecture changes. $KFAC + ADAM$ generally outperforms $ADAM$, which was previously considered to be state of the art. Moreover, due to its robustness, it generally outperforms *KFAC* and *EKFAC*, at least when not much parameter tuning is performed.

KFAC and EKFAC seem to be able to achieve the best objective function value with sufficient hyper-parameter tuning. Moreover, these methods can outperform $ADAM$ even in terms of convergence speed, at least for large d and batch size. However, the methods' performances are observed to be sensitive to parameter changes. Thus finding

KFAC and EKFAC might perform best with enough fine tuning.

the right hyper-parameters can be challenging, especially since their number is relatively large. Moreover, sometimes, the methods can blow up. Whilst $EKFAC$ seems to outperform $KFAC$ in certain situations, finding the right hyper-parameters seems more challenging. $KFAC$ and $EKFAC$ are thus most appropriate (i) for large d, (ii) when sufficient computational power, and (iii) time is available for fine tuning, and a very accurate solution is desired.

ADAM and AMSGrad seem to be the most versatile first order methods. They perform very well in most situations when using standard hyper-parameters. ADAM is a good choice if a fast solution with little tuning is required, since it is generally faster than $KFAC + ADAM$. However, it can stall sometimes, but in those situations we observe $AMSGrad$ performs really well. Nevertheless, if $ADAM$ does not stall, then it generally performs better than $AMSGrad$, and so the methods could be run in parallel. However, from our numerical experiments, there does not seem to be a clear scenario where these methods would be preferred over their second order counterparts.

GGN seems to outperform all the other methods in the setting where it is computationally efficient. This is expected, since it captures the most curvature information and does not make any curvature approximations. Increasing the batch size gives larger function decrease per iteration but increases the computational cost. The method is observed to be somewhat sensitive to its damping parameter, which offers a trade-off between noise level and convergence speed.

We conclude that second order methods can outperform first order ones even in stochastic settings, although finding good hyper-parameters seems more challenging, at least for $KFAC$ and $EKFAC$. This is because the accuracy of the approximation depends on the values of the hyper-parameters and the optimisation problem being solved. Nevertheless, using $KFAC$ as a preconditioner for $ADAM$ seems very robust and constantly achieves the best results with standard parameter choices.

Finally, GGN seems to perform best even when compared to $KFAC + ADAM$. However, the method does not scale up well for high dimensional output and thus its use is rather limited. Nevertheless, the potential of the Gauss Newton method is clear, and our analysis suggests that much is to be gained if we develop a computationally efficient method that retains the core strengths for a large number of output.

5 Potential Impact

For NAG, it is important to understand many different types of optimisation problems and their characteristics in order to be able to choose the best solution for their customers. Machine learning is a rich source of optimisation problems, however, the size of the data combined with the structure of the optimisation problem prevents the application of any classical optimisation methods. Our overview and assessment of the latest optimisation methods suitable for training deep neural networks, as a representative of hard optimisation problems within machine learning, will help NAG to improve their machine learning services.

Jan Fiala, a senior technical consultant at NAG, said: *"We always try to explore new areas either to understand how to advance our mathematical software, the NAG Library, or to support our services offering. Machine learning is a very active field where new ideas and publications appear almost on weekly basis. Alex's mini-project gives us a current snapshot and assessment of the trends in higher order methods used in training of neural networks and we will be able to build further on top of the insight gained."*